Unifying State and Policy-Level Explanations for Reinforcement Learning

Nicholay Topin

July 2022
CMU-ML-22-103

Machine Learning Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Manuela Veloso, Chair
Tom Mitchell
Ameet Talwalkar
Marie desJardins, Previously at Simmons University

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

This research was funded by: Air Force Research Laboratory award FA87501720152; Office of Naval Research award N000141712899; and a National Defense Science and Engineering Graduate Fellowship.

Keywords: reinforcement learning, explainable AI, interpretable AI, deep learning

To my friends, mentors, and family.

Abstract

Reinforcement learning (RL) is able to solve domains without needing to learn a model of the domain dynamics. When coupled with a neural network as a function approximator, RL systems can solve complex problems. However, verifying and predicting RL agent behavior is made difficult by these same properties; a learned policy conveys "what" to do, but not "why."

This thesis focuses on producing explanations for deep RL, summaries of behavior and their causes that can be used for downstream analysis. Specifically, we focus on the setting where the final policy is obtained from a limited, known set of interactions with the environment. We categorize existing explanation methods along two axes:

1. Whether a method explains single-action behavior or policy-level behavior
2. Whether a method provides explanations in terms of state features or past experiences

Under this classification, there are four types of explanation methods, and they enable answering different questions about an agent. We introduce methods for creating explanations of these types. Furthermore, we introduce a unified explanation structure that is a combination of all four types. This structure enables obtaining further information about what an agent has learned and why it behaves as it does.

First, we introduce CUSTARD, our method for explaining single-action behavior in terms of state features. CUSTARD's explanation is a decision tree representation of the policy. Unlike existing methods for producing such a decision tree, CUSTARD directly learns the tree without approximating a policy after training and is compatible with existing RL techniques.

We then introduce APG-Gen, our approach for creating a policy-level behavior explanation in terms of state features. APG-Gen produces a Markov chain over abstract states that enables predicting future actions and aspects of future states. APG-Gen only queries an agent's Q-values, making no assumptions about an agent's decision-making process.

We integrate these two methods to produce a Unified Explanation Tree (UET). A UET is a tree that maps from a state directly to both an action and an abstract state, thus unifying single-action and policy-level behavior explanations in terms of state features.

We extend existing work on finding important training points in deep neural networks. Our method, MRPS, produces explanations of single-action behavior in terms of past experiences. MRPS can find importance values for sets of points and accounts for feature magnitudes to produce more meaningful importance values.

Finally, we find the importance values of sets of past experiences for any node within a UET. Additionally, we introduce methods for computing approximate and exact influence for UET nodes. Since a UET conveys both single-action and policy-level behavior, these importance and influence values explain both levels of behavior in terms of past experiences. Our overall solution enables identifying the portion of the UET that would change if specific experiences were removed or added from the set used by the agent.

Acknowledgements

I want to thank my family, who taught me to always keep learning.

I would like to thank my advisor, Manuela Veloso, who has been crucial in completing this thesis. Her guidance throughout my PhD has been invaluable, and her insights and questions played a critical role in the development of my work. When I first began working on reinforcement learning, I had the pleasure of attending a talk by Manuela. Since then, I wanted to be part of her lab. Having her as my advisor has been every bit as great as I had dreamed.

I also want to thank Marie desJardins, another member of my committee and my research supervisor during undergrad. She was the one who initially introduced me to reinforcement learning and who encouraged me to pursue a PhD. When I was just an enthusiastic highschool student, she graciously offered me a summer research assistant position. This experience galvanized my interest in artificial intelligence and motivated me to focus on research. Without her influence, this thesis would likely not exist.

I would also like to thank the other two members of my committee: Tom Mitchell and Ameet Talwalkar. Tom gave me the opportunity to TA for his reinforcement learning class which he was co-teaching. This was an amazing experience and reminded me how much I enjoyed teaching. I have enjoyed collaborating with Ameet and others in his research group. He has been crucial in shaping my understanding of explainability and its role in machine learning.

I also would like to thank Mark Owen, my NDSEG mentor. His encouragement made completing my thesis substantially easier. Through the talk opportunities which he arranged I was able to meet many new researchers that I would not have met otherwise.

I also want to thank all my collaborators on the MineRL project. It was incredibly rewarding to launch a new ML competition centered around a new task and dataset.

Finally, I would like to thank my lab mates at CORAL and the other MLD students, both current and past. I am grateful for all of the advice and input I received as well as the encouragement and support. All of these aspects were essential to making my time at CMU as great as it has been.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Recent advances in neural networks have enabled the use of powerful function approximators alongside reinforcement learning (RL) techniques to solve difficult problems. However, the deployment of RL systems may be hampered by the difficulty to verify and predict the behavior of RL agents, as well as an inability to intervene based on this knowledge. Existing work on Explainable Reinforcement Learning (XRL) explains either single-action behavior or policy-level behavior. *Single-action behavior* explanations identify causes for an agent's action choice in a single state. *Policy-level behavior* explanations illustrate the long-term behavior of the agent. We focus on explanations that convey the significance of either state features or past experiences. *State feature* explanations identify the features of a state that affect how an agent treats that state. *Past experience* explanations show which past experiences led the agent to learn the current behavior.

In this thesis, we advocate for the use of explanations that simultaneously provide information of all four types. A unified explanation identifies the link between local behavior and policy-level behavior, so it enables inferences about the cause of policy-level behavior. Furthermore, identifying the impact of past experiences on both levels of behavior is a step toward predicting the global effects of a local intervention.

1.1 Thesis Question

This thesis seeks to answer the question,

> *How can we endow agents in an **RL setting** with the ability to **explain single-action** and **policy-level** behaviors in terms of both **state features** and **past experiences**?*

We focus on the **RL setting**, where an agent interacts with an environment by selecting an action, observing the resulting transition, and obtaining a reward. Notably, an agent is not permitted arbitrary queries to a domain model nor direct access to the domain dynamics. We believe this setting reflects properties present in real-world environments. For this same reason, we avoid learning approximate models of the environment, since we understand that modeling real-world environments is often not feasible. Finally, in our setting, an agent does

not necessarily explore the entire state-action space, so we assume a policy is learned from a limited (even if rich) set of past experiences.

By "**explain**," we mean the process of creating intermediate representations that serve to summarize agent behavior. We refer to such representations as *explanations*. These explanations can then be used for down-stream use-cases such as verifying agent correctness or safety (e.g., ensuring the agent takes certain actions only when certain conditions are met). We do not necessarily seek to produce human-interpretable representations; our focus is on the summarization aspect rather than how to best convey information to a human user.

Single-action behaviors are the most fine-grained behavior of an agent, so an explanation at this level permits analyzing local behavior at individual time steps. Moving beyond this low-level explanation, we seek to summarize an agent's overall behavior, competency, and policy complexity. We refer to explanations of such a global type as **policy-level** explanations. With both kinds of explanations, one can not only predict individual actions but also approximate entire trajectories.

Finally, we consider explanations in terms of both **state features** and **past experiences**. State features are the basic unit in which an agent receives information prior to selecting an action; by producing explanations in terms of this same unit, an agent can convey "what" behavior it has learned. By referring to past experiences, an agent is able to justify behavior based on past interactions with the environment. As a result, the learning process itself can appear in the explanations, allowing the agent to explain "why" an action was learned in addition to "what" was learned.

This thesis presents algorithms for creating explanations of all the aforementioned types. We also present an approach for joining these techniques to create a single explanatory structure. Furthermore, we experimentally evaluate the introduced methods on domains designed to validate a method's ability to avoid common pitfalls in explaining RL agents.

1.2 Motivating Example

To further motivate the explanation types we consider in this thesis, we describe an example GridWorld scenario.

1.2.1 Problem Description

Figure 1.1a shows the environment. In this environment, an agent is tasked with navigating to the goal, the top-left corner. Each state consists of two features: an x-coordinate value and a y-coordinate value. In the figure, the states are arranged based on their spatial relationships, following GridWorld conventions. Actions take the form of "move in this direction until an obstacle (e.g., edge boundary) is reached." Note that these actions differ from the common "move one unit in this direction" format; each action effectively moves the agent down the entire length of a hallway. A reward of -1 is provided for each action, except for some instances where a reward of 0 is randomly provided. An episode starts in a random state and ends when the goal state, s_G, is reached.

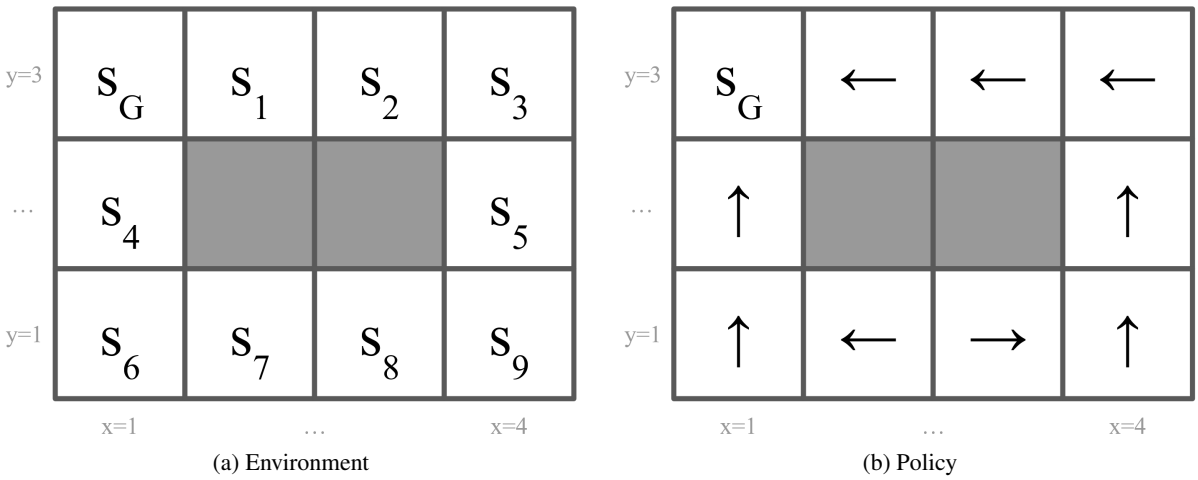


Figure 1.1: An example GridWorld environment (a) and a learned policy (b) overlaid onto this environment.

A hypothetical learned policy is shown in Figure 1.1b, overlaid onto the grid of states. In this example, a policy was first learned in an online fashion, and the final policy is the result of training to convergence with the final gathered set of experiences (i.e., training with the limited set of experiences until the policy stops changing). This policy is not necessarily optimal–for illustrative purposes, we consider a situation where the domain may not behave exactly as desired.

When an agent follows this policy, from some starting states, it takes three actions to reach the goal. An expert familiar with the domain may have the expectation that the traversable path is a rectangle, so any corner should be reachable within two actions. This outcome deviates from this expectation. Therefore, a human observer may wish to know why the agent behaves as it does and, secondarily, whether there is an intervention that yields better behavior.

The true cause of sometimes requiring three actions to reach the goal is taking the "move right" action in state s_8 (marked with hatching in Figure 1.2). If the policy were to move left in state s_8 instead, then the goal state is reached using one or two actions regardless of starting state. Note that, in non-toy environments, manual inspection of a policy (as can be done via Figure 1.1b) is not feasible, and even full enumeration of all states can be difficult.

1.2.2 Benefits of Explanations

Different explanation formats can provide different insights, which we briefly demonstrate here along with how these insights can build on each other.

Decision Tree Policy To determine how **feature values** influence **single-action behavior** (i.e., which immediate action is chosen), one could consult a decision tree policy, as shown in Figure 1.3. This is a decision tree that maps from a state to an action, so traversing the tree reveals which features impact a given choice. Investigating the leaves reveals a leaf where the "move right" action is taken (marked with hatching in the figure). Since the goal is maximally

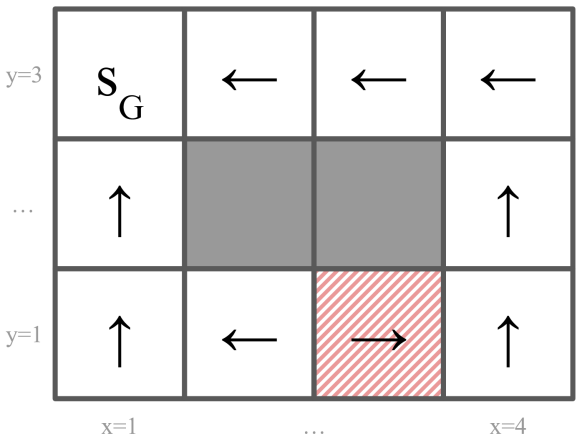


Figure 1.2: A policy for the example environment resulting from a limited set of past experiences. The suboptimal action in s_8 is hatched.

left in the rectangular region, intuition about this environment suggests that the "move right" action should never be taken. Therefore, the set of states where an incorrect action is chosen can be identified via the tree (i.e., s_8).

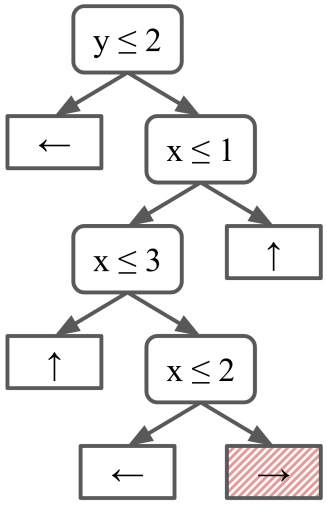


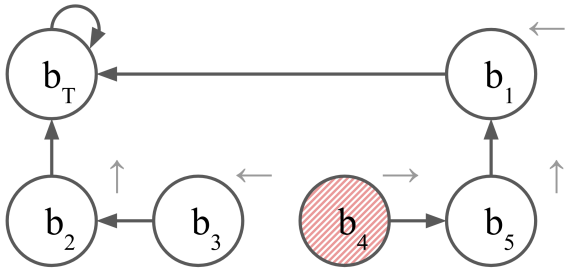
Figure 1.3: A Decision Tree Policy representation of the same policy as before. The leaf corresponding to s_8 is hatched.

Behavior Summarization To determine how **feature values** influence **policy-level behavior** (i.e., which sequence of actions will be taken), one could reference a Markov chain over abstract states, as shown in Figure 1.4b. This explanation format groups states into abstract states and then presents transitions between them via a Markov chain. Figure 1.4a shows the mapping from states to abstract states, and Figure 1.4b shows the Markov chain along with annotations of action chosen for all states within an abstract state. As shown with hatching, s_8 (and only s_8) maps to the abstract state b_4. The chain confirms that three actions are used

only when starting from b_4, also hatched. This process confirms that s_8 is the sole cause of the problematic behavior.

Env. State	Abs. State
s_1	b_1
s_2	b_1
s_3	b_1
s_4	b_2
s_5	b_5
s_6	b_2
s_7	b_3
s_8	b_4
s_9	b_5

(a) State mapping



(b) Markov chain over abstract states

Figure 1.4: An explanation that groups states into abstract states and then shows the Markov chain relating these abstract states. The abstract state corresponding to s_8 is hatched.

Experience Importance Now that the issue with the policy has been pinpointed, an observer may seek to determine why this aspect of the policy was learned. For this purpose, one needs an explanation in terms of **past experiences**, such as the importance of learning tuples with respect to a given action choice ("move right" in s_8). Figure 1.5 shows example importance values for this action choice, where a higher importance corresponds to a greater impact, so tuples with high importance values can be prioritized during investigation. The tuple with highest importance corresponds to instances where the "move left" action was taken in s_8, but the next state was still s_8. That is to say, on some occasions when the agent previously tried moving left, an unexpected obstacle prevented moving leftward, so the agent has learned to move right instead.

s	***a***	***r***	***s'***	**Imp**
s_8	←	-1	s_8	0.88
s_2	←	0	s_G	0.10
...				
s_4	↑	-1	s_G	0.01
s_3	←	-1	s_G	0.01

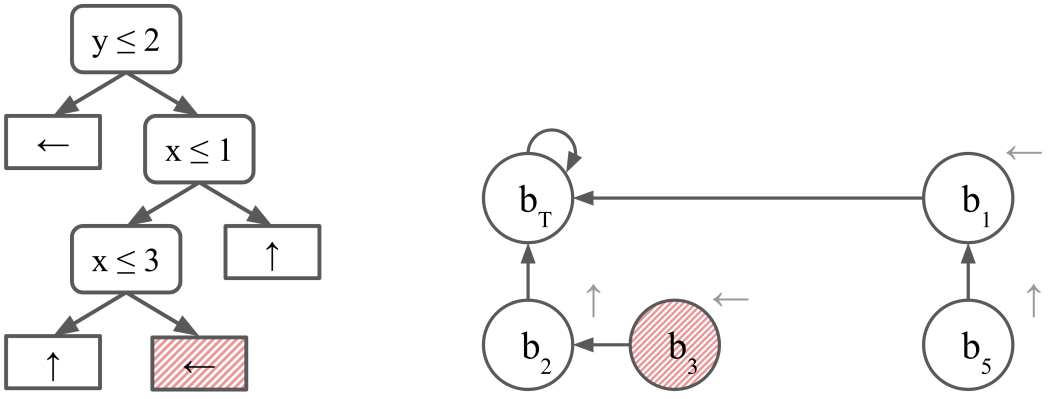
Figure 1.5: Importance values for past transition tuples. Identical transitions are grouped and given an importance value for the entire group. The top-ranked transition tuple group relates to moving left in s_8.

1.2.3 Benefit of Unified Explanation

The process outlined above identifies a potential cause of the unwanted behavior (an unexpected inability to move leftward in s_8 during training). As a next step, an observer may seek to remove this experience and change the behavior, but the current setup reveals little information about the effects of such a change. We propose to address this shortcoming through a unified explanation format that permits coordination between structures and nuanced explanations via past experiences.

Coordination between Structures In this example, the leaf marked in Figure 1.3 and the abstract state marked in Figure 1.4b conveniently correspond to each other. In general, additional states could fall into just the leaf or the abstract state, thus making the relationship between the two uncertain. Additionally, any changes to the tree or the Markov chain can be accompanied by arbitrary changes in the other structure. If the scope of possible changes were known, then changes in behavior could be predicted better. For example, in this case, changing the $x \leq 3$ node to a "move left" leaf only impacts the abstract states b_3 and b_4. The local nature of this impact can be identified and represented. Separate explanations of **single-action behavior** and **policy-level behavior** in terms of **state features** are unable to provide this information.

Nuanced Explanation via Past Experiences Simply removing the experience has an unknown effect on the entire policy. We seek to deliberately modify a single leaf in the tree (i.e., to obtain the tree shown in Figure 1.6a), so we specifically want importance with respect to the division at $x \leq 3$. Relatedly, the effects of the problematic transition should be limited to this subtree and not, e.g., directly affect the actions in s_1. This explanation of a **single-action behavior** in terms of **past experiences** produces a more nuanced explanation. Additionally, we wish to know if the desired result is obtained (i.e., only two actions are needed to reach the goal). For this, we effectively seek to obtain the Markov chain shown in Figure 1.6b. To determine the impact of removing the experience on the Markov chain, we need to know the experience's importance with respect to the chain. This is then an explanation of **policy-level behavior** in terms of **past experiences**.



(a) Decision Tree Policy (changed)

(b) Markov chain over abstract states (changed)

Figure 1.6: Updated explanations for the policy obtained by removing the unwanted experience.

This scenario demonstrates the problem addressed in this thesis. We advocate for a unified explanation format which maintains a link beween **single-action behavior** and **policy-level behavior** and provides an explanation in terms of both **state features** and **past experiences** for both types of behavior.

1.3 Approach

Our work unifies three explanation approaches:

1. Feature importance explanations for single actions in the form of a policy that is a decision tree,

2. Policy-level behavior explanations in the form of a Markov chain over abstract states and corresponding mapping of states to abstract states, and

3. Experience importance explanations that assign a numeric weight to each past experience based on impact on learned behavior.

To create a unified explanation, we first introduce methods for creating explanations that address a subset of our thesis question, then we present how to integrate these methods. Specifically, we merge our techniques for creating explanations of single-action and policy-level behavior in terms of state features, then we adapt our technique for creating explanations in terms of past experiences.

1.3.1 Decision Tree Policies

Our first approach, CUSTARD (Constrain Underlying Solution to a Tree; Apply RL to Domain), constructs a feature importance explanation in the form of a Decision Tree Policy (DTP). We introduce a novel Markov decision process (MDP) class, Iterative Bounding MDPs (IBMDPs), which encapsulates the problem of learning a DTP for a base MDP, the MDP for the original problem. An IBMDP requires an agent to take a sequence of observation actions before selecting an action from the base MDP. IBMDPs are designed so a sequence of observation actions is equivalent to traversing a decision tree, so we can extract a DTP from an IBMDP policy. Existing approaches for directly learning a DTP cannot use a neural network as their function approximator, which means these methods cannot leverage advances in deep RL. Approaches that do leverage neural networks approximate the learned policy with a DTP, so the learned policy cannot be fully explained. In contrast, the solution to an IBMDP is a DTP for the base MDP, regardless of the function approximator used in solving the IBMDP. This property allows IBMDPs to use neural networks during training and yield an equivalent DTP at any time. We present a way to convert existing deep RL methods to a form suitable for solving an IBMDP to find a DTP. This conversion is motivated by a two-agent approach that we show is equivalent to a modified single-agent system.

1.3.2 Abstract Policy Graphs

For policy-level behavior explanations, we introduce Abstract Policy Graphs (APGs). An APG is a Markov chain over abstract states with actions as edges. An APG allows the prediction of sequences of actions and qualities of future states. We introduce APG-Gen, a method for constructing an APG for an agent based only on a set of transition tuples and Q-value estimates for these transitions without relying on knowledge about the agent's function approximation scheme. APG-Gen forms abstract states using a process similar to divisive clustering: the set of transition tuples is repeatedly split on the feature with highest importance. This importance is computed with respect to the Q-value estimate for the cluster, so states which lead to similar future outcomes are grouped together. APG-Gen then constructs a Markov chain using empirical estimates of transition probabilities between abstract states. This process requires discrete actions and a binary featurization of all states.

1.3.3 Importance Scores for Past Experiences

To assign importance scores to past experiences, we build off of Representer Point Selection (RPS) [1], existing work on decomposing the weights of the final layer of a neural network into a weighted sum of the training points (as featurized by the rest of the network). We present Modified RPS (MRPS) that (i) provides an explanation in terms of clusters of similarly-treated points and (ii) accounts for variations in feature magnitudes. These modifications better account for near-duplicate points in the training set and feature rescaling, which yields more concise explanations than RPS to better reflect the actual impact of training points.

1.3.4 Unification

We merge CUSTARD and APG-Gen to create a method for producing a *Unified Explanation Tree* (UET). By using a tree to map from states to both an action and an abstract state, a UET provides both feature importance and policy-level behavior explanations. Using a UET rather than two separate explanations identifies the relationship between local behaviors and global behaviors. Specifically, we use the DTP partitions from CUSTARD as the initial clusters for APG-Gen, thereby creating specialized, shorter subtrees compared to a direct linking of trees.

When applied to a standard deep RL agent, MRPS identifies the transitions that influence overall action selection. We further adapt MRPS to provide these explanations at all levels of our UET. Internal UET nodes specify a feature and value for partitioning the state space, and they originate from the DTP or APG-Gen clustering process. For DTP nodes, we apply MRPS to the neural network used to solve the IBMDP that produced the DTP. For APG-Gen nodes, we use an RPS-style reparameterization of the feature importance values used for selecting division features. In both cases, we produce experience importance values with respect to partitioning features and values. These extensions allow a UET to compute experience importance at all of its levels, as well as identify the scope of potential changes should a set of experiences be removed.

1.4 Contributions

The key contributions of this thesis are as follows:

- CUSTARD, an approach for creating a Decision Tree Policy (DTP), an explanation of single-action behavior;
  - A novel class of MDPs, Iterative Bounding MDPs, which represent the problem of solving a base MDP using actions that correspond to a DTP for that base MDP;
  - A method for converting existing deep RL techniques to variants suitable for solving Iterative Bounding MDPs;
- APG-Gen, an algorithm for constructing an Abstract Policy Graph, a policy-level behavior explanation for closed-box RL agents;
  - A novel explanation form, the Abstract Policy Graph, which takes the form of a Markov chain over abstract states;
  - A set of algorithms for efficiently constructing an Abstract Policy Graph without assumptions about the RL agent's internal structure;
- MRPS, a method for explaining behavior in terms of past experiences;
  - An approach for identifying importance of sets of past experiences in addition to importance of single experiences;
  - A modification which accounts for varying feature magnitudes;
- A method for producing a single, unified explanation;
  - A new explanation format: Unified Explanation Trees that capture both an agent's action choice as well as abstract state membership;
  - An extension of CUSTARD and APG-Gen to produce an explanation of this format;
  - A modification of MRPS for identifying experience importance at all levels of the Unified Explanation Tree (i.e., importance with respect to decision tree partitions rather than only the final action);
- Novel environments for evaluating explainable RL methods and evaluation of our approaches using these domains;
  - PrereqWorld, a family of domains with configurable complexity and size, which have known prerequisite relationships between actions and therefore enable measuring how well an explanation method can identify these prerequisite relationships;
  - PotholeWorld, an environment where a limitation on policy complexity changes the best-performing policy, thus allowing evaluation of a method's ability to correctly perform this complexity/performance trade-off.

1.5 Thesis Outline

Figure 1.7 illustrates the relationship between chapters in this thesis, and the following outline summarizes each chapter. Note how we introduce methods for explanations of different types (e.g., single-action explanations in terms of state features in Chapter 3 vs. policy-level explanations in terms of state features in Chapter 4) and later join these techniques (e.g., explaining both single-action and policy-level behavior in terms of state features in Chapter 5).

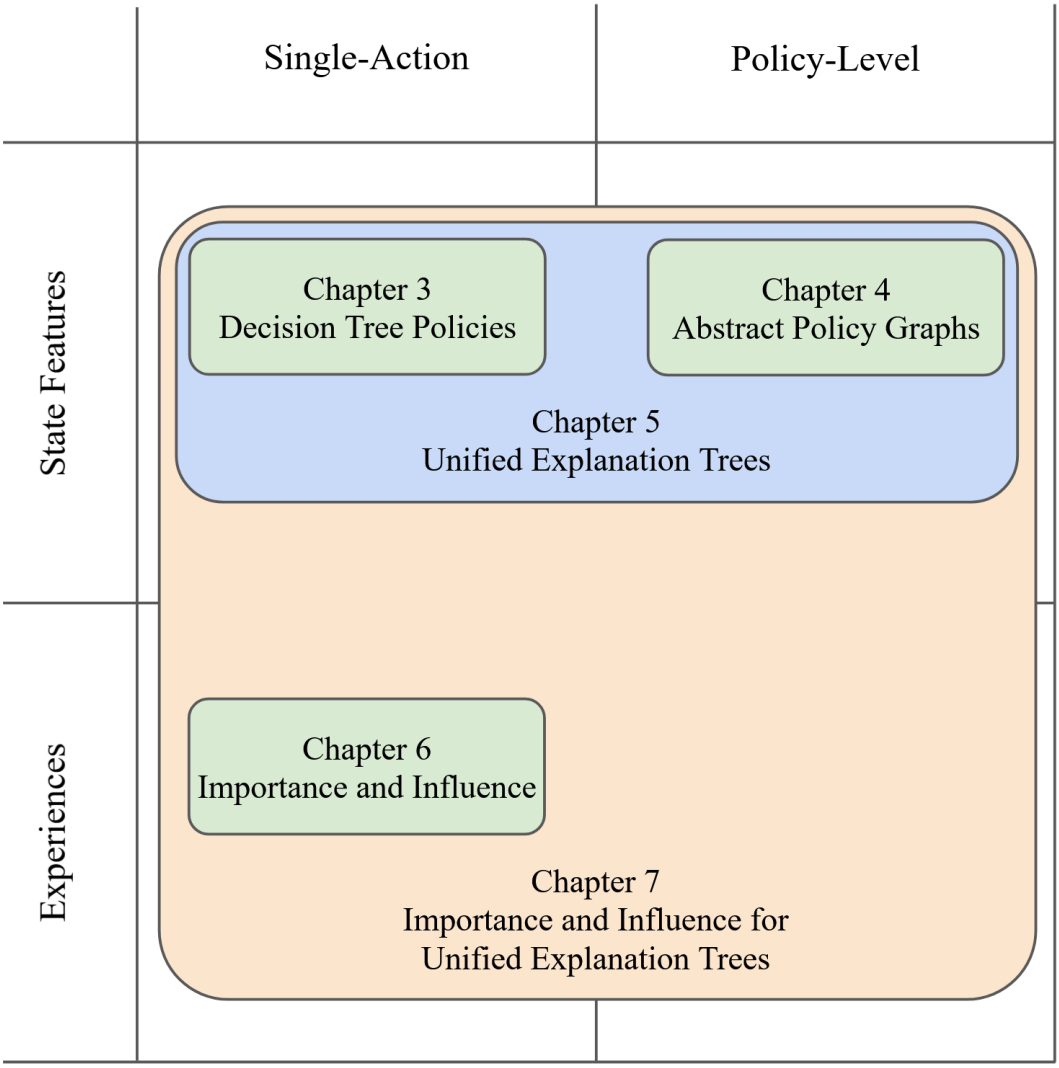


Figure 1.7: Organization of the chapters of this thesis along our two explanation type dimensions. Chapters that build upon other chapters are shown encapsulating those other chapters.

Chapter 2 – Preliminaries provides background helpful for reading this thesis as well as introducing the environments we use to evaluate our methods. Here, we clarify the notation and formulations we choose to use. We discuss potential issues that may arise when explaining an agent, and we use these potential issues to motivate the design of our environments.

Chapter 3 – Decision Tree Policies via DRL introduces Decision Tree Policies (DTPs): policies expressed in a form which is amenable to single action explanations in terms of state features. We introduce the Iterative Bounding MDP, which we show can be used to produce a DTP for a desired environment. Additionally, we introduce CUSTARD, a process to solve an Iterative Bounding MDP using a neural network function approximator. We show how this technique successfully learns concise DTPs without compromising agent performance. We also show how CUSTARD is able to make the correct complexity/performance trade-off, unlike existing methods that are not aware of complexity constraints.

Chapter 4 – Abstract Policy Graph Creation presents the Abstract Policy Graph, which is a Markov chain over abstract states. This structure summarizes an agent's behavior within a domain at a policy level in terms of state features; the contained information permits predicting an agent's future actions, determining whether the agent is competent, and measuring the complexity of its learned policy. We present APG-Gen, our algorithm for creating an Abstract Policy Graph for a trained agent. We show that APG-Gen's computational complexity is favorable, and we demonstrate its ability to produce accurate and concises explanations.

Chapter 5 – Unified Explanation Trees describes the Unified Explanation Tree, our explanation format which contains both single action and policy-level behavior information in terms of state features. We present a method to extract a set of decision trees over abstract states while using APG-Gen. We introduce how to join the trees produced via CUSTARD and from APG-Gen to yield a Unified Explanation Tree. We discuss how to produce a more concise tree by reducing repetitive structure, and we experimentally demonstrate the resulting reduction in complexity.

Chapter 6 – Importance and Influence in Neural Networks introduces our approach, Modified Representer Point Selection (MRPS), for assigning importance weights to an agent's past experiences. This method enables explaining single actions in terms of agent experiences. Our approach accounts for identical or similar points and can therefore assign importance weights to sets of experiences in a principled manner, unlike past approaches. We use a standard classification benchmark task to demonstrate the improved ability of our approach to detect experiences of interest.

Chapter 7 – Importance and Influence for Unified Explanation Trees describes how Modified Representer Point Selection can be further extended to allow more nuanced explanations in conjunction with a Unified Explanation Tree. While direct application of MRPS yields importance with respect to an action selection, we show how to obtain importance with respect to a decision node within the Unified Explanation Tree. This process consists of two sets of approaches, one for each of the two node types within a Unified Explanation Tree. We discuss how to leverage the properties of a Unified Explanation Tree to efficiently perform the required computations and avoid the approximations generally present within MRPS.

Chapter 8 – Related Work reviews previous work on explaining RL agents that is related to the methods presented in this thesis and compares to our contributions.

Chapter 9 – Conclusions and Future Work summarizes the contributions of this thesis and presents potential directions for future work.

Chapter 2

Preliminaries

We summarize key concepts used throughout this document in Section 2.1, along with clarifying our choice of formulations and notation. This is also where we note any assumptions that we leverage later. In Section 2.2, we describe the environments we introduce for evaluation of our work.

2.1 Background

2.1.1 Markov Decision Processes

In the context of RL, an agent acts in an environment defined by a Markov decision process (MDP). We use a six-tuple MDP formulation: $\langle S, A, P, R, \gamma, T \rangle$ [2], where:

- S is the set of states,
- A is the set of actions,
- P is the transition function ($S \times A \times S \rightarrow (0, 1)$),
- R is the reward function ($S \times A \times S \rightarrow \mathbb{R}$),
- γ is the discount factor, and
- T is the set of terminal states, which may be the empty set.

In this work, we assume all states consist of an assignment to each feature $f \in F$. Specifically, we use factored MDPs [3], in which each state consists of a set of feature value assignments $s = \{f_1, ..., f_{|F|}\}$. Note that we do not require a factored reward function. For certain presented methods (e.g., Chapter 4), we rely on the features consisting of binary values ($f \in \{0, 1\} \forall f \in F$). In cases where an MDP contains non-binary features, the MDP can be reparameterized in terms of binary features. For categorical features, this can be achieved by using a one-hot encoding. For continuous featurs, a discretization would be required.

2.1.2 Reinforcement Learning

2.1.2.1 Task

An agent is tasked with finding a policy, $\pi : S \rightarrow A$, which yields the highest expected discounted future return for all states. Note that, in general, the policy need not be deterministic, but we only consider the deterministic case in this work.

Likewise, note that we do not assume an optimal policy. In other words, the policy found by any given agent may not be the one that maximizes expected discounted future reward. We seek to explain the outcome of a learning process without assuming success on the part of the agent. Indeed, one of the applications of explanation methods is in identifying the cause of failure or learning suboptimal behavior.

2.1.2.2 Reinforcement Learning Constraints

We focus on the reinforcement learning (RL) setting. In this setting, the agent only has access to samples from P and R; the agent must interact with the world to learn a policy rather than directly querying the functions. There are RL methods that learn approximate models of P or R as part of the training process, but we focus on the case where only the policy and value function(s) are learned.

RL methods may operate in an offline setting (also called a batch setting) or an online setting. In an offline setting, the samples from P and R are fixed and cannot be influenced by the agent. In an online setting, the samples from P and R depend on the agent's choices, whether on the current best policy found by the agent or some other selected policy, such as a random exploration policy.

We do not limit ourselves to methods that operate in either the offline or the online setting, but there are two caveats. First, we perform our experiments with online methods and do prioritize techniques compatible with an online setting. Second, when computing experience importance values, we rely on the final policy being a function of a fixed set of experiences. This property naturally occurs in an offline setting. In an online setting, this can be achieved by collecting data during the typical online learning portion and then learning the final policy using a fixed set of collected data. This process can also be described as performing online learning and then fine-tuning on the final set of gathered data.

2.1.2.3 State- and Action-Value Functions

In the process of learning a policy, an RL agent generally approximates the state-value function or the action-value function [2]. The state-value function represents the expected return for a state s_0 when following policy π:

$$V_\pi(s_0) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t), s_{t+1})\right). \tag{2.1}$$

Analogously, the action-value function is the expected return when taking action a_t in s_t and following π afterward (when π is chosen to maximize expected discounted future reward). This

is known as the Q-function:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi}\Big(r_t + \gamma V^{\pi}(s_{t+1})\Big) = \mathbb{E}_{\pi}\left(r_t + \gamma \max_{a_{t+1}} Q^{\pi}(s_{t+1}, a_{t+1})\right). \tag{2.2}$$

Note that the state-value function can be obtained from the action-value function:

$$V_{\pi}(s_t) = Q_{\pi}(s_t, \pi(s_t)). \tag{2.3}$$

2.1.2.4 Learning Value Functions

Q-learning-based algorithms incrementally learn the Q-function and use it to infer the optimal policy π^*. The Q-function estimate is incrementally updated to be closer to a *target*, the bootstrapped estimate $r_t + \gamma \max_{a'} Q(s_{t+1}, a')$. In contrast, policy gradient methods directly model and optimize the policy. Actor-critic methods [4] additionally model the value function to leverage it in the policy update. They often use a critic for estimating the advantage function, $A(s, a) = Q(s, a) - V(s)$. Since methods generally use Q or V (and V can be obtained via Equation 2.3), the value-function is generally available alongside the policy of a trained agent. If an agent does not learn a value function while learning a policy, then a value function can be learned for an arbitrary policy using a set of (s, a, s') tuples.

2.1.2.5 Approximating Value Functions

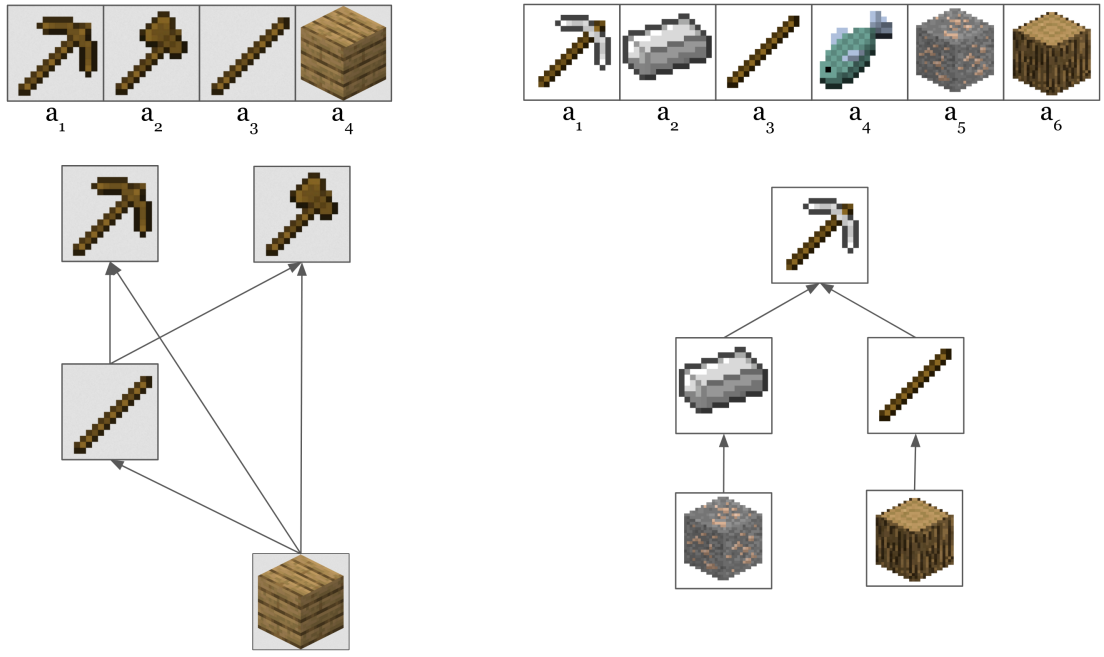
In practice, a Q-function or state-value function is approximated using a neural network [5]. Such a neural network generally consists of several layers (matrix multiplications on intermediate activation matrices) with nonlinear activation functions applied element-wise between layers. The final layer typically is not followed by a nonlinear activation function. With this network architecture, the network excluding the penultimate layer can be treated as learning a featurization of the input. The final layer can then be viewed as performing linear value function approximation atop this learned featurization. In Chapters 6 and 7, we take a similar perspective and rely on access to this final linear layer within a network.

2.2 Environments

Throughout this work, we evaluate on three different environments. Two of them are novel environments, designed to test aspects of explainable RL methods. We briefly describe the three environments and the motivation for their inclusion.

2.2.1 PrereqWorld

We introduce the PrereqWorld domain for evaluating our approach. This domain is an abstraction of a production task where the agent is to create a specific, multi-component item using a number of manufacturing steps. This environment is similar in structure to advising domains [6] and crafting, as in MineRL [7].



(a) PrereqWorld example with $m = 4$.

(b) PrereqWorld example with $m = 6$.

Figure 2.1: Example PrereqWorld environments showing item-action correspondence (top) and prerequisite hierarchy (bottom). (a) corresponds to the environment shown in Figure 2.2. (b) shows a different environment, to illustrate commonalities and differences between PrereqWorld instantiations.

2.2.1.1 Environment Description

The size of the state-space for an instance of this domain is controlled by the number of unique items, m. The agent may only have one of each item at a time. Production of each item may require some prerequisites, a subset of the other items, but no cycle of dependencies is permitted. In producing an item, the prerequisite items are usually consumed. A domain parameter, ρ, controls the probability that an item is consumed.

For ease of notation, we assume that the items are numbered according to their place in a topological sort (i.e., an item's prerequisites must be higher-numbered). Let i_d refer to the desired final item. For each item i_j, let C_j be the set of prerequisite items which i_j requires. A sample MDP is shown in Figures 2.1a and 2.2. Looking at Figure 2.2, note how the goal is to make i_1 and it requires having i_3 and i_4. In turn, i_3 also requires i_4. These dependencies induce the prerequisite hierarchy shown at the bottom of Figure 2.1a.

A state consists of m binary features, where the binary feature f_j corresponds to whether the agent has an item i_j. Any state where $s[f_d] = 1$ is a terminal state. The distribution of initial states is uniform over all possible non-terminal states. The reward is -1 for transitioning to a non-terminal state and 0 for transitioning to a terminal state. For simplicity, we take γ to be 1, but the optimal policies for any domain instance remain optimal for any γ in the interval $(0, 1]$.

$m = 4$, $\rho = 0$, $i_d = i_1$
$C_1 = \{i_3, i_4\}$
$C_2 = \{i_3, i_4\}$
$C_3 = \{i_4\}$
$C_4 = \{\}$
$S = \{0000, 0001, \ldots, 1111\}$
$A = \{a_1, \ldots, a_4\}$
$T = \{1000, 1001, \ldots, 1111\}$
$R(s, a, s') = 0$ for $s' \in T$
$R(s, a, s') = -1$ for $s' \notin T$
$\gamma = 1$

$P(0001|0000, a_4) = 1$
$P(0010|0001, a_3) = 1$
$P(0011|0010, a_4) = 1$
$P(1000|0011, a_1) = 1$
$P(0100|0011, a_2) = 1$
$P(0101|0100, a_4) = 1$
$P(0110|0101, a_3) = 1$
$P(0111|0110, a_4) = 1$
$P(1100|0111, a_1) = 1$
for other s and a,
$P(s'|s, a) = 0$ when $s \neq s'$
and
$P(s'|s, a) = 1$ when $s = s'$

Figure 2.2: The MDP for an example PrereqWorld instance.

There are m actions where the action a_j corresponds to attempting to produce item i_j. Actions for currently possessed items or for items with unmet prerequisites have no effect. That is, $P(s|s, a_j) = 1$ when feature $s[f_j] = 1$ or there is an $i_k \in C_j$ such that $s[f_k] = 0$. When an action is successful, f_j is set to 1 and each of item i_j's prerequisites is used with probability $1 - \rho$. That is, for all $i_k \in C_j$, f_k is independently set to 0 with probability $(1 - \rho)$ and left as 1 with probability ρ.

For the MDP in Figure 2.2, note that transitions are deterministic ($\rho = 0$) for simplicity and we do not show the transition function for states where $s[f_1] = 1$ (i_1 is present) since all such states are terminal. Notice how the domain can be solved optimally from the starting position 0000 (no items present) using the action sequence $[a_4, a_3, a_4, a_1]$. This ensures that an i_4 is present before i_3 is made, and another i_4 is created as a prerequisite to creating i_1. The correspondence between actions and items is seen at the top of Figure 2.1a. The bottom of Figure 2.1a then shows the relationship between items and, by extension, prerequisites for actions. We show another example environment in Figure 2.1b. In this case, m is 6, and the dependencies induce a different prerequisite hierarchy.

2.2.1.2 Motivation for Use

When applying a method for producing explanations, a user may seek to identify whether an agent is performing actions in the proper order. Similarly, a user may seek to know whether an agent is attending to ("looking at") only the relevant portions of the state. The PrereqWorld domain has inherent dependencies between actions which can be computed while creating an instantiation of the domain.

Since the "ground truth" for these different evaluations can be directly computed from the underlying mechanics of an environment instantiation, we can perform objective evaluation with respect to an upper-bound performance (i.e., compared to a method which can access all C, which is not available in practice).

A method for producing explanations can then be evaluated based on its ability to detect these dependencies. Specifically, explanations can be compared based on their ability to detect temporal patterns. For example, action 4 precedes action 1 when acting optimally within the MDP given in Figure 2.1a. Likewise, explanations can be compared based on ability to detect relevant and irrelevant features. For example, when acting optimally within the MDP given in Figure 2.1b, feature 4 is never relevant, but feature 6 is relevant only when feature 3 is 0. Furthermore, PrereqWorld naturally has sets of states that are treated identically, so any methods that group states can be evaluated based on their ability to correctly group these states that are equivalent under the given policy.

2.2.1.3 Inspiration

PrereqWorld is effectively a subset of the MineRL environment [7]. MineRL is an RL environment built around the game Minecraft. Within the environment, an embodied agent seeks to obtain a penultimate item. As in PrereqWorld, this item has prerequisite items, which themselves may have prerequisites.

Unlike PrereqWorld, obtaining an item in MineRL requires moving in a persistent, three-dimensional environment while observing the world in a limited way (i.e., only from a first-person point of view). The methods which perform best in MineRL in practice take a hierarchical approach where the high-level actions correspond to obtaining individual items and a sub-policy is used to perform all movement to obtain these items [8, 9, 10]. The high-level problem with the high-level actions is effectively the PrereqWorld environment.

The MineRL environment has been shown to be a compelling problem with analogues for many real-world difficulties such as a first-person perspective, a sparse reward function, and deep dependencies between subgoals [11, 12, 13, 14]. Since PrereqWorld captures a meaningful component of the MineRL environment (i.e., the top level of the policy hierarchy), evaluating explanations using PrereqWorld provides some insight into how explanations would perform when applied to problems relevant to broader RL research.

2.2.2 PotholeWorld

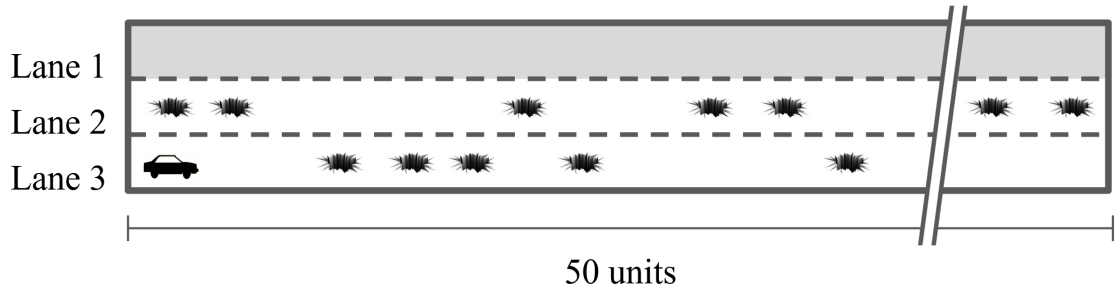


Figure 2.3: Example PotholeWorld environment showing pothole locations in lanes 2 and 3. Note how lane 1 is free of potholes but offers lower reward per unit travelled.

We introduce the PotholeWorld domain, shown in Figure 2.3. In this domain, the agent is tasked with traversing 50 units along a three-lane road. The first lane gives less reward per unit traveled, but the other two lanes contain "potholes" which lead to a reward penalty if traversed.

2.2.2.1 Environment Description

A state contains a single feature: the current position, in $[0, 50]$. The initial state is at position 0; an episode terminates when the position is 50. The three actions are $\{lane\_1, lane\_2, lane\_3\}$, which each advance the agent in the corresponding lane by a random amount drawn from $Unif(0.5, 1)$. Potholes are added starting from position 0 until position 50 with distances drawn from $Unif(1, 2)$. Potholes are assigned with equal probability to lane 2 or lane 3. When an action is taken, the base reward is equal to the distance moved. This is reduced by 10% if the $lane\_1$ action was taken. If a pothole is in the chosen lane along the traversed stretch (i.e., between previous agent position and next agent position), then the reward is reduced by 5. Though we randomly generate pothole locations, we use the same locations across all methods within any experiment trial.

A key aspect of this environment is that the pothole locations are not directly revealed to the agent as part of a state. To avoid the potholes, the agent must use its current position to infer whether the next stretch of each lane is devoid of potholes. Because of the relationship between movement distance and pothole spacing, at least one of lane 2 or lane 3 is always devoid of potholes for the maximum possible movement caused by taking an action. Thus, to attain maximum reward, an agent would initially encounter potholes, then learn their locations (relative to agent position), and finally avoid these potholes by weaving between lanes 2 and 3 in future episodes.

In the Figure 2.3 example, note how the optimal policy is to weave between lanes 2 and 3 by initially picking lane 3, then switching to lane 2 for a time and so on. Though the agent could always pick lane 1 to avoid the reward reduction caused by potholes, the agent receives a lower reward for each unit traversed along lane 1 than, e.g., the same distance within the first stretch of lane 3.

2.2.2.2 Motivation for Use

This domain is designed such that limiting a policy's complexity changes the best-performing behavior. When not limiting the policy, as noted above, the optimal behavior is to weave between lanes 2 and 3. When severely limiting the policy, such as forcing the agent to pick a single action (i.e., single lane) for the entire episode, then the best policy is to pick lane 1.

For policy complexities between these two extremes, the optimal behavior changes further. For example, if limiting how many times an agent can change actions from its choice in the previous state, then the agent is incentivized to change actions where many potholes appear in sequence within a specific lane.

This property enables evaluating a policy learning method's ability to successfully trade off complexity and performance. This is relevant where an interpretable policy is desired, so policy complexity must be limited. If a policy learning method does not account for this complexity

limitation, it is liable to poorly approximate the behavior that is optimal in the non-limited case and thus attain lower performance than merely always choosing lane 1. In contrast, a successful method would gracefully deteriorate in terms of total attained reward but never perform worse than the simple policy of always choosing lane 1.

2.2.3 CartPole

CartPole [15] is a commonly used domain for evaluating methods that produce DTPs, where the agent must balance a pole affixed to a cart by moving the cart back and forth.

2.2.3.1 Environment Description

We use the variant where the agent can select between applying a fixed amount of force to the left or to the right of the cart. As part of a state, the agent observes the cart's position, the cart's velocity, the pole's angle, and the pole's angular velocity. We use the variant where an episode terminates when the pole falls or 200 timesteps have elapsed. The agent is rewarded for each timestep, so an optimal policy is one where the pole does not fall for 200 timesteps for almost every starting position. Note that a successful policy is traditionally one that attains an average reward of at least 195.

2.2.3.2 Motivation for Use

We include this environment to provide standard benchmark results. This is a common environment for evaluation since (1) this environment has a long history of use in testing RL algorithms and (2) there exist policies that attain a high reward in this environment while consisting of a shallow decision tree. Following previous work, when using this environment, we limit methods to decision trees of depth two.

Chapter 3

Decision Tree Policies via DRL

In this chapter, we seek to explain individual actions in terms of the state features. For this purpose, we construct a Decision Tree Policy (DTP, a policy in the form of a decision tree) so that the tree can be subsequently analyzed, e.g., to identify which features are used in making a decision. Unlike existing work which converts a policy to a DTP or greedily constructs one using a tree-building method, we pose the tree-building problem as a reinforcement learning problem. This approach avoids the error caused by approximation and also enables use of deep learning function approximation rather than direct tree-building.

3.1 Motivation

The incorporation of deep neural networks into reinforcement learning (RL) has broadened the set of problems solvable with RL. Though these techniques yield high-performing agents, the policies are encoded using thousands to millions of parameters, and the parameters interact in complex, non-linear ways. As a result, directly inspecting and verifying the resulting policies is difficult. Without a mechanism for a human operator to readily inspect the resulting policy, we cannot deploy deep RL (DRL) in environments with strict regulatory or safety constraints.

3.1.1 Decision Trees and Decision Tree Policies

Decision trees (DTs) [16] are an interpretable model family commonly used to represent policies. DTs recursively split the input space along a specific feature based on a cutoff value, yielding axis-parallel partitions. *Leaf nodes* are the final partitions; *internal nodes* are the intermediate partitions. Some benefits of DTs include that they allow formal verification of policy behavior [17], counterfactual analysis [18], and identification of relevant features. DT-like models have been used to represent different components of the MDP or RL learning process. We focus on DT policies (DTPs), which map each state to a leaf node representing an action.

Sufficiently small DTPs are *interpretable* [19], in that people understand the mechanisms by which they work. DTs conditionally exhibit simulatability, decomposability, and algorithmic transparency [20]. When a person can contemplate an entire model at once, it is *simulatable*;

sufficiently small DTs exhibit this property. A *decomposable* model is one in which sub-parts can be intuitively explained; a DT with interpretable inputs exhibits this property. *Algorithmic transparency* requires an understanding of the algorithm itself: in particular, DTs are verifiable [17], which is important in safety-critical applications. Because of the additional constraints placed on a DTP, a DTP may perform worse than an unconstrained policy. Changing the featurization of an environment's state space can affect the relative performance of a DTP and unconstrained policy, but we consider a setting where the environment cannot be modified. We focus on environments where the given state featurization and action set is conducive to high-performing DTPs.

A DTP provides a way for users to answer questions such as:

- "What will the agent do in state s?"; and
- "Which features informed this choice?"

3.1.2 Downsides of Past Approaches

However, DRL techniques are not directly compatible with policies expressed as DTs. UTree [21] and its extensions [22, 23, 24] incrementally build a DT while training an RL agent. Transition tuples (or tuple statistics) are stored within leaf nodes, and a leaf node is split when the tuples suggest that two leaf nodes would better represent the Q-function. A concurrent work [25] uses a differential decision tree to represent the policy and approximates the soft-max tree with a DT after training. However, these methods require specific policy or Q-function representations, so they cannot leverage powerful function approximators like neural networks.

We discovered the need for an alternate approach while working on a UTree-style method, CQI [24]. CQI was designed to produce smaller trees than other UTree variants by changing the leaf-partitioning condition. When using a statistical test to split leaves, as is done in UTree, small differences in outcomes can lead to leaf partitions even when the chosen action does not change. In contrast, CQI tracks candidate partitions and only performs a partition when the policy's performance is expected to increase. Furthermore, CQI prioritizes the partitions that increase the policy's performance the most. However, we encountered scaling issues with CQI, since all methods in this family are fundamentally tabular methods across leaves. The only generalization that occurs is within a leaf; if a leaf is partitioned, no further generalization can happen across the new children. This experience motivates the desire for a method compatible with reinforcement learning algorithms that use a neural network as a value function approximator.

An alternative approach is to convert a DRL policy to a DT: a non-DT policy is learned and then is approximated using a DT. One such method is VIPER [17], which uses model compression techniques [26, 27, 28] to distill a policy into a DT. This work adapts DAGGER [29] to prioritize gathering critical states, which are then used to learn a DT. However, VIPER approximates an expert. When the expert is poorly approximated by a DT, the DTP that is found performs poorly. The resulting policy can be arbitrarily worse than the original one, and the DT can be large due to unnecessary intricacies in the original policy.

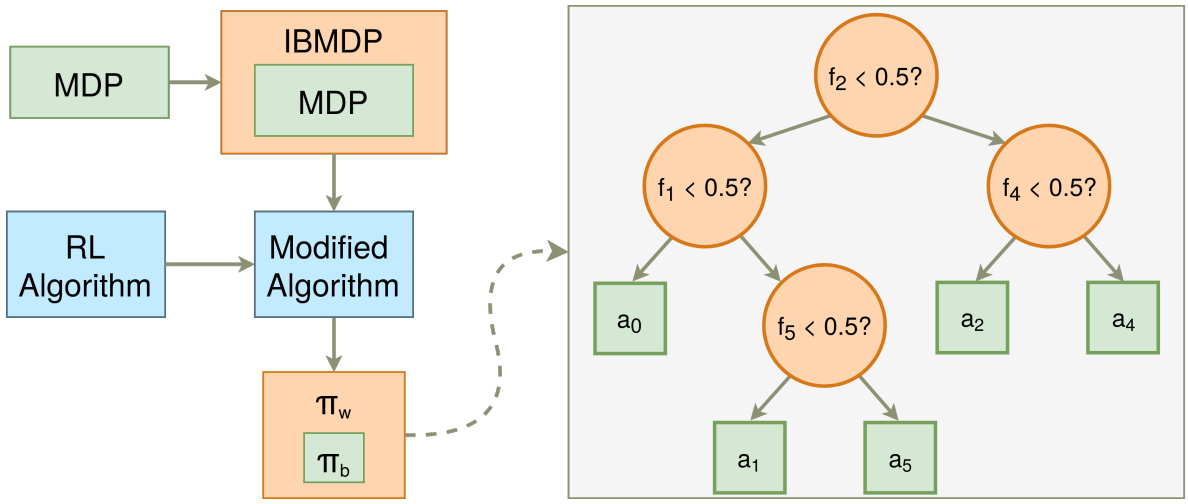


Figure 3.1: Method overview: we wrap a base MDP to form an IBMDP and solve using a modified RL algorithm. The solution is a decision tree policy for the base environment.

3.1.3 Benefits of a Meta-Problem Approach

To address the limitations of these techniques, we propose to solve a meta-problem using RL such that the solution corresponds to a DT-format policy for the original problem. We introduce CUSTARD (Constrain Underlying Solution to a Tree; Apply RL to Domain) [30], a process that uses RL to solve the meta-problem while ensuring that the embedded solution is equivalent to a DT-format policy throughout training (overview in Figure 3.1). We propose a novel Markov Decision Process (MDP) formulation for the meta-problem: Iterative Bounding MDPs. We present a general procedure to make RL techniques compatible with CUSTARD. CUSTARD allows modern DRL techniques to be applied to the meta-problem, since the learned policy weights can be fully replaced by an equivalent DT. Thus, CUSTARD maintains the interpretability advantage of a DT policy while using a non-interpretable function approximator during training. Additionally, CUSTARD ensures that the DT is an exact representation of the learned behavior, not an approximation. By directly finding a DTP, CUSTARD avoids the case encountered by VIPER where a high-performing expert is approximated, and this resulting DTP performs poorly.

In this chapter:

1. we introduce a novel MDP representation (IBMDPs) for learning a DT policy for a base MDP;

2. beginning with a two-agent UTree-like algorithm, we present an equivalent single-agent formulation that solves IBMDPs to produce DTs;

3. we show how to modify existing RL algorithms (policy gradient and Q-learning) to produce valid DTs for the base MDP; and

4. we empirically evaluate the performance of our approach and identify cases where it outperforms post-hoc DT fitting.

3.2 Approach: CUSTARD

We present CUSTARD, an approach for training an agent to produce a Decision Tree Policy using existing RL algorithms. We achieve this goal by training the agent to solve a wrapped version of the original, *base* MDP. The wrapped MDP, which we name an *Iterative Bounding MDP*, extends the base MDP by adding information-gathering actions and *bounding* state features to indicate the gathered information. The bounding features correspond to a position within a DT traversal, and the information-gathering actions correspond to partitions performed by internal nodes within a DT. By constraining an agent's policy to be a function of the bounding state features, the learned policy is equivalent to a DT.

In Section 3.2.1, we describe IBMDPs. In Section 3.2.2, we describe the process for extracting a DTP from an IBMDP policy during any point in training. In Section 3.2.3, we present methods for adapting existing RL algorithms to learn an implicit DTP for the base MDP. In particular, we describe modifications to Q-learning and actor-critic algorithms.

3.2.1 Iterative Bounding MDPs

We introduce Iterative Bounding MDPs (IBMDPs), a novel MDP formulation for producing DTPs. We seek to produce a DTP by ensuring that an agent's IBMDP policy is equivalent to a DTP for the original, base MDP. To use a DTP to select an action in the base MDP, a series of internal nodes are traversed, and then the leaf node specifies the action. To allow this behavior, an IBMDP has actions that are equivalent to traversing nodes and state features that indicate the current node.

The base MDP must be an MDP with a factored state representation, where each state feature has upper and lower bounds on its values. A *base state* is a state from the base MDP's state space, and a *wrapped state* is a state from the IBMDP's state space; other terms are defined analogously.

3.2.1.1 State Space

A wrapped state s_w consists of two parts: a base state s_b and bounding features, f_1^l-f_n^l and f_1^h-f_n^h. There exist two bounding features per base state feature, such that f_i^l represents a lower bound on the base feature f_i's current value, and f_i^h represents an upper bound for that same base feature's current value. The bounding features reflect the outcomes of binary comparisons performed during the traversal, and the bounds are tightened with more comparisons. A sequence of wrapped states represents a traversal through a DTP for a specific s_b. For simplicity, and without loss of generality, we consider s_b to be normalized such that all features are in $[0, 1]$. We use $s_w[c]$ to refer to a component c within s_w. An IBMDP state and state space are:

$$S_w = S_b \times [0, 1]^{2n}, \;\; s_w = \langle s_b, f_1^l, \ldots, f_n^l, f_1^h, \ldots, f_n^h \rangle.$$

3.2.1.2 Action Space

The action space for an IBMDP A_w consists of the base actions A_b and an additional set of information-gathering actions A_I:

$$A_w = A_b \cup A_I.$$

Base actions correspond to taking an action within the base MDP, as when reaching a leaf in a DTP. Information-gathering actions specify a base state feature and a value, which correspond to the feature and value specified by an internal node of a DTP. We present two different action space formats: a discrete set of actions and a Parameterized Action Space [31]. In both cases, the action can be described by a tuple, $\langle c, v \rangle$, where c is the chosen feature and v is the value. For simplicity, we consider $v \in [0, 1]$, where 0 and 1 respectively correspond to the current lower and upper bound on c.

With a discrete set of IBMDP actions, each of the n features can be compared to one of p possible values. This results in $p \times n$ discrete actions, with v values of $1/(p+1), \ldots, p/(p+1)$ for each of the n possible f. With this construction, the base actions must be discrete. In this case, the information-gathering actions are:

$$A_I = \{c_1, \ldots, c_n\} \times \left\{ \frac{1}{p+1}, \ldots, \frac{p}{p+1} \right\}.$$

In a Parameterized Action Space MDP (PASMDP), each action $a \in A_d$ has m_a continuous parameters. A specific action choice is specified by selecting $(a, p_1^a, \ldots, p_{m_a}^a)$. If the IBMDP is a PASMDP, then there is an action for each of the n features with a single parameter ($m_a = 1$), where the action specifies c and the parameter specifies v. With this formulation, the base MDP may have a continuous, multi-dimensional action space. This is supported by adding a single a with parameters corresponding to the base action choices. If A_b has discrete actions, then an a is added for each of them, with the corresponding m_a set to zero. The information-gathering actions in the PASMDP variant are:

$$A_I = \{c_1, \ldots, c_n\} \times [0, 1].$$

3.2.1.3 Transition Function

When an agent takes an information-gathering action, $\langle c, v \rangle$, the selected value v is compared to the indicated feature c. Since v is constrained to $[0, 1]$ but represents values in $[c^l, c^h]$, the un-normalized v_p is obtained by projecting $v_p \leftarrow v \times (c^h - c^l) + c^l$. The bounding features c^l and c^h are updated to reflect the new upper and lower bounds for c; the base features are unchanged. This process is equivalent to the behavior of an internal node in a DTP: a feature is compared to a value, and the two child nodes represent different value ranges for that feature. Thus, for an information-gathering action $\langle c, v \rangle$, the transition function of the IBMDP, T_w, is

deterministic, and the next state, s'_w, is based on s_w:

$$
\begin{aligned}
&s'_w[s_b] = s_w[s_b], \\
&s'_w[f] = s_w[f] \forall f \notin \{c^l, c^h\}, \\
&\text{If } s_b[c] \leq v_p\text{: } s'_w[c^h] = \min(s_w[c^h], v_p), s'_w[c^l] = s_w[c^l], \\
&\text{If } s_b[c] > v_p\text{: } s'_w[c^l] = \max(s_w[c^l], v_p), s'_w[c^h] = s_w[c^h].
\end{aligned}
$$

When a base action is taken, the base features are updated as though this action was taken in the base MDP, and the bounding features are reset to their extreme values. This is equivalent to selecting a base action in a DTP and beginning to traverse the DTP for the next base state (starting from the root node). This corresponds to a transition function of:

$$
\begin{aligned}
&a \in A_b \wedge ((s'_w[f^l_i] = 0) \wedge (s'_w[f^h_i] = 1) \forall i \in \{1, \ldots, n\}) \\
&\implies T_w(s_w, a, s'_w) = T_b(s_w[s_b], a, s'_w[s_b]).
\end{aligned}
$$

3.2.1.4 Reward Function

The reward for a base action is the reward specified by the base MDP for the base action, base original state, and base new state. The reward for information-gathering actions is a fixed, small penalty ζ. For a sufficiently low value of ζ, the optimal solution for the IBMDP includes the optimal solution of the base MDP. The overall IBMDP reward function is:

$$
\begin{aligned}
a \in A_b &\implies R(s_w, a, s'_w) = R(s_w[s_b], a, s'_w[s'_b]), \\
a \notin A_b &\implies R(s_w, \langle c, v \rangle, s'_w) = \zeta.
\end{aligned}
$$

3.2.1.5 Gamma

We introduce a second discount factor, γ_w. When a base action is taken in the IBMDP, the gamma from the base MDP, γ_b, is used to compute the expected discounted future reward. Otherwise, γ_w is used. For a γ_w sufficiently close to 1, the expected discounted future reward is identical for an s_w, if acted upon in the IBMDP, and its corresponding s_b, if acted upon in the base MDP.

3.2.1.6 Remaining Components

We present the additional components required for an episodic MDP, but the framework is also applicable to non-episodic environments. A transition in the IBMDP, (s_w, a_w, s'_w), is terminal if $a \in A_b$ and $(s_w[s_b], a, s'_w[s_b])$ is a terminal transition in the base MDP. The distribution over starting states of the IBMDP is derived from the distribution of starting states in the base MDP. The probability of starting in state s_w is 0 if some $f^l_i \neq 0$ or $f^h_i \neq 1$; otherwise, it is equal to the probability of starting in $s_w[s_b]$ in the base MDP.

[todo: provide concrete example for a specific domain?]

Algorithm 1 Extract a Decision Tree Policy from an IBMDP policy π, beginning traversal from obs.

1: **procedure** SUBTREE_FROM_POLICY(obs, π)
2: $a \leftarrow \pi(obs)$
3: **if** $a \in A_b$ **then** ▷ Leaf if base action
4: **return** Leaf_Node(action: a)
5: **else**
6: $c, v \leftarrow a$ ▷ Splitting action is feature and value
7: $v_p \leftarrow v \times (obs[c^h] - obs[c^l]) + obs[c^l]$
8: $obs_L \leftarrow obs$; $obs_R \leftarrow obs$
9: $obs_L[c^h] \leftarrow v_p$; $obs_R[c^l] \leftarrow v_p$
10: $child_L \leftarrow$ Subtree_From_Policy(obs_L, π)
11: $child_R \leftarrow$ Subtree_From_Policy(obs_R, π)
12: **return** Internal_Node(feature: c, value: v_p, children: $(child_L, child_R)$)

3.2.2 Tree Extraction

Not all policies for the IBMDP correspond to valid DTPs; the presence of s_b within each wrapped state allows access to full state information at any point during tree traversal. However, all IBMDP policies that only consider the bounding features (i.e., ignore s_b) correspond to a DTP. We describe the process for extracting a DTP from a policy defined over bounding observations from the environment, $\pi(s_w \setminus s_b)$. We present the training of such policies in Section 3.2.3.

Algorithm 1 outlines the full DTP extraction procedure. SUBTREE_FROM_POLICY constructs a single node based on the current observation; that node's children are constructed through recursive calls to this same function. As described in Section 3.2.1, the bounding features ($s_w \setminus s_b$) describe a node within a DTP, with $s_w[f_i^l] = 0 \wedge s_w[f_i^h] = 1 \forall i \in [1, \ldots, n]$ corresponding to the root node. SUBTREE_FROM_POLICY$(s_w \setminus s_b, \pi)$ for a root node s_w yields the DTP for π.

An action a within the IBMDP corresponds to a leaf node action (when $a \in A_b$) or a DT split (when $a \notin A_b$). Lines 2-3 obtain the action for the current node and identify its type. The action taken for a leaf node defines that leaf, so Line 4 constructs a leaf if a is not an information gathering action. Information gathering actions consist of a feature choice c and a splitting value v (Line 6). The IBMDP constrains v to be in $[0, 1]$, which corresponds to decision node splitting values between $s_w[c^l]$ and $s_w[c^h]$, the current known upper and lower bounds for feature c. Line 7 projects v onto this range, yielding v_p, to which feature c can be directly compared.

To create the full tree, both child nodes must be explored, so the procedure considers both possibilities ($s_b[c] \leq v_p$ and $s_b[c] > v_p$). Lines 8-9 construct both possible outcomes: a tighter upper bound, $c^h \leftarrow v_p$, and a tighter lower bound, $c^l \leftarrow v_p$. This procedure then recursively creates the child nodes (Lines 10-11). The final result (Line 12) is an internal DTP node: an

incoming observation's feature is compared to a value v_p ($obs[c] \leq v_p$), and traversal continues to one of the children, depending on the outcome of the comparison.

3.2.3 Training Procedure

If an agent solves an IBMDP without further constraints, then it can learn a policy where actions depend on s_b in arbitrarily complicated ways. To ensure that the base MDP policy follows a DT structure, the IBMDP policy must be a function of only the bounding features. Effectively, if the policy is a function of $s_w \setminus s_b$, then the policy is a DTP for the base MDP. However, with a policy of the form $\pi(s_w \setminus s_b)$, the standard bootstrap estimate does not reflect expected future reward because the next observation is always the zero-information root node state. Therefore, standard RL algorithms must be modified to produce DTPs within an IBMDP.

We present a set of modifications that can be applied to standard RL algorithms so the one-step bootstrap reflects a correct future reward estimate. We motivate this set of modifications by presenting a "two agent" division of the problem and then show the equivalent single-agent Q target. We then demonstrate how a target Q-function or critic can be provided with the full state (s_w) to facilitate learning while maintaining a DT-style policy. Finally, we present how the modifications are applied to Q-learning and actor-critic algorithms. Without loss of generality, we focus on learning a Q-function. If learning an advantage function or value function, an analogous target modification can be made.

3.2.3.1 Two Agent Division

Learning in an IBMDP can be cast as a two-agent problem: (i) a *tree agent* selects which information-gathering actions to take and when to take a base action, and (ii) a *leaf agent* selects a base action using the bounding features, when prompted to do so. Figure 3.2 shows this division, where the leaf agent selects actions in s_{l_1} and s_{l_2}, and the tree agent selects all other actions.

With this division of the problem, the leaf agent is equivalent to the agent in UTree-style methods. The tree agent replaces the incremental tree construction used in UTree and is akin to an RL agent constructing a DT for a supervised problem [32]. The leaf agent's observed transition sequence consists of leaf nodes and its own selected actions: $s_{l_1}, a_{l_1}, r_{l_1}, s_{l_2}, a_{l_2}$. The bootstrapped Q-value estimate is:

$$r_{l_1} + \gamma_b \max_{a' \in A_b} Q_l(s_{l_2}, a'),$$

where r_{l_1} is a reward obtained from the base MDP.

In this framing, the tree agent experiences a new episode when a base action is taken. The initial state is always the zero-information, root state, and the episode terminates when the agent chooses the stop splitting action, a_{stop}, which we add for the two-agent formulation. When the tree agent stops splitting, the reward is the value estimated by the leaf agent, $Q_l(s_l, a_l)$. The tree agent's Q-value target is:

$$r_d + \gamma_w \max_{a' \in a_{stop} \cup A_w \setminus A_b} Q_d(s'_d, a'),$$

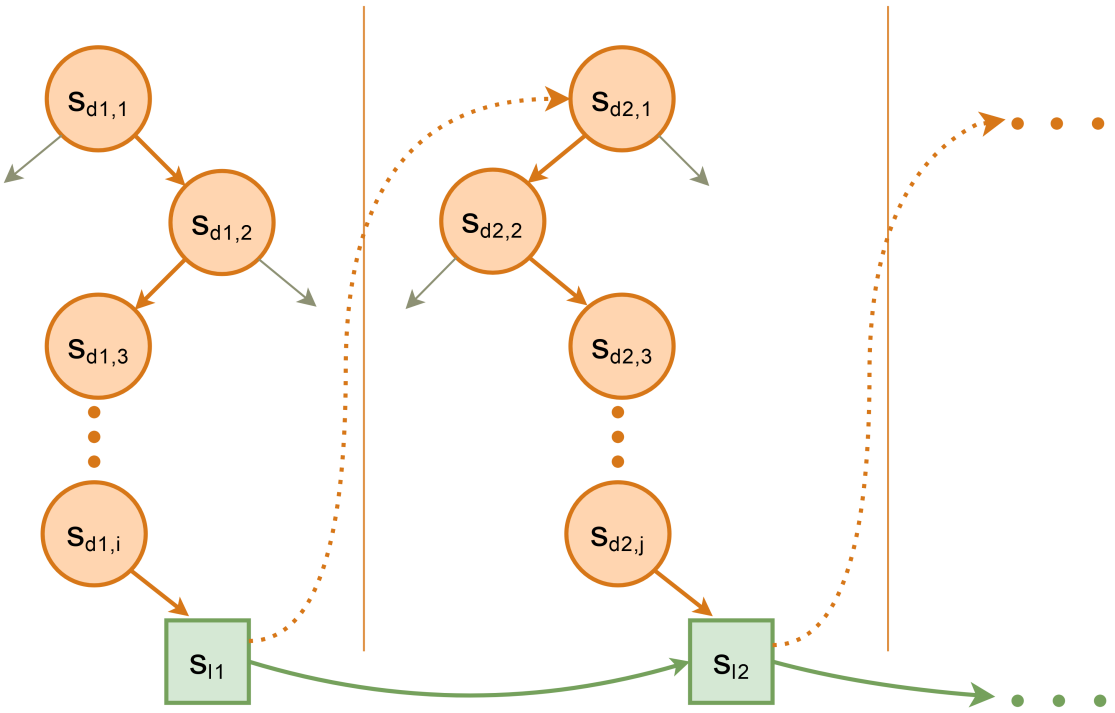


Figure 3.2: The division between the tree agent (circle states and arrow actions) and the leaf agent (square states and arrow actions). Each tree traversal is an episode for the tree agent and one transition for the leaf agent.

where r_d is $\max_{a' \in A_b} Q_l(s_d, a')$ if the a_{stop} action was chosen and ζ otherwise. When a_{stop} is taken, $Q_d(s'_d, a')$ is 0 for all a' since the transition is terminal for the tree agent.

These two equations for target Q-values allow an IBMDP to be solved using only the partial $s_w \setminus s_b$ observations. The tree agent does not directly receive a reward signal from future base actions but uses the leaf agent's estimates to update. The leaf agent learns long-term reward estimates based on rewards from the environment.

3.2.3.2 Merging of Agents

The target Q-value for a terminal tree agent action is r_d, which is $\max_{a \in A_b} Q_l(s, a)$. The tree agent's episode terminates if and only if a_{stop} is taken. Effectively, the tree agent seeks to learn $Q_d(s, a_{stop}) = \max_{a \in A_b} Q_l(s, a)$. Rather than learning this relationship, $Q_d(s, a_{stop})$ can directly query Q_l, simplifying the learning task without changing the underlying problem.

With this change to $Q_d(s, a_{stop})$, Q_d and Q_l are defined over disjoint subsets of A_w. A single, unified Q-function Q can be learned, which is defined over all a in A_w. This allows the target Q-values to be re-written as:

$$a \in A_b \implies \text{target} = r_{l_1} + \gamma_b \max_{a' \in A_b} Q(s_{l_2}, a'),$$
$$a \notin A_b \implies \text{target} = \zeta + \gamma_w \max_{a' \in A_w} Q(s', a'),$$

where s' is the next state, regardless of type. In the former equation, s_{l_2} is the next state in which a base action is taken when following the greedy policy. In the latter equation, if the $\max$ returns the Q-value for an $a \notin A_b$, the two terms correspond to the reward and expected discounted future reward. When the $\max$ returns the Q-value for an $a \in A_b$, the two terms are then the immediate reward and the reward from a_{stop} in the next state, effectively removing the terminal/non-terminal distinction for the tree agent.

As a result, this two-agent problem is equivalent to a single agent updating a single Q-function using two different targets, depending on the action taken. The equation for computing a target differs from the standard Q-function update equation (as applied to the IBMDP) in one way: if a base action is taken, the "next state" is the next state in which a base action is taken, rather than simply the next state. This single change is sufficient to learn DTPs for IBMDPs.

3.2.3.3 Omniscient Q-function

The above merged agent formulation can be directly used to learn DTPs. However, the merged formulation requires the next leaf state, s_{l_2}, when a base action is taken. This state is not naturally encountered when performing off-policy exploration, so s_{l_2} must be computed by repeatedly querying the Q-function with a sequence of s_d tree states until the next base action is chosen. As a result, computing a single base action target Q-value requires simulating the next choice of base action, roughly doubling the computation time.

As an extension of the merged agent formulation, we propose to approximate $Q(s_{l_2}, a)$ using a second Q-function, Q_o. We refer to this second Q-function as an *omniscient Q-function*, because its input is the full state s_w. Q_o is used in a supporting role during training; the policy is obtained directly from Q. As a result, providing Q_o the full state, s_w, does not violate the extraction process's requirement that the policy is a function of only $s_w \setminus s_b$.

The omniscient Q-function is trained to approximate $Q(s_{l_2}, a)$ based on a and the full state at s_{l_2}'s root, s_r. This root state is sufficient since s_{l_2} is obtained from s_r through a sequence of actions, each based on the previous s_d. Therefore, the current greedy policy corresponds to some function $F(s_r) = s_{l_2}$ for all (s_r, s_{l_2}) pairs. We have Q_o implicitly learn this function as it aims to learn an approximation $Q_o(s_r, a) \approx Q(s_{l_2}, a)$ for all base actions.

Additionally, the original merged formulation learns the Q-value at each level in the tree (for $s_{d1,1}, s_{d1,2}$, etc.) using targets computed from the next level. This leads to slow propagation of environment reward signals from the leaf nodes. In addition to using Q_o for the root node, we propose to have it learn to approximate $Q_o(s_w, a) \approx Q(s, a)$ for all states and all actions. Since Q_o has access to s_b, the rewards obtained in the leaf node, s_{l_1}, directly propagate through Q_o to earlier levels of the tree instead of sequentially propagating upward (from leaf to root).

As shown in Figure 3.3, during training, we use Q_o in cases where $Q(s, a)$ would be used as a target. The action choice is still based on $Q(s, a)$, but the value is obtained from Q_o. Both Q_o and Q are updated using the Q_o-based target.

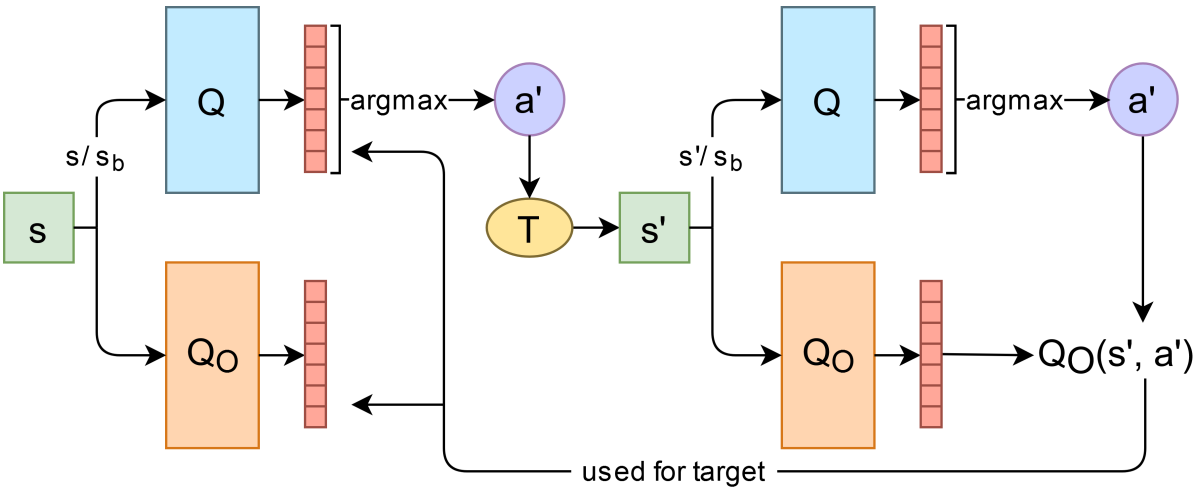


Figure 3.3: The method for using the omniscient Q-function, Q_o, for Q targets. The policy is based only on Q, so a DTP can be extracted despite Q_o being a function on the full state.

3.2.3.4 Modifying Standard RL Algorithms

For Q-learning-based methods, such as Dueling Deep Q-Networks (DDQN) [33] and Model-Free Episodic Control (MFEC) [34], we use the merged agent formulation target value of $Q_o(s_w, \arg\max_a Q(s_w \setminus s_b, a))$ in place of $\max_a Q(s, a)$ (for both $a \in A_b$ and $a \notin A_b$); the additional Q-function, Q_o, is updated using the same target value. For policy gradient methods, such as Proximal Policy Optimization (PPO) [4], the Q-function is used to compute advantage values only during training. Therefore, we use only Q_o, not Q, to compute advantages. Q_o is then trained using the merged agent formulation target value computations (replacing $Q(s, a)$ with $Q_o(s_w, a)$).

3.3 Experiments

We evaluate CUSTARD's ability to generate DTPs through solving an IBMDP using a non-interpretable function approximator during the learning process. An alternative to implicitly learning a DTP is to learn a non-tree expert policy and then find a tree that mimics the expert. We compare to VIPER, which takes this alternative approach and outperforms standard imitation learning methods. VIPER gathers samples using a DAGGER variant and weights the samples during tree training. For this evaluation, we use three environments, as described in Section 2.2. PrereqWorld is parameterized by m and ρ, which we set to 10 and 0, respectively. We produce smaller PrereqWorld variants by removing high-numbered items (based on the topological sort).

3.3.1 Learning with CUSTARD

To evaluate CUSTARD's ability to produce DTPs with a non-interpretable function approximator for the IBMDP, we apply the CUSTARD modifications to three base methods: DDQN, PPO, and MFEC with improvements from Neural Episodic Control [35]. DDQN is a Q-learning

	CartPole Reward	PrereqWorld Reward	Depth	PotholeWorld Reward	Depth
VIPER (DQN)	200.00 (0.00)	-4.00 (0.00)	5.70 (0.62)	46.30 (0.39)	6.00 (0.61)
VIPER (BI)	200.00 (0.00)	-4.00 (0.00)	6.00 (0.00)	46.31 (1.32)	9.18 (1.08)
CUSTARD (DQN)	198.72 (4.74)	-4.08 (0.34)	4.28 (0.67)	46.92 (2.14)	5.36 (1.41)
CUSTARD (PPO)	199.32 (3.23)	-4.04 (0.20)	4.16 (0.47)	45.39 (0.42)	1.04 (0.75)
CUSTARD (MFEC)	200.00 (0.00)	-4.00 (0.00)	3.92 (0.27)	49.18 (1.04)	9.74 (0.49)

Table 3.1: Final average reward and tree depth for different methods that make a DTP. The values in parentheses are Standard Deviation values.

approach that uses a neural network to learn a state-value function and action-value function, which are combined to form a Q-function. PPO is a policy gradient method that uses a critic for estimating the advantage function. We use a neural network for both the actor and critic. MFEC is a Q-learning approach that uses a nearest neighbor model to estimate Q-values.

The modifications from Section 3.2.3 are applied to all three methods. Actions are selected based on IBMDP states without the base state ($s_w \setminus s_b$); this affects the actor for PPO and the Q-function for DDQN and MFEC. DDQN and MFEC are used with a target Q-value function, Q_o, when performing updates, as in Figure 3.3. The target function and the critic for PPO are used with full IBMDP states.

We compare to VIPER using two expert types: DQN and Backward Induction (BI). In Table 3.1, we show the final average reward and tree depth for 50 trials on CartPole, PrereqWorld ($m = 7$), and PotholeWorld. Optimal final average rewards would be 200, -4, and 50, respectively. CUSTARD finds DTPs with high average reward for all environments and tends to find shorter DTPs than VIPER. To further evaluate depth-vs.-reward trade-offs, we use VIPER(BI) and CUSTARD(MFEC) since these methods have the fewest hyperparameters and are least computationally expensive.

3.3.2 Response to Environment Size

CUSTARD discourages the learning of unnecessarily large trees through the use of two penalty terms, ζ and γ_w. These penalties are akin to regularization of the implicit DTP: when multiple optimal DTPs exist for the base MDP, the optimal IBMDP policy corresponds to the DTP with the lowest average leaf height. In contrast, if a tree mimics an expert policy, then the resulting tree will include complex behaviors that are artifacts of the expert's intricacy.

We evaluate the decrease in tree size attained by using CUSTARD to directly learn a tree. We compare the tree depth and node count for DTPs found by VIPER and CUSTARD on PrereqWorld. The environment size is varied through m, which specifies the number of states (2^m) and the number of actions (m). For a given m, average reward is equal for both methods. The results are shown in Figure 3.4 (50 trials per method/m pair). CUSTARD produces smaller trees for $m \geq 4$, and the size differences increases with the environment size. This is because an unconstrained expert can learn more complex behaviors with a larger state space, and VIPER faithfully mimics the expert policy.

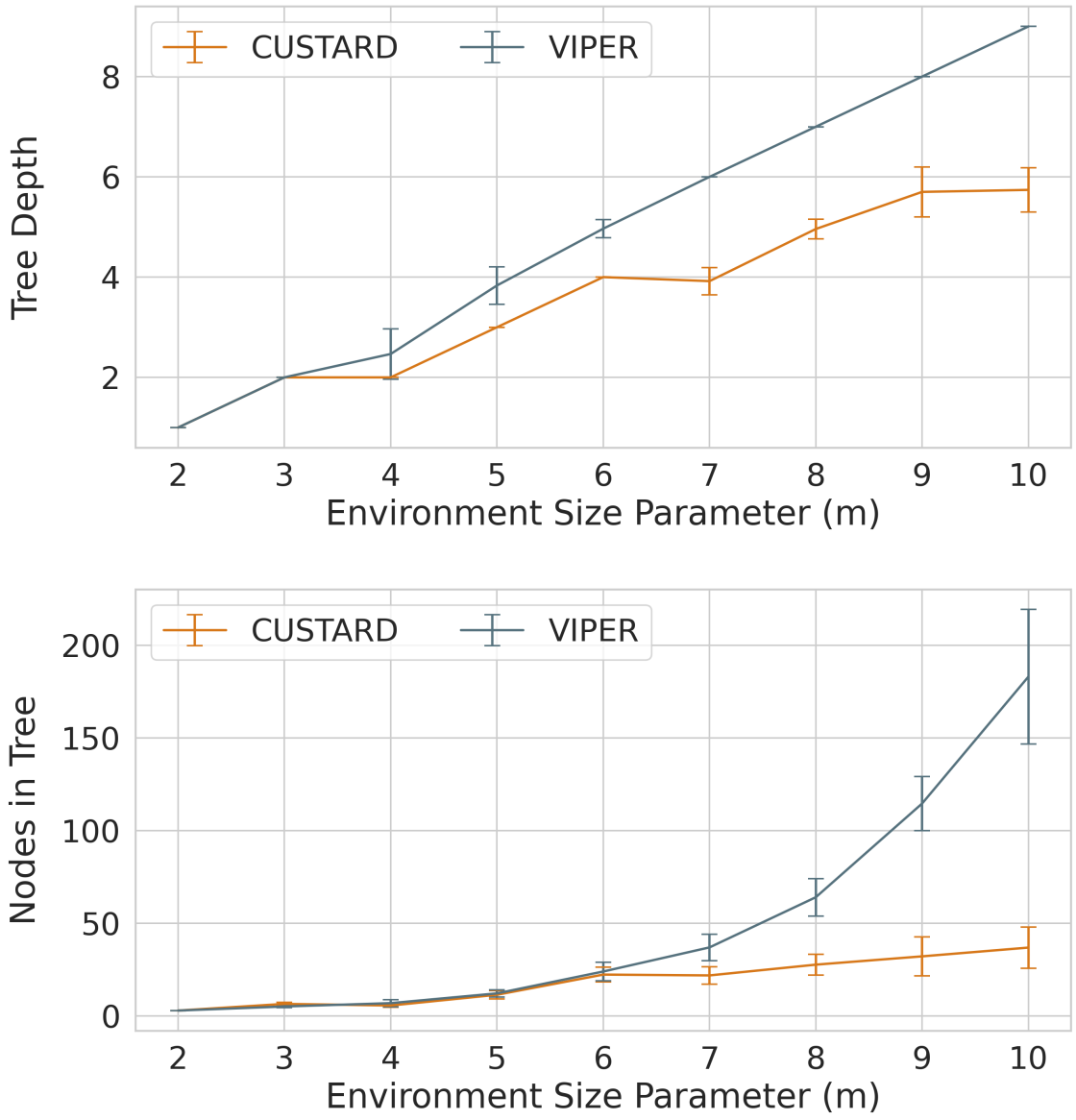


Figure 3.4: Tree depth and node count as the PrereqWorld environment size increases. The bars indicate the Standard Deviation. CUSTARD yields smaller trees for larger environments than VIPER.

3.3.3 Response to Tree Depth

If an application requires a DTP of fixed depth, then fitting a DT to an expert policy can yield a poor policy of that depth. This is because the expert is not learned in the context of the depth limitation; imperfectly imitating that expert can lead to low reward. CUSTARD yields better policies at a given depth since it directly solves an IBMDP that can be augmented with a depth limit. An IBMDP can include affordances [36], so that information-gathering actions cannot be

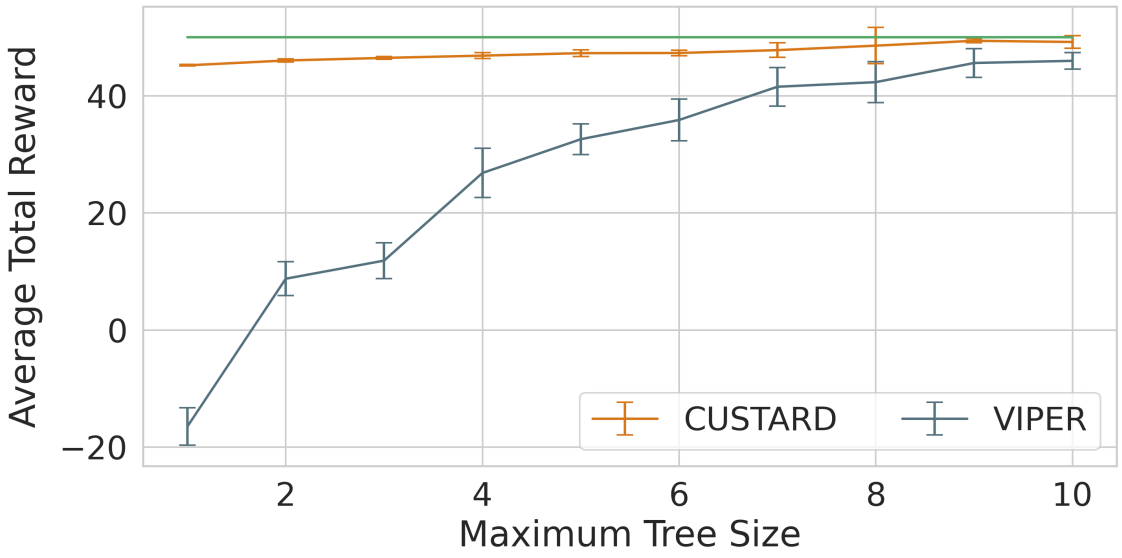


Figure 3.5: Average per-episode reward for trees of a fixed depth for PotholeWorld. The bars indicate the Standard Deviation. CUSTARD DTPs consistently achieve higher reward than VIPER's DTPs. The line at 50 indicates maximum possible per-episode reward.

chosen n actions after the most recent base action. With this modification, an RL algorithm can directly find the best DTP subject to the depth restriction.

We evaluate CUSTARD's ability to find DTPs with high average reward for PotholeWorld subject to a tree depth limit. This domain is designed so the overall optimal DTP cannot be pruned to obtain the optimal DTP for a smaller depth. We present the average episode reward as a function of the depth limit in Figure 3.5 for VIPER and CUSTARD (50 trials per method/depth pair). CUSTARD attains higher reward through using $lane\_1$ when the DTP depth is too shallow to avoid potholes in the other lanes. In contrast, VIPER always attempts to imitate the expert and attains a low reward when the DTP poorly represents the expert policy.

3.4 Summary

In this chapter, we introduced Iterative Bounding MDPs, an MDP representation that corresponds to the problem of finding a decision tree policy for an underlying MDP. Additionally, we identified how the standard value update rule must be changed so that all IBMDP solutions correspond to decision tree policies for the underlying MDP. We showed how existing RL algorithms can be modified to solve IBMDPs, so a non-interpretable function approximator can be used in conjunction with an existing RL method to solve an IBMDP and produce a decision tree policy. In addition, we provided empirical results showing the tree size and reward improvements possible through solving an IBMDP rather than approximating a non-interpretable expert with a decision tree after training.

We later incorporate the learned DTPs into our Unified Explanation Trees in Chapter 5. Inclusion of DTPs permits the UET to explain individual actions in a local context. In Chapter 7, the

overall structure is then augmented with the ability to identify which past experiences were most influential. As a result, the nodes within a DTP can also be explained in terms of influential past experiences.

Chapter 4

Abstract Policy Graph Creation

In this chapter, we move beyond the "single action" explanation introduced in Chapter 3 to explain longer-term behavior. To create a policy-level explanation, we express the policy as an Abstract Policy Graph (APG) [37], a Markov chain over abstract states. We present an algorithm for creating APGs, APG-Gen, that uses a learned value function and a set of observed transitions. We prove that the worst-case time complexity of our method is quadratic in the number of features and linear in the number of provided transitions, $O(|F|^2|tr\_samples|)$. By applying our method to a family of domains, we show that our method scales well in practice and produces APGs that reliably capture relationships within these domains.

Later, in Chapter 5, we demonstrate how to build decision trees over APG abstract states and then combine these trees with a DTP to form a Unified Explanation Tree. This process of building APG DTs reveals which features determine abstract state membership and permits analysis of an APG by recursively analyzing subtrees. The combination then permits a UET to describe both immediate behavior (for the next action) and longer-term behavior (for the sequence of all subsequent actions). In Chapter 7, we then find influential experiences for all nodes of a Unified Explanation Tree. Because of our incorporation of APG DTs into a UET, this identification of influential experiences also finds the influential experiences for partitioning abstract states.

4.1 Motivation

In the context of RL, autonomous agents learn to operate in an environment through repeated interaction. After training, the agent is able to make decisions in any given state, but is unable to provide a plan nor rule-based system for determining which action to take. Generally, a policy which selects actions ($\pi(s) = a$) is available along with its value function ($V_\pi(s) \in \mathbb{R}$), which predicts future reward from a state. However, neither the outcome of the actions nor the sequence of future actions taken is available. Without these, a human operator must blindly trust an RL agent's evaluation.

Existing techniques for explaining Deep Reinforcement Learning agents borrow techniques used for explaining neural network predictions, so they focus on explaining one state at a time.

These techniques pinpoint the features of the state that influence the agent's decision, but do not provide an explanation incorporating expected future actions. Therefore, the explanation is insufficient for a human supervisor to decide whether to trust the system. Likewise, no whole-policy view is available, so evaluating the agent's overall competency (as opposed to single-state evaluation) is impossible. For these reasons, we are interested in explaining policies as a whole: giving the context for action explanations and providing an abstraction of an entire policy.

To address the aforementioned issues, we propose the creation of a full-policy abstraction, which is then used as the basis for generating local explanations. We introduce Abstract Policy Graphs (APGs) as such a full-policy abstraction. Each APG is effectively a graph where each node is an abstract state and each edge is an action with associated transition probability between two abstract states. Using a mapping from states to abstract states, one can identify which groups of states the agent treats similarly, as well as predict the sequence of actions the agent will take. This explanation provides local explanations along with a global context.

An APG provides a way for users to answer questions about a specific state such as:

- "What other states are treated as equivalent?";
- "Which action will the agent take next?";
- "What are expected properties (feature values) of future states?"; and
- "Will the agent ever loop back to a state that is treated similar to this one?"

Additionally, we propose an algorithm, APG-Gen, for creating an APG given a policy, a learned value function, and a set of transitions. Starting with a single abstract state which encompasses the full state-space, APG-Gen uses a feature importance measure to repeatedly divide abstract states along important features. These abstract states are then used to create an APG. The splitting procedure additionally identifies which features are important within each abstract state. Notably, this general procedure is compatible with existing methods for learning a policy and value function.

The main contributions of this chapter are as follows:

1. we introduce a novel representation, Abstract Policy Graphs, for summarizing policies to enable explanations of individual decisions in the context of future transitions,
2. we propose a process, APG-Gen, for creating an APG from a policy and learned value function,
3. we prove that APG-Gen's runtime is favorable ($O(|F|^2|tr\_samples|)$, where F is the set of features and $tr\_samples$ is the set of provided transitions),
4. we empirically evaluate APG-Gen's capability to create the desired explanations.

4.2 Approach: APG-Gen

In Section 4.2.1, we first briefly describe an existing approach for computing feature importance; we use this without APG-Gen to partition sets of states. In Section 4.2.2, we describe Abstract Policy Graphs, our representation for explaining a policy. In Section 4.2.3, we propose APG-Gen, a method for constructing such explanations. In Section 4.2.4, we describe local explanations we produce from our policy explanations. Finally, in Section 4.2.5, we show that our method has favorable asymptotic runtime: quadratic in the number of features and linear in the number of transition tuples considered, where there are usually few features and runtime sub-linear in the number of transitions is unattainable.

4.2.1 Feature Importance Function

We use an *importance measure* for grouping states from an original MDP into abstract states. An importance measure is a function $I_f(c)$ that represents the *importance* of feature f in determining how a system treats a set of inputs (e.g., states), c. If f takes on the same value for all $s \in c$ or its value does not influence the system's output, then f is not important. We use the Feature Importance Ranking Measure (FIRM) [38] since it is fast to compute exactly for binary features and can be meaningfully interpreted. As a result, our method is applicable to domains where the states are featurized with binary features (i.e., all features are 0 or 1).

To calculate importance, FIRM uses $q_f(v)$, the *conditional expected score* of s for a feature f with respect to an arbitrary function $g(s)$. This score is the average value of $g(s)$ for all s within the set c where feature f takes value v:

$$q_f(v) = \mathbb{E}(g(s)|s[f] = v). \tag{4.1}$$

Intuitively, if $q_f(v)$ is a flat function, then v, the value of f, has no impact on the average value of $g(s)$ over $s \in c$, so provides little information. However, if v significantly impacts $g(s)$, then the value of $q_f(v)$ will vary. This motivates FIRM's importance measure $I_f(c)$, the variance of the conditional expected score:

$$I_f(c) = \sqrt{\mathbb{V}(q_f(s[f]))}. \tag{4.2}$$

In specific cases, the exact value of $I_f(c)$ can be computed quickly. One such case is when f is a binary feature. For a binary feature, the importance measure is given by

$$\begin{aligned} I_f(c) &= (q_{f0}(c) - q_{f1}(c))\sqrt{p_{f0}(c)p_{f1}(c)}, \\ p_{fv}(c) &= \mathbb{P}(s[f] = v), \\ q_{fv}(c) &= \mathbb{E}(g(s)|s[f] = v). \end{aligned} \tag{4.3}$$

In the case of binary features, FIRM corresponds to the expected change if the feature switches from 0 to 1. Conveniently, sign is preserved in the binary case, showing magnitude of importance as well as direction of effect.

4.2.2 Abstract Policy Graphs

To create a policy-level explanation, we express the policy as a Markov chain over abstract states where edges are transitions induced by a single action from the original MDP, which we term an Abstract Policy Graph (APG). We present an example in Figure 4.1. Consider a mapping function, $l(s)$, which maps states in the original MDP ($grounded$ states) to abstract states. In effect, each abstract state represents a set of grounded states from the original MDP. We use the phrase "an agent is *in abstract state* b" to mean that the current state of the domain, s, maps to b (i.e., $l(s) = b$). Let each set contain all states *interchangeable* under the agent's policy such that the agent behaves similarly when starting in a state from the set in a similar fashion. As a result, states in which the agent behaves similarly lead the agent to states in which the agent also behaves similarly. If the agent's transitions between grounded states are approximated using a Markov chain between these abstract states, then states that are treated similarly are readily identified and the agent's transitions between abstract states can be predicted.

For example, let the agent's distribution of actions be approximately equal for all future time-steps for all grounded states in the set:

$$\mathbb{E}_{s_{1,t}}\mathbb{P}(\pi(s_{1,t}) = a) \approx \mathbb{E}_{s_{2,t}}\mathbb{P}(\pi(s_{2,t}) = a) \forall a, t \tag{4.4}$$

for all s_1 and s_2 within a set, where $s_{i,t}$ is the state reached from s_i in t further time-steps using policy π.

If the transition function is deterministic, then the agent takes the same sequence of actions from each grounded state in the set because there is only a single $s_{i,t}$ for each i and t. In addition, since no two sets could combine to form an interchangeable set, then the abstract states for all $s_{i,t}$ are identical, too. Since the probability of transitioning from one abstract state to another after one action is then either zero or one (regardless of grounded state), the agent is effectively traversing a Markov chain of abstract states induced by its policy.

However, most interesting domains have stochastic transition functions. Under a stochastic transition function, two grounded states can satisfy Equation 4.4 while having different transition probabilities to future abstract states. The transition probability from one abstract state to another can be approximated as the average transition probability for grounded states in the source abstract state. For stochastic policies, the probability of taking any given action can be similarly approximated as the average over all grounded states in the source abstract state. The transition probability is no longer exact for any given grounded state in the set but is the transition probability for a randomly chosen state in the set. To make predictions for a series of transitions, we make a simplifying Markov assumption: the abstract state reached, b_{t+1}, when performing an action depends only on the current abstract state, b_t. This assumption leads to approximation error but works well in practice, as shown in Section 4.3.3.

The abstract states now form a Markov chain, as desired. This final product allows human examination of higher-level behavior (e.g., looking at often-used trajectories and checking for loops), prediction of future trajectory (along with accompanying probability), and verification of agent abstraction (e.g., ensuring agent's behavior is invariant to certain features being changed).

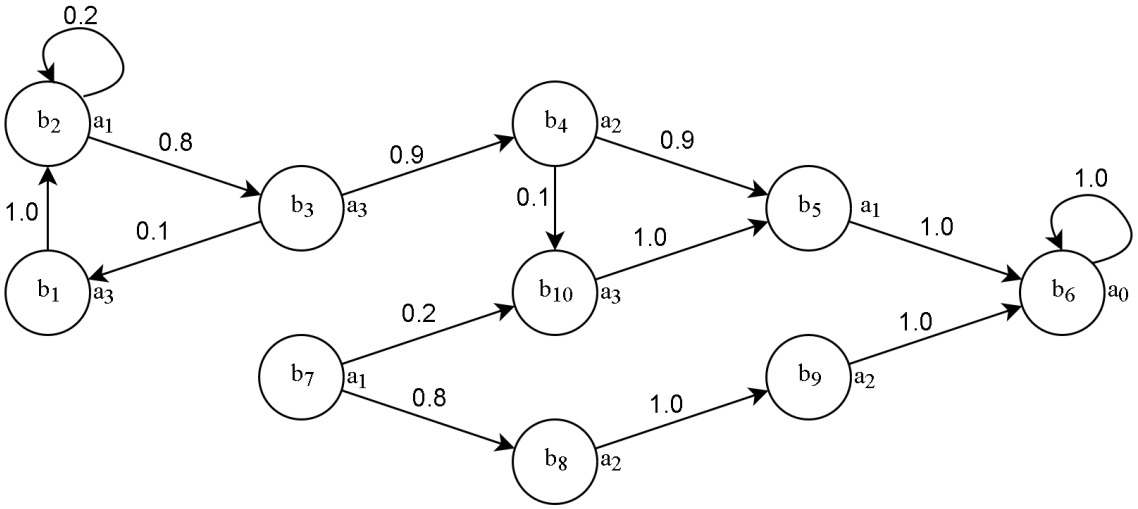


Figure 4.1: An example Abstract Policy Graph with edge labels indicating transition probabilities. The abstract state identifier is shown within each node, and the action taken is written adjacent to the node.

4.2.3 APG Construction

We propose an algorithm for creating APGs, APG-Gen. It first divides states into sets to form abstract states, then computes transition probabilities between them.

4.2.3.1 Importance Measure

APG-Gen is compatible with arbitrary interchangeability measures. We choose $V_\pi(s)$ as it is readily available, but the method does not rely on this choice. Using an interchangeability measure based on Equation 4.4 is difficult since it would depend on an expectation over all future states, which is often computationally expensive and requires knowing transitions for all states. We note that the definition of $V_\pi(s)$ in Equation 2.1 also includes an expectation over all future future states, as well as a dependency on the policy. Since the full state-value function is generally available and does not require computing additional expectations, we use it as our measure of interchangeability. The intuition is that two states with similar state-values lead to similar future outcomes in terms of reward, so are likely treated similarly by the agent. A different measure could be used instead

With an importance measure $I_f(c)$ for $V_\pi(s)$, a set of states which is interchangeable under the agent's policy should have low $I_f(c)$ for all f. Consider the case of $c_1 \bigcup c_2$, a set containing the original MDP states which should be contained in two abstract states. At least one f should have high $I_f(c_1 \bigcup c_2)$ because the grounded states from the two abstract states are treated differently. If there is no such f, then the two sets are treated the same and therefore belong to the same abstract state.

4.2.3.2 Splitting Binary Features

In the case where all features are binary, if the set is split based on the value of that f (into one subset if $f = 0$ and the other if $f = 1$), then both subsets will have I_f be 0, since f has a constant value within the subset. This holds for any set of grounded states which will ultimately form several abstract states. Therefore, this splitting procedure can be repeatedly performed to create abstract states from initially larger sets until all features have low importance.

Since each binary feature can only be important once and it is straightforward to split along a binary feature, the use of binary features allows quick computation. Therefore, in cases where the original MDP does not have solely binary features, pre-processing can be done to create features for APG-Gen. Note that these features are not used when evaluating $V_\pi(s)$ (i.e., an unmodified, arbitrary model can be used for approximating $V_\pi(s)$). The binary features are instead used when deciding to which set a specific tuple belongs while performing APG-Gen.

4.2.3.3 Abstract State Division

Since binary features allow efficient splitting, our approach is to initially form sets based on action taken under the current policy and then repeatedly split the set which has the greatest I_f value, as computed within that set. When the importances of all features for all abstract states are sufficiently low, then the abstract states consist of sets of states which are interchangeable under the agent's policy.

The pseudocode for our method is given in Algorithm 2. To perform the procedure, we require a set of sample transitions. As mentioned in Section 2.1.1, RL agents generally learn through interacting with a domain, meaning a set of (s, a, s') transitions is generally available. The notation we use for this set is a vector $tr\_samples$ consisting of entries t where action t_a is taken in state t_s, leading to a transition to state $t_{s'}$, an observed reward t_r, and a termination flag t_t (0 or 1). The policy is used in line 5 to discard transition tuples where the provided policy would perform a different action from the action in the stored tuple. This is done so that the generated explanation reflects only the current policy and not transition tuples observed under past policies.

Lines 2-6 separate the tuples based on the action taken. We pre-compute the feature importance for each set and save it in lines 7-8. Line 9 forms the core procedure, where abstract states are divided until no feature has importance greater than ϵ. The abstract state with the most important feature is found, then divided based on the most important feature. The importance of each feature is then re-computed in lines 18 and 20.

4.2.3.4 APG Edge Creation

Once the abstract state sets have been created, we create the mapping function l and Markov chain transition matrix using Algorithm 3 (a sparse matrix can be created in an almost identical fashion). In lines 2-4, the contents of each set are used to create the necessary entries in a lookup table for the mapping function. Simultaneously, the transition matrix is initialized to be zero-valued by lines 5-6. Then, in lines 7-13, the mapping function is used in conjunction with the transition tuples within each abstract state set to compute transition probabilities. That is, if

Algorithm 2 Compute abstract states based on transition samples and learned policy.

```
1:  procedure DIV_ABS_STATES(tr_samples, policy)
2:      for i in {1, ..., |A|} do
3:          c[i] ← ∅                                          ▷ initially, all sets empty
4:      for t in tr_samples do                                ▷ separate by action
5:          if policy(t_s) = t_a then
6:              c[t_a] ← c[t_a] ∪ t
7:      for i in {1, ..., c} do                               ▷ pre-compute feat. imp.
8:          m[i] ← [|I_f(c[i])| for f ∈ {1, ..., |F|}]
9:      while max_i max_j (m[i][j]) > ε do
10:         i_max ← argmax_i max_j (m[i][j])
11:         j_max ← argmax_j (m[i_max][j])
12:         c_{n_0}, c_{n_1} ← ∅
13:         for t in c[i_max] do                              ▷ split on most imp. feat.
14:             if t_s[j_max] = 0 then
15:                 c_{n_0} ← c_{n_0} ∪ t
16:             else
17:                 c_{n_1} ← c_{n_1} ∪ t
18:         m[i_max] ← [|I_f(c_{n_0})| for f ∈ {1, ..., |F|}]
19:         c[i_max] ← c_{n_0}
20:         m[|c|] ← [|I_f(c_{n_1})| for f ∈ {1, ..., |F|}]
21:         c[|c| + 1] ← c_{n_1}
22:     return c
```

a transition tuple t is in set $c[i]$, then the origin state, t_s, is in the abstract state represented by $c[i]$. The destination state, $t_{s'}$, then indicates a connection between $c[i]$ and $l(t_{s'})$. The transition probability from $c[i]$ to $c[n]$ is the portion of tuples in $c[i]$ which lead to a state in $c[n]$, so each tuple in $c[i]$ should increment $transition(i, l(t_{s'}))$ by $1/|c[i]|$, as done in line 13. Terminal transitions are identified in line 9 and instead lead to the special b_T abstract state. This abstract state represents termination and is represented by the highest-numbered row and column in $transition$. There will be no incoming edges to b_T if the set of terminal states, T, is empty. Line 14 sets b_T to have an edge to itself to create a valid Markov chain.

4.2.4 Abstract State Summarization

The policy graph algorithm presented creates a summary of the overall policy out of abstract states, which are each defined by a set of states from the original MDP. Due to the process that we use to create the abstract states, we can also create a characterization of the states that are in their set. Note that Algorithm 2 splits an abstract state into two based on a feature f because f is "important" based on chosen function g. These fs can be trivially recorded and stored for each abstract state. Once the final abstract states have been created, these fs indicate which

Algorithm 3 Create mapping function and transition matrix based on policy graph.

procedure COMPUTE_GRAPH_INFO(c)
  for i in $\{1, \ldots, |c| + 1\}$ **do**
    for t in $c[i]$ **do**
      $lookup[t_s] \leftarrow i$ ▷ create lookup table
    for n in $\{1, \ldots, |c| + 1\}$ **do** ▷ zero matrix
      $transition(i, n) \leftarrow 0$
  for i in $\{1, \ldots, |c|\}$ **do**
    for t in $c[i]$ **do**
      if $t_t = 1$ **then** ▷ terminal t_t go to dummy b_T
        $n \leftarrow |c| + 1$
      else ▷ others go to abstract state of next state
        $n \leftarrow lookup[t_{s'}]$
      $transition(i, n)$ += $1/|c[i]|$
  $transition(|c| + 1, |c| + 1) \leftarrow 1$ ▷ add b_T self-loop
  return $lookup, transition$

features were previously important. From this, the important features of an abstract state can be determined.

For any state in the transition set, s_n, and a specific abstract state, b, if $\pi(s_n) = \pi(s)$ and $s_n[f] = s[f]$ for any s in b's set and for all f which were used to create b, then s_n will also be in b's set. Similarly, if $s_n[f] \neq s[f]$ (for similarly defined s and f), then s_n cannot be in b's set. These feature-value assignments are necessary and sufficient to be part of b, so this creates an "if and only if" relationship. As a result, for any chosen state s, based on the features used to create its abstract state, the "relevant" features can be determined. If the value for any of these features changes, then s would be in a different abstract state and treated differently. Similarly, the agent is oblivious to changes in the other features given the values assigned to the relevant features. This relationship allows a human supervisor to determine which features affect how an agent treats a specific state. In addition, a summary of an abstract state can be formed using these same feature-value assignments.

4.2.5 Computational Complexity

We present the big-O runtime of our method. Since an agent may be trained on millions of transitions, it is important that an interpretability method scale favorably with respect to the number of transitions. We show that our approach takes time linear in the number of samples.

4.2.5.1 Computing FIRM

Since $p_{f1}(c) = 1 - p_{f0}(c)$ and $q_{f1}(c) = (\mathbb{E}(g(s)) - p_{f0}(c)q_{f0}(c))/p_{f1}(c)$, computing $p_{f0}(c)$ and $q_{f0}(c)$ is enough to calculate the importance. These can be computed for all f with a single pass through the set of states, as shown in Algorithm 4.

Algorithm 4 Compute feature importance for all features for given set of transitions.

1: **procedure** FIRM($tuples$)
2: $q_{tot} \leftarrow 0$ ▷ expected value over full set
3: **for** f in $\{1, \ldots, |F|\}$ **do**
4: $p_0[f] \leftarrow 0$ ▷ ratio of set with $s[f] = 0$
5: $q_0[f] \leftarrow 0$ ▷ $\mathbb{E}(s|s[f] = 0)$ for s in set
6: **for** t in $tuples$ **do**
7: $g\_val \leftarrow g(t_s)$
8: q_{tot} += g_val ▷ store sum for q_{tot}
9: **for** f in $\{1, \ldots, |F|\}$ **do**
10: **if** $s[f] = 0$ **then** ▷ p_0 is tally, q_0 is sum
11: $p_0[f]$ += 1
12: $q_0[f]$ += g_val
13: $q_{tot} = q_{tot}/|tuples|$ ▷ convert sum to average
14: **for** f in $\{1, \ldots, |F|\}$ **do** ▷ intermediate terms
15: $q_0[f] \leftarrow q_0[f]/p_0[f]$
16: $p_0[f] \leftarrow p_0[f]/|tuples|$
17: $p_1[f] \leftarrow 1 - p_0$
18: $q_1[f] \leftarrow (q_{tot} - p_0[f]q_0[f])/p_1[f]$
19: $q_{diff}[f] \leftarrow q_0[f] - q_1[f]$
20: **return** $[q_{diff}[f]\sqrt{p_0[f]p_1[f]}$ **for** f in $\{1, \ldots, |F|\}]$

The bulk of the computation is performed in lines 6 to 12. Here, every transition in the set is separately considered. Only a single evaluation of g is required regardless of the number of features. This evaluation is used to calculate $\mathbb{E}(g(s))$ and q_{f0} for each feature where $s[f] = 0$. The overall complexity of Algorithm 4 is therefore $O(|F||tuples|)$, where $|F|$ is the number of features and $|tuples|$ is the number of transitions over which FIRM is computed.

4.2.5.2 APG-Gen Runtime

The runtime of Algorithm 2 is quadratic in the number of features and linear in the number of provided transitions, $O(|F|^2|tr\_samples|)$.

Creating the initial abstract states (i.e., those based only on action taken) takes time $O(|A| + |tr\_samples|)$, where we assume $|A| \leq |tr\_samples|$. Computing FIRM for all of these abstract states takes $O(|F||tr\_samples|)$ time. The while loop in lines 9 to 21 forms the bulk of the algorithm, which we will analyze last. Creating the lookup and transition tables takes $O(|tr\_samples|)$, assuming a zero matrix can be created in constant time for line 6.

For lines 9 to 21, during each iteration of the while loop, the runtime is $O(\log_2(|c|))$ to insert the new i_{max}s if a max-heap is used to store the j_{max}s, $O(|c[i_{max}]|)$ to partition the set $c[i_{max}]$, and $O(|F||c[i_{max}]|)$ to compute FIRM for both new sets. Note that each time a set is divided, the number of features within that set with non-fixed values (and therefore positive importance) is reduced by one. Therefore, any given tuple may only be part of an evaluated set $|F|$ times.

As a result, over all iterations of the loop, the set division and FIRM computation takes at most $O(|F|^2|tr\_samples|)$ time. This can happen over the course of up to $2^{|F|}$ divisions, so the max-heap insertion takes time at most $O(|F|)$.

The overall worst-case runtime for APG-Gen is then on the order of $O(|F|^2|tr\_samples|)$. This is favorable since runtime must be at least linear in $|tr\_samples|$ and F is generally small compared to the number of tuples.

4.3 Experiments

We evaluate APG-Gen on our domain with scalable state space and controllable stochasticity. This domain, PrereqWorld, is described in Section 2.2.1. Experimental settings are described in Section 4.3.1.

An example APG made by APG-Gen for an instance of PrereqWorld is given in Figure 4.2. APG-Gen additionally describes each abstract state. For example, b_{16} corresponds to all states where features 2 and 3 are 1. This corresponds to always taking action a_1 when an i_2 and i_3 are present, which corresponds to $C_1 = \{i_2, i_3\}$ in this domain instance. This correspondence between the domain constraints and the explanation would allow a human operator to verify that an agent is behaving as expected.

4.3.1 Experimental Settings

We briefly describe how the subsequent experiments were performed. In particular, we describe: (1) how transitions were generated, (2) how APG-Gen was configured, (3) how plot points were created, and (4) how the domains were paramterized.

4.3.1.1 APG Inputs

For consistency, we use value iteration [39] to create the policies and value functions used for experiments, but other methods could be used instead. We iterate until the state-value function no longer changes. To generate the transitions, we generated trajectories from a random starting state until the maximum number was reached.

4.3.1.2 APG-Gen Stopping Criterion (ϵ)

In the case of binary features, FIRM corresponds to the expected change should the feature be changed from 0 to 1. Conveniently, sign is also preserved in the binary case, showing magnitude of importance as well as direction of effect. As a result, if no feature for any abstract state has FIRM magnitude greater than ϵ, then changing any given feature is not expected to change the value of $g(s)$ by more than ϵ (e.g., $\mathbb{E}_{s \in c}(|g(s, s_f = 0) - g(s, s_f = 1)|) < \epsilon \forall c$). We use this as a guideline for setting ϵ: we set ϵ to be the minimum difference in action-value between the best action and second-best action. For the PrereqWorld domain, this is $\epsilon = 1$.

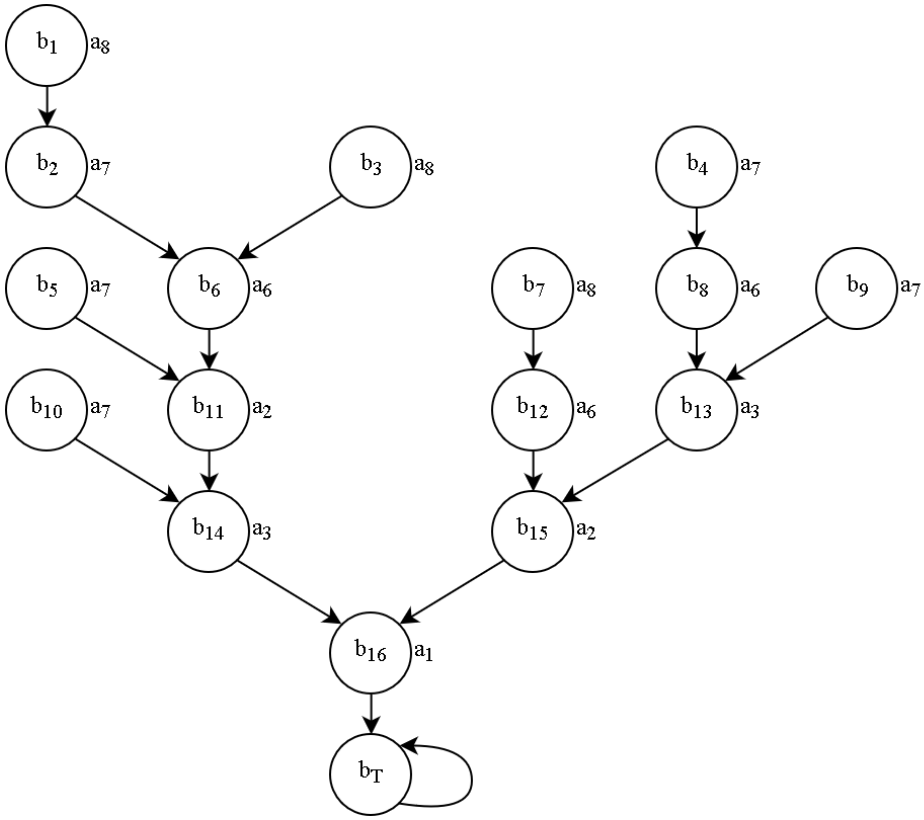


Figure 4.2: An example APG made by APG-Gen for a small PrereqWorld domain instance with $m = 8$ and $\rho = 0$. All edges have transition probability 1. The abstract state identifier is shown within each node, and the action taken is written adjacent to the node.

4.3.1.3 Trials

For each plotted data-point, we generate 100 different PrereqWorld instances. We evaluate each instance 1,000 times (i.e., we compute the feature importance for 1,000 different states or predict the nth action for 1,000 different trajectories), except for the points in Figure 4.5, since the explanation size is fixed per APG.

4.3.1.4 Domain Generation

Each domain instance is parameterized by ρ and m as specified for each experiment. For simplicity, d is always i_1. For each instance, we randomly add prerequisite relationships by selecting an item i_j uniformly at random and then an item i_k uniformly at random such that $k > j$. When adding prerequisite relationships, we constrain the expected number of actions to reach a terminal state to be within 10% of $2m$. This ensures that the domain can be solved in a reasonable amount of time using value iteration.

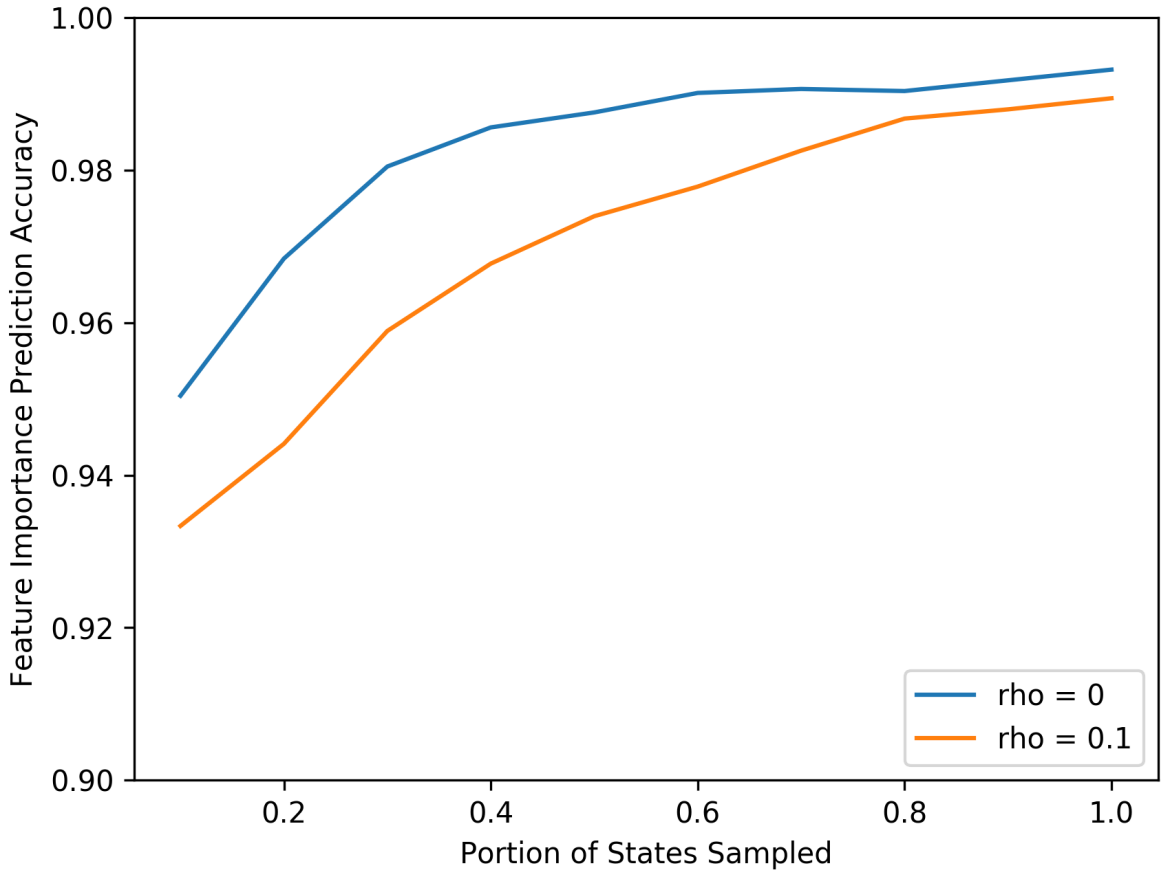


Figure 4.3: Comparison of feature importance prediction accuracy for increasing portion of non-terminal states for stochastic and deterministic PrereqWorld instantiations.

4.3.2 Local Explanation Generalization

Based on the way we construct our abstract states, we can create "if and only if" conditions for a state in the transition sample set to be part of an abstract state's set, as described in Section 4.2.4. From this, we can create a local explanation consisting of the set of features that are important in that state. To evaluate how well APG-Gen can generalize when predicting important features, we generate APGs using a set of transitions less than the full set of non-terminal states (i.e., we provide a set of transitions where no (s, a, s') tuple shares an s such that only a portion of non-terminal states appear as s). We then evaluate the local explanations by comparing to a ground truth computed for individual PrereqWorld instances with a domain parameter of $m = 15$ ($|S| = 2^{15}$).

The portions of correct feature classifications (important vs. not important) are shown in Figure 4.3. APG-Gen almost always correctly identifies the important features. Even with only 10% of the states, the prediction is correct over 93% of the time. When given 80% of the states, predictions are correct 98.7% of the time for both the stochastic and deterministic environments, which suggests that the model is able to identify genuine patterns in the policy. We believe

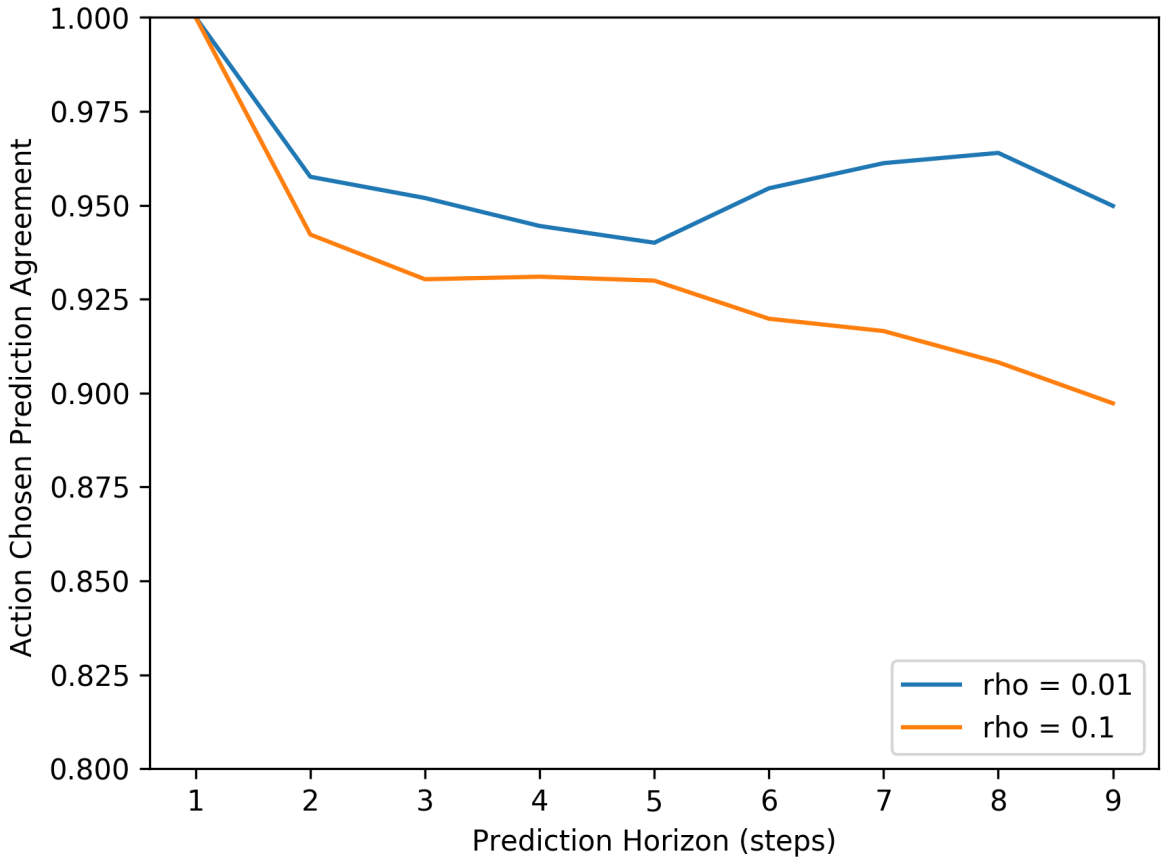


Figure 4.4: Action prediction for increasing time horizon for stochastic and deterministic PrereqWorld instantiations.

the errors the system makes are caused by the splitting order induced by APG-Gen's greedy splitting strategy.

4.3.3 n-hop Prediction Evaluation

An APG is able to predict the actions an agent will take, but this ability comes from an assumption made in Section 4.2.2. For each pair of abstract states, we produce a transition probability: the probability that the agent will be in the second abstract state, assuming the agent is following a transition tuple chosen at random from that first abstract state. This holds for a single action for states in the provided transition sample set, but a sequence of states is not independent, so performing several of these predictions in sequence will not yield the true probability of a state sequence.

To evaluate the error caused by making this assumption, we have APG-Gen predict the distribution of actions the agent will take n time-steps in the future. We compare it to the true computed distribution and report the portion of actions for which the true and predicted distributions

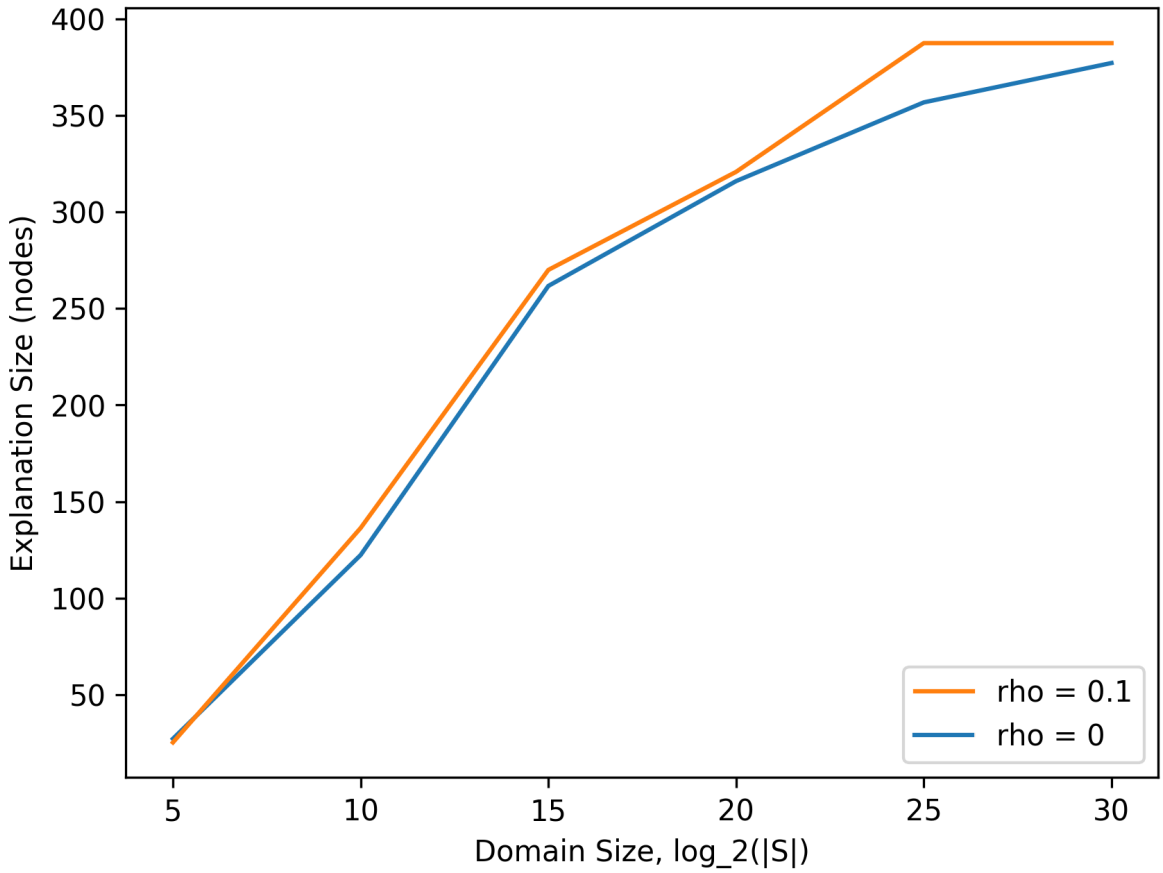


Figure 4.5: Comparison of explanation versus state-space size for stochastic and deterministic PrereqWorld instantiations.

agree. This is for a domain parameter of $m = 15$ ($|S| = 2^{15}|$). The size of the transition sample set is half the size of the set of non-terminal states.

The action prediction is consistently correct when the domain is deterministic, so we report results for two stochastic domains in Figure 4.4. Even with a small ρ, the prediction is less accurate as the number of steps increases, as is expected. However, there is no dramatic decrease, suggesting that the Markovian assumption made in Section 4.2.3 is reasonable. The steady decline is likely due to computing transition probabilities as an average of the transition sample set.

4.3.4 Explanation Size

The purpose of APGs is to be more human-interpretable than a Markov chain made from the base MDP. Therefore, the number of nodes in an APG should be much lower than the number of grounded states in the base MDP. To test this, we construct domains with a number of states ranging from 32 to 1,073,741,824 and count the number of abstract states in the corresponding APG. As in Section 4.3.3, for each generated APG, the size of the transition sample set is half

the size of the set of non-terminal states. The results are presented in Figure 4.5. Note that the x-axis is in log-scale.

The explanation size grows sub-linearly in m while the state-space size grows exponentially in m. This suggests that the explanation size is based more on the number of actions required to reach a terminal state than the number of states, which indicates that compact policy representations are being automatically extracted.

4.4 Summary

In this chapter, we introduced Abstract Policy Graphs, a whole-policy explanation from which state-specific explanations can be extracted. In addition, we presented APG-Gen, an algorithm for creating an APG given a policy, learned value function, and set of transitions, without constraints on how these are created. We showed that APG-Gen runs in time quadratic in the number of features and linear in the number of transitions provided, $O(|F|^2|tr\_samples|)$. Additionally, we demonstrated empirical results showing the small size of the APGs relative to the original MDPs, as well as the types and quality of explanations that can be extracted. Together, these show that APG-Gen can produce concise policy-level explanations in a tractable amount of time.

Chapter 5

Unified Explanation Trees

In Chapter 3 and Chapter 4, we present methods for single action and policy-level explanations in terms of state features, but these methods yield separate explanations of different aspects of the agent's behavior. Given our objective of producing an explanation that simultaneously conveys all these aspects, we now focus on how CUSTARD and APG-Gen can be used to create such a joined explanation. We first introduce the target explanation structure, Unified Explanation Trees. Next, we describe how to modify APG-Gen to extract a set of decision trees mapping from states to abstract states during the cluster partitioning process. We then introduce two ways to create Unified Explanation Trees: one combines the trees produced by CUSTARD and APG-Gen after each has been used; the other integrates the two methods more closely. Finally, we demonstrate the reduction in overall explanation size obtained by using our integrated approach.

5.1 Motivation

Decision Trees have favorable properties when used as an explanation format. We briefly note some relevant ones: (1) A DT directly conveys which features are considered within different portions of the input space. (2) A DT can be used to determine which features are relevant when deciding whether to pick a specific label of interest. (3) Meta-information about a DT's structure (e.g., its size) conveys the relative complexity of the DT. (4) The recursive structure of a DT permits explaining it recursively. (5) A single traversal through a DT uses only a small portion of the tree's parameters, allowing a single decision to be explained more readily. In contrast, consider a neural network, where all parameters are used for each decision. (6) Similarly, a traversal through a DT consists of a sequential application of simple rules; in our case, each partition requires a single binary comparison to a single other parameter to producing a binary output. By comparison, a neural network requires matrix multiplication to produce an output matrix of continuous values.

Decision Tree Policies benefit from these properties, but a DTP only conveys this information (e.g., which features are used) for *local* behavior (i.e., choosing a single action based on the state). In contrast, an APG provides information about general properties of the policy as well as longer-term information about states: states are grouped into abstract states such that all

states within an abstract state lead to similar future behavior by the agent. This grouping enables abstract state membership alone to convey longer-term behavior information. However, an APG is not expressed as a DT, so it does not inherit the favorable properties. The potential for improved interpretability motivates us to modify APG-Gen to produce decision tree structures.

To maximally benefit from the recursive nature of a DT and its repeated partitioning of the input space, we seek a single tree rather than separate DTP and APG-based trees. The benefits of a single tree motivate us to integrate DTPs and APG-based trees into a single Unified Explanation Tree (UET). This unified tree accepts a state as an input and returns both an action and an abstract state. As a result, a single tree conveys both the agent's current action and information about future behavior. Explanation methods that rely on a tree structure can then be applied to the abstract state categorization performed by APG-Gen.

A UET provides a way for users to answer questions such as:

- "What will the agent do in state s?";
- "Which features informed this choice?";
- "What other states are treated as equivalent?"; and
- "What are expected properties (feature values) of future states?"

5.2 Approach: Unified Explanation Trees

We seek a Unified Explanation Tree (UET): a tree mapping from a state s to both an action a and an abstract state b. The abstract states should have a relationship as specified within an APG. The partitions within the tree should each be based on a single feature, as is done in a Decision Tree Policy.

We leverage CUSTARD and APG-Gen to produce a UET: CUSTARD produces a tree mapping from s to a, and APG-Gen produces a mapping from s to b. We first modify APG-Gen to also produce a forest of trees in Section 5.2.1. Then, in Section 5.2.2, we demonstrate how to combine this forest with a DTP to produce a UET. Finally, in Section 5.2.3, we present a further modification of APG-Gen that yields a more concise UET by more closely integrating CUSTARD's DTP.

5.2.1 Extracting a Forest via APG-Gen

As an intermediate step to creating a UET, we must obtain trees that convey longer-term behavior. We can extract suitable trees based on the clustering of states performed by APG-Gen. For each action, APG-Gen partitions the state space such that each region corresponds to an abstract state. Because this partitioning is performed based on one feature at a time and by splitting on this feature, the partitioning process implicitly creates one DT per action in the environment.

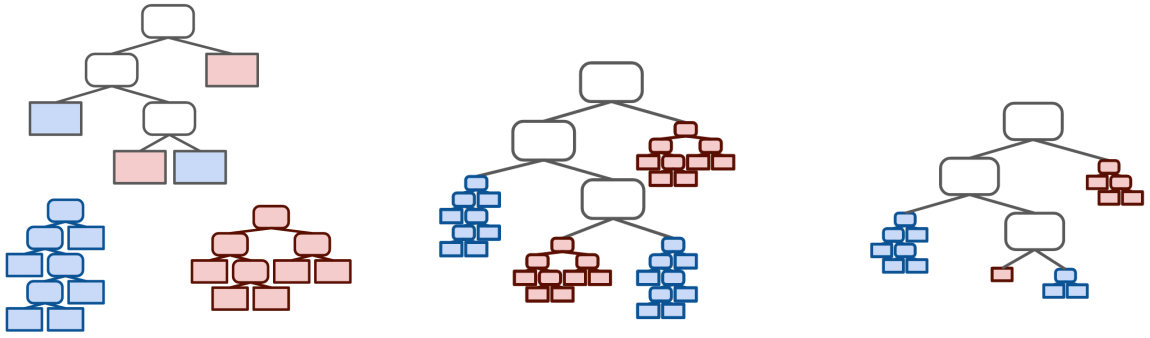
Algorithm 5 Compute abstract states based on transition samples and learned policy; simultaneously track performed partitions within a forest.

1: **procedure** DIV_ABS_STATES_WITH_FOREST($tr\_samples, policy$)
2: **for** i in $\{1, \ldots, |A|\}$ **do**
3: $c[i] \leftarrow \varnothing$ ▷ initially, all sets empty
4: $r[i] \leftarrow$ new_tree() ▷ create root node for each tree in forest
5: $l[i] \leftarrow r[i]$ ▷ entry in leaf list initially points to root node
6: **for** t in $tr\_samples$ **do** ▷ separate by action
7: **if** $policy(t_s) = t_a$ **then**
8: $c[t_a] \leftarrow c[t_a] \cup t$
9: **for** i in $\{1, \ldots, c\}$ **do** ▷ pre-compute feat. imp.
10: $m[i] \leftarrow [|I_f(c[i])|$ **for** $f \in \{1, \ldots, |F|\}]$
11: **while** $\max_i \max_j (m[i][j]) > \epsilon$ **do**
12: $i_{max} \leftarrow \text{argmax}_i \max_j (m[i][j])$
13: $j_{max} \leftarrow \text{argmax}_j (m[i_{max}][j])$
14: $c_{n_0}, c_{n_1} \leftarrow \varnothing$
15: $l_{n_0}, l_{n_1} \leftarrow l[i_{max}]$.make_child_leaves() ▷ create new children at leaf; get pointers
16: **for** t in $c[i_{max}]$ **do** ▷ split on most imp. feat.
17: **if** $t_s[j_{max}] = 0$ **then**
18: $c_{n_0} \leftarrow c_{n_0} \cup t$
19: **else**
20: $c_{n_1} \leftarrow c_{n_1} \cup t$
21: $m[i_{max}] \leftarrow [|I_f(c_{n_0})|$ **for** $f \in \{1, \ldots, |F|\}]$
22: $c[i_{max}] \leftarrow c_{n_0}$
23: $l[i_{max}] \leftarrow l_{n_0}$ ▷ store pointer to left child, replacing previous pointer
24: $m[|c|] \leftarrow [|I_f(c_{n_1})|$ **for** $f \in \{1, \ldots, |F|\}]$
25: $c[|c| + 1] \leftarrow c_{n_1}$
26: $l[|c| + 1] \leftarrow l_{n_1}$ ▷ store pointer to right child, extending list
27: **return** c, r ▷ return list of clusters and list of trees in forest

We call the outcome of this process an Abstract Policy Forest (APF) since each tree in the forest maps from a state s to an abstract state b; the abstract states are the same ones that appear within the corresponding Abstract Policy Graph.

Rather than recovering or discovering an APF after using APG-Gen, we modify APG-Gen to produce an APF at the same time as it produces an APG. The modified algorithm, APF+APG-Gen, differs only in the "Div_Abs_States" function. The modified function is shown in Algorithm 5; the portions in red are those that have been changed from APG-Gen.

Algorithm Explanation The core modification is maining a list of leaf pointers, l, alongside the list of clusters, c. The leaf pointers are references to nodes within the APF. The forest is initialized with one tree per action type in line 4. The initial leaf pointers consist only of root



(a) DTP (top) and APF (bottom) (b) UET produced by Extension (c) UET produced by Initialization

Figure 5.1: Starting with the DTP and APF in (a), our two approaches yield different UETs: (b) via Extension and (c) via Initialization.

nodes for the APF (line 5). As divisive clustering is performed in lines 11-26, the leaf pointers are updated in accordance with cluster partitions. Specifically, each cluster division leads to a new partition in the appropriate tree (line 15). Once the cluster is divided and the new clusters are stored (lines 22 and 25), the corresponding leaf pointers are set to reference these new leaves (lines 23 and 26, respectively). In addition to the final list of clusters, the output of the modified function now also includes a list of trees in the corresponding APF.

Computational Complexity Notably, the computational complexity of APF+APG-Gen is the same as for APG-Gen: $O(|F|^2|tr\_tuples|)$. The changes in Algorithm 5 consist of additional book-keeping whenever operations are already performed (i.e., a cluster is divided), while APG-Gen's complexity is dominated by computing feature importance (lines 10, 21, and 24) within a loop (line 11). Likewise, no additional assumptions are made and no additional constraints are added: only transition tuples and a policy are required; an alternative feature importance calculation may be used; and the agent may take any form.

5.2.2 Extending a DTP with an Abstract Policy Forest

Given a DTP and an APF, we can create a UET by appropriately linking together the trees. Note that each tree within an APF provides an s to b mapping only for states where the agent takes a specific action. Specifically, for each action a, there is a separate tree in the APF to provide the s to b mapping. In a DTP, each leaf node corresponds to a specific action choice by the agent; when the DTP is viewed as a partitioning of the state space, each leaf corresponds to a set of states where the same action is taken. Therefore, for all states within a given leaf node, the same APF tree would be used to identify the corresponding abstract states. This allows us to directly extend a DTP with APF trees by replacing each leaf node with the corresponding APF tree (i.e., if the DTP leaf corresponds to action a, then that leaf should be replaced with the APF tree for action a). This extended tree is then a valid UET.

Example Consider the example in Figure 5.1. A DTP for the example environment is shown at the top of Figure 5.1a: a state is mapped to one of two actions, blue or red. APF+APG-Gen

yields one tree per action, blue and red, as shown at the bottom of Figure 5.1a. Each of these trees maps a state *where the agent takes the appropriate action, blue or red,* to an abstract state. Since all states that fall into, e.g., a blue leaf of the DTP are valid inputs to the blue APF tree, each blue DTP leaf can be replaced with the blue APF tree. The same operation can be performed on the red DTP leaves using the red APF tree. This process yields the UET shown in Figure 5.1b.

5.2.3 Initializing APG-Gen with Leaves from a DTP

The Extension approach has the advantage of being post-hoc: any method of producing a DTP and an APF is compatible with this process. However, the resulting UET will necessarily exhibit recurring sub-structures (note the matching sub-trees within Figure 5.1b). This leads to two related disadvantages.

First, the UET may be larger than necessary. Each APF tree partitions the entire state space into abstract states, but each DTP leaf represents only a subset of the state space. As a result, some of the APF-derived partitions within the UET may be completely unnecessary; these partitions may be redundant with DTP-derived partitions closer to the root. Additionally, the best way for an APF tree to partition different portions of the state space may differ. By sharing a single APF tree, only a single, common structure can be found.

Second, the way that different portions of the UET are created is linked. This aspect reduces the interpretability of the tree: the partition created at a given node is based on a larger portion of the tree. As a result, the recursive nature of the tree is compromised (a subtree can be analyzed in isolation as its own tree) in addition to the local nature of its behavior (a single output can be analyzed in terms of the sequence of nodes traversed). Since we seek to later explain what influenced the creation of a UET, it is crucial that each partition depends only on the current node.

These disadvantages motivate us to more tightly couple the creation of the DTP and APF that are used to produce a UET. We further modify APF+APG-Gen to consider the leaves of the DTP. Specifically, we choose initial clusters for APF+APG-Gen based on leaf membership in the DTP, thus removing the relationship between states in different DTP leaves. This process allows a specialized tree for different regions of the state space even when the same action is taken; effectively, this approach leverages the partitioning performed by the policy tree.

For clarity, we refer to our resulting approach of creating an APF as APF+APG-Gen via Initialization; we refer to the base APF+APG-Gen method as using Extension. The difference between this variant and APF+APG-Gen is in the Div_Abs_States algorithm. The modified version is shown in Algorithm 6. The portions which have been modified or added (relative to Algorithm 5) are shown in red.

Example As an example, with the same DTP as shown in Figure 5.1a, APF+APG-Gen via Initialization would produce four trees within its APF (one per leaf in the DTP). These APF trees can then be joined with the DTP following the process outlined in Section 5.2.2, yielding the UET shown in Figure 5.1c. Note how though the left blue subtree and right blue

Algorithm 6 Compute abstract states based on transition samples and learned policy that is split into a mapping from states to leaves and a mapping from leaves to actions; simultaneously track performed partitions within a forest.

procedure DIV_DTP_ABS_STATES_WITH_FOREST($tr\_samples, s\_to\_l, l\_to\_a$)
  $leaf\_set \leftarrow \varnothing$
  $leaf\_to\_samples \leftarrow$ dict()
  for t in $tr\_samples$ **do**
    $t_l \leftarrow s\_to\_l(t_s)$
    if $l\_to\_a(l) \leftarrow t_a$ **then**
      $leaf\_set \leftarrow leaf\_set \cup t_l$
      $leaf\_to\_samples[t_l] \leftarrow leaf\_to\_samples[t_l] \cup t$
  $leaf\_list \leftarrow$ order($leaf\_set$)
  for i in $\{1, \ldots, |leaf\_list|\}$ **do**
    $c[i] \leftarrow leaf\_to\_samples[leaf\_list[i]]$ ▷ initialize with policy leaf membership
    $r[i] \leftarrow leaf\_list[i]$ ▷ initialize each tree in forest with policy leaf
    $l[i] \leftarrow r[i]$ ▷ entry in leaf list initially points to starting leaf node
  for i in $\{1, \ldots, c\}$ **do** ▷ pre-compute feat. imp.
    $m[i] \leftarrow [|I_f(c[i])|$ **for** $f \in \{1, \ldots, |F|\}]$
  while $\max_i \max_j (m[i][j]) > \epsilon$ **do**
    $i_{max} \leftarrow \text{argmax}_i \max_j (m[i][j])$
    $j_{max} \leftarrow \text{argmax}_j (m[i_{max}][j])$
    $c_{n_0}, c_{n_1} \leftarrow \varnothing$
    $l_{n_0}, l_{n_1} \leftarrow l[i_{max}]$.make_child_leaves() ▷ create new children at leaf; get pointers
    for t in $c[i_{max}]$ **do** ▷ split on most imp. feat.
      if $t_s[j_{max}] = 0$ **then**
        $c_{n_0} \leftarrow c_{n_0} \cup t$
      else
        $c_{n_1} \leftarrow c_{n_1} \cup t$
    $m[i_{max}] \leftarrow [|I_f(c_{n_0})|$ **for** $f \in \{1, \ldots, |F|\}]$
    $c[i_{max}] \leftarrow c_{n_0}$
    $l[i_{max}] \leftarrow l_{n_0}$ ▷ store pointer to left child, replacing previous pointer
    $m[|c|] \leftarrow [|I_f(c_{n_1})|$ **for** $f \in \{1, \ldots, |F|\}]$
    $c[|c| + 1] \leftarrow c_{n_1}$
    $l[|c| + 1] \leftarrow l_{n_1}$ ▷ store pointer to right child, extending list
  return $c, r, leaf\_list$ ▷ return lists of clusters, trees in forest, and input leaves

subtree correspond to the same DTP action, the subtrees have different structures. In particular, the right blue subtree is smaller in Figure 5.1c than the one in Figure 5.1b. Intuitively, since APF+APG-Gen via Initialization is partitioning a strictly smaller portion of the state-space, we would generally expect a smaller APF tree in this case.

Algorithm Explanation For this variation of the algorithm, the policy must be decomposable into a mapping from states to groups and a mapping from groups to actions. For our purposes, each group corresponds to states that fall into a specific leaf of a DTP. The algorithm provides a similar output as APF+APG-Gen: a list of clusters and a list of trees that constitute an APF. However, the APF contains one tree per leaf in the DTP rather than one tree per action. Thus, to pair APF trees with DTP leaves, the algorithm also returns the list of leaves found, in corresponding order (line 32).

The difference between the methods lies in how clusters are initially assigned (lines 2-13); the cluster division is identical (lines 14-31). First, the set of DTP leaves is identified (line 7), and transitions are grouped based on leaf membership (line 8). For compatibility with the rest of the algorithm, the DTP leaves must be put into an arbitrary order (line 9). As shown in lines 10-13, the initial clusters are then based on this leaf membership, with all states from the same DTP leaf starting within the same cluster.

Complexity and Assumptions Since the fundamental difference is only a change in initial grouping, the overall complexity of this version is identical to previous versions. The only additional assumption that is required is that the policy is a DTP or can be expressed as one. Likewise, the favorable properties remain. One notable benefit of this version is that the set of actions does not need to be iterated over. The requirement to iterate over actions limits unmodified APG-Gen to policies over discrete actions, but this variation is compatible with arbitrary action spaces.

5.3 Experiments

We demonstrate our method's ability to create concise UETs through combining CUSTARD DTPs and APF+APG-Gen's APFs. We have introduced two methods for creating a UET: via Extension and via Initialization. Another approach would be to create a decision tree over the trees in the APF and then sequentially use a DTP and the APF tree to mimic the output of a UET. We compare our two methods with this baseline approach in terms of explanation size (Section 5.3.1) and effectiveness of splits (Section 5.3.2). For these comparisons, we use PrereqWorld, as introduced in Section 2.2.1.

5.3.1 Explanation Size

Each of the approaches produces a tree as a final output. In general, a more concise tree is preferred: a smaller tree has fewer nodes to consider and fewer levels of recursion. In particular, one of the key motivations for our Initialization approach is to reduce the resulting tree size. We evaluate the size of the resulting UETs and baseline approach, and we measure the effectiveness of Initialization in reducing this size. For these experiments, we learn a DTP with CUSTARD, apply the appropriate APG-Gen variant, and then measure the tree size.

Note that, since APG-Gen repeatedly divides clusters until all features have low importance, all APF trees achieve the desired objective. This process means that a smaller tree cannot be caused by insufficient partitioning, which produces an unacceptable clustering of the transition

tuples. This property allows us to consider only tree size; in a different setting, thorough evaluation would also have to include measuring tree performance (e.g., in a classification setting, comparing both tree size and tree accuracy).

5.3.1.1 Metrics

We select average leaf depth as the key metric due to (1) its intuitive meaning and (2) how it arises in practical applications. If a leaf is chosen at random, the average leaf depth is the number of intermediate nodes that must be explained or considered while traversing from the root to that leaf. We also report the total number of nodes, though we argue that this is of less interest than the average leaf depth. In practice, traversing a DT (whether for prediction or explanation) is more common than considering the entirety of the tree at once, and the average leaf depth is already affected by the overall tree size. Finally, we report the tree height, which is effectively the worst-case numebr of nodes to consider while traversing from the root to any leaf.

A common metric in comparing explanations is the number and type of operations to perform common operations such as one inference (i.e., traversal from root to leaf). In our comparisons, all methods are decision trees that are restricted to partitions that compare a single feature to a single parameter. As a result, the operations are proportional to the average depth of the leaves. This metric is generally used for showing the benefits of a DT compared to a neural network, but only DTs exhibit the properties that we seek. We are comparing approaches to *find* a suitable DT rather than different explanation formats (e.g., DTs vs feature importance vectors). For this same reason, the total number of parameters or the "byte length" of an explanation (i.e., the amount of space required to store the parameters) are proportional to the number of nodes in each tree (which we already report).

5.3.1.2 Setup

We perform our evaluation on PrereqWorld, as described in Section 2.2.1. For this experiment, we select $m = 5$ and $\rho = 0$. We create the DTP using CUSTARD on top of Quantile Regression DQN since this is the variation that tends to perform best on PrereqWorld. We report results as an average across 15 trials. Because we are comparing APG-Gen variants and not DTP-producing methods, we use the same DTP for all methods within a trial. Likewise, within a trial, we use the same transition set across all evaluated methods. In this way, all methods are given the same starting point and additional variance is not introduced due to different method inputs.

5.3.1.3 Results

We report the tree height, average leaf depth, and total node count in Table 5.1.

In the baseline approach, a separate DTP and APF are used together. For the purposes of investigating all paths within the tree (i.e., understanding the entire explanation), one can treat the APF as appended to each leaf of the DTP– this is the "Stacking Baseline" in Table 5.1. If not investigating all paths but simply all nodes, then it may be possible to investigate the APF a

Method	Tree Height	Average Leaf Depth	Total Node Count
Stacking Baseline	8.13 (0.83)	5.82 (0.50)	105.33 (41.83)
Sequence Baseline	8.13 (0.83)	5.82 (0.50)	25.87 (5.37)
UET via Extension	4.73 (0.96)	3.84 (0.78)	22.07 (12.69)
UET via Initialization	4.07 (0.46)	3.23 (0.41)	14.07 (3.84)

Table 5.1: Comparison of tree sizes for different ways of combining a DTP and APF. The values in parentheses are Standard Deviation values.

single time rather than as part of each DTP leaf. In this case, the number of nodes to investigate is th sum of the DTP node count and APF node count. This is the "Sequence Baseline" in Table 5.1. Note that this baseline represents an optimistic case that may not be applicable for all use-cases, and we do not assume that our approaches benefit from such de-duplication of APF nodes.

Using Extension results in meaningfully shorter trees than directly stacking the trees; this is expected since stacking requires the addition of partitions that distinguish between APF trees while Extension leverages the fact that a DTP already performs this partitioning. Initialization with DTP leaves further reduces tree height by 0.66 nodes (14%). We believe this difference scales with the complexity of the environment since different portions of the state-space would yield APF trees that differ more, thus increasing the benefit from creating them separately.

We note that both the tree height and the average leaf depth decrease by a similar amount betwen Extension and Initialization. This suggests that using Initialization reduces leaf depth consistently across the entire tree rather than merely decreasing the length of the longest path.

Finally, a UET contains far fewer nodes overall (by at least a factor of four) than directly stacking a DTP and APF. Using Initialization in particular drops the number of nodes by a further 36%. This improvement indicates that a UET created using Initialization can more concisely present the same information and would therefore contain fewer unnecessary partitions. The intuition for this is that each leaf corresponds to a particular abstract state and the tree is merely a mechanism to group states into these abstract states; all the methods identically map states to abstract states, but any partitioning beyond that which is needed to group states appropriately adds unnecessary complexity without adding more information to the UET.

5.3.2 Split Effectiveness

In Section 5.3.1, we argued that a smaller tree performs the same function as a larger tree, so it removes unnecessary partitions or uses partitions in a more efficient manner. We further demonstrate the effectiveness of our methods by measuring the usefulness of splits. Specifically, we look at the average information gain within a combined DTP and APF to determine how well node partitions divide transitions based on their corresponding abstract state.

5.3.2.1 Metrics

We use average information gain as the metric for this experiment. The information gain within a node is the change in Shannon entropy due to a partition. This change is the difference between entropy of the entire transition set and weighted entropy of the subsets created by the node. The weights assigned to the subsets sum to one and are proportional to the number of transitions in each subset. We use Shannon entropy with the natural log. In cases where a node does not correspond to any transitions, then the information gain is taken to be zero.

Information gain is a common metric in evaluating the effectiveness of decision tree splits. In some use cases, it is used in selecting node partitions. This process effectively greedily maximizes information gain. We use abstract state membership for the class label required to compute entropy. This label is not available while constructing our UETs since abstract state membership is derived from the states found by APG-Gen after all partitioning has been performed. As a result, we cannot compare to methods that directly use information gain to build a decision tree.

For our Sequence Baseline results, the use of two sequential trees introduces complications. When computing entropy, using the whole transition set as input to the APF tree re-shuffles the transitions after the DTP has already separated them. As a result, repetition of partitioning between the DTP and APF tree would result in higher information gain–both trees would effectively get credit for the same reduction in entropy. To avoid double-counting information gain, we measure information gain with respect to the separate subsets found at each of the DTP's leaves. Though each APF tree node may process several different subsets, for purposes of computing the mean, we only count each APF tree node a single time to reflect the use of a single APF tree copy. This process leads to not penalizing the Sequence Baseline for using the same APF tree multiple times, but this property is at the expense of an optimistic average information gain for this baseline.

5.3.2.2 Setup

We follow the same setup as in Section 5.3.1. However, we use only 10 trials for these results. Note that these results are for a different set of trials, so the information gain measurements are for different trees than the previously reported heights, depths, and node counts.

5.3.2.3 Results

The average information gain for different methods is shown in Table 5.2.

Note that all of the methods partition the transitions into the same abstract states for the purposes of computing entropy. Each leaf is pure. As a result, one would expect trees with fewer nodes to have a higher average information gain. Our results match this intuition: ranking the methods from lowest to highest average information gain leads to the same order as ranking methods from highest to lowest node count.

The Stacking Baseline has a very low average information gain compared to the other approaches. This outcome is primarily due to the entire APF tree being appended to each DTP

Method	Mean Information Gain
Stacking Baseline	0.075 (0.014)
Sequence Baseline	0.286 (0.034)
UET via Extension	0.439 (0.127)
UET via Initialization	0.582 (0.079)

Table 5.2: Comparison of average information gain for different ways of combining a DTP and APF. The values in parentheses are Standard Deviation values.

leaf which leads to many APF tree nodes that are not used in practice. The Sequence Baseline attains higher average information gain, but it still performs substantially worse than our approaches. As noted previously, the APF tree used by this method must include additional nodes that distinguish between APF trees. In contrast, our approaches leverage the partitions made by the DTP to omit these nodes. Since these nodes provide no information gain, their inclusion meaningfully impacts the average information gain of this baseline.

Between our two approaches, Initialization performs better, as one would expect. We observed that Extension sometimes attains the same performance as Initialization. This outcome can occur when the trees found by APF+APG-Gen via Extension match all of the trees found by APF+APG-Gen via Initialization. We expect this to occur less frequently in more complex environments. As the environment size increases, the cases where different actions are chosen will differ more, so the trees are less likely to be equivalent. Unsurprisingly, there was no case where a UET via Extension had higher average information gain than its corresponding UET via Initialization.

5.4 Summary

In this chapter, we introduced Unified Explanation Trees (UETs). We presented two methods for creating a UET based on APG-Gen and the DTP learned via CUSTARD. As a prerequisite for these methods, we demonstrated how APG-Gen creates an implicit set of decision trees which can be readily extracted while performing APG-Gen's divisive clustering. First, we proposed a process to combine the trees into a UET. Next, we presented a modification to APG-Gen that integrated the DTP created by CUSTARD. Finally, we experimentally compared the resulting UETs to demonstrate the usefulness of the integrated variant.

Chapter 6

Importance and Influence in Neural Networks

We now address the third type of explanation in our thesis: identifying which past experiences are impactful with respect to the agent's current behavior. In this chapter, we introduce our approach for measuring experience importance in the case that the agent is using a neural network for value function approximation. This approach is based on Representer Point Selection (RPS), so our extended approach is Modified Representer Point Selection (MRPS). We develop our approach in a supervised learning setting so we can use standard benchmarks; in this setting, training points are used instead of experiences.

First, we identify the simplifications made by RPS: importance of a training point is computed independently of other points and of the feature values, and importance is computed based only on the final layer of a network. We then present remedies for these shortcomings by extending RPS in three distinct ways: computing importance for clusters of training points, normalizing feature values before computing importance, and measuring importance with respect to a greater portion of the network. We evaluate our proposed methods on a standard benchmark task and show that both clustering and normalization result in improved performance. In contrast, importance with respect to earlier layers does not meaningfully improve performance, so we argue that MRPS at the final layer is sufficient to explain a neural network.

In Chapter 7, we will then extend and apply MRPS to the structure introduced in Chapter 5. This extension will enable identifying impactful experiences at all levels of the UET. As a result, we will find impactful experiences with respect to both the overall action choice as well as intermediate decision tree partitions.

6.1 Motivation

Deep neural networks have revolutionized many technical and non-technical industries. These networks enabled human-level accuracy on tasks that were recently considered too difficult for AI. With the increasing availability of powerful toolkits and computation power, training a deep neural network has become a mundane task. However, deep learning systems can

exhibit unexpected behavior. Because of their non-interpretable nature, identifying the cause for specific behaviors requires additional methods. This motivates us to improve techniques for explaining deep networks.

Methods for explaining the predictions of a deep neural network have mostly focused on explaining the last layer or explaining the impact of different input features. Our goal is to develop a method that we will use in tandem with Decision Tree Policies. Decision Tree Policies already provide information about the impact of varying the input features, so we focus on the former technique: explaining the final layer of a neural network. In particular, we extend Representer Point Selection (RPS) [1] through three modifications:

- Computing importance in the context of the entire training set,
- Removing the sensitivity to parameter re-scaling, and
- Computing importance based on a larger portion of the network.

We compare the performance of these extensions to the baseline technique on a benchmark task. We find that the first two modifications result in large improvements in performance, whereas the third extension exhibits similar results to RPS. When these first two modifications are combined to form Modified RPS (MRPS), we achieve even better performance on the benchmark task.

6.2 Problem Formulation

Prior to introducing our approach, we briefly introduce the concepts upon which we build. First, we present the notation for the case of general deep learning (i.e., not necessarily RL). In Chapter 7, we will then introduce the RL-specific instatiation. We then define influence and importance. These closely relate to our overall objectives. Finally, we summarize Representer Point Selection, which we will extend in Section 6.3.

6.2.1 General Notation

Consider a function Φ parameterized by Θ with input x: $\Phi(x, \Theta)$. The output of Φ for a given x is the prediction for that x. Φ could be a neural network, Θ the weights of the network, and x an input to this network. In this chapter, we will focus on the case where Φ is a neural network predicting the class of each input image.

Θ is the result of some training procedure G, applied to (x, y) training pairs. In a deep learning classification setting, G would be the full learning procedure (gradient descent with a specific optimizer and hyperparameters) and the (x, y) pairs are training data input-label pairs.

The pairs are ordered in an arbitrary fashion and numbered sequentially. For simplicity, we treat G as a deterministic process. To distinguish Θ arising from different sets of pairs, we subscript Θ with the set of indices of the used pairs. For convenience, we use $[i..k]$ to mean $\{i, \ldots, k\}$ and $[k]$ to mean $\{1, \ldots, k\}$. Therefore, $\Theta_{[n]} := G((x_1, y_1), \ldots, (x_n, y_n))$, $\Theta_{[n-1]} :=$

$G((x_1, y_1), \ldots, (x_{n-1}, y_{n-1}))$, etc. Likewise, for simplicity, we use $[n] \setminus i$ to mean removing i from the set, leaving the set $\{1, \ldots i-1, i+1, \ldots, n\}$ (i.e., using $\setminus i$ rather than $\setminus\{i\}$ for brevity).

6.2.2 Influence

The *influence* of a training point (x_i, y_i) on x_j is the effect of including (x_i, y_i) on the prediction for x_j. This is equivalent to the change in prediction for x_j if (x_i, y_i) were added to the training pairs:

$$\text{INFL}_D((x_i, y_i), x_j) := \Phi(x_j, \Theta_D) - \Phi(x_j, \Theta_{D \setminus i}). \tag{6.1}$$

A subtle point to note is that while influence measures the impact of including a point, the counterfactual training set is one where (x_i, y_i) is *removed*. In other words, influence conveys the change in prediction had a given point *not* been included.

Note how the influence is defined with respect to a given set of training data, D. In practice, one seeks the influence with respect to some initial dataset. For simplicity, we will use a default D of $[n]$ and imply the $[n]$ for the sake of brevity:

$$\text{INFL}((x_i, y_i), x_j) := \Phi(x_j, \Theta_{[n]}) - \Phi(x_j, \Theta_{[n] \setminus i}). \tag{6.2}$$

The definition is analogous for sets of points. The influence of a set of points P on x_j is the effect of including the entire set on the prediction for x_j:

$$\text{INFL}(P, x_j) := \Phi(x_j, \Theta_{[n]}) - \Phi(x_j, \Theta_{[n] \setminus P}). \tag{6.3}$$

Note that this is equivalent to setting $P = (x_i, y_i)$; we slightly overload notation for brevity within this chapter.

Influence directly provides a way to explain the impact of a specific point or set of points on a specific prediction (i.e., on a local behavior). Additionally, this information is actionable in the sense that influence can be used to decide whether to remove specific points to obtain a desired change in behavior. With influence, the objective is to compute influence values in an efficient manner for different i and j.

However, exactly computing influence can be computationally expensive, even for a single point. This computation generally requires computing $G([n] \setminus i)$ from scratch (i.e., not based on the known $G([n])$ value). Even if a single $\Phi(x_j, \Theta_{[n] \setminus i})$ can be computed, influence provides a single value per x_j — an explanation of the broader impact of i would require applying Φ to many x_j.

6.2.3 Importance

Given the difficulty of measuring influence and the local (point-specific) nature of influence, an alternative is more appropriate in some use-cases. This difficulty motivates the use of importance, where an importance value, a scalar, is found for each training pair or a set of pairs, P. Unlike influence, the importance value does not depend on another x_j:

$$\text{IMP\_VAL}_D(P) \in \mathbb{R} \tag{6.4}$$

This independence from an x_j permits importance to serve as a global measure of a point's impact. However, again unlike influence, importance does not correspond to any specific relationship with G or Φ. Therefore, the objective in developing a way to compute importance is both to compute importance values efficiently and to find importance values which are empirically useful. This idea of usefulness is based on downstream use-cases. In this chapter, we focus on the standard use-case of identifying training points which (1) have incorrect y values and (2) maximally improve Φ's performance on the original task when labeled with their correct y values.

6.2.4 Representer Point Selection

Representer Point Selection (RPS) is a method for assigning an importance value to each point. Note that RPS is only able to find importance for individual points rather than for sets of points. RPS requires that Φ is a linear function of a featurized input. When applied to a deep network, RPS computes importance only for the final layer. The inputs to this final layer are the featurizations by the rest of the network. RPS effectively treats the featurization as fixed, so we treat Θ as only parameterizing Φ.

Thus, the overall network is split into Φ and Ω, where

$$\Phi(\Omega(x), \Theta) = \Theta\Omega(x).$$

Following [1], we define a featurized input as $\psi := \Omega(x)$. We use ψ rather than the f used in [1] to avoid confusion with our f features within a featurized state. We then use (ψ, y) pairs in an anlogous way to (x, y) pairs.

RPS requires that the training process minimize average loss L using squared L2 regularization:

$$\Theta_{[n]} = \arg\min_{\Theta} \left(\frac{1}{n} \sum_{i=1}^{n} L(\psi_i, y_i, \Theta) + \lambda ||\Theta||_2^2 \right). \tag{6.5}$$

When this is true (i.e., the gradient is effectively zero), the solution can be expressed as:

$$\Theta_{[n]} = \sum_{i=1}^{n} \frac{-1}{2\lambda n} \left(\frac{\partial L(\psi_i, y_i, \Theta)}{\partial \Phi(\psi_i, \Theta)} \right) \psi_i^T.$$

For each additive term, the non-ψ_i^T portion is termed α_i:

$$\alpha_i = \frac{-1}{2\lambda n} \left(\frac{\partial L(\psi_i, y_i, \Theta)}{\partial \Phi(\psi_i, \Theta)} \right). \tag{6.6}$$

The original function is then

$$\Phi(\psi_j, \Theta_{[n]}) = \sum_{i=1}^{n} \alpha_i \psi_i^T \psi_j.$$

From this, RPS obtains the scalar importance value and corresponding vector:

$$\text{IMP\_VAL}(\psi_i, y_i) = \alpha_i,$$

$$\text{IMP\_VEC}(\psi_i, y_i) = \alpha_i \psi_i^T.$$

The importance value corresponds to the quantity described previously. The importance vector is unique to RPS and stems from the relationship of α to the output of Φ: an ψ_j is effectively multiplied by a single vector to obtain a prediction, and this weight vector is a weighted sum of ψ_i^T vectors. The importance vector is then the contribution from a single training point to this weight vector.

6.3 Approach: MRPS

We present three potential extensions to RPS: find importance values for sets of similar sets of points rather than individual points, remove the sensitivity of importance to parameter rescaling, and find importance based on an earlier layer in the network. We describe our approaches in Sections 6.3.1, 6.3.2, and 6.3.3, respectively.

6.3.1 Reparameterize Last Layer in Terms of Exemplar Points

RPS is motivated by a use-case where a human is shown a list of training points ordered based on their importance values. The importance values are meant to help the human prioritize training points for review.

RPS identifies the training points with the greatest $|\alpha|$ values without considering the values nor embeddings of other training points. This approach has two shortcomings when there are training points with similar embeddings. First, introducing a copy of a point splits the original α value across both copies, so RPS reports the point as less important. However, the actual importance of all copies taken together is unchanged, so RPS will overlook important points if duplicates are introduced. Second, if there are multiple similar points with high $|\alpha|$, they are all identified as important without indicating a relationship between them. This results in an inefficient, homogeneous explanation.

In order to address these problems, we propose to find importance values for a set of points rather than for individual points. For any given set of points, we can create a hypothetical exemplar point for each class using a weighted average of all the points in the cluster, where weights are the original α importance of those:

$$c_{k,j} = \frac{\sum_{i \in C_k} \alpha_{i,j} \psi_i}{\sum_{i \in C_k} \alpha_{i,j}}. \tag{6.7}$$

We then assign an importance weight for each exemplar as

$$\alpha_{c_{k,j}} = \sum_{i \in C_k} \alpha_{i,j}. \tag{6.8}$$

Then we reparameterize the last layer in terms of these exemplar points:

$$\Theta = \sum_{k,j} \alpha_{c_{k,j}} c_{k,j}^T. \tag{6.9}$$

These sets can be selected in a number of ways. To address the first shortcoming, we can form sets out of x values that are identical or within some edit distance. We could also use the ψ representations for this, though it is possible for different inputs (e.g., images) to lead to similar embeddings. However, empirically, we find this does not happen. When there are multiple exact duplicate inputs, the exemplar point for all classes is then exactly the same as the duplicated input. This approach effectively recovers the original, non-duplicated case where a single point has its entire α value.

For the second shortcoming, we can create sets of similar points. The learned featurization function Ω is already trained to produce a useful featurization; Ω is meant to arrange the training points such that Φ, a linear function, can categorize all points into their respective classes. Thus, distance within this featurized space gives some notion of the similarity between points. We propose to cluster points using hierarchical clustering and treat each cluster as a set. This approach is computationally efficient and leverages the featurization that is already learned by Ω. With this method, the exemplar points are the cluster centers within the embedding space and capture the importance of the entire cluster.

Through our exemplar point importance values, we produce explanations that are more succinct and that better respond to duplicate points. Note that Equations 6.7 and 6.8 can be used with any set of points. Though we use local clusters for our experiments, any set of selected points can be assigned an exemplar point and a corresponding overall importance value. This ability is in contrast to standard RPS which produces importance values for one point at a time.

6.3.2 Remove Alpha Sensitivity to Weight Rescaling

RPS uses the computed α values to identify important training points based on the intuition that increasing the α for a training point would directly increase that training point's contribution to the network's output. However, a point contributes to the network weights through both α and its own feature values. Directly comparing the α values is not sufficient since the magnitude of the embedded training point features is not taken into account.

As an example, consider scaling the output of a specific neuron in the previous layer and the corresponding weights in the subsequent layer. This scaling effectively changes one component of the ψ vectors while keeping the network output unchanged. If this change is made, then the dot product of training point and testing point embeddings can change substantially. Since the output remains unchanged but the ψ vectors change, then the α values will also change. However, the importance vector for any data point need not change. In this way, the α values are sensitive to changes in weight rescaling in a way that undermines their utility as an indication of the actual impact a training point has on the learned function.

We propose to **normalize** the previous layer's activations so that the α values will be more representative. This approach is similar to using Batch Normalization [40], except no scale

nor shift parameter is learned; the normalization layer learns only to normalize the data to be mean zero and variance one. This normalization function, N, is directly applied on the learned featurization, leading to this architecture:

$$\Phi(N(\Omega(x)), \Theta) = \Theta N(\Omega(x)) = \Theta N(\psi).$$

In practice, RPS is applied by fixing Ω and then fine-tuning Θ to reduce the gradient of the loss, as shown in Equation 6.5, to effectively zero. Our approach can be applied within this procedure, without changing Ω or making any additional assumptions about it. After fixing Ω, the normalization parameters for N are learned. Then, Θ is computed as in standard RPS.

In the context of fine-tuning Θ after fixing Ω, our proposed normalization has an additional intuitive motivation. When treating the ψ values as fixed inputs (i.e., not affected by changing Θ), then Θ is obtained via Ridge Regression [41]. When using Ridge Regression, it is important to normalize the inputs so that each feature has mean zero and variance one. If this normalization is not performed, then the L2 penalty biases Θ away from the data and using some elements of ψ is unnecessarily penalized more. Our approach is exactly the normalization recommended when performing Ridge Regression.

6.3.3 Compute Importance based on Larger Portion of Network

RPS computes importance values using Equation 6.6. As a result, these values only reflect the relative contribution of the training points at the last layer of a network; the other layers are treated as part of a fixed Ω. We perform a similar reparameterization at the previous layer to obtain a second α vector for each training point:

$$\frac{1}{n}\sum_{i=1}^{n}\left(\frac{\partial L(\psi_i, y_i, \Theta)}{\partial \Phi(\psi_i, \Theta)} \cdot \frac{\partial \Phi(\psi_i, \Theta)}{\partial \Theta^{(2)}}\right) + 2\lambda\Theta^{(2)} = 0 \quad \Rightarrow \quad \Theta^{(2)} = \sum_{i=1}^{n} \alpha_i^{(2)} \psi_i^{(2)T} \tag{6.10}$$

where

$$\alpha_i^{(2)} = \alpha_i \Theta^T \sigma'(\Theta^{(2)} \psi_i^{(2)}). \tag{6.11}$$

Here, $\Theta^{(2)}$, $\alpha^{(2)}$, and $\psi^{(2)}$ are defined analogously to Θ, α, and ψ except they correspond to the next-to-last layer of the network.

Using all α values, it is possible to compute the contribution of a training point to the sequence of two layers (as opposed to just the layers individually). By including more layers in the importance weight calculation, one may expect to produce better explanations than RPS, which only uses importance at the last layer as an approximation for importance for the whole network.

6.4 Experiments

We focus on quantitative evaluations of RPS and our extensions of RPS. Since these are methods for producing explanations, there is no ground truth available to which to compare. Therefore, we evaluate all methods by using the produced explanations in downstream tasks. Though [1]

perform a few experiments using RPS, only one experiment is a quantitative comparison with an existing method: "Dataset Debugging." In this experiment, a network is trained on a corrupted dataset, and each method is evaluated based on its ability to identify potentially corrupted datapoints as well as its ability to improve learning performance. We use this same experiment to evaluate our proposed extensions and compare them to RPS and Influence Functions [42], the method to which RPS was compared in [1].

6.4.1 Baseline Approaches

We compare to RPS, as described in Section 6.2.4, as well as the method to which RPS was originally compared, Influence Functions [42].

The authors of [42] propose to use influence functions to identify the most important training points. The "most important" points are taken to be those whose removal would most change the model's predictions. Since repeatedly retraining a model with different training sets is infeasible, the influence of a point is approximated based on what would happen if the training point had slightly greater weight. This problem has a closed-form solution, but requires inverting the Hessian. By approximating the inverted Hessian via stochastic estimation, the authors can readily compute importance of all training points. The authors show that the approximations made are reasonable since the predicted change in model loss is similar to the actual change in loss.

To evaluate the method itself, the authors perform a series of experiments for different use cases (determining model behavior, detecting adversarial examples, identifying domain mismatches, and identifying mislabeled examples). This last experiment is the same one used by RPS. Before RPS, influence functions was the state of the art for finding the important training points for identifying mislabeled points.

6.4.2 Dataset Creation

As done in [1], we use part of the CIFAR-10 [43] dataset which corresponds to the images of horses or automobiles. A network is then trained to perform binary classification on this data. We closely follow the experimental methodology used to evaluate RPS in order to best compare our extensions to the base method; we use the same architectures, hyperparameter values, and dataset processing.

Note that we intentionally do not use a validation set in order to follow the experimental methodology used in [1]. If we were to use a validation set, then we would unfairly favor our method over RPS and influence functions, neither of which uses a validation set to tune hyperparameters. We use the same hyperparameter settings for all methods, following the ones provided by the authors of RPS.

6.4.3 Experiment Setup

The binary classification dataset is corrupted by flipping the class label of 40% of the training instances. An oracle is available for correctly re-labelling datapoints, but this oracle is only

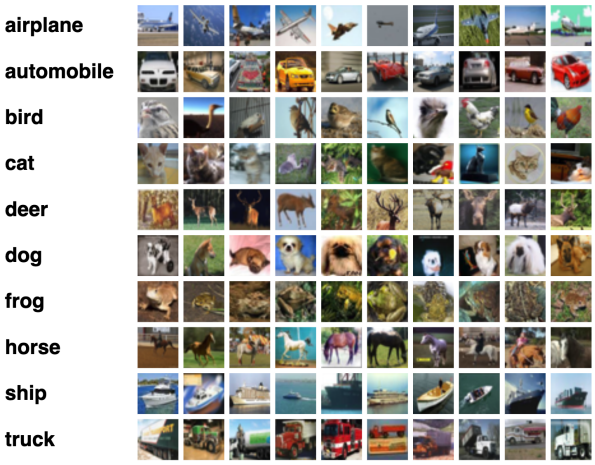


Figure 6.1: Sample data from the CIFAR-10 dataset, showing the ten classes, of which the benchmark task uses two.

permitted to label a limited number of points (n points). To select which points to label using the oracle, a network is trained on the corrupted data, and each explanation method is tasked with assigning a weight to each training point on the basis of this trained network. For a given value of n, the methods decide which n points to request from the oracle and how to re-label inputs based on the oracle's response. Then, the network is re-trained using the updated dataset. The resulting test accuracy and the fraction of identified corrupted input points are recorded. Intuitively, methods which improve testing accuracy and identify more corrupted points are potentially more useful for real-world dataset debugging.

In [1], n ranges from 5% of the training data to 40% of the training data, in increments of 5%. We evaluate using the same n values, but also evaluate for n equal to 45% of the training set. We do this because Figure 2 in [1] shows a change in behavior at 40% which may suggest a regime change for greater n values. Ultimately, we see only a smooth increase, but we still report results for that n value.

One of our proposed modifications produces an importance weight per exemplar point rather than for each training point. Specifically, we cluster training points in the embedding space into 2000 clusters using scipy's hierarchical clustering with Euclidean distance. We then create an exemplar point and importance value for each class for each cluster using Equations 6.7 and 6.8, respectively. To allow a fair comparison to RPS, this method only queries the oracle for these exemplar points (not with the whole set of points which that exemplar point represents). The total number of selected points is still limited to n. After the labels are obtained, each cluster associated with an exemplar point is relabelled based on the label assigned to the exemplar point. In practice, an exemplar point is not a member of the original training set and real-world oracles may be unable to label them, so we instead have the oracle label the point closest in embedding space to an exemplar point (as measured by Euclidean distance) rather than the exemplar point itself.

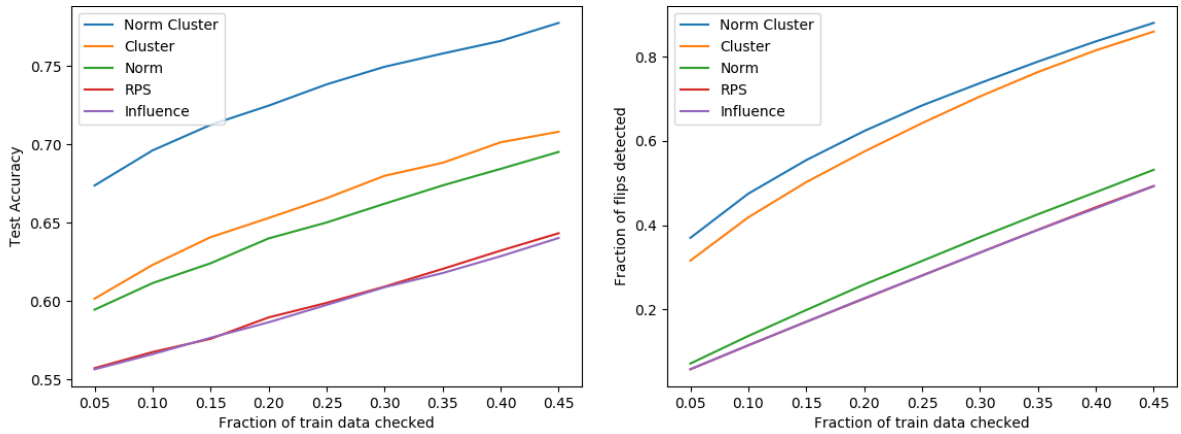


Figure 6.2: Comparison of our approaches to RPS and Influence Functions on the CIFAR-10 benchmark task. The left plot corresponds to success in the original task, and the right plot corresponds to the success in identifying mislabeled datapoints.

6.4.4 Evaluation Metrics

Following [1], we report the test accuracy and the portion of corrupted points found for each value of n. The test accuracy is the accuracy computed on a withheld test set with non-corrupted ground-truth labels. This computation is done with respect to predictions from a model trained on the corrupted training set after points have been labelled by the oracle. The proportion of corrupted points found is equal to the number of points which the method relabeled correctly after oracle was queried, divided by the total number of corrupted points in the training set.

6.4.5 Results

The results of 30 trials of the experiment are shown in Figure 6.2. Two of our proposed approaches (clustering and normalization) increase the test accuracy and the portion of corrupted points found. Moreover, when we combine the two techniques, the test accuracy increases even further. This shows that the two approaches address different types of issues found within the original RPS model. It is apparent that even though RPS beat the previous state-of-the-art method of influence functions, the improvements are tiny compared to the improvements we obtain by combining our clustering and normalization approaches.

There is a clear intuition why clustering improves performance. Figure 6.3 shows two examples of clusters with high total importance value (sum of α products of individual images). As we can see, all three images of the cars are very similar to each other. The original RPS method would query the oracle for a true label individually for all three images. However, it is sufficient to treat these three images as a single entity, since they are so close in the embedding space.

On the other hand, results obtained using our third approach of using $\alpha^{(2)}$ are less than 1% different from the original RPS results. This result suggests that using $\alpha^{(1)}$ values is a reasonable approximation of the overall importance of training points, so finding importance with respect to other layers is unnecessary.

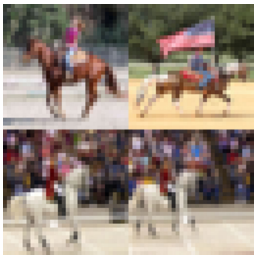


Figure 6.3: A pair of example clusters found using our method. The one on the left shows grouping of near-identical images while the one on the right shows grouping of a similar type of image.

6.5 Summary

In this chapter, we introduced three extensions to the state-of-the-art network explanation method RPS: computing importance for clusters of training points, normalizing feature values before computing importance, and measuring importance with respect to a greater portion of the network. We evaluated our proposed methods on the standard benchmark problem used for evaluating RPS and showed that both clustering and normalization resulted in huge boosts to performance. These results have motivated our incorporation of these techniques into Modified RPS, which we will subsequently incorporate into our unified explanation method.

Chapter 7

Importance and Influence for Unified Explanation Trees

In Chapter 5, we introduced our Unified Explanation Tree (UET), which merges a Decision Tree Policy obtained via CUSTARD (Chapter 3) and an Abstract Policy Forest obtained via a variant of APG-Gen (Chapter 4). In Chapter 6, we presented the concepts of experience importance and influence, and we introduced our method for computing importance: Modified Representer Point Selection (MRPS).

In this chapter, we demonstrate how to identify importance and influence for experiences with respect to nodes within a UET. This finishes the integration of the explanations considered in this thesis: a UET presents both single-action and policy-level behavior explanations in terms of state features by grouping states and identifying relevant features; computing experience importance at all levels of the tree explains single-action and policy-level behavior in terms of past experiences.

We first note that a UET consists of two node types. After training, both nodes perform the same operation (partitioning state space based on a feature value), but they originate from either CUSTARD or APG-Gen. We then introduce closed-form solutions for the importance and the influence for both of these node types. We show that these solutions achieve the desired objective of finding importance or influence, as appropriate.

7.1 Motivation

We seek an approach for identifying representative clusters of points for all nodes within the Unified Explanation Tree (UET). Extending MRPS to the UET provides a way for users to answer questions such as:

1. "Which experiences lead to this feature split?";
2. "Which experiences lead to this split value?"; and
3. "Which experiences lead to this action choice?"

If CUSTARD is used with a neural network for its value function approximation, then MRPS can be directly applied to that network. Though this does produce importance values with respect to policy choices, this approach is insufficient. First, it simultaneously finds importance based on the entire network and therefore explains all partitions at once rather than a specific partition. Second, it does not explain the partitions performed by APG-Gen.

We instead seek to find impactful experiences with respect to individual nodes in a UET. A UET has nodes from CUSTARD and from our proposed modification of APG-Gen; we must compute importance/influence differently in these two cases. An RL agent directly selects CUSTARD node parameters, and CUSTARD uses a single network to facilitate internal transfer. To explain individual nodes obtained via CUSTARD, we must disentangle the values of training points across different tree depths. Nodes obtained via APG-Gen are the result of applying a feature importance metric to Q-value estimates. To obtain importance and influence for such nodes, we must decompose the feature importance metric into individual contributions.

7.2 Problem Formulation

We had previously defined the problems of finding importance and influence for an entire neural network in a classification setting. We now seek importance and influence for our Unified Explanation Trees in a reinforcement learning setting. We describe the change from classification to reinforcement learning in Section 7.2.1. We then explain our objective and what it means to have importance and influence within a tree node in Section 7.2.2.

7.2.1 Notation for Reinforcement Learning Setting

We begin with the terminology introduced in Chapter 6. We now focus on the case where Φ represents a function learned by an RL agent: e.g., the Q-function, where the neural network predicts the future total discounted reward for an input state. In a reinforcement learning setting, G would be the full learning procedure (the learning algorithm as well as gradient descent with a specific optimizer and hyperparameters) and the (x, y) pairs are pairs of states along with the "target" value for that state (i.e., bootstrapped estimate for future total discounted reward).

An experience is a specific (s, a, r, s', d) tuple obtained from the environment. It corresponds to taking action a in state s, resulting in reward r and subsequent state s'. The boolean d indicates whether an episode has ended during this experience. An experience can be converted to a training point by setting x equal to s and obtaining a target value based on the tuple, as described in Section 2.1.2.4.

As in the classification setting, each input is featurized using a Ω function, so we use (ψ, t) pairs as inputs to G. The way we split a trained network into Φ and Ω effectively turns G into linear value function approximation with featurized states as input. This approach of treating a DRL agent as using linear value function approximation for the purposes of analysis is common.

The definitions for influence and importance are the same.

7.2.2 Influence/Importance at a Tree Node

A tree node partitions the input space based on a feature and a value to which that feature is compared. Within a UET, the choice of feature and value is based on some metric: potential feature/value pairs are assigned numeric scores and the highest-scoring pair is selected. Therefore, to explain why a given feature/value pair was chosen, we explain the numeric score assigned to this pair. We use the phrase "influence at a tree node" to mean "influence on the numeric score of the feature/value pair chosen at a tree node." We use "importance at a tree node" analogously. These influence and importance values are then node-dependent, and we seek a way to compute the influence and importance at a tree node of a set of experiences.

Though we focus on the importance and influence with respect to the selected feature/value pair, importance and influence can be calculated in the same way for other, non-selected feature/value pairs. Therefore, a method for computing importance or influence could be used to compare the current behavior within a node with a hypothetical alternative choice. For example, the influence of a transition with respect to a feature/value pair enables one to compute the numeric score of the feature/value pair if that transition were removed. If influence values are used to compute the numeric scores of the selected feature/value pair and an alternative pair, then one can determine whether the alternative pair would be selected if the transition were removed.

7.3 Approach: Influence for CUSTARD Nodes

We first look at nodes that arise from CUSTARD creating a DTP. First, in Section 7.3.1, we look at how Q-learning in general relates to the supervised problem which was considered in Chapter 6. In Section 7.3.2, we then discuss what the last layer of a network is learning when learning a Q-function; this is relevant since MRPS is applied to the last layer, and much of RL theory is based on a linear function approximator atop learned features. Then, in Section 7.3.3 we present how to disentangle the impact of experiences on different CUSTARD nodes and how to compute importance for these nodes. From here, we get a set of separate function approximations (one per node), so we can compute separate influences at each node. This then leads to our closed-form solutions for influence in Section 7.3.4.

7.3.1 Regression in Q-learning

When using a Q-learning-based method, we are effectively performing regression with squared loss. Q-learning methods use a set of (s, a, r, s', d) transition tuples. Each tuple corresponds to an experience where the agent was in state s, took action a, which yielded scalar reward r, led to state s', and may have ended the episode, as indicated by boolean d. These tuples are used to learn a Q-function $Q(s, a)$ that estimates the expected future discounted total reward after taking action a in state s. The optimal Q-function $Q^*(s, a)$ satisfies the Bellman equation

$$Q^*(s, a) = \mathbb{E}\left(r + d\gamma \max_{a'} Q^*(s', a')\right)$$

for all (s, a) pairs. The term inside the expectation is termed the *target* for a given (s, a, r, s', d) tuple. In practice, the Q-function is incrementally adjusted based on transition tuples so $Q(s, a)$ is closer to the target.

When using a neural network to represent the Q-function, this adjustment is done using some form of stochastic gradient descent along with squared loss regression on $Q(s, a)$, the current estimate, and the target. The loss is then:

$$L(s, a, r, s', d) := (r + d\gamma \max_{a'} (Q(s', a')) - Q(s, a))^2.$$

In practice, DRL methods learn a separate Q_a for each a (i.e., a is not an input). Additionally, the $Q(s', a')$ term is treated as fixed, in the sense that when computing the derivative of the loss with respect to Q, $Q(s', a')$ is treated as a constant. To make this clear, we define the target function T_Q

$$T_Q(r, s', d) := r + d\gamma \max_{a'} Q_{a'}(s').$$

The loss is then expressed as a function of a state s and a target t:

$$L(s, t) := (t - Q_a(s))^2. \tag{7.1}$$

During training, the loss is then used with a specific tuple by setting $t = T_Q(r, s', d)$.

7.3.2 Ordinary Least Squares and Ridge Regression

Though neural networks in RL generally consist of several sequential layers, they can be treated as performing linear value function approximation atop a learned feature extractor. Under this perspective, the final layer constitutes the value function approximator and the other layers form the feature extractor. This approach is akin to the one taken by RPS and, by extension, MRPS: Ω is a feature extractor and Φ is the function learned by the final layer.

7.3.2.1 Q-Learning as Ordinary Least-Squares

If we focus only on training the final layer of a model by minimizing Equation 7.1, we are effectively finding the ordinary least-squares solution (OLS):

$$OLS((\psi_1, t_1), \ldots, (\psi_n, t_n)) := \arg\min_{\beta} \left(\frac{1}{n} \sum_{i=1}^{n} (t_i - \beta\psi_i)^2 \right),$$

where ψ is a featurized state, $\psi_i := \Omega(s_i)$.

We use β here to distinguish from the Θ learned by RPS and MRPS. Note that the learned β is for a specific Q_a; it is action-specific. The tuples provided as input must therefore share this same action a.

7.3.2.2 RPS-Compatible Q-Learning as Ridge Regression

RPS and MRPS require squared L2 norm regularization in order to perform their reparameterization. If we use this regularization to allow the use of RPS and MRPS, then we are performing Ridge Regression (RR) [41]:

$$RR((\psi_1, y_1), \ldots, (\psi_n, y_n)) := \arg\min_{\Theta} \left(\frac{1}{n} \sum_{i=1}^{n} (y_i - \Theta\psi_i)^2 + \lambda_{[n]} ||\Theta||_2^2 \right).$$

We use $\lambda_{[n]}$ to indicate that the regularization term is generally chosen based on the dataset rather than some fixed value.

The closed-form solution for RR is then

$$RR((\psi_1, y_1), \ldots, (\psi_n, y_n)) := (\Psi_{[n]}^T \Psi_{[n]} + \lambda_{[n]} I_{|\psi|})^{-1} \Psi_{[n]}^T Y_{[n]}, \tag{7.2}$$

where $\Psi_{[n]}$ is the matrix formed by stacking $\psi_1, \ldots, \psi_n$, and $Y_{[n]}$ is the vector formed by stacking $y_1, \ldots, y_n$, and $I_{|\psi|}$ is the $|\psi| \times |\psi|$ identity matrix, where $|\psi|$ is the number of features in each featurized state.

This optimization problem is a member of the class supported by RPS: the average loss is being minimized, Φ is a linear function, and L2 regularization is used.

Selecting Lambda We choose $\lambda_{[n]} = n \times \lambda_c$, where λ_c is a fixed regularization term, independent of the set $[n]$. We scale by n based on two motivations. First, when using deep learning in practice, the weight-decay and similar regularization terms are generally not modified from their default values. Since the regularization is applied during each mini-batch, the number of minibatches dictates the degree of regularization. The mini-batch size is generally kept constant, so more mini-batches are used when there are more training points, which leads to regularization proportional to the number of points. As a result, scaling λ_c proportional to n in our non-batch setting is closer to how regularization is used in deep learning where batches are almost always used. Second, if λ_c were not scaled, then its contribution would decrease as n increases. For example, if each training point were duplicated, then the closed-form solution would change though the distribution of points is identical. By using $n \times \lambda_c$, the closed-form solution is invariant to duplication of the dataset.

7.3.3 Explaining CUSTARD

When explaining a DRL policy in terms of transitions, one may want to know "which past transitions contributed to choosing this current action?" In addition to identifying relevant transitions for a selection of a base action, we can perform a similar analysis on each node of the DTP– i.e., determine why the agent chose this type of node and the current parameters.

We first summarize the key aspects of CUSTARD in Section 7.3.3.1. We then describe how we separate the Q-value estimates learned across different nodes within a DTP in Section 7.3.3.2 Finally, we introduce our approach for finding importance with respect to any node and discuss the shortcomings with importance in this context in Section 7.3.3.3.

7.3.3.1 Relevant CUSTARD Properties

As discussed in Chapter 3, CUSTARD learns a DTP for a base MDP through a three-stage process:

1. "wrapping" the base MDP to form an IBMDP,
2. solving the IBMDP with a modified DRL method, and
3. extracting a DTP for the base MDP from the learned IBMDP policy.

A state s_w in the IBMDP consists of a state in the base MDP, s_b, and additional, IBMDP-specific features. Actions in the IBDMP are either base actions (a_b, those from the base MDP) or splitting actions that correspond to gathering more information as in an internal node within a DT.

Since the splitting actions do not change s_b, an agent's interaction with an IBMDP can be factored into an information-gathering step (where the agent performs splitting actions) and an interaction with the base environment (where the agent selects a base action and obtains an updated base state). The learned IBMDP policy, π_w, is forced to be invariant to s_b as part of step 2 by removing s_b from π_w's input. This policy then corresponds to a DTP with only $s_w \setminus s_b$ as input, where the internal nodes correspond to the information-gathering and each leaf corresponds to an interaction with the base environment.

7.3.3.2 Disentanging Q-value Estimates

We seek to explain individual CUSTARD nodes in terms of past experiences. As mentioned previously, this explanation process corresponds to identifying past experiences that have high importance or influence with respect to the feature and partition value chosen for a given node. The choice of feature and partition value is based on a Q-value estimate made by CUSTARD.

However, CUSTARD is designed to use a single neural network for all of its nodes. This design choice permits generalization across nodes, but it also causes all nodes to be related. As a result, if MRPS is directly applied to explain a node, then all past experiences are potentially impactful, not just the ones that correspond to that node. This entangling of nodes reduces the utility of a decision tree structure, which typically enables explaining a sequence of nodes while disregarding non-visited nodes.

We first note that CUSTARD with a tabular representation for its Q-value estimates does not have this entangling complication. If each node had separate Q-values, then updates to one node would have no effect on the others. Likewise, the Q-value estimate for a node would only be based on transitions that correspond to that node.

We then note that CUSTARD's single neural network for all nodes is useful only during training. In other settings, a neural network may be needed to generalize beyond the states encountered during training. However, when converting an IBMDP policy to a DTP for the base environment, CUSTARD already iterates over all nodes in the final DTP. As a result, once the DTP is found, the single neural network can be replaced by a modified network for each

node. In particular, we propose to keep the Ω portion of each network and treat it as fixed. Then, the final layer is fine-tuned separately for each node, effectively learning a separate Φ for each node. This process ensures that each node's Φ can be separately explained; each node's Q-value estimates are then based only on transitions that correspond to itself.

Changes During Fine-Tuning During this fine-tuning process, the DTP may change; the feature and partition value with the highest Q-value may change. One might expect such a change to disrupt our process and thus require a work-around. However, if such a change occurs, then we can readily alter the DTP in line with this change and create the new subtree following this same system. Such a change is automatically handled by fine-tuning nodes starting from the root. This approach ensures that any potential changes to the DTP do not invalidate fine-tuning that has already been performed.

Extra Weights This change can be made partway through learning, which allows generalization for faster training and subsequent independence. Only the final, linear layer is duplicated while the learned embedding Φ can still be shared. The increase in number of parameters is generally low, as only the linear weights are duplicated per node in the DTP, which itself generally has few nodes. Specifically, the number of new weights is $g \times |\psi|$, where g is the number of nodes in the DTP. In the worst case, the DTP has one leaf per past experience, so $g = 2n$. In this case, storing the past experiences requires $n \times |\psi|$ space, so the new weights can never require more than twice the space to store than the set of past experiences.

Extra Computation The fine-tuning is also quick to perform, unlike fine-tuning a single neural network for all nodes. A more thorough explanation will be given in Section 7.3.4.1, but the core idea is that each node corresponds to a specific $s_w \setminus s_b$ input. Thus, not only can we use the closed-form RR solution, but we can also use a special case where the time complexity is linear in the number of inputs and linear in the number of features. Furthermore, though each s_b corresponds to a number of nodes within a DTP (i.e., all the nodes along the path from the root to the corresponding leaf), each node can leverage the computation performed within its children. Thus, rather than needing $O(hn + h|\psi|)$ time, where h is the height of the tree, only $O(n + h|\psi|)$ time is needed. In practice, the time comlexity is $O(n)$, since, typically, $h < log_2(n)$ and $|\psi| < log_2(n)$.

7.3.3.3 Importance Computation and Importance Shortcomings

Using CUSTARD with a RR penalty and separating the parameters used for partitioning different nodes produces a problem compatible with RPS and MRPS. The MRPS reparameterization of the final layer can be directly used to produce a closed-form solution for importance:

$$\alpha_i = \frac{1}{\lambda_{[n]} n} \left(t_i - \Theta \psi_i \right). \tag{7.3}$$

As with importance in other cases, this solution provides a sense of the relative impact of different tuples encountered during training. As in Chapter 6, the α values for multiple points

can be combined to find the overall influence of the set of points. Note that the disentangling we previously performed ensures that all past experiences that correspond to a given CUSTARD node share the same ψ value. Therefore, the importance values are effectively a scaled error: a past experience where the target value differs more from the learned estimate will have higher importance. This relationship between error rate and importance matches intuition.

However, importance values do not have the same functionality as influence. They cannot be used to find the hypothetical change in learned weights and change in estimates should the set of past experiences be changed. The influence of a transition on the Q-function corresponds to the exact change in the Q-estimate if that transition were removed/added. Thus, influence values allow us to predict changes in the DTP by means of predicting changes in the Q-function. Effectively, the influence value is sufficient to determine whether and what impact adding/removing a transition would have on the DTP itself.

7.3.4 Computing Exact Influence

Ultimately, we seek a closed-form solution for influence for CUSTARD nodes. Ideally, this solution would contain separate components for different points that are added and removed. These components could be computed separately, and the decomposition would allow direct identification of the most and least influential sets of points.

To obtain such a solution, we first note that the RR solution has a convenient form in our case (Section 7.3.4.1). Then, we note the relationship of this solution to the prediction made at a node (Section 7.3.4.2). The mean target value affects the optimal RR estimator and, in turn, the Q-value estimate within a CUSTARD node. (Section 7.3.4.3). We combine all these to get the influence on the estimate (Section 7.3.4.4) and the change in learned parameters (Section 7.3.4.5).

7.3.4.1 Ridge Regression with Identical Inputs

We start by leveraging the closed-form solution for RR, Equation 7.2, to get Lemma 1.

When all inputs are the same, we can replace the matrix multiplication with just vector multiplication of a single input:

$$\begin{aligned} RR_{ss}(\psi_0, (y_1, \ldots, y_n)) &:= RR((\psi_0, y_1), \ldots, (\psi_0, y_n)) \\ &= (\Psi_{[n]}^T \Psi_{[n]} + \lambda I)^{-1} \Psi_{[n]}^T \times \bar{y}_{[n]} \\ &= (n \times \psi_0^T \psi_0 + \lambda I)^{-1} (n \times \psi_0^T) \times \bar{y}_{[n]} \\ &= \frac{1}{||\psi_0||_2^2 + (\lambda/n)} \psi_0^T \times \bar{y}_{[n]}, \end{aligned} \tag{7.4}$$

where $\bar{y}_{[n]}$ is the mean y value, $\frac{1}{n} \sum_{i=1}^{n} y_i$.

Computational Complexity This alternate form is favorable in two ways. First, the original matrix multiplication and inversion has polynomial time complexity in $|\psi|$ and linear in n (at least $O(n|\psi|^2)$). In contrast, Equation 7.4 takes time linear in $|\psi|$ for the ψ_0 portion and linear in n for $\bar{y}_{[n]}$, leading to an overall complexity of $O(n + |\psi|)$.

When applied to our setting with CUSTARD, we get Lemma 1.

Lemma 1. *For a set of transitions with identical state representations,* $\{(\psi_0, t_1) \ldots, (\psi_0, t_n)\}$*, the optimal Ridge Regression estimator is*

$$RR_{ss}(\psi_0, (t_1, \ldots, t_n)) := RR((\psi_0, t_1), \ldots, (\psi_0, t_n)) = \frac{1}{||\psi_0||_2^2 + \lambda_c} \psi_0^T \times \bar{t}_{[n]}, \tag{7.5}$$

where $\bar{t}_{[n]} = \frac{1}{n} \sum_{i=1}^{n} t_i$*.*

Computational Complexity In addition to the base time complexity of $O(n + |\psi|)$, we can leverage the $\bar{t}$ computations across different nodes to further decrease the time complexity within the entire tree. As mentioned previously, when performing this computation in all nodes of the DTP, we can compute $\bar{t}$ only at the leaves and then use a weighted average of the children's nodes at each inner node. This approach leads to an overall complexity of $O(n+h|\psi|)$ rather than $O(hn + h|\psi|)$, where h is the height of the tree.

7.3.4.2 Q-value Estimates within CUSTARD

Since we successfully separated the estimates produced by the different CUSTARD nodes, we get a separate RR solution for each node. This gives us Lemma 2.

Lemma 2. *For a CUSTARD DTP node* N *(created using Q-learning with an L2 penalty) and any state within that node* s'_w*, the Q-value estimate for an action* a *is*

$$Q_{a,N}(s'_w) := q_{a,N} = RR_{ss}(\psi_{w \setminus b,N}, (t_1, \ldots, t_n)) \times \psi_{w \setminus b,N}, \tag{7.6}$$

where $\psi_{w \setminus b,N}$ *is a node-specific assignment of bounding features.*

7.3.4.3 Influence on Mean Target Value

As an intermediate to the influence of training points on a Q-value estimate, we get the influence on the target values. Algebraic manipulation yields Lemma 3. Here, we introduce E, which can be considered the error of estimating a given target value by using the mean target value.

Lemma 3. *When removing transitions with indices Rem and adding transitions with indices Add,*

$$\bar{t}_{[n]} - \bar{t}_{[n] \setminus Rem \cup Add} = \frac{\sum_{j \in Rem} E_{[n]}(t_j) - \sum_{k \in Add} E_{[n]}(t_k)}{n - |Rem| + |Add|}, \tag{7.7}$$

where

$$E_{[n]}(t_j) = t_j - \frac{1}{n} \sum_{i=1}^{n} t_i. \tag{7.8}$$

7.3.4.4 Influence on CUSTARD Q-Value Estimate

Taken together, these then yield a closed-form solution for influence for a CUSTARD node, presented in Theorem 1.

Theorem 1. *The change in action Q-value estimate of a CUSTARD DTP node N on any state s'_w within the node, $Q_{a,N}(s'_w)$, when removing the points with indices in Rem and adding the points with indices in Add is*

$$\Delta Q_{a,N}(s'_w) = \frac{1}{1 + \lambda_c / ||\psi_{w \setminus b,N}||_2^2} \times \frac{\sum_{j \in Rem} E_{[n]}(t_j) - \sum_{k \in Add} E_{[n]}(t_k)}{n - |Rem| + |Add|} \tag{7.9}$$

where $E_{[n]}(t_j)$ is as defined in Equation 7.8, when s_b is within the bounds set by $s_{w,N}$ and is 0 otherwise.

This solution has the desirable property of largely independent contributions by different points with indices in Rem and Add. Therefore, given a constraint on |Rem| and |Add|, it is straightforward to find the largest potential change.

Informal Proof The CUSTARD DTP is directly extracted from the Q-function learned by CUSTARD. CUSTARD's Q-function is deliberately not shown the base state portion of s_w. The remaining features of s_w uniquely identify a potential node within CUSTARD's DTP, though not all nodes are present within a given learned DTP. As a result, each $Q_a(s_w)$ is a Q-value estimate within a specific node.

Since $Q_a(s_w)$ is deliberately only a function of $s_w \setminus s_b$ and this portion of s_w is the same for all s_w within a node, $Q_a(s_w)$ outputs the same value for all applicable s_w. The prediction made by the OLS solution in this instance is $\bar{t}_{[n]} := \frac{1}{n} \sum_{i=1}^{n} t_i$, the mean target value for all input points. The RR solution is the scaled OLS solution (Lemma 1). Since the CUSTARD Q-value estimate is tied to the RR solution (Lemma 2), the influence on the CUSTARD Q-value estimate is the scaled influence on the OLS prediction (Lemma 3).

7.3.4.5 Change in CUSTARD Weight Vector

In addition to the influence on the CUSTARD Q-value, we can also find the exact change in the weights of the final layer.

Theorem 2. *The change in the weights at the final layer of a CUSTARD DTP node N when a set of transition tuples with indices in Rem is removed and a set of transition tuples Add is added is*

$$\Delta\Theta = \frac{1}{||\psi_{w \setminus b,N}||_2^2 + \lambda_c} \psi_{w \setminus b,N}^T \times \frac{\sum_{j \in Rem} E_{[n]}(t_j) - \sum_{k \in Add} E_{[n]}(t_k)}{n - |Rem| + |Add|}, \tag{7.10}$$

where $E_{[n]}(t_j)$ is as defined in Equation 7.8.

Informal Proof Without loss of generality, consider removing the pair (s_n, t_n).

By the definition of influence (Equation 6.1) and since the final layer is a linear function of its featurized input, the change in weights is tied to the influence in Theorem 1:

$$\text{INFL}((\psi_n, t_n), \psi_j) = (\Theta_{[n]} - \Theta_{[n-1]}) \times \psi_j. \tag{7.11}$$

Since CUSTARD DTP node influence is identical for all states that fall within the node, the ψ_j term can be readily removed from the left-hand side.

7.4 Approach: Influence for APG-Gen Nodes

In the case of CUSTARD-created nodes, Q-values are derived from a network, so we can directly apply RPS and MRPS. For the nodes appended to a DTP by APG-Gen, feature partitions are based on feature importance. This feature importance is a function of Q-network outputs, so we can combine an MRPS decomposition of the Q-network with additional MRPS-style importance decompositions of these feature importance values.

In Section 7.4.1, we first note the relationship between feature importance and the decomposition found by RPS and MRPS. We use this relationship to compute an MRPS-style importance. We then present three methods in Sections 7.4.2-7.4.4 for computing influence. We introduce our methods in order of decreasing approximation, so latter methods account for more interdependencies at the expense of more computation. Finally, in Section 7.4.5, we present an exact computation of influence that does not rely on an MRPS decomposition. As described in Chapter 6, influence provides the exact change if a given training experience is removed, so this computation is not an approximation in any way.

Method Summarization We briefly summarize the differences between our methods in Table 7.1. Rem is the set of indices of past experiences to be removed, Add is the set of indices of past experiences to be added, and n is the number of past experiences currently used by the agent.

Note how the first three methods do not closely depend on Rem nor Add. These methods can first compute the contributions of individual points and then Rem and Add can be chosen. However, these methods only provide importance or approximate influence. In contrast, the last two methods provide exact influence, but this capability requires selecting Rem and Add before meaningful computation can be performed.

If computing importance or influence for a specific Rem and Add, then all methods take time linear in n. However, if several different sets are evaluated, then the first three methods can cache intermediate computations. As a result, the time to perform further influence computations with those methods does not depend on n. The final two methods always take time linear in n for any given Rem and Add.

These tradeoffs between the methods motivate our inclusion of both types.

Section	Method	"Retrain" with Rem and Add?	Time complexity in n (overall)	Time complexity in n (per pair)	Time Complexity in |Rem| and |Add|
7.4.1	MRPS-Style Importance	No	Linear	Constant	Linear
7.4.2	Initial Influence Estimate	No	Linear	Constant	Linear
7.4.3	Influence with Fixed α Values	No	Linear	Constant	Linear
7.4.4	Influence via α Value Changes	Yes	Linear	Linear	Linear
7.4.5	Influence without α values	Yes	Linear	Linear	Linear

Table 7.1: Properties of different methods for computing importance or influence for UET nodes obtained via an APG-Gen variant.

7.4.1 Feature Importance and MRPS Decomposition

As noted in Chapter 4, APG-Gen is compatible with any feature importance method. We choose FIRM for computing feature importance due to its performance, intuitive meaning, and simple computation. For the same reasons, we focus on the case of binary features: with binary features, exact computation is faster and the full effect of the feature is captured within the importance value.

7.4.1.1 FIRM within APG-Gen

FIRM's importance measure of a feature f on an arbitrary function g with respect to a set of inputs c is

$$\begin{aligned} I_f(c) &= (q_{f0}(c) - q_{f1}(c))\sqrt{p_{f0}(c)p_{f1}(c)}, \\ p_{fv}(c) &= \mathbb{P}(s[f] = v), \\ q_{fv}(c) &= \mathbb{E}(g(s)|s[f] = v). \end{aligned} \tag{7.12}$$

The $p_{fv}(c)$ values correspond to an empirical estimate of the probability that the feature f takes on value v based on the input set c. The $q_{fv}(c)$ values correspond to the expected value of the function for which importance is being computed for inputs in c that match the constraint set by v.

In our case, we seek the importance of transition tuples with respect to the learned Q-function ($g = Q$). As a result, we treat the $p_{fv}(c)$ values as fixed: slight changes in the distribution of encountered states have a limited impact compared to the difference in Q-value estimates between different states. Thus, if we find the influence on both $q_{f0}(c)$ and $q_{f1}(c)$, then their difference is the influence on $I_f(c)$.

The two q quantities are computed in the same way, just over different sets of states (those in c with $f = 0$ and those in c with $f = 1$). Without loss of generality, we consider how q_{f0} is computed. We set n as the number of states that satisfy the feature-value filtering and number these states from 1 to n. The value of $q_{f0}(c)$ is based on the Q-value estimates learned by the agent:

$$q_{f0}(c) = \frac{1}{n}\sum_{i\in[n]} Q(s_i). \tag{7.13}$$

Note that we can use Q_o (see Section 3.2.3.3), since APG-Gen just needs an arbitrary way to compute importance. Using Q_b is not strictly necessary. By using Q_o, we can use the maximum information about future outcome in determining feature importance. Since APG-Gen is specifically for conveying potential future outcomes, this choice improves APG-Gen's ability to achieve its primary objective.

7.4.1.2 MRPS-Style Importance for FIRM

Since Q is obtained from a network compatible with MRPS, we can perform an MRPS decomposition, where ψ_i is the featurized state s_i, $\Phi(s_i)$. To obtain the impact of a transition

ψ_z, we seek the contribution of the zth point to the final $q_{f0}(c)$ value. With some algebraic manipulation, we separate $q_{f0}(c)$ into three terms:

$$q_{f0}(c) = \frac{1}{n}\left(\sum_{j\neq z}\left(\left(\sum_{i\neq z}\alpha_i\psi_i^T\right)\psi_j\right) + \sum_{j\neq z}\left((\alpha_j+\alpha_z)\psi_j^T\psi_z\right) + \alpha_z\psi_z^T\psi_z\right). \tag{7.14}$$

The last two terms are those where α_z plays a role, so we can follow the approach of RPS and get an importance value for ψ_z with respect to $q_{f0}(c)$:

$$\frac{1}{n}Q(s_z) + \frac{1}{n}\alpha_z\psi_z^T\sum_{j\neq z}\psi_j. \tag{7.15}$$

Intuitively, this quantity captures the direct contribution of s_z via the $\frac{1}{n}Q(s_z)$ term and then the contribution of s_z's α term. The summation can be efficiently performed for multiple different s_z by finding the overall sum of all ψ and then subtracting ψ_z as needed.

Though this process yields an importance value, the approach leans heavily on the MRPS decomposition of Θ and does not provide an influence value. Therefore, we now take a different approach, which also has the benefit of clearer assumptions.

7.4.2 Initial Influence Estimate

Whenever the set of data used to train a network changes, the predictions made by this network will generally slightly change on all datapoints. However, when removing or adding only a small amount of data, these prediction changes can be quite small. By assuming that our learned Q does not change through adding or removing points, we can obtain a computationally simple influence estimate.

Lemma 4. *If Q is assumed to not change as a result of adding or removing points ($\Theta_{[n]} = \Theta_{[n]\setminus Rem\cup Add}$), then the change in $q_{f0}(c)$ when removing points with indices in Rem and adding points with indices in Add is:*

$$\frac{1}{n-|Rem|+|Add|}\left(\sum_{j\in Rem}\left(Q(s_j) - \frac{\sum_{i\in[n]}Q(s_i)}{n}\right) - \sum_{k\in Add}\left(Q(s_k) - \frac{\sum_{i\in[n]}Q(s_i)}{n}\right)\right). \tag{7.16}$$

Note how the contribution of each individual point with index from Rem or Add does not depend on other points. With this estimate, we can perform procedures that are generally difficult to perform with influence. For example, we can maximize the change in $q_{f0}(c)$ subject to a limit on how many points we can add or remove. By first computing the potential individual contributions, we can sort the points by these values and select the ones that would change $q_{f0}(c)$ the most.

Assuming a fixed Q is effectively reducing the link between the data used for APG-Gen and the data used to learn Q. By making this assumption, we compute the effect on the APG we find but not the effect on the underlying agent. As a result, this assumption is correct (i.e., does

not lead to an approximation) when seeking to explain only the APG itself. For example, if one uses a different dataset to train the agent and to create the APG, then changing the latter dataset will indeed not change the agent's Q.

7.4.3 Influence while Accounting for Change in Q

The previous influence estimate relied on a fixed Q. If the same data is used for both learning Q and constructing the APG, then removing or adding points can change Q. To account for this, we can use the MRPS decomposition of Q to find the effect on Q.

7.4.3.1 Old and New Q via Alpha Values

We now use Q^0 to signify the original Q, before points are added or removed. Q^1 is then the new Q, which is the Q obtained by training on the set $[n] \setminus \text{Rem} \cup \text{Add}$. The distinction between Q^0 and Q^1 prevents the cancellation of some Q terms used to obtain Lemma 4.

Lemma 5. *If Q^0 is the Q obtained using dataset $[n]$ and Q^1 is the Q obtained using $[n] \setminus Rem \cup Add$, then the change in $q_{f0}(c)$ when removing points with indices in Rem and adding points with indices in Add is:*

$$\textit{init\_est} + \frac{1}{n - |\textit{Rem}| + |\textit{Add}|} \sum_{i \in [n] \setminus \textit{Rem} \cup \textit{Add}} \left(Q^0(s_i) - Q^1(s_i) \right), \tag{7.17}$$

where init_est is the original estimate from Lemma 4.

Equation 7.17 provides the exact influence without any assumptions or approximations, but it requires obtaining the updated Q. In practice, this means retraining a Q-network, which we seek to avoid.

The term within the summation is related to the influence of points with indices in Rem and Add on Q. We can use the MRPS decomposition of Q^0 and Q^1 to express that term as a sum of individual point contributions:

$$Q^0(s) - Q^1(s) = \left(\sum_{i \in [n]} ((\alpha_i - \alpha_i')\psi_i) + \sum_{j \in \text{Rem}} \left(\alpha_j' \psi_j \right) - \sum_{k \in \text{Add}} (\alpha_k' \psi_k) \right)^T \psi, \tag{7.18}$$

where α_i is the α obtained from Q^0 and α_i' is the α obtained from Q^1.

Using this decomposition, we obtain a new equation for the change in $q_{f0}(c)$:

$$\text{init\_est} + \frac{1}{n - |\text{Rem}| + |\text{Add}|} \times \left(\sum_{i \in [n]} ((\alpha_i - \alpha_i')\psi_i) + \sum_{j \in \text{Rem}} \left(\alpha_j' \psi_j \right) - \sum_{k \in \text{Add}} (\alpha_k' \psi_k) \right)^T \sum_{i \in [n] \setminus \text{Rem} \cup \text{Add}} \psi_i. \tag{7.19}$$

Here, the weighted ψ values (not including the ψ_i sum) provide a contribution of the points as a weight vector of sorts. The α values are then analogous to MRPS importance. Note how the final result is similar to MRPS in that (1) the overall structure is $\alpha\psi_i\psi_j$ and (2) the overall importance is a summation of individual importances.

Notably, $\sum_i \psi_i$ can be pre-computed and then re-used for all different Rem and Add sets.

7.4.3.2 Influence if Assuming Alpha Values Do Not Change

Equation 7.19 requires α' values. RPS implicitly assumes that $\alpha'_i = \alpha_i$. The two values are indeed close in general, particularly when |Rem| and |Add| are low. Making this assumption yields the next step in terms of a better estimate (compared to init_est).

Lemma 6. *If the MRPS importance values are assumed to not change as a result of adding or removing points ($\alpha_i = \alpha'_i$), then the change in $q_{f0}(c)$ when removing points with indices in Rem and adding points with indices in Add is:*

$$\textit{init\_est} + \frac{1}{n - |\textit{Rem}| + |\textit{Add}|} \left(\sum_{j \in \textit{Rem}} (\alpha_j \psi_j) - \sum_{k \in \textit{Add}} (\alpha_k \psi_k) \right)^T \sum_{i \in [n] \setminus \textit{Rem} \cup \textit{Add}} \psi_i \tag{7.20}$$

where init_est is the original estimate from Lemma 4.

The use of α_k may seem bizarre since these α values are for points not in the original set of experiences. However, MRPS provides a closed-form solution for α that does not depend on the point being used to train the original network. Therefore, α_k values can be readily computed though they are not part of the MRPS expression for Θ in terms of α values.

A benefit for this approximation is that individual α_j and α_k terms do not depend on the entire sets, neither Rem nor Add, so operations such as maximizing a change subject to $|\text{Rem}| + |\text{Add}|$ constraints is easier than when allowing α values to change.

7.4.4 Accounting for Change in Alpha Values

Instead of assuming α_i and α'_i are equal, we can compute the difference. This then links to the next step of a better estimate, the true influence value. However, this computation of α'_i depends on the full Rem and Add sets, so using the true α differences prevents computing contributions of potential members of Rem and Add before considering the entire sets.

Following the MRPS approach of treating Q to be a linear model atop a learned featurization of inputs and given that we use an L2-based loss function, the exact alpha values are

$$\begin{aligned} \alpha_i &= \frac{-1}{2\lambda n} \frac{\partial L(\psi_i, t_i, \Theta)}{\partial \Phi(\psi_i, \Theta)} \\ &= \frac{-1}{\lambda n} \left(\Phi(\psi_i, \Theta) - t_i \right) \\ &= \frac{-1}{\lambda n} \left(\Theta \psi_i - t_i \right). \end{aligned} \tag{7.21}$$

The change in α_i is then closely related to the change in Θ, the optimal Ridge Regression weights:

$$\alpha_i - \alpha_i' = \frac{-1}{\lambda n}\left(\Theta\psi_i - t_i\right) - \frac{-1}{\lambda(n - |\text{Rem}| + |\text{Add}|)}\left(\Theta'\psi_i - t_i\right), \tag{7.22}$$

where Θ' is the set of weights used with Q^1.

For our problem, the change in Θ values can be exactly computed:

$$\Delta\Theta := \Theta - \Theta' = (\Psi^T\Psi + \lambda I)^{-1}\Psi^T T - (\Psi'^T\Psi' + \lambda I)^{-1}\Psi'^T T', \tag{7.23}$$

where Ψ' and T' are the matrix of featurized states and vector of target values used with Q^1, respectively.

For CUSTARD, we were able to reduce the change in Θ to a constant times the change in $\bar{t}$ since the ψ values were all related (within a given leaf). Here, there is no such relationship, so the change in Θ effectively requires solving the RR problem to find the new weights for the final layer of the original Q-network. In practice, the change can be computed more easily than solving the problem from scratch since the terms involving Ψ' are closely related to the terms involving Ψ, but this is a detail beyond the scope of this current work.

The change in Θ, in conjunction with the equations for the change in α above, allows us not only to exactly compute the change when a point is removed or added but also the degree to which $\alpha_i' := \alpha_i$ is a reasonable approximation.

7.4.5 Exact Influence Computation

The above computation is convenient when changes in α MRPS values are desired, but computing only the influence on $q_{f0}(c)$ allows skipping the steps that involve α terms. In particular, we can avoid finding $Q^0(s_i) - Q^1(s_i)$ through an MRPS decomposition. Instead, we can use the definition of $q_{f0}(c)$ along with the equation for influence on Q. First, the influence with respect to $q_{f0}(c)$ is:

$$\frac{\sum_{i\in[n]} Q^0(s_i)}{n} - \frac{\sum_{i\in[n]\setminus\text{Rem}\cup\text{Add}} Q^1(s_i)}{n - |\text{Rem}| + |\text{Add}|}.$$

With the relationship between Q^0 and Q^1 that we identified before, we can get a closed-form solution.

Lemma 7. *The change in $q_{f0}(c)$ when removing points with indices in Rem and adding points with indices in Add is:*

$$\frac{1}{n - |\mathit{Rem}| + |\mathit{Add}|}\left(\frac{1}{n}\left(|\mathit{Add}| - |\mathit{Rem}|\right)\Theta\sum_{i\in[n]}\psi_i + \Theta\left(\sum_{j\in\mathit{Rem}}\psi_j - \sum_{k\in\mathit{Add}}\psi_k\right) + \Delta\Theta\left(\sum_{i\in[n]}\psi_i - \sum_{j\in\mathit{Rem}}\psi_j + \sum_{k\in\mathit{Add}}\psi_k\right)\right) \tag{7.24}$$

where $\Delta\Theta$ is as given in Equation 7.23.

The dominating term in the computational complexity of using this solution comes from finding $\Delta\Theta$. As with the previous solution for influence, the dependence of $\Delta\Theta$ on the entire Rem and Add sets precludes separately computing contributions of potential points to remove or add and later selecting the Rem and Add sets. In these instances, an approximation (from earlier sections, such as Section 7.4.2) can be used for the purpose of selecting points based on approximate influence prior to determining exact influence.

7.5 Experiments

We have introduced methods for computing importance and influence for all nodes within a UET. As noted previously, influence is the exact change induced by removing or adding a set of experiences (i.e., transition tuples). Our overall goal is to compare the impact of removing or adding different sets of experiences, particularly with the goal of finding the most impactful set to remove/add based on a set size constraint.

In Section 7.3, we introduced an efficient approach for computing exact influence for nodes originating from the CUSTARD DTP. In Section 7.4, we introduced a number of different methods for computing influence. These latter methods make different assumptions in order to reduce computational complexity at the expense of computing approximate influence.

We demonstrate the utility of approximate influence values for finding impactful points with respect to nodes obtained via APG-Gen. In particular, in Section 7.5.1, we show that ranking experiences based on approximate influence rather than importance leads to a ranking closer to that obtained when using exact influence values. Then, in Section 7.5.2, we show that the approximate influence values lead to an ordering of experiences that strongly correlates with the ordering induced by the exact influence values.

7.5.1 Identifying Most Influential Experiences

One of the use-cases for influence values is to assist in deciding whether to remove or add given experiences. Thorough evaluation of each experience (e.g., via a human expert or experiment that allows the modified agent to act in the real world) is generally not feasible. Instead, experiences that have the highest influence values can then be screened to determine whether they are anomalous (outliers) or crucial (rare positive examples).

We consider a setting where a fixed number of experiences are to be evaluated. Each method selects a set of experiences of fixed size, and the different sets are compared. Effectively, we are evaluating methods based on their ability to identify the "top k" most influential experiences. We compare selecting experiences at random, based on importance, based on approximate influence (by assuming a fixed Q-function), and based on exact influence.

7.5.1.1 Metrics

We use the set chosen based on exact influence values as the ground truth. We make this choice since the experiences with highest exact influence are, by definition, those that are most impactful. The sets chosen by other methods are then compared to this set chosen via exact influence values. A successful method is then one that has high recall, treating the experiences selected based on exact influence as positive samples and all others as negative samples. We refer to this as the "intersection percentage" because this metric is the "size of intersection with ground truth set" divided by the size of the set retrieved. Note that this set size is fixed– all methods must select sets of identical size.

We perform this evaluation for a range of different set sizes. We report these sizes as "sample percentage." The sample percentage ranges from 0 to 1 (with a step size of 0.02) and corresponds to the set size divided by the number of total experiences.

Influence and importance may be positive or negative. Some past work on importance, such as [1], only considers the absolute value of influence and importance. For some use-cases, one seeks experiences that specifically reinforce or reduce a given behavior. Therefore, we evaluate methods by looking at the "most negative" and "most positive" experiences separately. This evaluation leads to two separate sets of plots.

7.5.1.2 Setup

We largely follow the setup used in Chapter 5. We perform our evaluation on PrereqWorld, as described in Section 2.2.1. For this experiment, we select $m = 8$ and $\rho = 0$. The DTP is created using CUSTARD on top of Quantile Regression DQN, and we use the APG-Gen with Initialization method.

7.5.1.3 Results

We report the intersection percentage for most negatively influential experiences in Figure 7.1 and for most positively influential experiences in Figure 7.2.

As expected, in both plots, random selection of experiences leads to an intersection percentage that closely matches the sample percentage. The two percentages are expected to match; the slight deviations present are due to empirical sampling.

Importance generally performs better than random selection for intermediate sample percentage values in the 0.2-0.8 range. This behavior suggests that the importance values do not distinguish well between the most negatively and positively influential experiences.

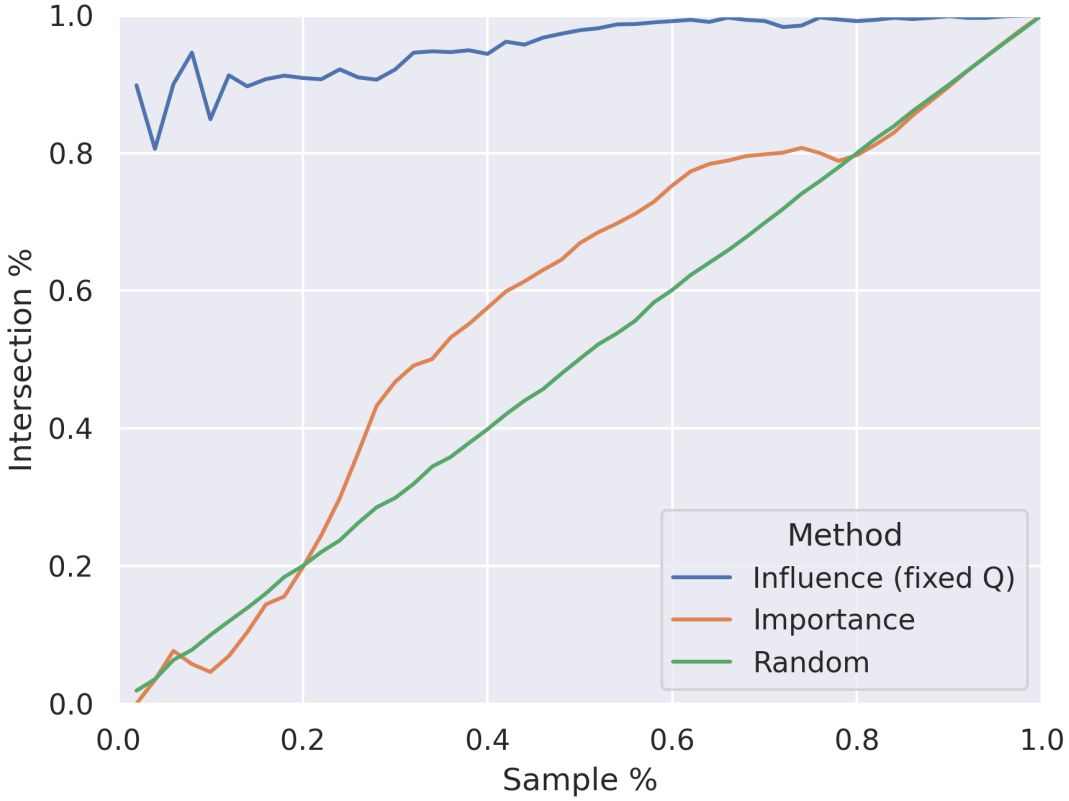


Figure 7.1: Comparison of methods for selecting sets of negatively impactful experiences. The resulting sets are compared to the sets obtained via exact influence values.

In contrast, approximate influence maintains a relatively high intersection percentage. In particular, the most positively influential experiences are identified with an intersection of more than 90% even for small sample percentages. These results indicate that approximate influence can be used to identify the large majority of truly influential points for evaluation using a subsequent evaluation method (e.g., executing the agent to observe its performance).

7.5.2 Ranking Influential Experiences

Building on the previously mentioned use-case, one may seek not only a set of impactful experiences but a ranking of the experiences based on their relative impact. To that end, we directly compare the rankings induced by approximate influence and importance relative to exact influence.

7.5.2.1 Metrics

We use the Spearman Rank Correlation (SRC) metric to quantify how well the order produced by different methods matches the order produced by exact influence values. We omit details on

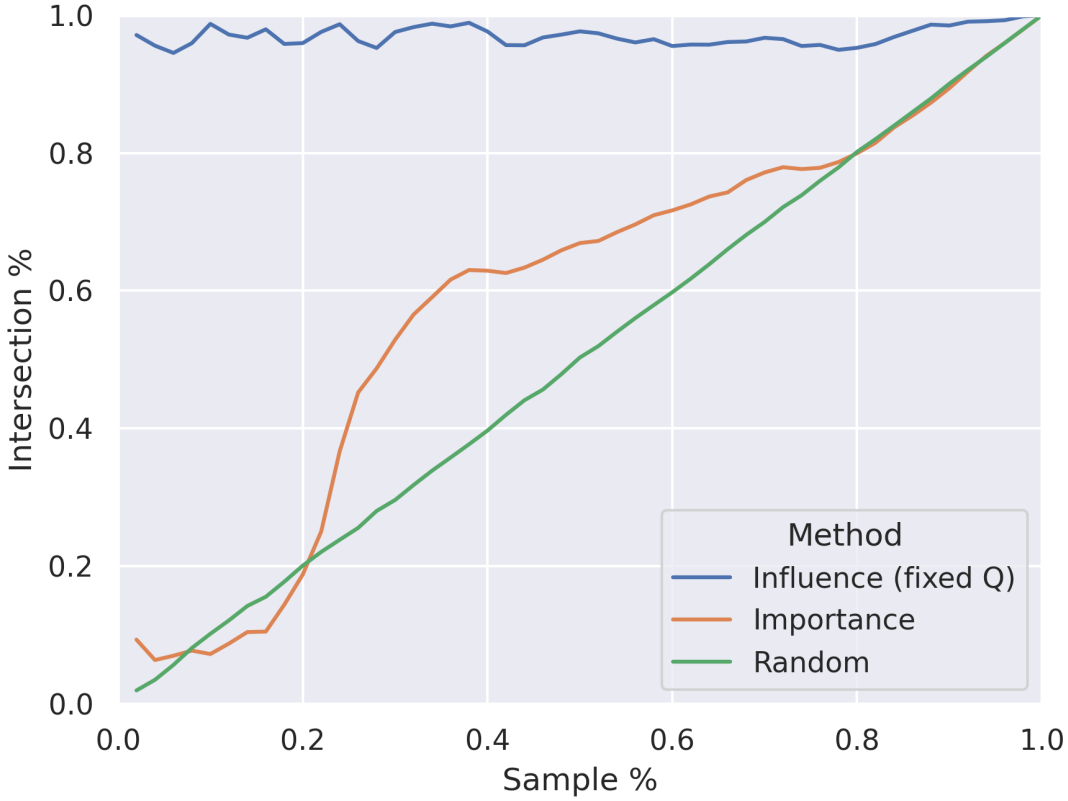


Figure 7.2: Comparison of methods for selecting sets of positively impactful experiences. The resulting sets are compared to the sets obtained via exact influence values.

SRC, but briefly note that the SRC reflects the correlation between two rankings of the same points. An SRC coefficient is bounded by -1 and 1. In our case, values closer to 1 correspond to a closer match between the rankings, and a value of 1 means that the rankings are identical.

7.5.2.2 Setup

We use the same setup as Section 7.5.1.

7.5.2.3 Results

We report the Spearman Rank Correlation coefficients for importance and approximate influence relative to exact influence values in Table 7.2.

Notably, the SRC for approximate influence is substantially higher than that of importance. This result indicates that our approach for computing approximate influence is more effective at ranking points based on actual influence than importance. Additionally, the SRC for approximate influence is quite close to 1. This outcome suggests that, for this domain, the assumption that the Q-function does not change is reasonable.

Method	Spearman Rank Correlation
Influence (fixed Q)	0.9866
Importance	0.3667

Table 7.2: Comparison of Spearman Rank Correlations with respect to the ranking produced by exact influence values.

7.6 Summary

In this chapter, we presented our approach for finding the importance and influence of experiences on nodes within a UET. We introduced our closed-form solutions for these quantities.We further introduced methods for efficiently computing importance and influence based on these solutions. These solutions enable identifying which experiences impacted a partition within a UET and the degree of this impact. With this ability, a UET can be explained in terms of both state features and agent experiences.

Chapter 8

Related Work

In this chapter, we review the literature related to our work. This thesis builds upon and presents a unified method for providing four types of explanations, so we provide an overview of existing work around each of the relevant explanation types. Section 8.1 focuses on works that explain single-action behavior in terms of state features. Section 8.2 focuses on methods that explain the impact of aspects of the MDP or learning process (such as the reward function or observed transitions) that led to the agent's learned behavior; this includes explanations in terms of past experiences, though existing work explains only single-action behavior in this way. Section 8.3 focuses on approaches that explain the agent's longer-term behaviors (i.e., policy-level behavior) in terms of state features or another featurization of the states. Section 8.4 compares our approach to existing methods.

In addition to this review of related work, we contributed to a broader survey of explainable reinforcement learning methods [44]. This survey organizes works into a hierarchy that is derived from the categorization used for this chapter, but the survey includes methods beyond those immediately relevant to contextualizing the contributions of this thesis. The survey also provides a more thorough comparison across groups within the hierarchy that is orthogonal to the comparisons within this chapter of prior work to our methods. While surveying existing work for the survey paper, we did not identify work which fully addresses our thesis question.

8.1 Per-Action Feature Importance Explanation

Methods that explain a single action in terms of the current state's features fall into three distinct types: (1) those that use a post-learning step to convert the policy to an inherently interpretable form, (2) those that use an inherently interpretable format for the policy (i.e., at any point in time, an action can be explained), and (3) those that generate an explanation for an action without converting the policy to an inherently interpretable form. We summarize these categories within Sections 8.1.1, 8.1.2, and 8.1.3, respectively. This categorizations roughly correspond to the distinctions made by DARPA's XAI initiative [45] between Model Induction, Interpretable Models, and Deep Explanation.

8.1.1 Post-hoc Conversion to Explainable Format

Policies represented with suitable models enable explanations to be readily generated or the use of the representation itself as an explanation. The standard structures used in RL, lookup tables and neural networks, are not directly interpretable, but a conversion step can be used to approximate an original policy with an explainable model. Many explanation-generation methods effectively use imitation learning [46, 47], where the policy obtained via RL is the *expert* and the policy obtained via imitation learning is the *learner*. The most successful line of work, to date, began with VIPER [48], where a DT is trained using the Q-values from expert transitions gathered via DAGGER [29]. MoET [49] builds on VIPER, replacing the DT with a gating function over multiple DTs.

Other approaches use a standard supervised approach instead of DAGGER but vary the model type and optimization procedure. The Lumberjack algorithm [50] constructs a Linked Decision Forest (LDF). [51] use a learner consisting of an ensemble of decision trees, which is trained via gradient boosting. [52] train a set of Binary Space Partitioning (BSP) Trees to approximate a learned Q-function, with one tree per action. [53] use a U-Tree regression tree with a linear model at each leaf. [54] use Soft Decision Trees (SDTs). [55] grow a decision tree using an extension of CART which measures action, value, and derivative estimation accuracy. [56] introduce AI-Interpret, which ultimately produces a decision tree where internal nodes use boolean expressions over domain language expressions.

An alternative approach is taken by [57], who use genetic programming to extract control policies from a neural network. Similarly, [58] use a recursive bilevel evolutionary algorithm to find a nonlinear decision tree, then prune the tree by reoptimizing all rules using an evolutionary optimization procedure.

8.1.2 Learn as Inherently Explainable Format

Methods which learn an inherently explainable policy can be categorized by the format of the policy. The most common format is a decision tree, though other formats are also sometimes used.

8.1.2.1 Decision Tree Models

Given the relative interpretability of DTs [19, 59], they are frequently used as representations for explaining RL agents. Earlier work in learning a DT policy was not explicitly motivated by explainability, but these methods produce DT policies of the same style as newer approaches which seek explainable RL agents. [60] introduce the *G algorithm*, which incrementally builds a DT while training an RL agent for MDPs with binary features. Transition tuples are stored within leaf nodes, and leaf nodes are split when the Student's t-test suggests that two distinct distributions are present in the saved transition tuples.

UTree [21] uses the Kolmogorov-Smirnov test, which identifies violations in the Markov property. Continuous UTree [22] extends UTree to MDPs with continuous-valued features. [61] also extends the G algorithm to continuous-valued features and evaluates splitting criteria

of information gain, Gini index, and the Twoing Rule [62]. [63] extend [61] to a multi-agent setting. TTree [64] is a UTree-like algorithm for solving Semi-MDPs (SMDPs) [65, 66, 67]. [23] modifies UTree to use a variety of splitting criteria: Gini, Info Gain, t-test, and Twoing rule. The TG algorithm [68] extends the G algorithm to use internal nodes with tests on conjunctions of first-order literals (rather than a single comparison). TGR [69] extends TG to have pruning and revising operations. To allow these additional operations, TGR stores tuple statistics for all nodes (not just the leaves, as in TG and G).

[70] create tree policies in a batch RL setting. The authors use fitted Q-iteration with tree-based regression methods (CART, Kd tree, tree bagging, and two new ensemble algorithms). Policy Tree [71] creates trees with leaves that are Giffs softmax sub-policies. The splitting values are updated via policy-gradient RL. Similarly, [25] learns a softmax tree using an on-policy policy gradient method. After training, the tree can be discretized to form a DT policy approximation.

8.1.2.2 Other Models

[72] and [73] propose a policy-iteration scheme that produces a clustering over the state space, where a single action is associated with each cluster. [74] use a genetic algorithm to create an inherently interpretable model: trees with basic algebraic functions as internal nodes and constants and state variables as leaves. [75] use a biological neuronal cell model to represent a policy. [76] propose to use a search algorithm over closed-form formulas which are used to rank actions. [77] use Particle Swarm Optimization to learn a fuzzy controller policy, which is a set of linguistic if-then rules whose outputs are combined. [78] use fuzzy RL to learn a policy where state space regions around prototypes can each be expressed as a single distribution.

8.1.3 Directly Generating an Explanation

We categorize methods for directly generating explanations based on the form the explanation takes. Note that different approaches for generating similar styles of explanations may make different assumptions about the agent being explained, but there are generally commonalities within a group. For example, saliency-based methods require a differentiable policy or Q-function while attention-based methods sometimes require the ability to add an attention layer to the value function approximator used by the agent.

8.1.3.1 Incremental Explanations

[79] compare an agent's chosen action to a possible alternative using a known factored MDP; natural language templates are filled based on policy execution outcomes. [80] perform model reconciliation: the agent provides sufficient information to the user such that the user's predicted plan matches the agent's plan. [81] represent a user's understanding of the task and produce contrastive explanations in natural language and a form akin to PDDL [82] precondition/postcondition/effects/actions.

8.1.3.2 Natural Language Explanation

[83] provide information about the agent's target using a linear classifier that is trained in a supervised fashion. [84] provide per-action natural language explanations by training a supervised model on a corpus of human-provided action-explanation pairs. [85] fill natural language templates using a domain model trained on demonstrations and a set of operator-specified important program state variables and functions. [86] segment a path and map each segment to an utterance, as a function of the desired point in the verbalization space. [87] construct natural language explanations by comparing outcomes of a user-specified policy and a learned policy.

8.1.3.3 Saliency

[88] use standard gradient approaches ([89], [90]) to produce saliency maps for RL agents. [91] produce object saliency maps using a perturbation-based approach, where an object detector is assumed to be provided. [92] use Layer-wise Relevance Propagation (LRP) to create saliency maps. [93] propose a perturbation-based approach. Motion-Orient REinforcement Learning (MOREL) [94] identifies motion within the pixel-based observations. [95] directly use LIME [96] to identify groups of features which are used by the Pensieve [97] agent. [98] find that saliency is not sufficient as an explanatory model for RL agents; the authors argue that conclusions drawn from saliency map outputs are highly subjective.

8.1.3.4 Attention

After training, [99] use imitation learning to train an additional network that modifies the state input by applying an attention mask. [100] add an attention module within the LSTM portion of the agent's value function approximator. [101] add an attention-like module to the agent's value function approximator, though they do not use an LSTM within their agent. [102] use a two agent system, where one agent constrains what the second agent may use for predictions, and the first agent has an attention layer. [103] add an attention component to their agent, which they train using an evolutionary algorithm to allow non-differentiable attention components. [104] add a key-value store to their agent such that the agent's output is multiplied by each key, and these outputs are multiplied by the values. The agent's output is put through deconvolutional layers to visualize the attention. [105] question the validity and usefulness of attention in Natural Language Processing problems. The authors find that multiple different attention explanations can lead to the same behavior and that attention often does not correlate with gradients at feature inputs.

8.1.3.5 Other Explanation Types

[106] generate a counterfactual state which minimally differs from the query state. The authors split the policy network into a "state to latent representation" portion and a "latent representation to action" portion. A deep generative model is then used to shift a state's latent representation toward one which leads to a different action. [107] train a generative model over

the environment's state space, and they use its latent space to select states which maximize a user-specified objective.

8.2 Explanations in Terms of MDP or Learning Process

Beyond feature importance explanations, specific actions have been explained in terms of the MDP or the training process. Most commonly, the agent is made to reveal the transition dynamics it has learned. Such methods are described and compared in Section 8.2.1. More recently, different types of reward function decomposition have been used to explain the agent in terms of the MDP's reward function. Such methods are discussed in Section 8.2.2. Finally, methods that identify impactful training experiences are discussed in Section 8.2.3.

8.2.1 Modeling of Domain Information

[108] use a neural network to approximate the transition function. Since the network is a differentiable model of the environment, gradients from the state predictor network can be used to update the policy. In this work, the "explanations" are internal explanations for the agent to improve itself rather than explanations for a human user. [109] learn a representation of the transition dynamics as a dynamic Bayesian Network or DT. [6] learn per-feature pre-requisites for a factored domain to then use a case aggregation method to explain actions in terms of past experiences. [110] learn an environment model as a Probabalistic Graphical Model (PGM). This model is learned jointly with the policy using a latent state space, so future states and actions can be predicted. However, the components of the PGM are still CNNs, so they cannot be directly interpreted.

[111] ground opportunity chains in human-agent experimental data. They use an RNN to learn opportunity chains, and they use DTs to improve the accuracy of task prediction and of the generated counterfactuals. [112] use inductive learning to identify preconditions of agent actions in terms of first-order relational conjunctions of symbolic state variables. [113] learn the structure of a factored MDP as a DT for the reward function and each feature's transition function. [114] model the relative effect of actions with DTs. The DTs are trained in a supervised fashion to predict the change for each state feature and the reward for each state-action pair. [115] propose a memory-based RL approach that uses an episodic memory to explain decisions using probability of success and number of transitions required to reach the goal state. [116] learn a structural causal model (with pre-specified structure), which encodes causal relationships between variables of interest.

8.2.2 Decomposition of Reward Function

By conveying aspects of the reward that contributed to the agent's behavior, an explanation can provide information in terms of the agents objectives. [117] use reward decomposition charts (RDCs) as explanations. The reward function is assumed to be decomposable into a set of additive terms with semantic meaning. The Q-function is decomposed into a similar set of additive terms (with one Q-function term per reward function term). [118] produce explanations of the form, "I could do [action] to decrease $R(obj)$. However, this would decrease $R(obj_2)$"

through a vector value function based Multi-Objective RL approach that quantifies objective relationship via RL. [119] present a two-level hierarchical DRL system. The high-level agent effetively discretizes the environment and defines subgoals based on this discretization. The expected value of choosing a subgoal is then available via the higher-level policy, allowing the return for various subgoals to be accessed via the high-level agent.

8.2.3 Identification of Important Training Points

[120] examine which states an RL agent needs to memorize for efficient learning and explaining. The authors train an agent with DRL and use Sparse Bayesian RL (SBRL) to extract and store important memories. SBRL effectively selects which states to record, along with a weight for each state. [121] identify training points that are *influential* for the off-policy estimation of Q-values (i.e., whose removal causes a large change in Q-value estimates for initial states). The authors present closed-form solutions for cases where a linear model trained with least-squares error or a nearest-neighbors model is used as a value function approximator. [122] convey the objective function of an agent by presenting a set of transition tuples. In particular, the agent models the human's current belief about the agent's objective function, and the agent presents tuples which most contradict the human's current estimate.

8.3 Policy-level Behavior Explanations

Explanations of an agent's longer-term behavior have been studied less than explanations of single actions. Given the interactive nature of an agent, explanations for long-term behavior generally take the form of a graph over abstract states (or an equivalent formulation).

8.3.1 Summarization via Set of Transition Tuples

[123] explain agent behavior by presenting a user a set of trajectories. The trajectories are selected based on the diversity of states as well as the difference in Q-value between the best and worse actions among states within the chosen trajectories. Similarly, [124] seek to find "critical states," which are provided to a user so they can form a mental model of the agent. Critical states are those where the chosen action has a much higher Q-value than another. [125] builds on [124]. The authors pick trajectories to show to humans so they can see key interactions by the agent. The trajectories are selected based on "interestingness" statistics, which include frequency, estimated value, and certainty. [126] compare different ways of choosing state/action pairs for policy summarization. The authors find that the best method for selecting pairs depends on the method of policy reconstruction: there is no single best way to select pairs.

8.3.2 Conversion of RNN to Finite State Machine

[127] convert a trained agent's RNN into a Finite State Machine (FSM). The authors consider an agent where the output of a CNN is fed as input into an RNN. They apply a discretization process between the two networks and fine-tune the RNN. The resulting agent has a set number of possible inputs and possible transitions, so a FSM can be constructed to replace the RNN.

This family of techniques only applies in cases where the agent contains a RNN. Additionally, to obtain a fully explainable agent, the CNN would have to be explained using a different approach.

8.3.3 Extraction of Clusters or Abstract States

[128] embed states into a space such that states located close to each other in this space are treated similarly by the agent. A human operator can then identify clusters within this space as well as relationships between clusters. [129] present TLdR, an approach for identifying landmarks, which are propositional formulas that must be satisfied for a goal to be completed. Effectively, each landmark corresponds to a set of states in which a subgoal has been completed. The possible transitions between landmarks can then be displayed as a graph that shows how an agent would reach a goal state from the starting state. These two methods can be interpreted as creating abstract states that encapsulate sets of states from the base MDP. [130] provide an overview of previous methods for creating abstractions for MDPs. Though the authors do not consider explainability, the approaches they describe could be used to explain an agent's behavior in terms of the created abstract states. The methods either create an abstraction before or during training, so the agent must act in the abstract MDP. As a result, such approaches cannot be used to explain an arbitrary agent that learned to act on states from the base MDP.

8.4 Comparison with Our Work

In contrast with existing approaches, our explanation format explains both single-action behavior and policy-level behavior in terms of both state features and past experiences. Since we use a unified structure (rather than one structure per explanation type), we identify important training points at all levels of the structure. Furthermore, each element of our approach provides capabilities not present in existing approaches.

For single-action behavior explanations in terms of state features, we introduce Iterative Bounding MDPs (Chapter 3), which cast the problem of finding a decision tree policy as a meta-problem. This approach is the first that simultaneously does not approximate the learned policy after training and allows the use of standard deep RL techniques. For policy-level behavior explanations in terms of state features, we construct a Markov chain over abstract states (Chapter 4). This format provides quantitative information about future transitions without constraints on the agent type (e.g., requiring a Recurrent Neural Network during training). Using a single decision tree for both explanation types (Chapter 5) enables us to identify important past experiences for specific elements of an agent's behavior (Chapters 6 and 7). Using these feature importance values, a user is able to modify agent behavior in terms of the explanation itself (e.g., change the features considered and the categorization of states into abstract states).

Chapter 9

Conclusions and Future Work

We first summarize the contributions of this thesis. Then, we discuss the future lines of work that are enabled by the contributions of this thesis. We conclude with a summary of our overall work.

9.1 Contributions

In the motivating example in Chapter 1, we demonstrated the types of insights possible through explainable reinforcement learning methods with different properties: explanations of single-action vs policy-level behavior and explanations that are state-feature-based vs experience-based. We have shown techniques for producing explanations of all of these types as well as a unification of these techniques which permits further insights.

9.1.1 Decision Tree Policies

We introduced a new method, CUSTARD, for creating a Decision Tree Policy (DTP), which is an explanation of single-action behavior in terms of state features. CUSTARD is designed to be compatible with existing RL methods, including those that leverage neural networks as value function approximators. This compatibility is achieved by posing the "find a DTP for this MDP" problem as an RL problem that is then solved by an existing RL method. The meta-problem takes the form of a novel type of MDP, an Iterative Bounding MDP (IBMDP), which ensures that any solution for the underlying MDP is a valid DTP. We additionally presented a method for extracting a DTP from any policy that solves the meta-problem framed as an IBMDP. We showed how to use this approach with existing RL methods to avoid the downsides of other methods that find DTPs.

9.1.2 Abstract Policy Graphs

We introduced a new type of policy-level behavior explanation, the Abstract Policy Graph (APG). An APG effectively partitions the state space into abstract states and then forms a Markov chain out of the resulting abstract states. This type of explanation enables predicting an agent's future actions and identifying the sequence of subgoals it will complete. Additionally,

we introduced APG-Gen, a method for constructing an APG using only sample transitions from an agent and its Q-value estimates for its chosen actions. APG-Gen treats the Q-function as an opaque box and makes no assumptions about how it was trained or how it functions internally. We showed that APG-Gen can efficiently construct APGs. We also demonstrated that an APG successfully captures information about future actions and relevant features.

9.1.3 Importance Scores for Past Experiences

We introduced MRPS, a method for assigning importance scores to sets of past experiences for agents that use a neural network as their value function approximator. This score computation follows previous work where the final layer of the network is decomposed into a weighted sum of past experiences, and these weights are then used as importance values. We evaluate our approach on a supervised classification debugging task and show the benefits of our approach, assigning scores to sets of points rather than only to individual points. Since CUSTARD was designed to leverage RL methods that use neural networks, MRPS is directly compatible with CUSTARD and finds the importance values for past experiences. These values are with respect to the IBMDP actions so show importance not only for the base MDP actions but also for partitioning actions taken within the DTP.

9.1.4 Unified Explanation Trees

We introduced a new explanation structure, the Unified Explanation Tree (UET), which explains both single-action and policy-level behavior in terms of state features. We introduced a method for creating a UET which is effectively an application of a modified APG-Gen variant to a DTP learned by CUSTARD. Due to the way a UET is constructed, it captures the relationship between local and global behaviors. Furthermore, we presented several approaches for finding the importance and the influence of training points on all nodes of a UET. These approaches enable explaining not only agent actions in terms of past experiences but also the internal operation of the UET itself, thus explaining the effect of experiences on policy-level behavior. We introduced approaches that ranged from exact solutions that make no assumptions to approximations that are preferred for certain use-cases.

9.1.5 Domains and Evaluation

We introduced two novel domains for evaluating explainable RL methods. The first domain, PrereqWorld, is automatically constructed with known, ground-truth feature importances and subgoal ordering. These properties enable a quantitative evaluation of an explanation's ability to identify relevant features and the sequence of subgoals completed by an agent. The second domain, PotholeWorld, is a domain where the optimal policy substantially differs depending on the permitted policy complexity. This property causes interpretable approximations of an expert to perform poorly compared to alternative interpretable policies. This domain enables evaluating an RL method's ability to not only learn an interpretable policy but also find a high-performing one subject to this constraint. We demonstrated the effectiveness of our interpretable policy-

learning method, CUSTARD, on both of these domains and used PrereqWorld throughout our evaluation.

9.2 Future Work

We present potential research directions that extend the work presented in this thesis.

9.2.1 Extending UETs to Continuous Features

We introduced a method for creating a unified explanation tree that conveys both single-action and policy-level behavior in terms of both state features and past experiences. However, the policy-level behavior component is provided by APG-Gen with FIRM as the feature importance metric, and FIRM relies on all state features being binary. Though this permits using APG-Gen with categorical features, continuous features are not directly supported. APG-Gen should be extended to become compatible with continuous features. To achieve this, one would need to either create a method for dynamically discretizing continuous features or identify a suitable feature importance metric which is compatible with continuous features while maintaining the favorable computational complexity exhibited by APG-Gen with FIRM over binary features.

9.2.2 Extending UETs to Continuous Actions

We introduced CUSTARD, a way to learn a decision tree policy via an arbitrary reinforcement learning algorithm by casting the "find a decision tree policy for this environment" meta-problem as an RL problem. We then used the resulting decision tree in constructing our unified explanation tree. Though we presented a hypothetical way to leverage continuous actions in solving the meta-problem, our current methods all use discrete actions. In the same way that we modified RL methods that can handle discrete actions (e.g., DQN), we would need to modify methods that can handle a mixture of discrete and continuous actions (i.e., one that is compatible with Parameterized Action Spaces).

9.2.3 Enabling Efficient Intervention via Experience Removal/Addition

We presented methods for finding the influence of any set of past experiences on any node within a unified explanation tree. Influence provides the exact change in the node's behavior if that set of experiences is removed or added, as appropriate. As a result, we can determine whether removing/adding any given set would have an effect and, if it does, which subtree would change as a result of this intervention. However, if an intervention is performed by such a removal or addition, there is no efficient way to rebuild the affected subtree. We could create an approach that finds an optimal subtree based on the gathered experiences without affecting other portions of the unified explanation tree. Ideally, this approach would leverage the computation performed to create the original subtree so that this process is as computation-efficient as possible.

For this method, we would need differing approaches for CUSTARD-derived and APG-Gen-derived nodes, as when initially building the unified explanation tree. In the CUSTARD nodes,

we could use an IBMDP formulation along with a tabular method to find the new optimal subtree. This tabular method could be initialized with the Q-values for the original subtree, and we could prioritize updates for leaves that would contain the removed or added experiences. For all leaves of this subtree, we could then apply APG-Gen, which is already efficient to compute and whose computation does not depend on other leaves in the tree.

Such an approach would enable direct, simultaneous modification of a policy and its corresponding unified explanation tree whenever experiences are added or removed. Potentially, this could form the basis of an online method to automatically gather new experiences as needed. This extension would be a large step toward the overall goal of our work, to enable the use of RL in more domains by enabling interventions via requesting specific experiences instead of requiring trust in an opaque policy.

9.2.4 Applying to "Learning from Demonstrations" Problem

Our approaches for creating explanations are compatible with suboptimal policies. As a result, they can be used to explain arbitrary policies, even those that are not the result of a reinforcement learning process. In particular, our method for explaining policy-level behavior, APG-Gen, only requires examples of transitions induced by the policy and makes no assumptions about how the policy internally operates. Therefore, we can apply APG-Gen to a dataset of human behavior to gain insight into a human expert's choices. Beyond this, behavior cloning can be used to imitate a human expert and all our approaches can be used to explain the resulting agent. By applying our methods for identifying influential experiences, we can identify cases where the human demonstrations may not be representative of useful demonstrations for the imitating agent's learning.

This process would be most useful in cases where human demonstrations are expected to be of varying but unknown quality, so our approach could be used to filter out low-quality demonstrations that may lead to incorrect policy entries. A natural setting in which to evaluate such an extension would be with the MineRL data, where we have gathered human data from a large number of different experts of varying skill level. This extension would contribute to solving the same problems that we began to address with our MineRL line of work: "How can we make RL more computationally efficient? How can we reduce the need for exploration?" These barriers of computational complexity and need for random exploration reduce the applicability of RL, but efficient learning from human demonstrations can diminish the impacts of these barriers by initializing an agent with useful sub-policies and initial behavior. Automatic identification of low-quality demonstrations would facilitate learning from human demonstrations which, in turn, reduces the need for random exploration.

9.2.5 Applying to Health and Finance Domains

We have applied our methods to environments specifically designed for evaluating explanation methods. We designed these environments to have ground-truth dependencies between subgoals and different optimal policies depending on the maximum permitted policy complexity. These properties facilitated our development of methods that provide information on the

sequence of sub-goals an agent is seeking to reach and that successfully find high-performing policies when the policy representation cannot capture the overall optimal policy. These aspects are of great importance in finance and healthcare domains. We could potentially apply our methods to these domains. Such an application of our work is key to our overall goal of allowing RL to be used more broadly. By providing explanations instead of only prescribing a policy, our work has the potential to make RL more useful and more applicable in real-world settings.

9.3 Summary

We have introduced methods for creating explanations for RL agents. We categorize these methods using two axes: whether the method explains single-action behavior or policy-level behavior and whether the method provides explanations in terms of state features or in terms of past experiences. These four types of explanations allow answering different questions about an agent, and the combination of all four types enables gleaning further information about an agent and its learning process. To the best of our knowledge, this work is the first work that does reasoning about adding and removing past experiences in reinforcement learning explanations.

We introduced a method for explaining single-action behavior in terms of state features via a decision tree representation of a policy. Notably, this method is compatible with existing RL techniques that use neural networks as their function approximator. We have also introduced a method for explaining policy-level behavior in terms of a Markov chain over abstract states. We showed that our method has a favorable computational complexity despite using only queries to an agent's Q-values, making no assumptions about an agent's decision-making process. We showed how to integrate these two methods to produce a Unified Explanation Tree, which maps from a state directly to both an action and abstract state, thus unifying single-action and policy-level behavior explanations in terms of state features. We evaluated our methods on domains designed specifically for testing RL explanation techniques. We then showed how to find the importance of sets of past experiences. We extended this general method to our Unified Explanation Tree and also presented closed-form solutions for exact influence. This addition of explanations in terms of past experiences allows identifying the exact subtree that would change if experiences were removed or added.

Bibliography

[1] C.-K. Yeh, J. Kim, I. E.-H. Yen, and P. K. Ravikumar, "Representer point selection for explaining deep neural networks," in *Advances in Neural Information Processing Systems*, pp. 9291–9301, 2018.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[3] C. Boutilier, R. Dearden, M. Goldszmidt, *et al.*, "Exploiting structure in policy construction," in *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence*, 1995.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[6] T. Dodson, N. Mattei, and J. Goldsmith, "A natural language argumentation interface for explanation generation in markov decision processes," in *International Conference on Algorithmic Decision Theory*, pp. 42–55, Springer, 2011.

[7] W. H. Guss*, B. Houghton*, N. Topin, P. Wang, C. Codel, M. Veloso, and R. Salakhutdinov, "MineRL: a large-scale dataset of Minecraft demonstrations," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 2019.

[8] S. Milani, N. Topin, B. Houghton, W. H. Guss, S. P. Mohanty, K. Nakata, O. Vinyals, and N. S. Kuno, "Retrospective analysis of the 2019 MineRL competition on sample-efficient reinforcement learning using human priors," *Proceedings of Machine Learning Research (PMLR): NeurIPS2019 Competition & Demonstration Track Postproceedings*, 2020.

[9] W. H. Guss, S. Milani, N. Topin, B. Houghton, S. Mohanty, A. Melnik, A. Harter, B. Buschmaas, B. Jaster, C. Berganski, *et al.*, "Towards robust and domain agnostic reinforcement learning competitions: Minerl 2020," in *NeurIPS 2020 Competition and Demonstration Track*, pp. 233–252, PMLR, 2021.

[10] A. Kanervisto, S. Milani, K. Ramanauskas, N. Topin, Z. Lin, J. Li, J. Shi, D. Ye, Q. Fu, W. Yang, *et al.*, "Minerl diamond 2021 competition: Overview, results, and lessons learned," *arXiv preprint arXiv:2202.10583*, 2022.

[11] W. H. Guss, C. Codel, K. Hofmann, B. Houghton, N. Kuno, S. Milani, S. Mohanty, D. P. Liebana, R. Salakhutdinov, N. Topin, *et al.*, "Neurips 2019 competition: the minerl competition on sample efficient reinforcement learning using human priors," *arXiv preprint arXiv:1904.10079*, 2019.

[12] B. Houghton, S. Milani, N. Topin, W. Guss, K. Hofmann, D. Perez-Liebana, M. Veloso, and R. Salakhutdinov, "Guaranteeing reproducibility in deep learning competitions," in *The 33rd Conference on Neural Information Processing Systems, Challenges in Machine Learning Workshop*, 2020.

[13] W. H. Guss, M. Y. Castro, S. Devlin, B. Houghton, N. S. Kuno, C. Loomis, S. Milani, S. Mohanty, K. Nakata, R. Salakhutdinov, *et al.*, "The minerl 2020 competition on sample efficient reinforcement learning using human priors," *arXiv preprint arXiv:2101.11071*, 2021.

[14] R. Shah, C. Wild, S. H. Wang, N. Alex, B. Houghton, W. Guss, S. Mohanty, A. Kanervisto, S. Milani, N. Topin, *et al.*, "The minerl basalt competition on learning from human feedback," *arXiv preprint arXiv:2107.01969*, 2021.

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

[16] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, 1986.

[17] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," in *Advances in Neural Information Processing Systems*, 2018.

[18] K. Sokol and P. Flach, "Desiderata for interpretability: Explaining decision tree predictions with counterfactuals," in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI-19), Student Abstract Track*, vol. 33, pp. 10035–10036, 2019.

[19] C. Molnar, *Interpretable Machine Learning*. 2019. `https://christophm.github.io/interpretable-ml-book/` (July 2020).

[20] Z. C. Lipton, "The mythos of model interpretability," *ACM Queue*, vol. 16, no. 3, pp. 31–57, 2018.

[21] R. McCallum, "Reinforcement learning with selective perception and hidden state," *PhD Thesis, University of Rochester, Department of Computer Science*, 1997.

[22] W. T. Uther and M. M. Veloso, "Tree based discretization for continuous state space reinforcement learning," in *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pp. 769–774, 1998.

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

[24] 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.

[25] I. D. J. Rodriguez, T. W. Killian, S. Son, and M. C. Gombolay, "Optimization methods for interpretable differentiable decision trees in reinforcement learning," *arXiv preprint, arXiv:1903.09338*, 2019.

[26] C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proc. of the 12th ACM Int. Conf. on Knowledge Discovery and Data Mining*, 2006.

[27] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[28] 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.

[29] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, pp. 627–635, 2011.

[30] N. Topin, S. Milani, F. Fang, and M. Veloso, "Iterative bounding mdps: Learning interpretable policies via non-interpretable methods," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021.

[31] W. Masson, P. Ranchod, and G. Konidaris, "Reinforcement learning with parameterized actions," in *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, 2016.

[32] M. Preda, "Adaptive building of decision trees by reinforcement learning," in *Proceedings of the 7th WSEAS International Conference on Applied Informatics and Communications*, 2007.

[33] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of the 33rd International Conference on Machine Learning*, 2016.

[34] C. Blundell, B. Uria, A. Pritzel, Y. Li, A. Ruderman, J. Z. Leibo, J. Rae, D. Wierstra, and D. Hassabis, "Model-free episodic control," *arXiv preprint arXiv:1606.04460*, 2016.

[35] A. Pritzel, B. Uria, S. Srinivasan, A. Puigdomenech, O. Vinyals, D. Hassabis, D. Wierstra, and C. Blundell, "Neural episodic control," *arXiv preprint arXiv:1703.01988*, 2017.

[36] K. Khetarpal, Z. Ahmed, G. Comanici, D. Abel, and D. Precup, "What can i do here? a theory of affordances in reinforcement learning," *arXiv preprint arXiv:2006.15085*, 2020.

[37] N. Topin and M. Veloso, "Generation of policy-level explanations for reinforcement learning," in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI-19)*, 2019.

[38] A. Zien, N. Krämer, S. Sonnenburg, and G. Rätsch, "The feature importance ranking measure," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2009.

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

[40] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pp. 448–456, PMLR, 2015.

[41] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.

[42] P. W. Koh and P. Liang, "Understanding black-box predictions via influence functions," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1885–1894, JMLR. org, 2017.

[43] Alex Krizhevsky, "The cifar-10 dataset," 2009. [Online; accessed 14-November-2019].

[44] S. Milani, N. Topin, M. Veloso, and F. Fang, "A survey of explainable reinforcement learning," *arXiv preprint arXiv:2202.08434*, 2022.

[45] D. Gunning and D. W. Aha, "Darpa's explainable artificial intelligence program," *AI Magazine*, vol. 40, no. 2, pp. 44–58, 2019.

[46] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the 21st International Conference on Machine Learning*, 2004.

[47] 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.

[48] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," in *Advances in Neural Information Processing Systems*, pp. 2494–2504, 2018.

[49] M. Vasic, A. Petrovic, K. Wang, M. Nikolic, R. Singh, and S. Khurshid, "Moët: Interpretable and verifiable reinforcement learning via mixture of expert trees," *arXiv preprint, arXiv:1906.06717*, 2019.

[50] W. T. Uther and M. M. Veloso, "The lumberjack algorithm for learning linked decision forests," in *The 4th International Symposium on Abstraction, Reformulation, and Approximation*, pp. 219–232, 2000.

[51] A. Brown and M. Petrik, "Interpretable reinforcement learning with ensemble methods," *arXiv preprint, arXiv:1809.06995*, 2018.

[52] A. Jhunjhunwala, "Policy extraction via online q-value distillation," *Masters Thesis, University of Waterloo*, 2019.

[53] G. Liu, O. Schulte, W. Zhu, and Q. Li, "Toward interpretable deep reinforcement learning with linear model u-trees," in *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 414–429, 2018.

[54] Y. Coppens, K. Efthymiadis, T. Lenaerts, A. Nowé, T. Miller, R. Weber, and D. Magazzeni, "Distilling deep reinforcement learning policies in soft decision trees," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence Workshop on Explainable Artificial Intelligence*, 2019.

[55] T. Bewley and J. Lawry, "Tripletree: A versatile interpretable representation of black box agents and their environments," *arXiv preprint arXiv:2009.04743*, 2020.

[56] J. Skirzynski, F. Becker, and F. Lieder, "Automatic discovery of interpretable planning strategies," *arXiv preprint arXiv:2005.11730*, 2020.

[57] H. Zhang, A. Zhou, and X. Lin, "Interpretable policy derivation for reinforcement learning based on evolutionary feature synthesis," *Complex & Intelligent Systems*, pp. 1–13, 2020.

[58] Y. Dhebar, K. Deb, S. Nageshrao, L. Zhu, and D. Filev, "Interpretable-ai policies using evolutionary nonlinear decision trees for discrete action systems," *arXiv preprint arXiv:2009.09521*, 2020.

[59] C. Kingsford and S. L. Salzberg, "What are decision trees?," *Nature Biotechnology*, vol. 26, no. 9, pp. 1011–1013, 2008.

[60] D. Chapman and L. P. Kaelbling, "Input generalization in delayed reinforcement learning: An algorithm and performance comparisons," in *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, vol. 91, pp. 726–731, 1991.

[61] L. D. Pyeatt and A. E. Howe, "Decision tree function approximation in reinforcement learning," in *Proceedings of the 3rd International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models*, vol. 2, pp. 70–77, 2001.

[62] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[63] K. Tuyls, S. Maes, and B. Manderick, "Reinforcement learning in large state spaces," in *Robot Soccer World Cup*, pp. 319–326, 2002.

[64] W. T. Uther and M. M. Veloso, "Ttree: Tree-based state generalization with temporally abstract actions," in *Adaptive agents and multi-agent systems*, pp. 260–290, Springer, 2003.

[65] R. E. Parr and S. Russell, *Hierarchical control and learning for Markov decision processes*. University of California, Berkeley Berkeley, CA, 1998.

[66] P. Marbach, O. Mihatsch, M. Schulte, and J. N. Tsitsiklis, "Reinforcement learning for call admission control and routing in integrated service networks," in *Advances in Neural Information Processing Systems*, pp. 922–928, 1998.

[67] S. J. Bradtke and M. O. Duff, "Reinforcement learning methods for continuous-time markov decision problems," in *Advances in Neural Information Processing Systems*, pp. 393–400, 1995.

[68] K. Driessens, J. Ramon, and H. Blockeel, "Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner," in *Proceedings of the 12th European Conference on Machine Learning*, pp. 97–108, 2001.

[69] J. Ramon, K. Driessens, and T. Croonenborghs, "Transfer learning in reinforcement learning problems through partial policy recycling," in *Proceedings of the 18th European Conference on Machine Learning*, 2007.

[70] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning," *Journal of Machine Learning Research*, vol. 6, no. Apr, pp. 503–556, 2005.

[71] U. D. Gupta, E. Talvitie, and M. Bowling, "Policy tree: Adaptive representation for policy gradient," in *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI-15)*, pp. 2547–2553, 2015.

[72] R. Akrour, D. Tateo, and J. Peters, "Towards reinforcement learning of human readable policies," in *The European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases: The 1st Workshop on Deep Continuous-Discrete Machine Learning*, 2019.

[73] R. Akrour, D. Tateo, and J. Peters, "Reinforcement learning from a mixture of interpretable experts," *arXiv preprint arXiv:2006.05911*, 2020.

[74] D. Hein, S. Udluft, and T. A. Runkler, "Interpretable policies for reinforcement learning by genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019.

[75] M. Lechner, R. M. Hasani, and R. Grosu, "Interpretable neuronal circuit policies for reinforcement learning environments," *Proceedings of the 27th International Joint*

Conference on Artificial Intelligence Workshop on Explainable Artificial Intelligence, 2018.

[76] F. Maes, R. Fonteneau, L. Wehenkel, and D. Ernst, "Policy search in a space of simple closed-form formulas: Towards interpretability of reinforcement learning," in *Proceedings of the 15th International Conference on Discovery Science*, pp. 37–51, 2012.

[77] D. Hein, A. Hentschel, T. Runkler, and S. Udluft, "Particle swarm optimization for generating interpretable fuzzy reinforcement learning policies," *Engineering Applications of Artificial Intelligence*, vol. 65, pp. 87 – 98, 2017.

[78] J. Huang, P. P. Angelov, and C. Yin, "Interpretable policies for reinforcement learning by empirical fuzzy sets," *Engineering Applications of Artificial Intelligence*, vol. 91, p. 103559, 2020.

[79] O. Z. Khan, P. Poupart, and J. P. Black, "Minimal sufficient explanations for factored markov decision processes.," in *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, 2009.

[80] M. Zakershahrak, Z. Gong, N. Sadassivam, and Y. Zhang, "Online explanation generation for human-robot teaming," *arXiv preprint, arXiv:1903.06418*, 2019.

[81] S. Sreedharan, S. Srivastava, and S. Kambhampati, "Hierarchical expertise level modeling for user specific contrastive explanations," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pp. 4829–4836, 2018.

[82] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "Pddl-the planning domain definition language," 1998.

[83] M. van der Meer, M. Pirotta, and E. Bruni, "Exploiting language instructions for interpretable and compositional reinforcement learning," *arXiv preprint, arXiv:2001.04418*, 2020.

[84] U. Ehsan, B. Harrison, L. Chan, and M. Riedl, "Rationalization: A neural machine translation approach to generating natural language explanations," *Proceedings of the 1st AAAI/ACM Conference on Artificial Intelligence, Ethics, and Society*, 2018.

[85] B. Hayes and J. A. Shah, "Improving robot controller transparency through autonomous policy explanation," in *Proceedings of the 2017 12th ACM/IEEE International Conference on Human-Robot Interaction (HRI*, pp. 303–312, IEEE, 2017.

[86] S. Rosenthal, S. P. Selvaraj, and M. M. Veloso, "Verbalization: Narration of autonomous robot experience," in *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pp. 862–868, 2016.

[87] J. van der Waa, J. van Diggelen, K. v. d. Bosch, and M. Neerincx, "Contrastive explanations for reinforcement learning in terms of expected consequences," in *Proceedings of*

the 27th International Joint Conference on Artificial Intelligence Workshop on Explainable Artificial Intelligence, 2018.

[88] L. Weitkamp, E. van der Pol, and Z. Akata, "Visual rationalizations in deep reinforcement learning for Atari games," in *Proceedings of the 30th Benelux Conference on Artificial Intelligence*, pp. 151–165, 2018.

[89] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint, arXiv:1312.6034*, 2013.

[90] W. Samek, T. Wiegand, and K.-R. Müller, "Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models," *arXiv preprint, arXiv:1708.08296*, 2017.

[91] R. Iyer, Y. Li, H. Li, M. Lewis, R. Sundar, and K. Sycara, "Transparency and explanation in deep reinforcement learning neural networks," in *Proceedings of the 1st AAAI/ACM Conference on Artificial Intelligence, Ethics, and Society*, 2018.

[92] T. Huber, D. Schiller, and E. André, "Enhancing explainability of deep reinforcement learning through selective layer-wise relevance propagation," in *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pp. 188–202, 2019.

[93] S. Greydanus, A. Koul, J. Dodge, and A. Fern, "Visualizing and understanding atari agents," in *Proceedings of the 35th International Conference on Machine Learning*, pp. 1792–1801, 2018.

[94] V. Goel, J. Weng, and P. Poupart, "Unsupervised video object segmentation for deep reinforcement learning," in *Advances in Neural Information Processing Systems*, pp. 5683–5694, 2018.

[95] A. Dethise, M. Canini, and S. Kandula, "Cracking open the black box: What observations can tell us about reinforcement learning agents," in *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pp. 29–36, 2019.

[96] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 1135–1144, ACM, 2016.

[97] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 197–210, 2017.

[98] A. Atrey, K. Clary, and D. Jensen, "Exploratory not explanatory: Counterfactual analysis of saliency maps for deep rl," in *Proceedings of the 8th International Conference on Learning Representations*, 2020.

[99] W. Shi, Z. Wang, S. Song, and G. Huang, "Self-supervised discovering of causal features: Towards interpretable reinforcement learning," *arXiv preprint, arXiv:2003.07069*, 2020.

[100] A. Mott, D. Zoran, M. Chrzanowski, D. Wierstra, and D. J. Rezende, "Towards interpretable reinforcement learning using attention augmented agents," in *Advances in Neural Information Processing Systems*, pp. 12329–12338, 2019.

[101] Z. Yang, S. Bai, L. Zhang, and P. H. Torr, "Learn to interpret Atari agents," *arXiv preprint arXiv:1812.11276*, 2018.

[102] X. Wang, Y. Chen, J. Yang, L. Wu, Z. Wu, and X. Xie, "A reinforcement learning framework for explainable recommendation," in *Proceedings of the 2018 IEEE International Conference on Data Mining (ICDM)*, pp. 587–596, 2018.

[103] Y. Tang, D. Nguyen, and D. Ha, "Neuroevolution of self-interpretable agents," *arXiv preprint, arXiv:2003.08165*, 2020.

[104] R. M. Annasamy and K. Sycara, "Towards better interpretability in deep q-networks," in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI-19)*, vol. 33, pp. 4561–4569, 2019.

[105] S. Jain and B. C. Wallace, "Attention is not explanation," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 3543–3556, 2019.

[106] M. L. Olson, L. Neal, F. Li, and W.-K. Wong, "Counterfactual states for atari agents via generative deep learning," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence Workshop on Explainable Artificial Intelligence*, 2019.

[107] C. Rupprecht, C. Ibrahim, and C. J. Pal, "Finding and visualizing weaknesses of deep reinforcement learning agents," in *Proceedings of the 8th International Conference on Learning Representations*, 2020.

[108] T. M. Mitchell and S. B. Thrun, "Explanation-based neural network learning for robot control," in *Advances in Neural Information Processing Systems*, pp. 287–294, 1993.

[109] A. L. Strehl, C. Diuk, and M. L. Littman, "Efficient structure learning in factored-state mdps," in *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pp. 645–650, 2007.

[110] J. Chen, S. E. Li, and M. Tomizuka, "Interpretable end-to-end urban autonomous driving with latent deep reinforcement learning," *arXiv preprint, arXiv:2001.08726*, 2020.

[111] P. Madumal, T. Miller, L. Sonenberg, and F. Vetere, "Distal explanations for explainable reinforcement learning agents," *arXiv preprint, arXiv:2001.10284*, 2020.

[112] D. Dannenhauer, M. W. Floyd, M. Molineaux, and D. W. Aha, "Learning from exploration: Towards an explainable goal reasoning agent," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence Workshop on Adaptive Learning Agents*, 2018.

[113] T. Degris, O. Sigaud, and P.-H. Wuillemin, "Learning the structure of factored markov decision processes in reinforcement learning problems," in *Proceedings of the 23rd International Conference on Machine Learning*, pp. 257–264, 2006.

[114] T. Hester, M. Quinlan, and P. Stone, "Generalized model learning for reinforcement learning on a humanoid robot," in *Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2369–2374, 2010.

[115] F. Cruz, R. Dazeley, and P. Vamplew, "Memory-based explainable reinforcement learning," *AI 2019: Advances in Artificial Intelligence*, 2019.

[116] P. Madumal, T. Miller, L. Sonenberg, and F. Vetere, "Explainable reinforcement learning through a causal lens," in *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI-20)*, 2020.

[117] A. Anderson, J. Dodge, A. Sadarangani, Z. Juozapaitis, E. Newman, J. Irvine, S. Chattopadhyay, A. Fern, and M. Burnett, "Explaining reinforcement learning to mere mortals: An empirical study," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 2019.

[118] H. Zhan and Y. Cao, "Relationship explainable multi-objective reinforcement learning with semantic explainability generation," *arXiv preprint, arXiv:1909.12268*, 2019.

[119] B. Beyret, A. Shafti, and A. A. Faisal, "Dot-to-dot: Explainable hierarchical reinforcement learning for robotic manipulation," in *Proceedings of the 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[120] G. Dao, I. Mishra, and M. Lee, "Deep reinforcement learning monitor for snapshot recording," in *Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 591–598, IEEE, 2018.

[121] O. Gottesman, J. Futoma, Y. Liu, S. Parbhoo, L. A. Celi, E. Brunskill, and F. Doshi-Velez, "Interpretable off-policy evaluation in reinforcement learning by highlighting influential transitions," *arXiv preprint, arXiv:2002.03478*, 2020.

[122] S. Huang, D. Held, P. Abbeel, and A. Dragan, "Enabling robots to communicate their objectives," in *Autonomous Robots*, pp. 309–326, 2019.

[123] D. Amir and O. Amir, "Highlights: Summarizing agent behavior to people," in *Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems*, pp. 1168–1176, International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[124] S. H. Huang, K. Bhatia, P. Abbeel, and A. D. Dragan, "Establishing appropriate trust via critical states," in *Proceedings of the 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3929–3936, IEEE, 2018.

[125] P. Sequeira and M. Gervasio, "Interestingness elements for explainable reinforcement learning: Understanding agents' capabilities and limitations," *arXiv preprint arXiv:1912.09007*, 2019.

[126] I. Lage, D. Lifschitz, F. Doshi-Velez, and O. Amir, "Exploring computational user models for agent policy summarization," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 2019.

[127] A. Koul, S. Greydanus, and A. Fern, "Learning finite state representations of recurrent policy networks," *arXiv preprint, arXiv:1811.12530*, 2018.

[128] T. Zahavy, N. Ben-Zrihem, and S. Mannor, "Graying the black box: Understanding dqns," in *Proceedings of the 33rd International Conference on Machine Learning*, 2016.

[129] S. Sreedharan, S. Srivastava, and S. Kambhampati, "Tldr: Policy summarization for factored ssp problems using temporal abstractions," 2020.

[130] L. Li, T. J. Walsh, and M. L. Littman, "Towards a unified theory of state abstraction for mdps," in *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics*, 2006.