Contents

List of Tables ................................................................. vi
List of Figures ............................................................... viii
Abstract ........................................................................ xi
1. Introduction ................................................................. 1
   1.1 Contributions .......................................................... 3
   1.2 Thesis Organization .................................................... 4
2. State of the Art ............................................................. 5
   2.1 Learning from Demonstration ........................................ 5
     2.1.1 Open Problems in Learning from Demonstration .......... 5
   2.2 Common LfD Algorithms .............................................. 8
     2.2.1 Supervised Learning ................................................. 8
     2.2.2 Similarity-Based Methods ......................................... 9
     2.2.3 Programming by Demonstration ................................ 10
     2.2.4 Structured Prediction ............................................... 10
     2.2.5 Learning Goal Hierarchies ........................................ 11
     2.2.6 Reinforcement Learning .......................................... 11
     2.2.7 Inverse Reinforcement Learning ............................... 12
   2.3 Learning from Human Demonstrators ............................. 14
     2.3.1 Markovian-Type Approaches ..................................... 14
     2.3.2 Approaches Designed for Robotics ............................. 14
     2.3.3 Other Approaches .................................................. 15
   2.4 Capturing the Sequential Nature of Data ......................... 15
     2.4.1 Standard Approaches .............................................. 15
     2.4.2 Rule-Based Approaches .......................................... 17
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.3</td>
<td>Similarity-based Approaches</td>
<td>18</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Other Approaches</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td>Internal State of the Demonstrator</td>
<td>20</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Do Not Consider Internal State</td>
<td>20</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Demonstration Annotation</td>
<td>21</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Direct Inference</td>
<td>22</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Markov Inference</td>
<td>22</td>
</tr>
<tr>
<td>2.6</td>
<td>Evaluation Metrics</td>
<td>23</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Domain Specific</td>
<td>23</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Multi-Agent Methods</td>
<td>24</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Generalized Methods</td>
<td>25</td>
</tr>
<tr>
<td>2.7</td>
<td>Active Learning from Demonstration (ALfD)</td>
<td>25</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Uncertainty-Based Approaches</td>
<td>25</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Iterative Approaches</td>
<td>26</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Other Approaches</td>
<td>26</td>
</tr>
<tr>
<td>3.</td>
<td>Application Domains</td>
<td>27</td>
</tr>
<tr>
<td>4.</td>
<td>Similarity-Based LfD</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>State Representation</td>
<td>33</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Trace Collection</td>
<td>33</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Experimental Evaluation</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Similarity Assessment</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>Feature Selection and Sequentiality</td>
<td>37</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Notation</td>
<td>38</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Wrappers</td>
<td>39</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Filter-Wrappers</td>
<td>41</td>
</tr>
<tr>
<td>4.3.4</td>
<td>NSFS</td>
<td>41</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Time Windows</td>
<td>42</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>4.3.6 Experimental Setup</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>4.3.7 Results (Minecraft)</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>4.3.8 Results (Super Mario and Thermometers)</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>4.3.9 Feature Selection Conclusions</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>4.4 Conclusions</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>5. Active Learning</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>5.1 Overview</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>5.2 The SALT Algorithm</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>5.3 Strategies</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>5.3.1 $\rho_s$ Strategy</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>5.3.2 $\rho_d$ Strategy</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>5.4 Experimental Setup</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>5.5 Results</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>5.6 Reward-Free SALT Strategies</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>5.6.1 Reward-free SALT strategies</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>5.6.2 Reward-Free Strategies Versus Reward-Requiring Strategies</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>5.6.3 Reward-Free Strategies Versus Baselines</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>5.7 Conclusions</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>6. Understanding SALT</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>6.1 Examination of Duplicated Data</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>6.1.1 Experimental Setup</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>6.1.2 Results - J48</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>6.1.3 Results - k-nn</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>6.1.4 Removing Duplicate Data</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>6.1.5 Summary</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>6.2 Distribution Comparisons</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>6.2.1 Experimental Setup</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Results</td>
<td>66</td>
</tr>
<tr>
<td>6.3</td>
<td>Conclusions</td>
<td>67</td>
</tr>
<tr>
<td>7</td>
<td>Towards Human Demonstrators</td>
<td>68</td>
</tr>
<tr>
<td>7.1</td>
<td>Demonstrator Budget</td>
<td>68</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Budget-Aware Strategies</td>
<td>69</td>
</tr>
<tr>
<td>7.2</td>
<td>Experimental Evaluation</td>
<td>69</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Results</td>
<td>70</td>
</tr>
<tr>
<td>7.3</td>
<td>Conclusions</td>
<td>73</td>
</tr>
<tr>
<td>8</td>
<td>Learning from Humans using SALT</td>
<td>74</td>
</tr>
<tr>
<td>8.1</td>
<td>Study Setup</td>
<td>74</td>
</tr>
<tr>
<td>8.2</td>
<td>Study Results</td>
<td>76</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Mental Effort</td>
<td>77</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Enjoyability</td>
<td>77</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Perceived Learning</td>
<td>77</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Reason for Ending Training</td>
<td>77</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Boards Trained</td>
<td>78</td>
</tr>
<tr>
<td>8.2.6</td>
<td>Actual Learning</td>
<td>78</td>
</tr>
<tr>
<td>8.3</td>
<td>Conclusions</td>
<td>78</td>
</tr>
<tr>
<td>9</td>
<td>Conclusions</td>
<td>79</td>
</tr>
<tr>
<td>9.1</td>
<td>Contributions</td>
<td>79</td>
</tr>
<tr>
<td>9.2</td>
<td>Future Work</td>
<td>80</td>
</tr>
<tr>
<td>10</td>
<td>Appendix A - List of Publications</td>
<td>82</td>
</tr>
<tr>
<td>10.1</td>
<td>Accepted Peer-Reviewed Papers</td>
<td>82</td>
</tr>
<tr>
<td>10.2</td>
<td>Pending Peer-Reviewed Papers</td>
<td>82</td>
</tr>
<tr>
<td>Bibliography</td>
<td>83</td>
<td></td>
</tr>
</tbody>
</table>
List of Tables

2.1 An Example of Learning Agent and Demonstrator Moves, Where the Demonstrator’s Policy Involves Randomness ................................................................. 23

4.1 Predicates that are discretized (or transformed from absolute to relative) and the resulting predicates. Note that this table only includes the predicates which are discretized. .......................... 34

4.2 Average loss for various algorithms and loss functions, evaluated using a one-vs-one procedure (lower is better). Results which are statistically significantly better than the rest are underlined. ................................................................................ 35

4.3 Average loss for various algorithms and loss functions, evaluated using a leave-one-out procedure (lower is better). Results which are statistically significantly better than the rest are underlined. ................................................................................ 35

4.4 Average loss for various feature selection methods (lower is better) for various time window sizes (k). Experiments were done using an average of 6,507 training instances. .......................... 43

4.5 Average loss of NSFS for varying iterations (lower is better), and approximate number of similarity measure calls needed to perform feature selection for a single test. Experiments were done using an average of 6,507 training instances. ................................................................. 44

4.6 Average loss (lower is better) and required number of calls to the similarity measure during feature selection. ................................................................................ 44

4.7 Average task reward for various feature selection methods for various time window sizes (k), for the Simple Thermometers domain. Experiments were done using 1,000 training instances. ................................................................................ 45

4.8 Average task reward of NSFS for time windows of size k = 1 and for varying numbers of iterations (lower is better), within the Simple Thermometers domain. Experiments were done using 1,000 training instances. ................................................................................ 45

4.9 Average task reward for various feature selection methods for various time window sizes (k), for the Complex Thermometers domain. Experiments were done using 1,000 training instances. ................................................................................ 46

4.10 Average task reward of NSFS for time windows of size k = 1 and for varying numbers of iterations (lower is better), within the Complex Thermometers domain. Experiments were done using 1,000 training instances. ................................................................................ 46

4.11 Average task reward for various feature selection methods for various time window sizes (k), for the Super Mario Domain. Experiments were done using 1,000 training instances. ................................................................................ 46

4.12 Average task reward of NSFS for time windows of size k = 1 and for varying numbers of iterations (lower is better), within the Super Mario domain. Experiments were done using 1,000 training instances. ................................................................................ 46
6.1 This table shows the average proportion of unique instances in the training data (using J48 as an underlying learner) for each method and each domain (Simple Thermometers, Complex Thermometers, Super Mario, and averaged across all domains, in that column order). Lower numbers indicate that there are more duplicated instances in the training data. .................................................. 58

6.2 This table shows the average proportion of unique instances in the training data (using k-nn as an underlying learner) for each method and each domain (Simple Thermometers, Complex Thermometers, Super Mario, and averaged across all domains, in that column order). Lower numbers indicate that there are more duplicated instances in the training data. .................................................. 59

7.1 Average number of demonstrator calls (“Context-Switches”), and the proportion of states in which the demonstrator either has to control or relabel for various methods and domains (“Proportion of States Requiring Actions”, or “PSRA”), averaged over both underlying learners. ................................................................. 73

8.1 Aggregate results from the post-study questionnaire. This table shows how many people stopped training each algorithm, categorized by the general reason they stopped. ............ 76

8.2 Aggregate results from the post-study questionnaire. This table shows the average user rating for each algorithm for each category, along with the results of a 2-tailed paired t-test statistical significance test. .................................................. 77
List of Figures

1.1 Left: An image of DOOM, an early First Person Shooter game. Right: An image of Quake, another early First Person Shooter game. ................................. 1

1.2 Left: An image of the first person shooter game Half Life 2. Right: An image of the popular soccer game FIFA 18. ................................................................. 2

1.3 Left: An image of Alien Isolation, a horror game where the main antagonist adapts to your play style. Right: An image of the adaptive horror game Hello Neighbor. ... 2

2.1 Illustration of a demonstrator steering a learning agent back into the set of states for which it has training data. Red represents that the agent has moved out of the training space, the solid line represents that the agent is controlling, and the dotted line represents that the demonstrator has taken back control. ........................................... 6

2.2 A simplified example of the flight controller decision tree. ................................. 8

2.3 Illustration showing the relation between Reinforcement Learning (RL), which derives a policy given a reward function; Inverse Reinforcement Learning (IRL), which derives a reward function from either a policy or a set of demonstrations; and Learning from Demonstration (LfD), which derives a policy from a set of demonstrations. ............................... 13

2.4 Example of Sliding Window(left) and Recurrent Sliding Window (right). The red arrows represent the action(s) from the previous timestep being used as part of the state data for the next timestep. ...................................................... 16

2.5 The Graph Transfer Network architecture, containing two graph transformers. .... 17

2.6 An example of a user-recorded program in SmallStar. This program opens the Negotiations folder, moves a single file in it onto the desktop, and closes the folder again. ... 22

2.7 The Simulated Car-Driving Domain. ................................................................. 24

3.1 A screenshot of Minecraft. .................................................................................. 28

3.2 An example world state and corresponding action. .............................................. 29

3.3 A screenshot of Super Mario. ............................................................................. 29

3.4 A screenshot of the Thermometers puzzle game. .................................................. 30

4.1 An excerpt of the taxonomy of concepts used in the domain of Minecraft. ........ 37

4.2 An example world state and corresponding action from the Minecraft domain. ... 38

5.1 Illustration of how the the strategies used in SALT govern its behavior. ............. 49
5.2 The left graph shows the amount of reward gained in Super Mario (vertical axis) as a
function of the amount of training data (horizontal axis) for various SALT strategies and
baselines, and the right graph shows the percentage of constraints satisfied (vertical axis)
as a function of the amount of training data (horizontal axis) for 5x5 Thermometers grids,
for various SALT strategies and baselines. Note the lines are different lengths because
all methods are ran for the same number of iterations, but some query the demonstrator
more and therefore gather more training data. ........................................ 52

5.3 The left column of graphs shows the amount of reward gained in Super Mario (vertical
axis) as a function of the amount of training data (horizontal axis) for various SALT
strategies and baselines, and the right column of graphs shows the percentage of con-
straints satisfied (vertical axis) as a function of the amount of training data (horizontal
axis) for 5x5 Thermometers grids, for various SALT strategies and baselines. Note that
for graph c, some of the strategies gave identical results, so some of the lines perfectly
overlap. ................................................................................................. 53

5.4 Task reward (y-axis) plotted against amount of training data (x-axis), for various learning
methods. Note both $\rho_s$ and $\rho_d$ are stated for each SALT variant, and that dotted lines
represent SALT variants that require a reward function. The left side (a,b,c) compares
various SALT variants against each other, and the right side (d,e,f) compares SALT
against our baselines. .................................................................................. 55

6.1 Task reward (y-axis) plotted against amount of training data (x-axis), for various learning
methods, using k-nn as an underlying learner. Note both $\rho_s$ and $\rho_d$ are stated for SALT
and that all methods were ran for the same amount of iterations. Since some methods
gain much less training data than others, this results in some lines being much shorter
than others. ................................................................................................. 60

6.2 Reward gained in the Simple Thermometers domain with J48 as an underlying learner
(vertical axis) as a function of the amount of training data (horizontal axis), when keeping
in or removing duplicates from the training data for various learning methods. ........ 61

6.3 Reward gained in the Complex Thermometers domain with J48 as an underlying learner
(vertical axis) as a function of the amount of training data (horizontal axis), when keeping
in or removing duplicates from the training data for various learning methods. ........ 62

6.4 Reward gained in the Super Mario domain with J48 as an underlying learner (vertical
axis) as a function of the amount of training data (horizontal axis), when keeping in or
removing duplicates from the training data for various learning methods. ............ 62

6.5 Reward gained in the Simple Thermometers domain with k-nn as an underlying learner
(vertical axis) as a function of the amount of training data (horizontal axis), when keeping
in or removing duplicates from the training data for various learning methods. ........ 63

6.6 Reward gained in the Complex Thermometers domain with k-nn as an underlying learner
(vertical axis) as a function of the amount of training data (horizontal axis), when keeping
in or removing duplicates from the training data for various learning methods. ........ 63

6.7 Reward gained in the Super Mario domain with k-nn as an underlying learner (vertical
axis) as a function of the amount of training data (horizontal axis), when keeping in or
removing duplicates from the training data for various learning methods. ............ 64
6.8 Average expected probability for a state from $D_t$ to be in the distribution created by $D_t$ (y-axis) versus training data (x-axis). Higher values provide evidence that $D_t$ and $D_i$ are closer together. ................................................................. 66

6.9 An illustration of four boards in the Complex Thermometers state space. Each colored set of circles represents one trace, with each circle being one training instance. The traces start in the bottom right and expand roughly towards the top left as time proceeds. . . 67

7.1 Reward gained in Super Mario and each Thermometers domain as a function of the amount of training data (horizontal axis) for various strategies and baselines. The vertical axis represents the amount of reward gained (in the case of Super Mario) or the % of constraints satisfied (in the case of both Thermometers domains) The top row are results gained using a global budget, and the bottom row are results gained splitting the budget up evenly over iterations. ................................................................. 71

7.2 Percentage of time that the learner’s chosen move matched the move the demonstrator would of made during validation (vertical axis) as a function of the amount of training data (horizontal axis) for various strategies and baselines. The top row are results gained using a global budget, and the bottom row are results gained splitting the budget up evenly over iterations. ................................................................. 72

8.1 A bar plot showing the reported programming experience of users participating in the study. A rating of 0 means the user had no programming knowledge at all, and a rating of 5 means they believed they were an expert programmer. ................................. 75

8.2 Task reward gained in the Simple Thermometers domain (vertical axis) as a function of the amount of training data (horizontal axis) for SALT and DAGger. As different demonstrators played for different numbers of iterations, a 2-value simple moving average was taken to smooth the curves. ................................................................. 76
Learning from demonstration (LfD) is a branch of machine learning that focuses on learning how to perform a given task by observing an demonstrator perform one or several demonstrations of it. Moreover, many of the current LfD techniques assume a large pool of training examples from which to learn. The long term goal of our research is to develop general LfD methods which can more easily learn from human demonstrators than state-of-the-art methods, through requiring less training data and more carefully selecting when to obtain more training data. The main contribution of this thesis is a novel Active Learning from Demonstration algorithm called SALT (Selective Active Learning from Traces), an algorithm which can match or outperform other state-of-the-art algorithms in multiple domains, while using less training data. We have also gathered evidence that human demonstrators find it preferable to another state-of-the-art LfD algorithm, that it is less mentally burdensome for them to train, and that it learns better from them.
Chapter 1: Introduction

Learning from demonstration (LfD) is a branch of machine learning that focuses on learning how to perform a given task by observing an demonstrator perform one or several demonstrations of it. An LfD agent operates by first garnering training data from a demonstrator, and then using that data to attempt to learn to accomplish the same task.\footnote{4}

The field of LfD has been intensively studied over the past 30-40 years with the goal of replicating humans’ ability to learn complex tasks from observation. Work in this area has been reported under the labels of Learning from Observation, Imitation Learning, Behavioral Cloning, Apprenticeship Learning, or Programming by Demonstration, which are largely synonymous with LfD. Although some of them make special emphasis in some specific approach to LfD (for example, Learning from Observation emphasizes learning from pure observation without any form of interaction with the demonstrator), the overall goal is to learn to replicate behavior via observing the performance of another agent.

Applications of LfD range from controlling a car or airplane\footnote{4} to creating robot controllers\footnote{3} to providing medical information such as the location of tumors\footnote{58}. Another important application domain is that of computer games. Examples of this are real-time strategy games such as learning to play Starcraft\footnote{71,75,118}, Atari games such as Asteroids, Pitfall, or Road Runner\footnote{102}, and racing games such as Super Tux Cart\footnote{96} (A Mario Cart clone). Currently, AI in games is usually coded manually (and often from scratch\footnote{28}), which is known to be a significant bottleneck (for example, the 20-minute experience of Facade took two man-years to hand-craft the AI of the game’s two Non-Player Characters\footnote{55}). This is because many games are just too unique to easily reuse representations or code from other games\footnote{28}. Although this has began to change, the idea of a generalized AI “toolkit” is still heavily sought after.

LfD has the potential to provide this toolkit by allowing developers to simply demonstrate the behavior that they wish the AI to have, and having the AI learn that behavior automatically. This would enable the crafting of Non-Player Characters (such as townspeople, allies, and enemies) without requiring programming knowledge - by simply performing the actions that are desired of the NPCs.

Additionally, being able to author AI’s by demonstrating behavior also has the potential to lead to more complex AI behavior. Take the domain of First-Person Shooters as an example. Many of the early games in this genre (such as DOOM or Quake, shown in Figure 1.1) have enemies with extremely limited AI: When they sense the player, they walk towards them firing their weapon. As

![Figure 1.1: Left: An image of DOOM, an early First Person Shooter game. Right: An image of Quake, another early First Person Shooter game.](image_url)
the genre advanced, games such as Half-Life and Half-Life 2, pictured in Figure 1.2 (Left), which can give the illusion of a smart AI via finite-state-machines or expert systems were created\textsuperscript{30}. However, even though game AI has advanced dramatically since these games, advanced players are still able to figure out how to exploit the AI in many games. One example of this is players using the weaknesses of AI to farm FIFA coins in the popular soccer game FIFA 18\textsuperscript{20}, pictured in Figure 1.2 (Right).

This exploit helps to demonstrate that as behaviors get more complex they also become more difficult to successfully script without errors in the AI’s behavior. Using LfD in the AI of enemies in games has the potential to create very complex behaviors relatively easily, making games both more challenging and more interesting. This idea has been presented in some games already. For example, in Alien Isolation, if you hide from the alien in lockers too often, he will start checking lockers more. If you use the flamethrower to ward him off too often, it will become less effective. In fact, the creators of Alien Isolation, pictured in Figure 1.3 (Left), claim the alien to be “almost sentient”\textsuperscript{18}. Another example of this can be found in Hello Neighbor, pictured in Figure 1.3 (Right), where the player has to avoid an antagonist that learns how the player moves throughout a house\textsuperscript{62}. The field of LfD has the potential to make creating advanced AIs like this much easier, allowing the creation of AI by simply performing the desired behaviors. Furthermore, this same notion can be applied to serious/educational games, either allowing the crafting of tutors without needing to program them directly\textsuperscript{41}, or by adapting to the human learner to help them more effectively learn\textsuperscript{69} (for example, by selecting future questions based on which questions the human learner has gotten correct so far, as a human teacher would).

LfD is also very common in humans\textsuperscript{35,100}. Among other strategies, humans commonly look to a demonstrator for information on how to perform a task. For example, a human might learn to perform an action a certain way by observing a demonstrator making the desired motions (such as when a parent teaches a child how to brush their teeth or wash their hands). In fact, game designers
often learn how to create AI in NPCs by observing a mentor (such as a teacher) creating NPC behavior. But what if instead of a game designer having to perform LfD themselves to learn how to script an AI, the game designer could simply demonstrate the desired behaviors and have the AI itself perform the LfD portion instead? This would allow for the much easier creation of AI and make the creation of AI more accessible to non-programmers (since there would be no programming required), goals which heavily motivate our work. Methods which allow learners to master complex behaviors from human demonstrators are an important step in achieving this goal.

However, one assumption that most current LfD approaches make is access to a large amount of training data, which is not always feasible. If LfD is to be used to solve behavior authoring, this would imply the human author would have to demonstrate the desired behavior an unreasonable number of times in order to generate enough training data (for example, 660 levels of Super Mario\textsuperscript{85}). In order to address this problem of LfD from humans, methods need to be derived which require less training data and less effort from the demonstrator, so that they may more effectively learn from human demonstrators.

To address this problem, we have developed a new LfD framework called \textit{SALT}, with the two-fold goal of 1. being able to learn from less training data and 2. being more feasible for learning from human demonstrators than state-of-the-art methods. We have also shown via empirical analysis that both \textit{SALT} learns better from human demonstrators than another state-of-the-art method and that human demonstrators prefer \textit{SALT} over the other method. This is the key contribution of our work, but all of our contributions are listed in detail below.

### 1.1 Contributions

The contributions of my work are five-fold:

- Deriving algorithms which can learn from less training data than other State-of-the-Art algorithms: As previously discussed, many LfD algorithms assume access to a large amount of training data. However, this is not always feasible, as in some domains we simply cannot gather a large amount of training data (such as teaching robots to complete a task by physically demonstrating that task\textsuperscript{8}). Furthermore, if we want to derive algorithms which are feasible for human demonstrators, we need to consider that they can only provide so much training data. Our algorithm, \textit{SALT}, has been shown to learn tasks more effectively with less training data in multiple domains.

- Deriving algorithms which are more feasible for human demonstrators than other State-of-the-Art algorithms: Similarly, most LfD algorithms are simply not designed with human demonstrators in mind, but rather for use with a synthetic demonstrator. However, there are factors that apply to human demonstrators that do not bother synthetic demonstrators (such as context-switching\textsuperscript{91}). For example, \textit{DAgger}, another Learning from Demonstration algorithm, stochastically switches between whether the demonstrator or the learning agent is controlling, which is fine for a synthetic demonstrator but may be jarring to a human demonstrator. As such, algorithms are needed which impose less cognitive burden on humans, or are otherwise more feasible for learning from them. \textit{SALT} has also been shown to be more feasible for human demonstrators: both by capturing cognitive burden metrics from synthetic demonstrators and in a user study comparing it to another State-of-the-Art algorithm.

- Demonstrator Budget: We have introduced the idea of a “demonstrator budget” for LfD domains, where the learner has a limit of how much it can request information from the demonstrator. This is shown to improve results in our domains, and can help devise LfD algorithms which can learn from a limited amount of data.

- Cognitive Burden Metrics: We have derived two metrics for determining how much cognitive burden is imposed on a demonstrator training a learner, and compare our algorithm and other state-of-the-algorithms in these metrics.
• A Study of Feature Selection, Distance Functions, and Time Window methods for LfD: We have performed a comparison of various feature selection methods (including one novel one), distance functions, and time window sizes for multiple LfD domains, showing which strategies appear to work better for LfD problems.

In summary, for my thesis dissertation I have created an Active LfD algorithm which has been shown to learn with less training data, better learn from human demonstrators, and produce learners which act more similar to the demonstrator than State-of-the-Art methods.

1.2 Thesis Organization

Chapter 2 reviews the open problems in LfD and the current State-of-the-Art progress towards addressing these problems. Specifically, we cover the need for specialized LfD algorithms, making algorithms which are feasible for learning from human demonstrators, capturing the sequential nature of LfD data, capturing the internal state of the demonstrator, and deriving evaluation metrics which accurately measure learning in LfD domains. Each problem is described and then work related to each problem in the literature is detailed.

Chapter 3 covers the various domains that have been used throughout our work. Specifically there are four domains: Minecraft, Super Mario, Simple-Thermometers, and Complex-Thermometers. Each domain is described in detail including what the state representation looks like and what actions are performable in that domain.

Chapter 4 covers our starting work with similarity based methods, including the representation of data. Also detailed are Similarity Assessment or how similarity between states is determined, automatically modifying the set of features used for learning, and a technique for handling data sequentiality. This is done to identify a baseline for how current techniques work and how much data they require.

Chapter 5 then describes our work in Active LfD, including introducing our novel algorithm SAL-T. SAL-T is an Active Learning from Demonstration algorithm which uses a set of strategies to determine when and for how long to query the demonstrator for more data. The various SAL-T strategies are also explained in detail and empirically evaluated against each other and other state-of-the-art methods.

A further analysis of SAL-T’s behavior is then provided in Chapter 6. Specifically, two aspects of the learners trained by each algorithm are empirically analyzed. The first is the amount of duplicated data in each algorithms dataset as well as what happens when duplicates are removed. The second is the comparison of the distributions of states encountered during training and testing.

Chapter 7 describes our work evaluating and improving SAL-T’s feasibility for human demonstrators. Specifically, one new modification to SAL-T is introduced: The idea of a demonstration query budget which determines how much data the learner can request from the demonstrator. Additionally, two metrics of cognitive burden are introduced and used to analyze SAL-T’s and two state-of-the-art baselines’ feasibility for human demonstrators.

Chapter 8 then details the results of a user study testing SAL-T on human demonstrators. This is done because the cognitive burden metrics were used with synthetic demonstrators, and it is crucial to see how feasible actual human demonstrators find SAL-T. This chapter details the study and compares SAL-T versus DAgger on feasibility for demonstrators according to 6 different factors.

Finally, Chapter 9 wraps up with overall conclusions. It first summarizes the niche our work fills, and then goes on to summarize our work, concluding that SAL-T is an important step in the direction of creating LfD algorithms which can feasibly learn from humans.
Chapter 2: State of the Art

In this chapter we introduce background concepts relevant to our research and that will be used throughout the rest of the document. The remainder of this section is structured as follows. First, we present a list of open research problems in LfD in Section 2.1, then give a description of some of the work being done in LfD in general in the context of those open research problems in Sections 2.2 through 2.6. Next, in Section 2.7 we discuss the work done in Active Learning, a specific subfield of machine learning. Finally, Section 2.8 describes current work in Learning from Human Demonstrators.

2.1 Learning from Demonstration

In the remainder of this work, we adopt the formalization proposed by Argall et al.\textsuperscript{4}: The world can be in one of a set of states \(S\), and at each time instant, the agent can perform one out of a set of actions \(A\). With the mapping between states by way of actions being defined by a probabilistic transition function \(F(s'|s,a) : S \times A \times S \rightarrow [0,1]\). It is assumed that the state is not fully observable. The learner instead has access to observed state \(Z\), through the mapping \(M : S \rightarrow Z\). A policy \(\pi : Z \rightarrow A\) selects actions based on observations of the world state. Actions can range from low-level motions to high-level behaviors. A demonstration \(T_i = [(z_1,a_1),\ldots,(z_n,a_n)]\) is defined as a sequence of observation-action pairs (called instances), where \(z_i \in Z\), and \(a_i \in A\). The goal of LfD is then, given training data consisting of a set of demonstrations \(T = \{T_1,\ldots,T_n\}\), to derive a policy which allows the learner to choose an action based on the current observed world state.

Although LfD and standard supervised learning share some features, their differences cause them to have a different set of research problems, as elaborated upon in Section 2.1.1. In this chapter we will focus on those open problems, giving a description of each as well as the work in the field that attempts to address them.

2.1.1 Open Problems in Learning from Demonstration

Work on LfD can be traced back to the early days of machine learning\textsuperscript{6}, and is still the subject of extensive research\textsuperscript{12,21}. However, despite the large body of work in this area, a number of key open challenges remain:

1. **Need for Specialized Algorithms**: Most past work on LfD\textsuperscript{4} assumes that the problem can be addressed using standard supervised algorithms. However, as formally proven by Ross and Bagnell\textsuperscript{93}, LfD violates one of the fundamental assumptions of supervised learning: examples are not independently and identically distributed (i.i.d. assumption). This happens because in LfD the next state for which the agent will need to predict an action depends on previous predictions (that is to say, the learned policy affects the future input). Intuitively, this is a problem because during training, the agent has been exposed to a collection of training examples that follow some given distribution. However, during performance, the prediction errors of the agent compound, and might make the agent visit states of the world that are farther and farther away from this distribution, thus making the prediction error grow even faster (basically because the agent reaches situations for which it has never seen any training example). Ross and Bagnell\textsuperscript{93} showed that, in practice, the result is that the accumulated error over a series of iterative predictions grows quadratically, rather than linearly. Ideally, once the agent has left the space of states for which it has seen training data, it should try to steer back into this “desirable state space” (see Figure 2.1 for a visual illustration of this notion). However, few current works address this issue. Furthermore, demonstrations of this recovery behavior can be very rare, depending on the problem being solved. An open problem
in the field is to design algorithms that can cope with this issue, help the learner recover from errors, and prevent their compounding.

2. **Feasibility of Learning from Human Demonstrators** Many existing algorithms for LfD often require a very large amount of training data – too much to be practical for human demonstrators. Active-learning LfD approaches tend to exacerbate the problem – not only does the demonstrator need to provide demonstrations, but they also need to be able to respond to queries that the learner makes. For example, in the experiments reported by Ross, Gordon, and Bagnell\textsuperscript{95}, DAgger required data from a demonstrator playing 660 Super Mario levels before the learner’s task performance plateaued. Therefore, specialized algorithms which can work with small amounts of training data are needed, as well as algorithms which impose less mental burden on human demonstrators in other ways.

3. **Sequential Nature of Data**: Tasks in some areas of machine learning (such as LfD) tend to be very sequential in nature, since the learned policy affects further input. That is to say, the actions taken in any given state do not depend purely on that state, but also on past states as well. As such, learning procedures which only take into account the current state often fail to capture the true nature of the data. Some work has been done which accounts for this sequentiality, but they do not completely solve the problem\textsuperscript{17}. For example, one technique for handling sequential data is by using time windows. By default, only the current state of the world is fed into an LfD algorithm. With time windows, however, instead of just the current state, the algorithm is given a fixed “window” of past states. Although this has provided decent results in several cases\textsuperscript{82;101}, there are a few issues with this approach. The first is that the larger the window size, the more features that have to be taken into account for learning. This quickly becomes intractable when learning real-world tasks, especially for those with long-range dependencies (when the action issued at a certain state by the demonstrator is affected by something that happened far prior). The other main issue is that even if it is reasonable to compute time windows for a given problem, how does one choose what size the time window needs to be? Too large of a window is wasteful and can greatly impair runtime, but too small of one might not capture enough of the dependencies of the data (note that capturing long-range dependencies is often incredibly difficult due to combinatorial explosion, so a window size must be chosen that captures as many dependencies as possible while retaining reasonable performance). Other approaches, such as Markov Model-based approaches, do exist, but also

![Figure 2.1: Illustration of a demonstrator steering a learning agent back into the set of states for which it has training data. Red represents that the agent has moved out of the training space, the solid line represents that the agent is controlling, and the dotted line represents that the demonstrator has taken back control.](image-url)
suffer from very limited scalability\textsuperscript{17}.

Therefore, one of the open problems in LfD is coming up with methods that are not only able to capture sequential data, but are also efficient enough to be used for real-world problems. In fact, many of the challenges involved in the addressing of sequential data on LfD are shared with other related areas such as sequential machine learning, where they are also still open challenges. This problem has seen a lot of recent progress, such as LSTMs (Long Short-Term Memory models)\textsuperscript{36}, but these approaches have limited applicability to LfD from humans since they require a lot of data to be trained.

4. **Internal State of the Demonstrator**: An internal state in an demonstrator can be thought of as a set of preferences, goals, etc. that are directly influencing the demonstrator’s actions, but are not observable nor explicit. In some domains, the demonstrator will perform actions based on an internal state. For example, a player in Minecraft might rearrange their inventory or craft extra tools in anticipation of future events, due to a non-explicit future goal in their mind. In these cases, something needs to be done in order to recreate this internal state (or else the agent will be missing key information in its efforts to accurately learn the behavior of the demonstrator)\textsuperscript{23}. Several techniques exist, such as capturing the internal state and making it part of the input or trying to infer the internal state as a hidden variable with machine learning. For example, some previous work has attempted to address this problem using the former method, either via action dependencies\textsuperscript{74} or annotations made by the demonstrator\textsuperscript{46}. In some cases, however, the demonstrator might not even be able to accurately describe their internal state.

5. **Evaluation Metrics**: An evaluation metric is the assessment that is used to compare the actions of the learned agent to those of the demonstrator (to evaluate the agent’s learning). In other forms of machine learning, such as supervised learning or reinforcement learning, evaluation metrics have been derived which are well-known and well-accepted. However, this is not the case in LfD, because measuring the success of an agent accomplishing tasks is not always enough to check that the agent learned the specific strategies that the demonstrator used\textsuperscript{73}. When learning from an demonstrator, we may wish for our agent to not only accomplish the task, but also accomplish it in the same way that the demonstrator is accomplishing it. This implies some measure of behavior similarity, which is a non-trivial task.

Adding to this difficulty is the fact that not all LfD algorithms will work for all problems. Therefore, a categorization of these behaviors is very important. One way to do so is to divide them into levels, where each level requires a different type of algorithm and possibly different evaluation metrics. Specifically, these behaviors can be split into three levels, with level one being the simplest, and level three being the most complex:

- **Level 1 - Strict imitation**: These behaviors do not require feedback from the environment, nor memory.
- **Level 2 - Reactive Behavior**: Markovian behaviors which correspond to input-to-action mappings.
- **Level 3 - Memory-based behavior**: Requires an internal state that represents memory and/or past states to learn.

However, no evaluation metric currently exists which can properly and accurately evaluate all of these categories. For level 1 behaviors, a direct comparison of the outputs might suffice, while higher levels will require more sophisticated measures. One strategy involves comparing the distribution of the outputs rather than the outputs themselves\textsuperscript{73}. However, very little work has been done in this area, and it is still an open problem in LfD\textsuperscript{85}.

Now that we have examined some of the open problems in the field of LfD, the next five sections will be dedicated to describing each one in detail.
2.2 Common LfD Algorithms

Over the past 30-40 years, a very large set of algorithms have been proposed to address LfD, from supervised learning to reinforcement learning-based approaches, to more specialized approaches\textsuperscript{6,31}. Many of these methods require a very large amount of training data. For example, the FullyConvLSTM algorithm presented by Vinyals et al.\textsuperscript{116} is experimented with in the Real-Time Strategy game of Starcraft II. Sampling at a rate of only once per 8 steps of the game, the reward (the in-game score) does not plateau until around 250,000,000 game steps (training instances). This section presents an overview of the major approaches, categorized by type.

2.2.1 Supervised Learning

A common supervised learning approach in LfD is learning decision trees. Decision trees are predictive models which map from the states to one or more actions. Depending on the situation, one might use a single decision tree that chooses the entire subset of actions the agent should perform or a series of trees that each choose whether the agent should perform a particular action. One could also use a decision tree to model each separate demonstrator, rather than one tree that attempts to model all of them, to better learn each demonstrator’s behavior.

An example of this is seen in the work of Sammut et al.\textsuperscript{98}, where they learn decision trees for the task of learning to pilot an airplane based on data from human airplane pilots. The algorithm they use for learning is C4.5\textsuperscript{83}. This tree represents a purely reactive policy, with concepts such as time being completely excluded. The states are the discretized, filtered data that is gained from preprocessing data recorded from the human pilots performing various flight tasks. The output is a decision tree with actions as the leaves, which can be employed to work the simulated aircraft’s controls, an example of which can be seen in Figure 2.2. When tested used a simulated airplane environment, they found that not only did the learner control the airplane as well as a human pilot, but in many cases it actually cleaned up inconsistencies and corrections that were present in the logs of the human pilots using the simulator.

Another supervised learning approach that has been applied to LfD is that of Neural Networks. For example, in the ALVINN network\textsuperscript{81}, an agent is taught to drive a van based on data from a demonstrator driving it. Specifically, a human demonstrator drives the van, and an image of the current road ahead is fed into the learner as input. Backpropogation is then used on-the-fly to try and learn what steering direction the human demonstrator would use given an image of the road. This method, however, runs the risk of repetitive inputs overwriting old learning, such as driving down a long straightaway making the learner forget how to handle turns. In order to address this problem, each image is processed to create 14 additional images, each of which is a lateral shift (as a rough example, think of how the road would look different from the driver’s seat versus the front passenger’s seat, and imagine 13 more of these viewpoints). The images are then all fed into the network as the input. These extra images greatly reduce the repetitiveness of the input, mostly solving the aforementioned issue. The ALVINN network was able to successfully steer the van in a
variety of situations from a single lane dirt road to a lines two-laned highway for up to half a mile, until it reached either a difficult intersection or the end of the road.

Another approach for using supervised learning for LfD is that of Gaussian Mixture Models (GMMs). An example of this comes from Chernova and Veloso\textsuperscript{13}, who use a set of GMMs to generate and represent the action policy. First they cluster all of the observations that lead to the same action together, and then create a separate Gaussian mixture for each cluster. They then use Expectation Maximization to learn the parameters for each mixture model. Classification of a new datapoint is then done by selecting the mixture (and therefore action class of that mixture) with the highest likelihood.

Some work has also been done using Locally Weighted Regression for LfD. For example, Ijspeert\textit{ et al.}\textsuperscript{37} focus on learning to perform complex but rhythmic patterns. They do this by sampling traces from the demonstrator performing a task, then modeling these problems using a nonlinear oscillator\textsuperscript{47}. These models are then used in a kernel solver to create rhythmic control strategies that attempt to perform the desired patterns.

Finally, Lau\textit{ et al.}\textsuperscript{52} proposed to use version space algebra. The idea is that, instead of immediately trying learn actions that transform complex objects into other complex objects, they build up a complex version space by composing together (using their algebra) version spaces which contain simpler functions. Their Programming by Demonstration system, SMARTedit, then leverages this version space in order to learn complex behaviors in a text-editing domain.

### 2.2.2 Similarity-Based Methods

Another common category of learning algorithms used for LfD are similarity-based methods. These are in fact also types of supervised learning, but they are quite common and therefore warrant being explained separately. Algorithms in this category classify states according to similarity, using the action corresponding to the most similar state as its prediction.

The most basic similarity-based method is $k$-nearest neighbor (kNN)\textsuperscript{15}, which can be defined as: given a new state $s$, for which we want to determine which action to perform, a nearest-neighbor algorithm works by finding the $k$ most similar states to $s$ in training data $T$, and an action is chosen from amongst those $k$ states (the method to do this varies, picking the first action on the list or picking the most common action are two possibilities). Some work that has been done in LfD using the kNN approach is the temporal backtracking algorithm presented by Floyd\textsuperscript{23} and teaching robots to play group soccer\textsuperscript{92}.

Another common similarity based method is that of case based reasoning (CBR)\textsuperscript{43,79}. Given a new problem, a typical CBR system uses a similarity measure to identify previous relevant cases, and then adapts the solutions in these retrieved cases to solve the problem at hand (in this case, picking a set of actions to execute). When applying CBR to LfD, there are therefore three key problems: how to transform demonstrations into cases, how to retrieve cases, and how to adapt the solutions in these retrieved cases. Strategies to address the former two will be expounded upon in Section 2.4.3.

There is not as much work related to the third problem, but an example of case adaptation in the context of LfD can be seen in the Darmok and Darmok2 systems\textsuperscript{70,72}. Darmok2 uses a couple of assumptions in order to efficiently adapt plans to the current situation. First, it is assumed that plan adaptation can be divided into 2 processes: parameter adaptation (via a domain independent method based on potential fields) and structure adaptation. The other major assumption is that no plan will undo what another plan does (for an example of the problems that are caused by plans undoing the work of other plans, the reader is referred to the well known Sussman Anomaly\textsuperscript{100}). These assumptions allow the Darmok2 system to maintain real-time performance even with plan adaptation (for more information on plan adaptation, the reader is referred to\textsuperscript{60}).

A broader application of CBR to LfD is the generalized framework proposed by Floyd and Esfandiari\textsuperscript{22}. This framework, named jLOAF, is intended to allow the development of agents from different domains and with different goals, and avoids hard-coding any domain knowledge. Although there are limitations to the framework, it is show than learners can successfully learn a variety of behaviors for four different domains.
Another important item to note about similarity based methods is that some strategies use a structured representation of data because it is a more natural encoding of the state data. This being the case, having similarity measures which operate on structured data has some importance. One such example of this that is not directly intended for LfD is RIBL. RIBL takes two factors into effect when computing similarities between two objects – their attributes and their relations to other objects. These factors then depend on those other objects’ attributes and relations, and so on. These relations are expounded upon to a user-specified depth, and then an analysis is performed of how often arguments appear together in the same position to get an overall similarity measure.

### 2.2.3 Programming by Demonstration

This category of algorithms separates itself from LfD in general by outputting this policy directly in the form of a program. One way to do this is to use generalization, as in the SYNTHESIZE algorithm by Bauer. Basically, this algorithm works by taking specific examples of a computation and combines them into a generalized procedure for that computation. This algorithm takes in a set of examples of performing a computation in specific cases in the form of c-trees which are data structures consisting of three components:

- **Name**: The name of the procedure that the c-tree describes
- **Input list**: The list of all the specific values which are used in the computation
- **An ordered, directed tree**: The nodes of this tree are the instructions used in the computation’s description.

It turns these c-trees into a generalized procedure that can be used for other cases. The basic idea is to find an id-cover for the examples, which is a partitioning of all the nodes in the computation trees (except for those nodes which are hypothesized to be added as introductory assignments) into classes. Classes are groupings such that nodes in the same class are ideally occurrences of the same instruction in the demonstrations (for example, if two different nodes both represented an "addition" instruction, they should be in the same class). An id-cover is found by performing a state-space search on the nodes where the starting point of this search is the set consisting of the leftmost node of each c-tree that involves a function, predicate, or procedure, and nodes are expanded starting with those that have the fewest classes. By then taking the least generalization of each class, the set of instructions in the new procedure are created. Finally, the variables are made consistent, and a parameter list is formed. This technique was tested by attempting to derive the general procedure for some common operations (multiplying two numbers, sorting an array, returning the roots of a quadratic equation, etc) given examples of that operation. All of the operations tested were able to be learned based on just a couple of examples, but operations which had many repeated instructions (such as recognizing strings of the form $0^n1^n$ using a Turing Machine) took significantly longer than those that did not. Other notable Programming by Demonstration approaches include automating common tasks via incremental development, learning to control robots to perform human-like tasks by the learning and encoding of specific skills, and controlling robots via conceptual or neural models of imitation learning.

### 2.2.4 Structured Prediction

Structured prediction problems are classification problems where there is a collection of output variables (rather than a single one as in standard supervised learning) which either constrained or dependent on one another. These interactions which manifest themselves as sequential, spatial, or combinatorial structures in the data. Structured prediction methods for LfD therefore attempt to capture these structures (especially ones related to sequentiality) and use them to boost learning.

Maximum Margin Planning starts by modeling problems as discrete Markov Decision Processes. The supplied solution trace is made to be more lucrative to the learner than worse policies by attempting to make it better than any other solution by a margin value (which scales appropriately with the loss of those other solutions). This notion then allows the problem to be formulated as
a quadratic program – that quadratic program is a representation of the Maximum Margin Planning problem. A modified subgradient method is then used to try and solve this problem. This approach was tested in an automated driving domain, and found that the learned behavior often matched the demonstrated behavior, and even when it did not the learned behavior follows rules implicitly provided in the demonstrations (for example, avoid buildings and favor roads over grassy areas).

An extension of MMP called LEARCh comes from Ratliff et al. LEARCh is an iterative algorithm which attempts to learn a cost function for which the demonstrator’s example path is the path with the least cost. It does this by suggesting local corrections to the cost function via attempting to increase the cost function along the agent’s path and decrease it along the demonstrator’s path, which forces the paths together. LEARCh traveled further to complete the same courses as a hand-tuned approach, but scored less total cost and was able to drive faster on average.

2.2.5 Learning Goal Hierarchies

Hierarchically decomposing the task that the LfD algorithm is learning into sub-tasks has been exploited for learning from demonstration for the purposes of giving a logical structuring to goals and strategies. The objective is to give structure to the learning approach, and split the learning goal into multiple subgoals, the achieving of which will achieve the overall goal.

For example, Könik and Laird create a relational learning by observation framework that uses a hierarchy of operators. This hierarchy allows for the automatic creation of an agent program that can achieve the overall goal of acting similar to an demonstrator by attempting to solve the smaller learning problems associated with each operator. In order to learn the desired behavior, the system uses an inductive logic programming algorithm, inverse entailment, to learn a theory. This theory represents the lower level decision concepts using examples from the training set, situational predicates from an episodic database which stores information about the lists of predicates that occur at each situation, and any hand-coded static domain knowledge. The search for a good theory is simplified by the operator hierarchy discussed before. First, any decisions about an operator are going to be learned in the context of its parent operator, and the parent operator’s conditions are also implicit conditions of the child operator. As such, the conditions themselves get simpler, and in some cases it might be sufficient to learn only the differences between sibling operators. Also, the parameters to the parent operator can help narrow down what state information is the most relevant, which also leads to a smaller state space. This simplifies learning greatly by guiding searching. At the end of the learning phase, learned concepts are compiled into an executable program. This approach was tested using a simulated building-navigation domain. When learning from correct positive and negative examples, this technique provided a hypothesis equivalent to the correct one in all 20 trials, provided there were sufficient examples (approximately 10 positive examples and 10 negative examples). Similar results were obtained for training from only correct positive examples and heuristic negative examples, but required many more examples (approximately 60-80 of each).

2.2.6 Reinforcement Learning

Many approaches based on reinforcement learning have been proposed to address LfD (such as the original work of Schaal). Standard reinforcement learning is a method of learning where the agent receives reward or punishment after the execution of each action according to some external function $R$. As the agent performs actions, it receives a reward/punishment based on $R$ and uses it to determine how desirable the resulting state is (a higher reward means a more desirable state). The agent attempts to pick actions in such a way as to maximize its accumulated reward over time, learning the desired policy as a result.

One example of using Reinforcement Learning for LfD comes from Bogdanovic et al., who try to imitate a human player in an Atari-games environment by making the environment record states (in the form of video frames) and actions (in the form of button presses). They then similarly convert each frame from a 210 x 160 128-color image to an 83 x 83 grayscale image. Experiments were done on the classic Freeway game, training on about 98% of the data and testing on a single gameplay episode. Their algorithm performs better than the SARSA algorithm (a temporal difference algorithm that
A very related set of methods are those based on Inverse Reinforcement Learning (IRL) \(^{64}\) and much better than random, receiving about half the score of a human demonstrator.

Another approach comes from \textit{Gao et al.} \(^{29}\). \textit{NAC}, normalizes the Q-function and reduces Q-values of unseen actions. It learns an initial policy from demonstrations then self-refines that policy while performing the task. \textit{NAC} is tested in three navigation domains: Toy Minecraft, Torcs, and GTA. They find that their algorithm improves on behaviors learned from imperfect demonstrations and was more robust to imperfect demonstrations than other Q-learning based methods.

Some Reinforcement Learning based methods combine RL with another technique. One example of this combines LfD and motion planning into a single framework \(^{87}\). Probabilistic inference is used to find trajectories which are optimal to a skill, but weight is also placed on being feasible in different situations. They show that when teaching a robot learner to open a box, open a drawer, and moving an object, performance is improved over previous similar methods, while also being able to handle additional obstacles more effectively. Another algorithm, \textit{DaD}, adds data-aggregation on the data set to traditional dynamics Reinforcement Learning and augments it with an additional iterative procedure \(^{115}\). This method is tested both in simulated domains (CartPole and a Helicopter Hovering domain), and with real robots (Driving to a specific spot and moving a robot arm to a specified joint configuration). They show that it incurred less cost in the simulated domains and performed more desirably in the real-world domains. A third strategy, Deep Q Learning from Demonstration (DQfD), combines Q-Learning and Markovian techniques \(^{34}\). Specifically, a Markov Decision Process is used which is updated to the loss gained from Q-Learning updates from supervised classifications of the actions provided from the demonstrators. After this initial phase is over, Reinforcement Learning is used to continue learning. They find that DQfD does better than pure imitation learning in 39 of 42 Atari games (of 11 different types), and outperforms the best demonstration given in 11 of those games.

Another strategy is to combine RL and LfD more directly. For example, \textit{Nair et al.} \(^{61}\) present a method which combines demonstrations with reinforcement learning. Their strategy is to use reinforcement learning, but to use demonstrations to help with tasks where exploration may be very difficult. The effect of the demonstrations is slowly annealed away once the learned policy starts to outperform them. They apply this strategy to three tasks: Pushing a block to a location, and picking up a block and moving it to a specific position in the air. They found that this sped up learning over just using reinforcement learning. Another example of this is \textit{THOR} \(^{108}\), which focuses on searching for policies which optimize total cost over a time-step window, instead of over an infinitely long horizon. It creates a Markov Decision Process which uses a gradient formulation that is a halfway point between Imitation Learning and Reinforcement Learning. \textit{Thor} was evaluated on the well known Mountain-Car, Acrobat, and CartPole domains from the OpenAI Gym \(^{10}\), finding it to give better performance than other similar methods.

### 2.2.7 Inverse Reinforcement Learning

A very related set of methods are those based on Inverse Reinforcement Learning (IRL) \(^{64}\). Rather than trying to directly predict actions, IRL operates on learning problems in which the reward function \(R\) of the demonstrator is unknown. While reinforcement learning generates a policy from a reward function, inverse reinforcement learning derives a reward function from a policy, as shown in Figure 2.3. To use IRL for learning from demonstration, instead of having an explicit policy, the learning agent has access to only a set of demonstrations. From these demonstrations, IRL is used to derive a reward function \(R'\), which is expected to approximate the true (unknown) demonstrator reward function \(R\). Once \(R'\) has been derived, the learning agent can use regular reinforcement learning to derive the desired policy.

One such algorithm is presented by \textit{Ng and Russell} \(^{64}\). The inputs to this algorithm (shown in Algorithm 1) are a series of demonstrations from the initial state and represent the actions of some unknown policy \(\pi\). The algorithm uses Monte Carlo search iteratively to derive reward functions. After running the algorithm for a specified number of iterations an estimation \(R\) of \(R'\) that was being used is returned, with the goal of having \(R\) be as close to \(R'\) as possible. This algorithm was tested in 3 domains. The first of which is a 5x5 grid world where the agent starts from the lower
Algorithm 1 IRL from Sampled Traces($policy, traces$)

1: D = \{ $d_i$ | $d$ is a initial state $\}
2: s_0 = \text{initial state}
3: P = \{ $p_0$ | $p_0$ = some policy$\}
4: \textbf{while} Not satisfied with current $R$ \textbf{do}
5: \hspace{1em} execute $m$ Monte Carlo traces under each $\pi_i$
6: \hspace{1em} \textbf{for} $i = \{1, \ldots, d\} \textbf{do}$
7: \hspace{2em} $V^\pi_i(s_0) = \text{average empirical return on these } m \text{ traces if reward had been } R = \phi_i$
8: \hspace{1em} \textbf{end for}
9: $R = \alpha_1\phi_1 + \ldots + \alpha_d\phi_d$
10: $\pi_{k+1} = \pi \mid \text{max}(V^\pi(s_0)) \text{ under } R$
11: $P = P \cup \pi_{k+1}$
12: \textbf{end while}

left square, and receives a reward of 1 when reaching the top right square. The available actions are to move in one of the four cardinal directions, but 30% of the time selecting an action will cause a random movement as well. In this case, the algorithm was able to find a reward that was very close to the true reward. The next experiment was the “mountain-car” task, where a car is to drive up a hill and park. The true reward for this task is -1 per step until the goal is reached, with the state data being the car’s position and velocity. For this experiment, the algorithm was also able to learn a reward that was very close to the true reward. Finally, the last experiment was run in a continuous version of the 5x5 grid world, with the algorithm learning a reasonable reward function after the first iteration, and a reward function which allowed the learner to select the same actions of the demonstrator around 95% of the time was obtained after about 15 iterations.

Another example of using IRL directly on experience replays comes from Konidaris et al. Their algorithm, known as CST, segments each trace into a skill chain, and then merges all of the skill chains into a skill tree. This tree can then be further refined while learning, and can be applied in continuous domains. Policies created by CST were then shown to perform better than agents which only used skill chaining in a pinball domain.

An iterative approach based on IRL is the Multiplicative Weights algorithm. This approach starts with an initial mixed strategy that is used for the first round of the game, and then computes a new mixed strategy after each round based on the game matrix, which reports the loss for each possible combination of moves between the two players. No empirical experiment is done, but proofs are provided guaranteeing that Multiplicative Weights will find approximately optimal min-max strategies.

An extension of the Multiple Weights algorithm is Multiplicative Weights for Apprenticeship Learning, which extends the algorithm to be applied to a game matrix and to be able to estimate...
an optimal policy instead of requiring that it be computed exactly. This algorithm was compared in a car driving simulator and was rated on minimizing collisions, avoiding driving off-road, and driving fast. They found that the agent drove at least as fast as the demonstrator, and reduced the number of collisions (the number of off-roads was 0 for both).

An example of IRL using Markov Decision Processes comes from Melo and Lopes, where they use Markov Decision Process metrics in a standard supervised environment in order to attemptLfD. In order to do so, they calculate the distance between states in the MDP via bi-simulation, specifically using a metric called the ground distance. Kernel-based methods that use this metric are then employed for learning. They compare this method to one which uses more “standard” metrics (Zero-one Distance and Transition Distance), and find that using the MDP metric outperforms both handily, and actually significantly jumps in classification rate once a certain threshold of noise has been met.

There has also been some work done in augmenting the reward function used in typical Inverse Reinforcement Learning. For example, Suay et al. use reward shaping, which provides additional information after each state transition. The agent is then rewarded for taking appropriate steps towards the goal, which augments the environmental reward. They show that adding this shaping to their inverse reinforcement learning method increases task performance in two domains: A version of Super Mario, and a simple Maze domain.

### 2.3 Feasibility of Learning from Human Demonstrators

Many current approaches to LfD assume an abundance of training data. This is fine for most AI demonstrators, but may be intractable to use with human demonstrators. To this end, LfD methods will either need to be adjusted or created anew to be able to effectively learn from human demonstrators. The rest of this section discusses the work that has been done in this area.

#### 2.3.1 Markovian-Type Approaches

One example of a Markovian approach to learning from humans comes from Nikolaidis et al. They first clustered human demonstrators into different types, and then learned a reward function for each type via inverse reinforcement learning. Finally, those models were included as part of a mixed-observation Markov decision process which also includes the human type (as a partially observable variable). This process was tested by having a robot aid humans in refinishing a box by positioning the box so that it would be most helpful. They found that the human subjects felt that the robot anticipated their actions better, was more responsive to the humans, and led to better team efficiency than for baselines where the reward functions were hard coded by experts.

Another example is a hybrid approach that first uses a Bayesian clustering algorithm to build a hidden semi-Markov model, which captures probabilities for demonstrated motions. This model is then stochastically sampled to generate motions, which are tracked using a linear quadratic regulator. This approach was tested by having a human demonstrate a “hot-stabbing” action, which consists of inserting a plug into a receptacle, and the metric used is the whether the plug has made it to the receptacle by the end of the motion. On a test of 10 trials each over 3 different targets, it was found that the agent was able to successfully perform the task every time.

#### 2.3.2 Approaches Designed for Robotics

A fair amount of the work from the literature in learning from human demonstrators focuses on robotic domains. For example, Calinon and Billard use an algorithm which uses Expectation Maximization to generate an initial model. They then use regression to generate stochastically new data based on the current model, and then refine a new set of parameters based on the generated and newly observed data. This approach was tested by attempting to teach a humanoid robot how to imitate 10 different basketball-inspired human gestures, and they found that it garnered results not significantly worse than a batch learning policy.
Another example is a two-phase algorithm known as the A-OPI algorithm\(^3\). In the demonstration phase, the learner uses a set of teacher demonstrations to create an initial policy. In the advising phase, the learner executes this policy and is offered advice by a human advisor on how to update its policy. A-OPI was tested via a spatial positioning task where a Segway RMP robot was moved around, and it was found that it increased both execution success and accuracy over just having the demonstrator human more demonstrations.

Finally, Akgun and Thomaz\(^1\) derives an approach which attempts to solve two different tasks (performing a task and detecting the success of a task). It does so by training 2 models for each set of demonstrations: An action model garnered via the motion of the robot (joint positions, motor commands, etc), and a goal model learned using external sensors to gain information about the state of the object being manipulated. The action model is then used to execute the skill, while the goal model is used to monitor the execution of the skill. This split comes from the notion that human demonstrators are likely to be good at demonstrating what to do, but have a harder time demonstrating how to do it. For a set of 8 humans demonstrating two tasks, they found that the agent was able to successfully close a box 57% of the time, and successfully pour from a cup 75% of the time, but was able to determine whether or not the task was a success 90% of the time.

### 2.3.3 Other Approaches

Besides the aforementioned categories, other approaches at learning from human demonstrators have been used. For example, Wang\(^117\) uses an inductive learning approach in order to learn complex action operators. Their system, OBSERVER, starts by observing traces provided by a demonstrator in order to get an initial set of planning operators. It then has a practice step where it performs learning-by-doing tasks to further refine its operators. This step reduces OBSERVER’s reliance on the demonstrator. OBSERVER was tested in a process planning domain, and was shown to learn operators which would solve problems as efficiently as human-coded operators would.

### 2.4 Capturing the Sequential Nature of Data

Unlike in traditional supervised machine learning, in LfD data is often sequential, meaning the action a demonstrator chooses depends on one or more states prior to the current state, instead of only the current state. Various approaches have been presented in the literature to account for this, described below.

#### 2.4.1 Standard Approaches

Although some recent work in the field has addressed the issue of sequential data, many others have not. Some focus on trying to predict sequential behavior without sequential data\(^25\), while others attempt to solve these learning problems in domains where a single record is assumed to contain all of the needed information to make a prediction at that time\(^98\).

Dietterich in particular identifies the following methods that present a variety of ways to account for sequential data\(^17\):

- **The Sliding Window Method** – This method converts a sequential supervised learning algorithm into the classic one, using an input window. Basically, this method constructs a window classifier which maps an entire window of state data (instead of just one record) to one or more actions.

- **Recurrent Sliding Windows** – Operates like sliding windows, except that each predicted value is fed as an input along with the window to help make the next prediction. An illustration of this and the previous method can be seen in Figure 2.4.

- **Hidden Markov Models (HMM)**– This strategy creates a probabilistic model of the way in which the states and actions are generated. Hidden Markov models are techniques to model systems that are assumed to be Markov processes with hidden states, a Markov process being...
Figure 2.4: Example of Sliding Window (left) and Recurrent Sliding Window (right). The red arrows represent the action(s) from the previous timestep being used as part of the state data for the next timestep.

A stochastic process that does not require memory beyond knowing the current state. The observed variables in an HMM are the output of the model. The state information is a set of hidden variables, which are not directly visible to the observer. It is defined by two stationary probability distributions: the transition distribution which states how actions are related, and the observation distribution which states how the states are related to the actions. This method, however, suffers from the label bias problem. The label bias problem stems from the fact that all of the “probability mass” which arrives at a given state in the HMM must be distributed amongst the possible successor states. Observations can affect which destinations along their outgoing transitions get mass, but not how much total mass to pass on. This leads to states with fewer outgoing transitions having a bias over those with many, since all of the “mass” is getting transferred to only a few states. In the worst case, states that only have one outgoing transition ignore the observation altogether.

- Conditional Random Fields (CRFs) – CRFs are conditional distributions that have associated graph structures. The advantage of this conditional distribution is that dependencies amongst input variables do not need to be explicitly represented, which allows for the use of rich, global features of the input. In the context of this paper, CRFs can be used to overcome the label bias problem. This is done by modeling the relationship among adjacent label pairs as a Markov Random Field which has been conditioned on the inputs – this determines the way that adjacent outputs influence each other based on the input values.

- Graph Transformer Networks – The input in a graph transformer network is a state data graph where each edge has a feature vector that is attached to it. A graph transformer network consists of two parts. The first is a neural network which takes in the state data graph and produces an intermediate graph which consists of pairs of class labels with their scores. This intermediate graph is then analyzed by a Viterbi transformer, which finds the path through the graph with the lowest total score, and then provides a graph that contains only this path. That graph provides the predicted output labels (in the context of this paper, the predicted actions). An illustration of this structure is shown in Figure 2.4.

A recent approach to deal with this problem are Recursive Neural Networks, the most prominent example of which are LSTMs (such as the FullyConvLSTM algorithm). Although LSTM’s do address the problem of learning in domains which have dependencies that go arbitrarily far in the past, they also often require a lot of training data to learn, which is unfeasible when learning from human demonstrators.
2.4.2 Rule-Based Approaches

Some work has been done in handling data sequentiality via using rule-based representations of strategies. For example, Khardon\textsuperscript{39} presents a production rule system, which is a strategy composed of condition-action rules. In addition, it contains information on the current goals that are trying to be achieved, as well as a representation of working memory. Their system works by, for each iteration (each time we need to match a currently observed state), the condition of every rule is evaluated, to see which rules match the current state. Out of the rules that match the state, the one with the highest priority is selected. The action portion of that rule is then either executed or simply added to working memory. This cycle is repeated until the overall goal is achieved. One thing that is important to note about this method is that the agent does not have access to the working memory of the demonstrator—it is considered internal to the strategy and therefore must be learned. By learning and maintaining an internal working memory, the strategy represented by the rule-set can consider information present in past states, thus capturing some of the sequentiality of the data.

This system was tested in the well known blocks world domain\textsuperscript{119}, using four operators and eight blocks. They found that, as long as example demonstrations with a length of or greater than 1000 were used, the learner was able to solve about 80% of problems that were eight blocks. The system's success dropped as the number of blocks in the testing problems rose, but maintained a solid 60% on significantly larger problems consisting of 20 blocks. It is also important to note that the system is only feasible for relatively small problems and domains, as the runtime of the learning algorithm is exponential in both the number of free variables and the complexity of the conditions of the rules.

Work towards a more generalized rule based method of handling data sequentially is given by Konik and Laird's relational learning by observation framework\textsuperscript{46}. This framework has an execution cycle which consists of two modes. In the first, a demonstrator generates behavior via selecting actions with a behavior interface, in the second, a previously learned agent program generates the behavior and the demonstrator gives feedback, which leads to an improved agent. One assumption that is made is that the goals, subgoals, and actions that the agent can take are broken into a hierarchical tree structure. The leaves of the tree represent actions that the agent can directly take in the world, and the internal nodes represent higher level actions, subgoals, or goals. By using this structure, the agent is given an idea of what actions affect other actions, capturing some sequentiality.

Two experiments were conducted using this framework. The first experiment involved artificial examples for selection condition concepts in a building navigation problem, using inverse entailment. Rather than use negative examples, positive examples that had the demonstrator choosing all of the best parameter selections for the given situation were given a special tag. Six maps were generated,
with varying numbers of items and doorways, for a total of 36 different combinations. For this group of combinations, 5 experiments consisting of 5 positive examples each were conducted. The testing set also consisted of 36 combinations, but it is important to note that the maps themselves were made larger, to dissuade size-specific hypotheses from being evaluated as accurate. The results of this experiment were that as the number of tagged examples increased, so too did the average accuracy—from about 50% with 0 tagged examples up to 90% with 5 tagged examples.

The second experiment used behavior data generated by Soar agents using the presented framework. All behavioral data for this experiment was created using a single map that consists of 13 fully connected rooms, each of which has a symbolic node to aid agent navigation. The Soar agent controls a virtual character that has an internal map of the rooms and items in the level. The agent will randomly choose an item and move towards it using the appropriate operators until they are in the target room. The goal of this experiment was to learn the concepts of selection and termination, using 600 positive examples and 200 negative examples for each concept. The results were that the termination condition was correct, but an over-general theory was produced for the selection condition that could lead to the selection of random nodes on occasion. This indicates that using negative examples obtained at situations which do not have operators selected might not be sufficient for learning operators that do contain parameters—an issue that might be solved by generating negative examples which have random parameters.

In summary, rule-based approaches are very good at capturing some aspects of data sequentiality, especially which actions are dependent on other actions. However, not much work has been done in capturing the sequentiality of the states themselves using rule-based approaches, and rule-based approaches tend to only be feasible for relatively small domains.

### 2.4.3 Similarity-based Approaches

Another common approach is that of similarity-based methods, such as \(k\)-nearest neighbor or cased-based reasoning (as mentioned above in Section 2.2.2), for which many strategies to handle the sequentiality of data have been devised in the literature of LfD. For example, in the context of case-based reasoning, many case acquisition strategies (extracting cases from demonstrations) have been proposed to address this problem:

- **Reactive Learning** – This strategy simply learns one case per entry in the demonstration. This means that it is purely reactive, having no concept of time or order of actions.

- **Monolithic Sequential Learning** – This strategy learns a case per demonstration. That is, given a demonstration with \(n\) actions, it learns a complete sequential plan consisting of all actions in precisely the same order that the demonstrator executed them. However, this means that no matter when a new case is retrieved that case will contain a plan of actions starting from the very beginning, instead of from the current step.

- **Hierarchical Sequential Learning** – This strategy splits a task into a series of subtasks, and attempts to learn a plan for each of those subtasks separately. It also attempts to find which tasks are subtasks of other tasks. This allows for a kind of “checkpoint”, where if a new case needs to be retrieved that sequence of actions will start from the beginning of that subtask, rather than the very beginning.

- **Dependency Graph Learning** – This strategy constructs a dependency graph, where nodes and edges represent actions and dependencies in a plan, respectively. This removes the assumption that the demonstrator has executed the actions in a total and correct order, and allows for plans that can be better executed and adapted.

- **Dependency Graph Hierarchical Learning** – This is similar to the previous method, but it does not learn sequential plans. Rather, it uses the dependency graph to learn plans which have a partial ordering of actions based on their dependencies.
• Timespan Learning – This strategy employs the previous strategy, with one major change. After the dependency graph is generated, this method uses timespan analysis to remove dependencies which cannot exist due to action duration. For example, if action \( a \) was started, and then action \( b \) was started, and action \( a \) ended after \( b \) was already started, it is impossible for \( b \) to have a dependency with \( a \).

• Timespan Hierarchical Learning – This strategy is identical to Dependency Graph Hierarchical Learning, but with a similar change as the previous method. After the dependency graph is created, timespan analysis is used to remove actions inconsistent with action duration.

These strategies were all tested using S3, a simplified RTS game that retains the complexity of the domain. Agents were taught to play the game using each acquisition strategy, given five random demonstrations of an expert AI playing and winning the game against various AIs. None of the strategies were able to produce a agent which played the game quite as well as the demonstrator, with timespan learning producing the best results.

One algorithm that was derived to attempt to account for data sequentiality is SMILe, by Ross and Bagnell\(^9\). This is an iterative algorithm that can use practically any classification algorithm to train a classifier at each iteration. The main idea behind SMILe is to derive a new policy at each iteration which is a stochastic mixture of a newly trained policy and the previous policies, with the newest policies having the largest weights. SMILe can be roughly viewed as the predecessor to DAgger.

DAgger, proposed by Ross et al.\(^\), is another active Learning from Demonstration algorithm that attempts to account for the violation of the i.i.d. assumption\(^9\). The idea behind DAgger is to take learning data from a series of traces of the demonstrator performing a task, and train a learner on that data. The algorithm then repeats the process, but reduces how often the demonstrator is used more and more in each iteration (replacing it more and more with the learner), still storing the states that are encountered as well as the actions that the demonstrator would have taken (regardless of who is controlling). After a set number of iterations, the version of the learner which performs the best on validation is chosen as the final learned policy.

An extension of DAgger called SafeDAgger\(^\) attempts to reduce the cognitive burden placed on the demonstrator by attempting to learn a reference policy that states what actions the demonstrator would take without having to consult the demonstrator, and querying the demonstrator when the learner's policy deviates too far from this reference policy.

While some approaches, such as DAgger, take into account temporal data by default, others only employ it when necessary. For example, the Temporal Backtracking algorithm\(^\), is a variant of a the nearest neighbor approach modified to take into account previous states and actions. The basic flow of Temporal Backtracking is shown in Algorithms 2 and 3, where PT stands for the problem threshold and ST stands for the solution threshold. When the agent is performing a task, after learning, not just the current state of the world, but also the sequence of past states (the run) is taken into account. The output of these functions (when used together) is the action to perform. The algorithm works by comparing pieces of the demonstrations to the current run. The algorithm starts by comparing the current state to the last state in one of the demonstrations, if more than one demonstration is more similar than a certain threshold, then temporal backtracking moves one step backwards in time, and compares the previous state with the previous state of the demonstrations. This process is repeated until only one demonstration remains. Additionally, similarity between past actions are also considered during this process.

This approach was tested using a simulation of a robot in a 50 unit by 50 unit environment, with obstacles scattered throughout. At each time step, the robot can perform one of more of five actions: move forward, move backward, turn left, turn right, and turn 180 degrees. The state data obtained was values of a touch sensor and a sonar sensor which indicates the distance to the nearest obstacle. The demonstrator was observed interacting with the environment over a period of time which resulted in 50,000 training cases and 25 sets of 2500 testing cases, with the obstacles and starting point changing for each set of test cases. Temporal Backtracking was then compared against a reactive retrieval approach in terms of how accurately it was able to retrieve cases that lead to imitating the demonstrators behavior. The reactive approach did slightly better at predicting when
Algorithm 2 \texttt{stateRetrieve}(run, demonstrations, time)

1: $NN = \emptyset; NNa\textit{ctions} = \emptyset; \text{bestSim} = -1; \text{bestRun} = \text{NULL}$
2: \textbf{for} each past $\in$ demonstrations \textbf{do}
3: \quad $\text{similarity} = \text{sim}(\text{state}(\text{run}, \text{time}), \text{state}(\text{past}, \text{time}))$
4: \quad \textbf{if} similarity $> \text{bestSim}$ \textbf{then}
5: \quad \quad $\text{bestSim} = \text{similarity}; \text{bestRun} = \text{past}$
6: \quad \textbf{end if}
7: \quad \textbf{if} similarity $> PT$ \textbf{then}
8: \quad \quad $NN \leftarrow NN \cup \text{past}$
9: \quad \quad \textbf{if} action(past) $\notin NNa\textit{ctions}$ \textbf{then}
10: \quad \quad \quad $NNa\textit{ctions} \leftarrow NNa\textit{ctions} \cup \text{action}(\text{past})$
11: \quad \quad \textbf{end if}
12: \quad \textbf{end if}
13: \textbf{end for}
14: \textbf{if} $NN == \emptyset$ \textbf{then}
15: \quad return action(bestRun)
16: \textbf{else if} $|NNa\textit{ctions}| == 1$ \textbf{then}
17: \quad return $\{NNa\textit{ctions}\}$
18: \textbf{else}
19: \quad return actionRetriev\textit{e}(run, \text{NN}, \text{time} + 1)
20: \textbf{end if}

it was supposed to reverse, but Temporal Backtracking did considerably better predicting when to turn left or right, as well as on the overall retrieval accuracy.

2.4.4 Other Approaches

An approach explored by Akgun et al.\textsuperscript{2} was to use keyframes to help capture the sequentiality of the data in teaching a robot movement skills. The idea was that providing keyframes would give a few discrete points that capture the necessary movements required to complete the skills, thereby making the sequentiality more accessible than it would be in a typical trace-learning case with a lot of data. When put into practice with goal-oriented (e.g. stacking blocks) and means-oriented (e.g. saluting) skills, they found that their keyframe based approaches required more time to train and required more demonstrations in general but were rated as more proficient at means-oriented skills by human demonstrators. However, the highest success rate of learning actions came from providing demonstrators with a keyframe-annotated trace representing a skill that is slightly failing (for example, touching a location a few centimeters away from the location that should be touched), and iteratively updating the frames in order to teach the proper behavior.

2.5 Internal State of the Demonstrator

Another difficulty in LfD is attempting to capture internal state data from the demonstrator. Often, when the demonstrator is performing a task and creating a demonstration, they have goals or remember parts of past states, which affects their actions. Some researchers have addressed this problem by asking the demonstrators to annotate their data traces with intentions\textsuperscript{46,72}, but in general this can not always be done. As such, several types of strategies have been developed to address this issue.

2.5.1 Do Not Consider Internal State

Strategies to capture internal state range from very simple to very complex. The simplest strategy is to ignore it completely. This strategy has been used in most works in the past\textsuperscript{25,98}, with varying
Algorithm 3 actionRetrieve(run, demonstrations, time)

1: \( NN = \emptyset; \) \( NNactions = \emptyset; \) bestSim = −1; bestRun = NULL
2: for each past \( \in \) demonstrations do
3: \[ \text{similarity} = \text{sim}(\text{state}(\text{run}, \text{time}), \text{action}(\text{past}, \text{time})) \]
4: if \( \text{similarity} > \text{bestSim} \) then
5: \[ \text{bestSim} = \text{similarity}; \text{bestRun} = \text{past} \]
6: end if
7: if \( \text{similarity} > \text{ST} \) then
8: \( NN \leftarrow NN \cup \text{past} \)
9: if \( \text{action}(\text{past}) \notin NNactions \) then
10: \( NNactions \leftarrow NNactions \cup \text{action}(\text{past}) \)
11: end if
12: end if
13: end for
14: if \( NN = \emptyset \) then
15: return action(bestRun)
16: else if \( |NNactions| = 1 \) then
17: return \( \{NNactions\} \)
18: else
19: return stateRetrieve(run, NN, time + 1)
20: end if

(amounts of success depending on the problem at hand – for tasks which are heavily memory-based, this strategy is typically insufficient.)

2.5.2 Demonstration Annotation

Another common strategy is to take demonstration data without any sort of internal state data, and then have the demonstrator go back and properly annotate the demonstrations with that data\(^{46,72}\). An example of this can be seen in the SmallStar system\(^{32}\). Smallstar obtains data by having users press a button to start recording, taking a transcript of their actions, and stopping collection when they choose to stop recording. An example of a program generated through this process is in Figure 2.6. This recording mechanism is low overhead and also outputs the user’s actions in a human readable format. Another thing to note is that only user actions which cause a change in state are recorded (selecting and then unselected a certain element would not be recorded, whereas selecting and then deleting that element would be recorded as a deletion).

To create the new behavior, the user edits the data descriptions of the various objects after recording a program. Three common instances of this are shown below.

- **Set Iteration** – The user records a program that operates on a single object (e.g., a single file), changes the file description to ANY to include all files, and introduces a loop to iterate over them.

- **Loops** – The user selects the actions in the transcript that they want to loop and choose the “Repeat...” option from the pop-up window. They then enter the appropriate condition piece (for looping over the set of files in Figure 2.6, it would be the Negotiations-any data description).

- **Conditionals** – To learn conditional statements, the user records a “straight-line” program, and then adds in if-then conditions after the recording using the “if” option from the pop-up menu.

Thanks to these annotations, SmallStar can have a user easily edit the exact steps that they took into a generalized program that follows their behavior. Other pieces of work employing demonstrator annotations include\(^{46}\) and\(^{56}\).
2.5.3 Direct Inference

One method for inferring the internal state of the demonstrator directly from the demonstration data involves looking at past states and actions\(^{98}\). The idea here is to look at records not as individual entities, but rather as temporally linked. The learning agent is then able to use the entire run (the sequence of all states and actions observed so far in the demonstration) of the demonstrator to try and approximate how the internal state of the demonstrator changes. Specifically the goal here is not to directly learn the internal state of the demonstrator, but rather learn it implicitly and then use that knowledge to have the learner learn to be *behaviorally equivalent* to the demonstrator, meaning that if they experience the same run they will produce the same action. Another way to state this is that the learner is taking advantage of the sequential nature of the data to try and interpolate what the internal state of the demonstrator would be at the current state. In this case, this leads to a significantly better overall prediction accuracy. However, the approach of using the entire run is only feasible for small demonstrations. For example, in the work presented in\(^{98}\), world-state data is only recorded when the pilot performs an action, such as adjusting the flaps. This leads to no more than 1,000 events recorded in a single demonstration, with 19 data points in each event – a relatively small dataset. For domains which have many more entries in a single demonstration or a much higher data dimensionality, using the entire run quickly becomes computationally infeasible. One strategy for expanding its utility would be to only pull in past events when necessary\(^{23}\).

2.5.4 Markov Inference

Another approach is to use Hidden Markov Models (HMMs), which can model the internal state of the demonstrator as hidden variables. An example of this is the concept of Input-Output Hidden Markov Models (IOHMMs)\(^{7,73}\). In cases where it can be assumed that the internal state of the demonstrator only relies on the current and directly previous states, this is a powerful method for capturing that internal state. An extension of Hidden Markov Models (HMM)\(^{85}\), these can be used to learn how to map states to actions. IOHMMs consist of 3 variables: input, output, and a hidden internal state. These directly correspond to the state data, the action performed, and the internal state of the demonstrator. As an IOHMM runs, the internal state that is constructed is an approximation of the demonstrator’s internal state, thereby reconstructing it based on the data from the current and directly previous state. Learning algorithms that employ these methods do exist (such as\(^{7}\), which is derived from the expectation-maximization algorithm\(^{16}\)), but only operate in restricted cases (such as the internal state variable being discrete). It is also important to note that if the input values to the IOHMM are also unobservable (as with the internal state of the demonstrator) then a normal HMM might still be able to infer the demonstrator’s internal state. Other examples of using IOHMMs for LfD include learning to perform technical support procedures\(^{53}\), automation of repetitive computer tasks\(^{68}\), and determining head gestures based on prosody\(^{99}\). Although methods such as this can be very effective in restricted cases, no general-case solution has been found. Another example of using HMMs for this inferring the internal state of the demonstrator is via Dynamic-Bayesian Networks\(^{73}\).

---

**Figure 2.6:** An example of a user-recorded program in SmallStar. This program opens the Negotiations folder, moves a single file in it onto the desktop, and closes the folder again.
Table 2.1: An Example of Learning Agent and Demonstrator Moves, Where the Demonstrator’s Policy Involves Randomness

<table>
<thead>
<tr>
<th>Learning Agent</th>
<th>Demonstrator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>Left</td>
</tr>
<tr>
<td>Right</td>
<td>Left</td>
</tr>
<tr>
<td>Left</td>
<td>Right</td>
</tr>
<tr>
<td>Right</td>
<td>Left</td>
</tr>
</tbody>
</table>

2.6 Evaluation Metrics

Relative to traditional supervised learning, metrics for LfD are much harder to define. As such, there is no unified consensus on which metrics are the right ones to use for the evaluation of LfD algorithms. A key reason for this is that metrics for LfD must capture two different aspects of a learner’s behavior at once. The first is how well an agent performs a task, which is typically easy to evaluate. The second aspect that needs to be captured is the resemblance of the learner’s behavior to the demonstrator’s. This is very hard to capture, especially since one must first attempt to define what it means to “behave similarly” to the demonstrator. Another reason that not much work has been done towards a generalized evaluation metric is that most LfD approaches are tested using a single domain. Algorithms in LfD tend to be highly specific (differing in such areas as data representation and demonstration methods) and therefore do not generalize to a wide variety of problems, so not much work has been done with evaluation metrics which address a wide variety of problems. Deriving a standard set of evaluation metrics which would let different algorithms from different domains be accurately compared is still an open and very important problem in the field of LfD.

The simplest way to attempt evaluation is to simply directly compare the actions that an agent and the demonstrator take given the same world states, and say that the more similar the actions are, the better the agent learned the demonstrator’s behavior. However, this is not always enough to capture whether the agent has truly learned the correct behavior or not. To illustrate, imagine that you have a domain where an agent can (always) move either left or right. Given a demonstrator and a learner, the results of 5 moves might look like in Table 2.1. If we were to judge how well the agent had learned based on its moves versus the demonstrator’s moves, we would judge it to not be a very good learner at all. However, what if the policy that the demonstrator follows is simply “pick either left or right at random and move in that direction”? In this case, the agent might have learned the behavior perfectly, so clearly more robust evaluation metrics are needed.

2.6.1 Domain Specific

Some work has been done in developing powerful domain-specific evaluation metrics. For example, Chernova and Veloso developed a metric for evaluating LfD algorithms in the context of simulated car driving. In their framework, the agent acts as the car and the demonstrator acts as the driver. As the agent is moving down the road, it is moving much faster than the other cars and can only take one of three actions (move one lane to the left, move one lane to the right, or stay in the current lane). The road consists of three adjacent lanes surrounded by two shoulders, and the goal is for the agent to avoid hitting other cars (see Figure 2.7 for a visualization). The demonstrator drives at first, but gauges the learner’s progress and once it feels the learner has reached a certain point it gives full control to it.

Two evaluation metrics were derived that are specific to this domain that evaluate the learner’s performance in other ways than how closely they mimicked the demonstrator’s actions. The first is the percentage of time that the agent spends in collisions with other cars. Every time-step that the agent is in contact with a car, it counts as another “collision”. The percentage of this time versus the total time the agent has control is one way in which the agent’s performance is evaluated, with...
the baseline being 30% (from driving in the center lane and never moving). The second evaluation metric is how often the agent is in each lane during a trial. Basically, as the demonstrator drives, the lane preferences that it exhibits are recorded. The same happens with the agent (while driving over an identical road segment). The evaluation metric is then calculated using an estimate of how similar the two driving styles were, based on these preferences.

The results of these evaluation metrics is that they paint very different pictures of how well the agent learned the demonstrator’s behavior in this domain. The first evaluation metric asserted that the agent learned the behavior very well, reducing themselves to a collision rate of only 1.3% after 500 demonstrations, indicating that the agent had learned the demonstrator’s behavior of avoiding cars quite well. However, the second evaluation metric determined that unlike the demonstrator, who fairly effectively used all five lanes, the agent spent most of the trials avoiding traffic by driving on the right shoulder of the road. This meant that their driving styles were very dissimilar, and as such the metric indicated that the agent had not correctly learned the demonstrator’s behavior as a whole, which gives some insight into why there is still much work to be done in this area. Many other researchers have employed similarly domain-specific metrics in their work.

Another example of domain specific evaluation metrics is provided by Silver et al. They define two metrics for evaluating driving an autonomous car “offline”. The first of these is to calculate the average loss between the agent’s planned path and a demonstrator’s example path, which will assess how similar the learner’s plan is to the demonstrator’s. The second metric is a cost ratio, defined as the cost of an example path divided by the cost of the agent’s corresponding planned path – a ratio closer to 1 means a path which is more consistent with the demonstrations. A somewhat similar metric comes from the domain of performing surgical tasks. They gather information from novice, intermediate, and expert demonstrators. This information is then used to train a classifier algorithm to classify traces generated by the learner as falling into one of those three categories.

2.6.2 Multi-Agent Methods

Various evaluation metrics have been conceived for LfD involving multiple agents. One example of this is the work of Chernova and Veloso, who derived several metrics to evaluate LfD algorithms for robots that are taught by a human demonstrator. Some of these metrics are very domain specific, while others are more general and could conceivably be applied to many areas in LfD. A brief listing of these items is included below.

- Fan-out – the number of independent learners than can be managed by a single demonstrator in real-time learning.
• **Teacher Response Time** – The delay between when a robot encounters a problem and the human intervenes (can be generalized to the delay between when a learner encounters a problem and requests assistance from the demonstrator, and when that request is fulfilled).

• **Level of Autonomy** – How much autonomy the learner has compared to the appropriate amount that should be given for the specified task.

• **Appropriate Utilization of Mixed Initiative** – the robot’s ability to effectively regulate whether itself or the human has greater control over its actions (can be generalized to a learner regulating how much control the demonstrator has over its actions).

• **Trust** – The degree of the operator’s trust in the robot and its ability to perform its learned job (can be generalized to the degree of the demonstrator’s confidence that the learner has successfully learned its behavior).

### 2.6.3 Generalized Methods

There has, however, been some work done towards deriving a generalized evaluation metric for LfD learners. One example of this is work done by Ontaño, Montaña, and Gonzalez, which addresses the issues presented by simply evaluating learning by comparing the actions of the demonstrator and learner in situations such as that in Table 2.1. The general idea is to look at the distributions from which the actions come and compare those using the Kullback-Leibler divergence (KL), rather than comparing the actions directly. The problem is that one does not already know the distributions of the demonstrator, or they would already know the demonstrator’s policy and could just use that. Instead, assumptions are made about what sort of distribution the actions take, and then those assumptions are used to create an approximation of the distribution for each the demonstrator and the learner (for the learner, a collection of demonstrations must first be obtained, via the same methods that demonstrations are retrieved from the demonstrator). These distributions are then compared using the KL divergence to gauge learning.

The assumption that is made depends on the level of the problem, according to the 3 level system in Section 2.1.1. For level 1 behaviors, the problem is trivial. For level 2 behaviors, the assumption that is made is that the probability distribution is of the form $P(Y|X)$. Using this assumption the two probability distributions are created and analyzed. For level 3 behaviors, the process is identical except that it is assumed that the probability distribution takes the form of a Dynamic Bayesian Network. In order to create the distributions from the assumed distribution families, they propose to use a Monte Carlo approach as the approximation method.

### 2.7 Active Learning from Demonstration (ALfD)

Active learning is a framework where the learner is given some initial training data, and then can query for more. The idea behind this is that it will lead to more better results with less training data, since the learner is trying to intelligently acquire data that will best help it to learn. The state of the art in ALfD can be placed into categories; the rest of this section will examine the work that falls into each category.

#### 2.7.1 Uncertainty-Based Approaches

Of the work that has been done on ALfD, a good portion of it uses the general notion of uncertainty as a policy for when to query the demonstrator. For example, Silver et al. examine novelty and uncertainty based approaches. The former creates a density model based on the features in the training data, and then given any set of features from a new world state, it could be determined whether or not it was a novel state. The latter trains multiple learners on random subsets of the training data, with the measure of uncertainty being how much they disagree. It was shown in a path-drawing domain that active learning with these policies outperformed a baseline which used the same number of examples but no active learning.
Another approach is to estimate the expected error, and have the learner query on the states which would garner the highest expected error, such as through Monte Carlo sampling, as seen in. Their algorithm has the learner query the demonstrator for a state if it is 1. uncertain about that state and 2. it is expected that querying the demonstrator will greatly reinforce its current belief. This method was tested on two text-classification domains and outperformed baselines which looked purely at the uncertainty of states.

2.7.2 Iterative Approaches

A somewhat recent example of approaching Active Learning from Demonstration is DAgger. DAgger is an algorithm which has the demonstrator control for the first iteration and then shifts control to the learner. The learner and demonstrator are then stochastically mixed, with the demonstrator relabeling the actions for which it was not in control. They elaborate on two different mixing policies - one that gives the demonstrator a 0% chance to control on each iteration after the first, and one that halves how often the demonstrator controls each time, and show that in their Super Mario domain both policies outperform standard supervised approaches.

One downfall to Dagger is that it requires the demonstrator to control or relabel for many, many world states. An extension of DAgger designed to address this issue is SafeDAgger. SafeDAgger attempts to reduce the cognitive burden placed on the demonstrator by attempting to learn a reference policy which states what actions the demonstrator would take without having to consult the demonstrator, and querying the actual demonstrator when the learner’s policy deviates too far from this reference policy.

2.7.3 Other Approaches

The concept of version space is roughly that there is a set of hyperplanes which separate the data in the space of induced features of a domain. One policy that can be used for active learning is to try and halve the version space with each query. To do this, they test three methods that approximate how evenly a query would divide the state space. When tested on text classification tasks, active learning with any of these policies outperformed passive learning.

Two active learning applications of CBR to LfD that are closely related to our own work are Mixed Initiative Observation Acquisition and Delayed Observation Acquisition. In the former, the demonstrator and learning agent share control of a task. The learner can cede control to the demonstrator when it needs advice, and the demonstrator cedes control after a single action, but the demonstrator can also seize control from the learner at any time. The latter is similar, but stores problem states for later labelling in case the demonstrator is unavailable or unable to immediately help. They found that both of these methods increased results in a tetris domain over passive case acquisition, where the learner is never in control of the task during training.

Another method uses the concept of states being risky instead of uncertain. This algorithm, called SHIV, defines states as risky if they either lie in an area without a lot of previously trained-upon states, or the training error at that state is high. States that are the riskiest are the states that are chosen for querying the demonstrator. They compare this approach against DAgger’s stochastic mixing in three domains and find that it obtains similar results but requires less queries to the demonstrator.
Chapter 3: Application Domains

Although LfD has broad applications beyond the realm of games, we propose to use games as application domains. Specifically, we have used several application domains in the preliminary work that we have done so far, including a first person exploration game, a puzzle game, and a platformer game. The remainder of this chapter describes each of these domains.

Minecraft

Minecraft (shown in Figure 3.1) is an expansive, open ended game that focuses on exploration and building. Although there are enemies to fight and items to collect, the game imposes no true goal on the player, instead letting them set their own. The players move about in a 3-dimensional world, which is divided into blocks that (with a few exceptions) the player can pick up and put down as they please. Players are able to kill enemies, build homes, tear down structures, and collect items to craft new tools. Due to these attributes, Minecraft requires both short term, almost reflexive decisions as well as long-term planning. Specifically, Minecraft exhibits the following properties:

- Almost deterministic: Most of Minecraft is deterministic, but it also boasts some stochastic features, such as the spawning of enemies or what items drop when certain blocks are broken.
- Partially Observable: How far a player can be seen depends on their settings, but even the farthest vision range is a very small fraction of the world.
- Dynamic: The environment is rich with other agents, in the form of passive and aggressive entities. Passive entities, such as cows or pigs, tend to be hunted by the player for food and items. Aggressive entities, however, will actively pursue the player and try to destroy him. These entities, along with the passage of time, comprise the dynamic parts of the environment, and act independently of the player.
- Continuous: The environment of Minecraft is continuous, with the position and rotation of the player and other entities being measured as real values.

For this domain, we represent both actions and world states as logical horn clauses. The world state is represented as logical clauses of the form \( X_t = x_0 \lor x_1 \lor \ldots \lor x_m \). Each predicate \( x_i \) takes the form of a Prolog term, and represents some aspect of the world. For example, the term \( health(20) \) would mean that the player has 20 health. Figure 3.2 shows an actual world state with its corresponding action. Although world states may have an arbitrarily large number of predicates, those predicates belong to 15 different types (position, rotation, block, etc.), some of which can appear multiple times per entry.

Similarly, actions are also represented as logical clauses \( Y_t = [y_0 \land y_1 \land \ldots \land y_m] \). Each predicate \( y_i \) represents an action the player is taking at that time, also in the form of Prolog terms. Notice that we need to represent actions as clauses, since players might execute more than one action at a time (walking while turning and swinging a weapon), or even no actions. For instance, in the example in Figure 3.2, the player was rotating at the same time as chopping wood (choppingWood(-1) means the player is chopping wood with their bare hands). There are 9 actions in all, which are detailed below:

- Rotating(UpDown, LeftRight): The player is rotating the character UpDown degrees up or down and LeftRight degrees left or right.
- Walking(Forward, Left, Right, Backwards, Running): The player is moving the character in up to 2 directions (if Forward, Left, Right, or Backwards are true, they are moving in that direction). If Running is true, the player is running in that direction instead of slowly walkin.
• ChoppingWood(Tool): The player is chopping a piece of wood with Tool equipped.

• Digging(Block, Tool): The player is breaking a “soft” block Block (sand, dirt, gravel, etc) with Tool equipped.

• Mining (Block, Tool): The player is breaking a “hard” block Block (stone, coal ore, bricks, etc) with Tool equipped.

• Breaking(Block, Tool): The player is breaking any block Block not covered in the previous 3 actions with Tool equipped.

• SwitchedTo(Item): The player has equipped Item.

• Placing(Block): The player has placed Block onto the block they are currently facing.

• Crafting(Item, Amount): The player has crafted Amount number of Item and added them to their inventory.

In addition to this symbolic representation, we also used a propositional representation which encoded the same information about the domain and the same actions as a feature vector of 98 propositional features. When more than one predicate with the same name functor appears in a world state (such as pMob in Figure 3.2), only the predicate which represents the enemy/object closest to the player is recorded in the feature vector. If any predicates do not appear in a world state, their associated feature in the feature vector is set to 0.

Unlike the domains that follow, this domain has no task reward. Rather, experiments in this domain were done via directly comparing the actions a learner took compared to the actions the demonstrator would have taken in the same situation, via a leave-one-out or one-versus-one procedure on the collected data.

Super Mario

Super Mario (shown in Figure 3.3) is the classic Super Mario platform game, where the player has to reach the end of a level while avoiding enemies and picking up coins. Although some players set their own goals such as picking up every coin or killing every enemy, the main goal (and therefore the goal that we focus on in the domain) is to get as close to the rightmost end of the level as possible. The reward function used is Mario’s X-coordinate in pixels plus a bonus of 500 if he is Fire
\[ z_{162} = \text{holding}(17, 1, 0) \land \text{rotation}(\text{sw}, \text{up}) \land \text{selectedBlock}(17) \]
\[ \land \text{target}(\text{none}) \land \text{block}(1, \text{extreme}) \land \text{block}(2, \text{extreme}) \]
\[ \land \text{block}(3, \text{extreme}) \land \text{block}(4, \text{high}) \land \text{block}(17, \text{high}) \]
\[ \land \text{block}(18, \text{extreme}) \land \text{level}(0) \land \text{health}(20) \land \text{food}(20) \]
\[ \land \text{pMob}(\text{pig}, \text{left}, \text{far}) \land \text{aMob}(\text{zombie}, \text{center}, \text{near}), \]
\[ \land \text{pMob}(\text{pig}, \text{right}, \text{far}) \land \text{groundItem}(17, \text{right}, \text{near}), \]
\[ \land \text{groundItem}(17, \text{left}, \text{far}) \land \text{cluster}(17, 3, \text{left}, \text{far}), \]
\[ \land \text{closestStone}(\text{none}, \text{none}) \land \text{closestWorkbench}(\text{none}, \text{none}) \]
\[ y_{162} = \text{rotating}(\text{sw}), \text{choppingWood}(-1) \]

**Figure 3.2:** An example world state and corresponding action.

![Image of Super Mario](image)

**Figure 3.3:** A screenshot of Super Mario.

Mario, and only 250 if he is Large Mario (Mario starts off as Fire Mario, and drops to Large Mario and then Small Mario when he takes one or two hits from enemies, respectively). This emphasizes Mario’s goal of moving as far right on the level as possible and trying to do so without being hit by enemies.

Specifically, Super Mario exhibits the following properties:

- **Deterministic:** The seed used to generate a Super Mario level determines the position of all tiles, enemies, etc. and the enemies movement/attacks do not have any randomness to them.

- **Partially Observable:** A player cannot see the whole level at any given time, only a few tiles behind Mario and about 2 dozen tiles ahead of him.

- **Dynamic:** The environment includes moving enemies and projectiles (both friendly and harmful), and the player can break some of the blocks.

- **Continuous:** The Super Mario domain is continuous, using real numbers in the physics and location calculations.

This domain exhibits strong sequentiality because how the controlling agent moves will affect what part of the level it sees next, and actions such as jumping require time (to jump successfully, the jump button has to be held for several frames in a row). Specifically, world states in Super Mario are represented by 1083 features representing the surroundings of Mario (information about the terrain, location of enemies/powderups/projectiles, and values such as the number of coins collected). The action space of Super Mario consists of 5 boolean features corresponding to what buttons the agent is pressing in the game (left, right, down, fire/speed, and jump). The demonstrator used is an A* agent used in the 2009 Mario AI Competition\textsuperscript{113} that plays nearly optimally.
Figure 3.4: A screenshot of the Thermometers puzzle game.

The reward function for both Thermometers domains is simply the overall percentage of constraints satisfied for the current board, where there are two kinds of constraints: The number of filled pieces in a row or column matches the number for that row or column, and that each thermometer has a legal configuration (meaning that it is filled starting from the circularly shaped bulb, and all filled pieces are adjacent).

This domain was inspired by the Android Game Thermometers Puzzles\textsuperscript{1}, and created for my work. It has been made publicly available for use in other people’s experiments.

**Thermometers**

The last domain is a puzzle game known as Thermometers (Figure 3.4), where the player sees a board with a collection of thermometers of different lengths and orientations, and needs to figure out how full or empty each of the thermometers is based on a collection of row and column constraints. Boards can be of varying sizes, but 5x5 boards are most commonly used in our experiments as they are sufficiently complex for learning. We have created two different versions of this domain (and are currently using both), Simple-Thermometers and Complex-Thermometers. This domain exhibits the following properties:

- **Deterministic:** All of the available actions have a deterministic result, and there is no randomness in the domain.
- **Fully/Partially Observable:** In the Complex-Thermometers version, the player can see the entire board at once. In the Simple-Thermometers, the player can only see one row or column of the board at a time.
- **Static:** The tiles on the grid never change other than when the player performs actions on them.
- **Discrete:** The board is a discrete grid of tiles, and all actions use the grid locations.

The first variation, Complex-Thermometers, represents the board and the constraints imposed on each row and column of the board as a vector of 245 features. The action space \( \mathcal{Y} \) consists of 75 actions, 3 actions for each tile:

- **fillTile(Tile):** Fill a tile on the board, where \( \text{Tile} \) can be any of the 25 tiles for 5x5 boards

\textsuperscript{1}Available here on the Google Play Store
• emptyTile(Tile): Empty a tile on the board, where \(Tile\) can be any of the 25 tiles for 5x5 boards

• clearTile(Tile): Clear a tile on the board, where \(Tile\) can be any of the 25 tiles for 5x5 boards

The second variation, Simple-Thermometers, represents a single row or column of the board and its constraints as a vector of 14 features. There are 12 actions in total for a 5x5 board:

• fillTile(Tile): Fill a tile in the current row/column, where \(Tile\) can be any value 0-4 for 5x5 boards

• emptyTile(Tile): Empty a tile in the current row/column, where \(Tile\) can be any value 0-4 for 5x5 boards

• moveToNext: Move to the next row or column on the board.

• clear: Set all tiles in a row/column to undetermined and move to the first column (if previously looking at a row) or row (otherwise).
Chapter 4:LfD via Similarity-Based Methods

In this chapter, we provide an initial investigation for subsequent chapters by exploring the performance of LfD approaches in the target domains and set a standard supervised learning baseline. The goal here is twofold: to assess how well LfD approaches perform in these domains, and to assess how much data is needed for these methods to train learners.

The work in this chapter is an investigation of the performance of supervised learning methods, specifically similarity-based ones. In similarity-based methods, a similarity function needs to be created or learned that determines how related two objects (or world states) are. This similarity function is then leveraged in order to learn. One example of this is the well known $k$-nn algorithm\textsuperscript{15}, which gets the closest $k$ world states to the current one and picks the action that corresponds with the most of these states.

There has been a lot of work in using similarity-based methods for LfD. Similarity-based methods have been used in LfD for robotics, such as having a simulated robot avoid obstacles\textsuperscript{23} or teaching a learner to play robot soccer\textsuperscript{92}. These methods have also been used to create cross-domain frameworks and formalisms\textsuperscript{22,60}. Similarity-based methods have also been applied specifically to games, such as teaching a learner to play Real-Time Strategy games\textsuperscript{74}. Since the application of these methods to LfD problems is so common, this was a sensible area to begin our investigation.

Specifically, we compare the performance of different pre-existing approaches as well as a couple novel ones. When creating a similarity-based method, there are several major decisions that need to be made. How the states are going to be represented for learning needs determined, as well as how the method will measure how similar those states are. It also needs to be determined if the original set of features will be used or if some sort of feature selection needs done, and how (if at all) sequentiality will be handled. Therefore, due to their importance in the creation process, these key aspects are what we examined:

- **State Representation:** Deals with how exactly the states are represented within the machine learning system. Concepts such as using a symbolic representation versus a propositional representation and performing preprocessing on the state database are discussed. The proper representation can decrease the amount of training data by making training examples more effective. This can also allow for the creation of specialized algorithms that can more effectively handle LfD problems.

- **Similarity Assessment:** Deals with how similarity between states is determined, both during learning and for how to evaluate learning. When using similarity-based methods, the underlying similarity measure can have a large impact on the performance of the method or how many training instances are needed to train the learner. Understanding these underlying similarity measures can also aid in the creation of specialized algorithms which can better handle LfD algorithms.

- **Feature Selection:** Deals with automatically modifying the set of features that is used for learning to further increase learning. The right set of features can make training examples more effective and therefore reduce the amount of training data needed. This could also lead to more effective algorithms for LfD problems, using feature selection as a preprocessing step.

- **Handling Sequentiality:** Deals with evaluating a common method for attempting to capture sequentiality in LfD systems. By adding data features which capture the sequentiality of LfD problems, we can sometimes reduce the amount of training data needed since the training examples will be more descriptive. As capturing the sequentiality of data is an open problem in LfD, this is an important area to explore.
These comparisons are done to gain an understanding of some of the common LfD techniques which currently exist. By viewing the comparison between these current techniques and how much training data is required in each case, we were able to identify a baseline upon which the thesis work will expound.

4.1 State Representation

How data is represented plays an important role in machine learning. As our application domains have already been presented, in this section we will describe the symbolic representation of our domains.

4.1.1 Trace Collection

To accomplish the goal of learning to perform complex tasks from observing a human, data was gathered in the form of traces. A trace \( R = [(X_0, Y_0), ..., (X_T, Y_T)] \) is a sequence of state-action pairs where each entry \((X_t, Y_t)\) contains the world state \(X_t\) at a given time \(t\), and the actions \(Y_t\) the demonstrator executed at time \(t\).

The world state is represented as logical clauses of the form \(X_t = x_0 \land x_1 \land ... \land x_m\). Each predicate \(x_i\) takes the form of a Prolog term, and represents some aspect of the world. For example, in Minecraft the term \(\text{health}(20)\) would mean that the player has 20 health.

Similarly, actions are also represented as logical clauses \(Y_t = [y_0 \land y_1 \land ... \land y_n]\). Each predicate \(y_i\) represents an action the player is taking at that time, also in the form of Prolog terms. Taking Minecraft as an example, notice that we need to represent actions as clauses, since players might execute more than one action at a time (walking while turning and swinging a weapon), or even no actions. For instance, in the example in Figure 3.2, the player was rotating at the same time as chopping wood (\(\text{choppingWood}(-1)\) means the player is chopping wood with its bare hands).

Some value can also be found in modifying the traces before attempting to learn from them. Four different kinds of traces were experimented with in our learning domain of Minecraft:

- **Original Traces**: Traces where all of the information is recorded exactly as it is collected from the game – including numerical values.
- **Discretized Traces**: Traces which take some of the numerical values and replace them with discretized values, and turns absolute coordinates into relative coordinates. For example, \(\text{rotation}(45, 75)\) might become \(\text{rotation}(\text{NorthWest}, \text{Up})\), or absolute positions might be made relative to the player. Four different discretizations were experimented with for the Minecraft domain, as seen in Table 4.1.
- **Filtered Traces**: Traces which have any instances where the player was doing nothing removed.
- **Discretized-Filtered Traces**: Traces which combine both of these post-processing steps.

4.1.2 Experimental Evaluation

This section examines how learning performance differs between a symbolic and propositional state representation of Minecraft (the latter capturing the same items as the former, but in a feature vector instead of symbolically). We experimented with two procedures: one-vs-one and leave-one-out. For the former, we train on one trace and test on a second trace (for example, given 2 traces you would have 4 datapoints: train on trace 1 and test on trace 1, train on trace 1 and test on trace 2, train on trace 2 and test on trace 1, and train on trace 2 and test on trace 2). For the latter, we train on all but one trace and test on the remaining one (for example, given 3 traces you would have 3 datapoints: Train on traces 1 and 2 then test on trace 3, train on traces 2 and 3 then test on trace 1, and train on traces 3 and 1 then test on trace 2).
Table 4.1: Predicates that are discretized (or transformed from absolute to relative) and the resulting predicates. Note that this table only includes the predicates which are discretized.

<table>
<thead>
<tr>
<th>Original Predicate</th>
<th>Original Values</th>
<th>New Predicate(s)</th>
<th>New Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotating(α, β)</td>
<td>α ∈ [-180, 180]</td>
<td>xRotation(X), yRotation(Y)</td>
<td>Where: X ∈ {Left, Center, Right} Y ∈ {Up, Center, Down}</td>
</tr>
<tr>
<td>rotation(α, β)</td>
<td>α ∈ [0, 360]</td>
<td>rotation(D, H)</td>
<td>Where: D ∈ {S, SW, W, NW, N, NE, E, SE} H ∈ {Down, Center, Up}</td>
</tr>
<tr>
<td>amob(type, xa, ya, za)</td>
<td></td>
<td>amob(type, xr, yr, zr)</td>
<td>Where (xr, yr, zr) are the relative coordinates of an enemy/creature to the player, and type is the specific type of enemy/creature.</td>
</tr>
<tr>
<td>pmob(type, xa, ya, za)</td>
<td></td>
<td>pmob(type, xr, yr, zr)</td>
<td>Where (xr, yr, zr) are integers representing the absolute position of the dropped item/projectile, and type identifies the specific type of item/projectile.</td>
</tr>
<tr>
<td>groundItem(type, xa, ya, za)</td>
<td></td>
<td>groundItem(type, xr, yr, zr)</td>
<td>Where (xr, yr, zr) are integers representing the position of the dropped item/projectile relative to the player, and type identifies the specific type of item/projectile.</td>
</tr>
<tr>
<td>projectile(type, xa, ya, za)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Experimental Setup – Loss Functions

Loss functions are one way to measure how well actions were selected without having the learner actually play the game. To do this, we separated out one trace from the training data to use as the testing data. For each state-action pair in the trace, we compare the action the learner selects against the action the demonstrator actually took at that state, using a loss method. The loss for an entire trace is then the average loss for each state, with a lower loss meaning more accurate predictions by the learner. Three different loss functions were experimented with:

- **0-1 Loss**: returns 1 if the predicted actions match the demonstrator’s actions exactly, and 0 otherwise.
- **Jaccard Loss (JL)**: We define the Jaccard Loss as one minus the Jaccard similarity between the predicted actions and the demonstrator’s actions.
- **Levenshtein Loss (LL)**: Levenshtein distance between the predicted actions and the demonstrator’s actions.

We employed this range of functions because 0-1 loss is very fast, but very coarse in its comparison. Jaccard Loss is more fine grained, but is a little slower, and Levenshtein loss is very fine grained, but is by far the slowest.

Experimental Results

In Tables 4.2 and 4.3, we compared the results against a series of baseline predictors (“Random”, “Most-likely”). The specific similarity functions used are defined in Section 4.2, for now we will focus on comparing across the different representations. The first row (“Random”) in the table shows the results of retrieving one of the instances in the training set at random, and using it as the prediction. The second row (“Most-likely”) shows the results from always predicting the set of actions that was the most common in the training set. The other 4 rows (“Jaccard”, “Levenshtein”, “Weighted Levenshtein”, and “Propositional”) show results for generating a prediction using a nearest neighbor algorithm with the corresponding similarity measure.
Table 4.2: Average loss for various algorithms and loss functions, evaluated using a one-vs-one procedure (lower is better). Results which are statistically significantly better than the rest are underlined.

<table>
<thead>
<tr>
<th></th>
<th>Original 0-1 JL LL</th>
<th>Filtered 0-1 JL LL</th>
<th>Discretized 0-1 JL LL</th>
<th>Filtered/Discretized 0-1 JL LL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Random</strong></td>
<td>0.841 0.823 0.823</td>
<td>0.973 0.919 0.919</td>
<td>0.973 0.919 0.919</td>
<td>0.949 0.907 0.907</td>
</tr>
<tr>
<td><strong>Most-likely</strong></td>
<td>0.630 0.630 0.630</td>
<td>0.794 0.765 0.765</td>
<td>0.630 0.630 0.630</td>
<td>0.776 0.759 0.759</td>
</tr>
<tr>
<td><strong>Jaccard</strong></td>
<td>0.634 0.576 0.542</td>
<td>0.710 0.606 0.534</td>
<td>0.680 0.625 0.601</td>
<td>0.707 0.604 0.560</td>
</tr>
<tr>
<td><strong>Levenshtein</strong></td>
<td>0.619 0.556 0.550</td>
<td>0.688 0.580 0.568</td>
<td>0.667 0.606 0.601</td>
<td>0.685 0.568 0.558</td>
</tr>
<tr>
<td><strong>Weighted Levenshtein</strong></td>
<td>0.619 0.557 0.551</td>
<td>0.696 0.578 0.566</td>
<td>0.662 0.606 0.601</td>
<td>0.682 0.566 0.557</td>
</tr>
<tr>
<td><strong>Propositional</strong></td>
<td>0.663 0.577 0.540</td>
<td>0.722 0.583 0.512</td>
<td>0.649 0.563 0.541</td>
<td>0.672 0.524 0.482</td>
</tr>
</tbody>
</table>

Table 4.3: Average loss for various algorithms and loss functions, evaluated using a leave-one-out procedure (lower is better). Results which are statistically significantly better than the rest are underlined.

<table>
<thead>
<tr>
<th></th>
<th>Original 0-1 JL LL</th>
<th>Filtered 0-1 JL LL</th>
<th>Discretized 0-1 JL LL</th>
<th>Filtered/Discretized 0-1 JL LL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Random</strong></td>
<td>0.841 0.823 0.823</td>
<td>0.973 0.919 0.919</td>
<td>0.973 0.919 0.919</td>
<td>0.949 0.907 0.907</td>
</tr>
<tr>
<td><strong>Most-likely</strong></td>
<td>0.630 0.630 0.630</td>
<td>0.794 0.765 0.765</td>
<td>0.630 0.630 0.630</td>
<td>0.776 0.759 0.759</td>
</tr>
<tr>
<td><strong>Jaccard</strong></td>
<td>0.589 0.480 0.467</td>
<td>0.679 0.505 0.484</td>
<td>0.572 0.462 0.455</td>
<td>0.621 0.444 0.386</td>
</tr>
<tr>
<td><strong>Levenshtein</strong></td>
<td>0.619 0.508 0.496</td>
<td>0.661 0.480 0.395</td>
<td>0.560 0.476 0.425</td>
<td>0.600 0.429 0.374</td>
</tr>
<tr>
<td><strong>Weighted Levenshtein</strong></td>
<td>0.558 0.438 0.389</td>
<td>0.661 0.480 0.395</td>
<td>0.560 0.458 0.425</td>
<td>0.600 0.429 0.374</td>
</tr>
<tr>
<td><strong>Propositional</strong></td>
<td>0.601 0.476 0.424</td>
<td>0.671 0.472 0.382</td>
<td>0.555 0.441 0.414</td>
<td>0.587 0.398 0.350</td>
</tr>
</tbody>
</table>

Overall, it can be seen that discretizing the traces doesn’t seem to improve results using a one-by-one methodology. This modification decreased loss in 1 of the methods (Propositional), had the same loss for one (Most-Likely), and increased loss in four (Jaccard, Levenshtein, Weighted Levenshtein, and Random). Filtering the traces is even worse, as it increases loss for all 6 methods – as does both filtering and discretizing them. When using a leave-one-out methodology, filtering the traces still increases loss in all 6 methods. Discretizing the traces, however, is able to decrease loss for 3 of the methods (Jaccard, Levenshtein, and Propositional), and only decreases them in 2 (Random and Weighted Levenshtein), with Most-likely’s loss not changing. Finally, both filtering and discretizing the traces leads to large loss reductions for 2 methods (Levenshtein and Propositional), and loss increases for 2 methods (Random and Most-likely). The last two methods, Jaccard and Weighted Levenshtein, show an increase in 0-1 loss but a decrease in the other two losses.

4.2 Similarity Assessment

Similarity based metrics such as the $k$-nn algorithm require some way of determining how similar two states are – similarity measures are used to do this. Five similarity measures were used – the first four focus on the symbolic representation that was used during our early work, and the last one operates over propositional data vectors.

- **Jaccard**: Given two clauses (represented as sets of terms), we define the Jaccard similarity as the size of their intersection over the size of their union, which can be characterized by the following equation:

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

where $X \cap Y$ is a clause that contains only those predicates present both in $X$ and in $Y$, and $X \cup Y$ is a clause that contains the union of predicates in $X$ and in $Y$. $|$ represents the number of predicates in a clause. Notice that for a predicate to be in the intersection, it must be a
perfect match. For example, \( \{pMob(pig, -94, 64, 143)\} \cap \{pMob(pig, -94, 64, 142)\} = \emptyset \), since the last parameter does not match.

- **Levenshtein:** Given two clauses, the Levenshtein distance counts the number of edit operations that need to be performed to one of the clauses in order to convert it into the other. The edit operations that are considered are adding a value, removing a value, and substituting a value by another. Our similarity measure between clauses is based on first computing the edit distance between the individual terms that compose the clauses, and then aggregating them to generate the final distance (the Levenshtein similarity is then simply one minus that distance).

To measure distance between two terms, we first represent each of them as a tree, and then use Pawlik and Augsten’s tree edit distance measure\(^{78}\).

Given that a clause can be seen as a set of terms, computing the similarity between two clauses \( X_1 \) and \( X_2 \) with a large number of terms using the edit distance might have a prohibitive cost. For that reason, we employ an approximation algorithm (Algorithm 4). This algorithm works by first computing a matrix with as many rows as terms in \( X_1 \) and as many columns as terms in \( X_2 \). Each position of this matrix contains the similarity between the corresponding row and column terms. This similarity is computed as \( 1 - \frac{d_L(x_i, x_j)}{\max(|x_i|, |x_j|)} \), i.e., by computing the edit distance, normalizing it, and then turning it into a similarity. Here, \(|x_i|\) represents the size of a term, measured as the number of edit operations needed to construct this term from scratch from the empty term. Then, the similarity between the two clauses is computed by adding the maximum similarity value in each column of this matrix, and dividing by the number of terms in \( X_1 \) (the largest of the two clauses).

This approximation algorithm thus has polynomial cost with respect to the number of terms in the clauses, instead of the exponential cost required to compute the exact edit similarity.

- **Weighted Levenshtein:**

In the basic definition of Levenshtein Distance each of the edit operations (adding, removing, or substituting any element or subelement of a tree) is given a cost of 1. To increase the accuracy of this technique, we provided background knowledge about how similar or different the different constants appearing in the terms are. This is done by creating a taxonomy of the different concepts appearing in Minecraft, grouping them by intuitions about their similarity (an excerpt of this taxonomy is shown in Figure 4.1). Then, we used this taxonomy to calculate edge weights between each of the concepts in this taxonomy according to the following formula (adapted slightly from\(^{38}\)):

\[
w(c, p) = \left( \beta + \frac{1 - \beta}{\bar{E}(p)} \right) \left( \frac{d(p) + 1}{d(p)} \right)^\alpha \left[ IC(c) - IC(p) \right]
\]

where \( w \) and \( c \) are concepts, \( \alpha \) and \( \beta \) are constants, \( E(p) \) is the entropy of \( p \), \( \bar{E} \) is the average entropy, \( d(p) \) is the depth of \( p \), and \( IC(x) \) is the information content of \( x \), defined as \( \log^{-1} P(x) \) (where \( P(x) \) is the probability that a given game entity is an instance of concept \( x \)). The distance between any two concepts is then the sum of all edge weights on the shortest path between them. Once these distances are calculated, they are used as the costs for the substitution operation in the Levenshtein distance.

- **MAC/FAC:** The Levenshtein distance is a more fine-grained measure than Jaccard (which operates at the granularity of predicates), but it also has a prohibitive computational cost. In order to cull the runtime to a reasonable level, a technique called MAC/FAC\(^{38}\) was used for retrieval in the implemented CBR system. MAC/FAC uses a two stage retrieval process: given that we want to find the nearest neighbor from a large collection of instances, we first use a computationally cheap similarity measure to filter among those instances, and then use a more expensive method to compute structural matches on the instances returned from the first stage. Specifically, we used a Jaccard Similarity to find the 10 nearest neighbors, and then the Levenshtein distance to choose which among those is the closest.
Algorithm 4.1 clauseSimilarity($X_1, X_2$)

1: if $|X_1| < |X_2|$ then
2: \quad return clauseSimilarity($X_2, X_1$)
3: end if
4: \quad n_1 = |X_1|, n_2 = |X_2|
5: M = n_1 \times n_2 matrix of zeroes.
6: for $x_i \in X_1$ do
7: \quad for $x_j \in X_2$ do
8: \qquad M(i, j) = 1 − δ_L(x_i, x_j)/max(|x_i|, |x_j|)
9: \quad end for
10: end for
11: return $\frac{1}{n_2} \sum_{j=1...n_2} (\max_{i=1...n_1} M(i, j))$

Figure 4.1: An excerpt of the taxonomy of concepts used in the domain of Minecraft.

- \textit{Hamming}: For the propositional data representation, Hamming distance was used, which simply counts the number of features that are an exact match between the two feature vectors that represent the world states.

\section*{Experimental Results}

These results also come from Tables 4.2 and 4.3. For the one-vs-one procedure, the Levenshtein similarities achieve the best results when using the original traces. However, surprisingly, the simple propositional distance with the propositional representation outperforms Levenshtein when the traces are discretized. Although further experiments would be needed to determine the cause, this may be caused by the two-step retrieval process (MAC/FAC) used for Levenshtein, where the first pass uses the Jaccard similarity, thus limiting the performance of Levenshtein. For the leave-one-out procedure, we once again observe that the Levenshtein similarity achieves the best results for the non-discretized traces, but that once discretized, a propositional representation achieves better results.

\subsection*{4.3 Feature Selection and Sequentiality}

Many times, when a domain is being encoded into a structured representation, irrelevant data tends to get encoded together with what the important data - primarily because it is not always clear what is needed for the machine learning algorithm to perform well. This confuses machine learning algorithms and can result in less effective learning. Therefore, feature selection can help extract what data is the most important, and therefore bolster learning.

Features can be roughly defined as the aspects of the data which are used to determine what world state from the learning database is closest to the currently encountered world state.
$z_{162} = \text{holding}(17, 1, 0) \land \text{rotation}(\text{sw, up}) \land \text{selectedBlock}(17) \land \text{target}(\text{none}) \land \text{block}(1, \text{extreme}) \land \text{block}(2, \text{extreme}) \land \text{block}(3, \text{extreme}) \land \text{block}(4, \text{high}) \land \text{block}(17, \text{high}) \land \text{block}(18, \text{extreme}) \land \text{level}(0) \land \text{health}(20) \land \text{food}(20) \land p\text{Mob}(\text{pig, left, far}) \land a\text{Mob}(\text{zombie, center, near}) \land p\text{Mob}(\text{pig, right, far}) \land \text{groundItem}(17, \text{right, near}) \land \text{groundItem}(17, \text{left, far}) \land \text{cluster}(17, 3, \text{left, far}) \land \text{closestStone}(\text{none, none}) \land \text{closestWorkbench}(\text{none, none})$

$y_{162} = \text{rotating(sw), choppingWood}(-1)$

**Figure 4.2:** An example world state and corresponding action from the Minecraft domain.

One well known category of feature selection methods the well known “wrapper” methods. Wrapper methods iteratively invoke a base learning algorithm with different features subsets and evaluate the performance of each in order to find the feature set that maximizes performance. Wrapper methods split the training set $T$ into two parts: the subtraining set $S$ and the cross-validation set $C$. Then, the wrapper method iteratively tests different feature subsets by training the base learning method using $S$ and evaluating the resulting loss using $C$.

In our research, we have examined various strategies for feature selection, most of which are based on wrapper methods:

- **Additive Wrappers**: Wrappers which start with an empty subset of features and greedily select the best feature at each iteration.

- **Subtractive Wrappers**: Wrappers which start with a subset of features and greedily dispose of the features whose removal provides the best results.

- **Filter Methods**: Methods which test each feature individually and gets rid of all but a certain percentage of them.

- **Filtered Wrappers**: Methods which first use a filter, and then perform an additive wrapper on the remaining set of features.

- **Naive Sampling Feature Selection (NSFS)**: This is an alternative, wrapper inspired method which searches for the best feature subset based on Monte Carlo sampling, which uses Naive Sampling $^{75}$.

### 4.3.1 Notation

Here we will introduce the notation used in the feature selection descriptions and algorithms. For the structured representations, we represent both actions and world states as logical clauses of the form $z_t = p_0 \land p_1 \land ... \land p_m$. Each predicate $p_i$ takes the form $h(a_1, ..., a_r)$, where $h$ is the functor and each $a_i$ represents the value that each of the attributes of the predicate takes (note that both $h$ and all the arguments $a_i$ are constants), and represents some aspect of the world. For example, $\text{health}(20)$ would have $\text{health}$ being the functor and 20 being the value of the only attribute for that functor. Let $\mathcal{H} = \{h_0, ..., h_n\}$ be the set of all functors and $\mathcal{A}_h$ the set of all attributes for the functor $h_i$. Actions are also represented as logical clauses, where each individual predicate corresponds to a Minecraft action, and if the player executed more than one action at a time (e.g., walking while swinging a sword), the clause representing the action will have more than one predicate. An example of an observation-action pair can be seen in Figure 4.2.

In structured representations, such as Horn Clauses $^{66}$, one equivalent of selecting features is to select which functors and attributes of each functor should be used. Specifically, we define the “set of features” $F$ for a structured domain as the set of functors being used plus the set of attributes for each functor. Consequently, we define a subset of features $F' \subseteq F$ as a subset of all the functors
Algorithm 5 PW($F, S, C$)

1: $F_{\text{best}} = F_{\text{subset}} = \emptyset, F_{\text{left}} = F, l_{\text{best}} = 1,$
2: while $F_{\text{left}} \neq \emptyset$ do
3: \hspace{1em} $f_{\text{best}} = \arg\min_{f \in F_{\text{left}}} \text{loss}(F_{\text{subset}} \cup f, S, C)$
4: \hspace{1em} $l = \text{loss}(F_{\text{subset}} \cup f_{\text{best}}, S, C)$
5: \hspace{1em} $F_{\text{subset}} = F_{\text{subset}} \cup f_{\text{best}}$
6: \hspace{1em} $F_{\text{left}} = F_{\text{left}} \setminus f_{\text{best}}$
7: \hspace{1em} if $l < l_{\text{best}}$ then
8: \hspace{2em} $l_{\text{best}} = l, F_{\text{best}} = F_{\text{subset}}$
9: \hspace{1em} end if
10: end while
11: return $F_{\text{best}}$

Algorithm 6 FSW($H, S, C$)

1: $F_{\text{best}} = F_{\text{subset}} = \emptyset, H_{\text{left}} = H, l_{\text{best}} = 1,$
2: while $F_{\text{left}} \neq \emptyset$ do
3: \hspace{1em} $h_{\text{best}} = \arg\min_{h \in H_{\text{left}}} \text{loss}(F_{\text{subset}} \cup h \cup A^h, S, C)$
4: \hspace{1em} $l = \text{loss}(F_{\text{subset}} \cup h_{\text{best}} \cup A^{h_{\text{best}}}, S, C)$
5: \hspace{1em} $F_{\text{subset}} = F_{\text{subset}} \cup h_{\text{best}} \cup A^{h_{\text{best}}}$
6: \hspace{1em} $H_{\text{left}} = H_{\text{left}} \setminus h_{\text{best}}$
7: \hspace{1em} if $l < l_{\text{best}}$ then
8: \hspace{2em} $l_{\text{best}} = l, F_{\text{best}} = F_{\text{subset}}$
9: \hspace{1em} end if
10: end while
11: return $F_{\text{best}}$

4.3.2 Wrappers

Propositional Wrapper (PW)

This method uses a greedy search to attempt to find the best possible subset of features (Algorithm 5)\(^{42}\). The algorithm starts with an empty feature set, and iteratively adds the feature which will minimize loss. Once all possible features are added, the subset that yielded the lowest loss is returned. We define the loss function $\text{loss}(\text{Subset}, S, C)$ as training the base learning algorithm on the set of demonstrations $S$ using only the features in $\text{Subset}$, then calculating the loss of the learned classifier using the set of demonstrations $C$.

Functor Structured Wrapper (FSW)

This is a wrapper method that works directly over the structured representation. It also employs a greedy search, but considers only the set of functors, i.e., it does not select among the set of attributes of each functor. As we can see in Algorithm 6, each time a functor $h$ is selected (added to $F_{\text{selected}}$), all of its attributes $A^{h_{\text{best}}}$ are also added to $F_{\text{selected}}$.

Attribute Structured Wrapper (ASW)

The *Attribute Structured Wrapper* (ASW) operates similarly to Functor Structured Wrapper, but also selects which of the attributes of each selected functor to include in the structured representation.
Algorithm 7 ASW(\(F, \mathcal{H}, S, C\))

1: \(F_{\text{best}} = F_{\text{subset}} = \emptyset, F_{\text{left}} = \mathcal{H}, l_{\text{best}} = 1\),
2: while \(F_{\text{left}} \neq \emptyset\) do
3: \(f_{\text{best}} = \text{argmin}_{f \in F_{\text{left}}} \text{loss}(F_{\text{subset}} \cup f, S, C)\)
4: \(l = \text{loss}(F_{\text{subset}} \cup f_{\text{best}}, S, C)\)
5: \(F_{\text{subset}} = F_{\text{subset}} \cup f_{\text{best}}\)
6: \(F_{\text{left}} = F_{\text{left}} \setminus f_{\text{best}}\)
7: if \(f_{\text{best}} \in \mathcal{H}\) then
8: \(F_{\text{left}} = F_{\text{left}} \cup (A_{f_{\text{best}}} \cap F)\)
9: end if
10: if \(l < l_{\text{best}}\) then
11: \(l_{\text{best}} = l, F_{\text{best}} = F_{\text{subset}}\)
12: end if
13: end while
14: return \(F_{\text{best}}\)

Algorithm 8 SASW\(_{\text{min}}\)(\(\mathcal{H}, S, C\))

1: \(F_{\text{best}} = F_{\text{subset}} = \text{FSW}(\mathcal{H}, S, C), l_{\text{best}} = \text{loss}(\text{FSW}(\mathcal{H}, S, C))\),
2: while \(F_{\text{subset}} \neq \emptyset\) do
3: \(f_{\text{best}} = \text{argmin}_{f \in F_{\text{subset}}} \text{loss}(F_{\text{subset}} \setminus f, S, C)\)
4: \(l = \text{loss}(F_{\text{subset}} \setminus f_{\text{best}}, S, C)\)
5: \(F_{\text{subset}} = F_{\text{subset}} \setminus f_{\text{best}}\)
6: if \(l < l_{\text{best}}\) then
7: \(l_{\text{best}} = l, F_{\text{best}} = F_{\text{subset}}\)
8: end if
9: end while
10: return \(F_{\text{best}}\)

(see Algorithm 7). The algorithm also uses a greedy search approach, and each time a functor is selected, the set of its attributes \((A_{f_{\text{best}}} \cap F)\) are added to the set of features to consider in future iterations (lines 7-9). Therefore, the algorithm might select a given functor, but only select a subset of its attributes to be added to the representation.

Subtractive from Min Attribute Structured Wrapper (SASW\(_{\text{min}}\))

One issue with the greedy search approach of ASW is when a functor is added to the selected features, none of its attributes are added initially. Thus, if a given functor is only important if a given attribute is included, it might not be added until very late in the feature selection process. To avoid this, we devised this alternative search process, which first calls Functor Structured Wrapper to obtain the initial set of functors, and then employs a greedy search that subtracts attributes or functors one by one out of the initially selected functors, until no removal can be done without increasing the loss (see Algorithm 8). The algorithm considers both removing a functor \(h\) (along with all its attributes, \(A_{h}\)), and removing just one attribute.

Subtractive from Max Attribute Structured Wrapper (SASW\(_{\text{max}}\))

Many different sets of features often achieve the same loss (since adding irrelevant features sometimes does not reduce the loss). This method operates identically to the previous (SASW\(_{\text{min}}\)), except it uses a modified version of the Functor Structured Wrapper which returns the largest set of features that achieves the minimum loss.
Algorithm 9 PF-ASW($\mathcal{H}, S, C, X$)

1: $F_{\text{best}} = F_{\text{subset}} = \emptyset$, $F_{\text{start}} = \mathcal{H}$, $l_{\text{best}} = 1$, storeLoss = {}
2: for $f \in F_{\text{start}}$ do
3:    storeFeatureAndLossPairs.add(loss(f, S, C))
4: end for
5: storeFeatureAndLossPairs.sortByLoss()
6: $F_{\text{left}} = \text{storeLoss.getBestFeatures}(X)$
7: return ASW($F_{\text{left}}, \mathcal{H}, S, C$)

4.3.3 Filter-Wrappers

Proportion-Filtered ASW (PF-ASW)

This method was created to address the high computational cost of the ASW wrapper. In this method, the features are preprocessed by testing each feature individually and recording the results, then taking the top $X\%$ of them, where $X$ is a user-defined parameter (see Algorithm 9 for the detailed procedure). This selection of top features then becomes the list of selectable features for ASW, the notion being that most of the helpful features will still be maintained while considerably reducing the computational cost of the algorithm.

4.3.4 NSFS

All the wrapper methods above are based on greedy search (either greedily adding features one by one or removing them one by one). This is since searching over the complete space of possible feature subsets using systematic search is unfeasible since there is an exponential number of feature subsets to consider. In this section we present an alternative search method based on Monte Carlo sampling.

The key insight is that feature selection can be seen as a Combinatorial Multi-Armed Bandit (CMAB) problem. A CMAB is an extension of the classic multi-armed bandit (MAB) problem, where at each iteration an agent picks a value for a set of $n$ variables $X = \{X_1, ..., X_n\}$, where variable $X_i$ can take $K_i$ different values in order to maximize the cumulative reward (where the unknown stochastic reward function depends on the selected values). When solving a CMAB, an agent needs to estimate the potential rewards of each variable combination based on past observations, balancing exploration and exploitation in order to find the combination that maximizes expected reward. There is a combinatorial number of possible values (or macro-arms) which the agent can select at each iteration. Additionally, not all possible value combinations might be legal, a function $L: X \rightarrow \{\text{true, false}\}$ determines which of them are.

Given a set of $n$ features $F$, we define a CMAB with $n$ binary variables, where $X_i$ determines whether feature $f_i \in F$ will be selected or not. The legality function $L$ prevents selecting an attribute but not its functor. Naive Sampling Feature Selection (NSFS) exploits such a formalization and works as follows (see Algorithm 10):

- The algorithm runs for a fixed number of iterations $T$ (the computation budget). At each iteration, a CMAB sampling policy is used to select a set of features (line 4).
- With each set of features, the base learning algorithm is trained using $C$, its performance is evaluated by a test set made out of a single instance (a single observation-action pair picked at random from $C$, and removed from the training set before training) to obtain a reward (lines 5-6). Since we employ instance-based learning methods, training time is basically zero.
- At the end of the computation budget, the set of features that has been selected most often (i.e., the one that the sampling policy considers as the one that maximizes the expected reward) is returned as the set of selected features (mostFrequentMacroarm function in the algorithm).
Algorithm 10 NSFS($\mathcal{H}, S, C$)

1: $F = \mathcal{H} \cup \left( \bigcup_{h \in \mathcal{H}} A^{h} \right)$
2: $\text{CMAB} = \text{new with } |F|$ binary variables.
3: for $t = 1..T$ do
4:   $F' = \text{Naïve Sampling}(\text{CMAB}, \epsilon_0, \epsilon_l, \epsilon_g)$
5:   $(z, a) = \text{randomly selected from any } T' \in C$
6:   $r = 1 - \text{loss}(F', S, \{(z, a)\})$
7:   Update $\text{CMAB}$ reward for $F'$ with $r$
8: end for
9: return mostFrequentMacroarm($\text{CMAB}$)

Intuitively, evaluating the performance of each feature subset via testing it against a single instance in each iteration, minimizes the amount of time it takes to obtain a reward for a given feature subset, letting the algorithm test a larger number of feature subsets in the same computation time. However, since we are testing against a single instance, the loss computed for each feature has a large degree of randomness. However, CMAB sampling policies are designed to handle such stochastic reward functions.

Specifically, we employed Naïve Sampling\textsuperscript{75}, a CMAB sampling strategy based on using nested $\epsilon$-greedy sampling strategies. Naïve Sampling has three parameters: $\epsilon_0, \epsilon_l$, and $\epsilon_g$, which are set to 0.3, 0.3, and 0.0 in our experiments, based on preliminary experiments. Naïve Sampling, initially behaves as if it selects macro-arms at random (when no information about the expected reward of each macro-arm is available), and iteratively converges to selecting only those macro-arms that are promising.

4.3.5 Time Windows

For some behaviors, all of the relevant information might not be available in the current state, in which case data from past states should be taken into account. For example, if an agent passes a sign saying “turn right at the next intersection”, when they approach the intersection they need to turn right, but the sign would no longer be in the current state information. For most standard supervised approaches however, only the current world state is taken into account when learning. For LfD problems, one common way of addressing this need for knowledge of past states is time windows\textsuperscript{17}, or simply passing in information from the past $k - 1$ states as input in addition to the current state. This can improve results, but it also increases the computational cost significantly even for low values of $k$.

4.3.6 Experimental Setup

We tested these various methods in all three of our application domains. Experiments in Minecraft consisted of an average trace length of 723 world state-action pairs, and were done using a 10-fold cross validation. Experiments in Super Mario were performed using $N = 1$ traces and $C = 25$ iterations, and experiments in both Thermometers domains were performed using $N = 1$ and $C = 25$ iterations as well. For Minecraft, 10 traces from a human demonstrator were used, but synthetic demonstrators were used for the other three domains. The loss reported for all experiments is the Levenshtein Loss.

4.3.7 Results (Minecraft)

For the Minecraft results, we compared against 3 baselines:

- **All Features Propositional (AFP):** uses a propositional representation where all features are used for learning.
Table 4.4: Average loss for various feature selection methods (lower is better) for various time window sizes ($k$). Experiments were done using an average of 6,507 training instances.

<table>
<thead>
<tr>
<th>Feat. Sel. Method</th>
<th>$k = 1$</th>
<th>$k = 2$</th>
<th>$k = 3$</th>
<th>$k = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFP</td>
<td>0.355</td>
<td>0.329</td>
<td>0.308</td>
<td>—</td>
</tr>
<tr>
<td>AFS</td>
<td>0.374</td>
<td>0.348</td>
<td>0.336</td>
<td>0.308</td>
</tr>
<tr>
<td>R</td>
<td>0.798</td>
<td>0.798</td>
<td>0.798</td>
<td>0.798</td>
</tr>
<tr>
<td>PW</td>
<td>0.333</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FSW</td>
<td>0.265</td>
<td>0.245</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ASW</td>
<td>0.244</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SASW&lt;sub&gt;min&lt;/sub&gt;</td>
<td>0.244</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SASW&lt;sub&gt;max&lt;/sub&gt;</td>
<td>0.244</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>NSFS</td>
<td>0.265</td>
<td>0.269</td>
<td>0.263</td>
<td>—</td>
</tr>
<tr>
<td>PF-ASW&lt;sub&gt;10%&lt;/sub&gt;</td>
<td>0.243</td>
<td>0.210</td>
<td>0.205</td>
<td>—</td>
</tr>
<tr>
<td>PF-ASW&lt;sub&gt;20%&lt;/sub&gt;</td>
<td>0.244</td>
<td>0.211</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>PF-ASW&lt;sub&gt;30%&lt;/sub&gt;</td>
<td>0.247</td>
<td>0.243</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

- **All Features Structural (AFS):** uses a structured representation using all predicates and attributes.
- **Random (R):** This method randomly selects an action from the training data.

Table 4.4 shows the average loss (lower is better) for each feature selection method used, for various time windows. Since even small time windows (of size $k > 1$) can significantly increase runtime, many of the feature selection methods are not computationally feasible to run. A dash (—) in the table indicates that the test did not terminate after 100 hours, and was therefore considered infeasible. Note that these experiments were done using an average of 6,507 training instances, indicating that they would not be feasible for use with a human demonstrator. Table 4.5 shows the loss achieved by NSFS for varying numbers of iterations.

The method that achieved the best results for time windows of size 1 was PF-ASW<sub>10%</sub>, by a very narrow margin. FSW and NSFS performed the worst of the symbolic methods, but not significantly so. In fact, none of the symbolic feature selection methods performed statistically significantly better than any other, according to a 2-tailed $t$-test with $p = 0.05$. However, all symbolic methods performed significantly better than the three baselines. Moreover, FSW, PF-ASW<sub>10%</sub>, and PF-ASW<sub>20%</sub> performed statistically significantly better than both the baselines and PW.

It can also be seen that a greater time window decreases loss, even without feature selection. In fact, using a time window of size 4 gives statistically significantly less loss that a time window of size 1 ($p < 0.05$). Time windows also helped many of the feature selection methods perform better, with the one exception being NSFS. We hypothesize that this is due to having a larger pool of irrelevant features for NSFS, which would make it take more iterations to converge on a subset of features, and also, the high level of randomness introduced by our reward assignment procedure (which we will revise in future work). However, increasing the number of iterations enough to solve this issue caused the algorithm to take longer than the 100 hour cutoff that was employed.

We assessed computational cost by measuring how many times the similarity measure was called during feature selection, since more calls to the similarity measure will typically result in a longer runtime. The number of similarity calls needed for each method with a time window of size 1 is shown in Table 4.6. As can be clearly seen, PF-ASW<sub>10%</sub> and PF-ASW<sub>20%</sub> perform by far the best in this regard, followed by NSFS. The next best are FSW, SASW<sub>min</sub>, and PF-ASW<sub>30%</sub>, which require more calls to the similarity measure, but still considerably less than the other methods. It is also interesting that all of the structured feature selection methods require less calls to the similarity measure, thanks mainly to the fact that attributes are only considered if the corresponding functors are added.
Table 4.5: Average loss of NSFS for varying iterations (lower is better), and approximate number of similarity measure calls needed to perform feature selection for a single test. Experiments were done using an average of 6,507 training instances.

<table>
<thead>
<tr>
<th>NSFS Iterations</th>
<th>Loss</th>
<th>Similarity Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.295</td>
<td>4.5 Million</td>
</tr>
<tr>
<td>2,000</td>
<td>0.288</td>
<td>9 Million</td>
</tr>
<tr>
<td>5,000</td>
<td>0.278</td>
<td>22.5 Million</td>
</tr>
<tr>
<td>10,000</td>
<td>0.274</td>
<td>45 Million</td>
</tr>
<tr>
<td>100,000</td>
<td>0.246</td>
<td>450 Million</td>
</tr>
<tr>
<td>200,000</td>
<td>0.248</td>
<td>900 Million</td>
</tr>
<tr>
<td>500,000</td>
<td>0.246</td>
<td>2,250 Million</td>
</tr>
</tbody>
</table>

Table 4.6: Average loss (lower is better) and required number of calls to the similarity measure during feature selection.

<table>
<thead>
<tr>
<th>Feat. Sel. Method ($k = 1$)</th>
<th>Loss</th>
<th>Similarity Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFP</td>
<td>0.355</td>
<td>N/A</td>
</tr>
<tr>
<td>AFS</td>
<td>0.374</td>
<td>N/A</td>
</tr>
<tr>
<td>PW</td>
<td>0.333</td>
<td>19,876 Million</td>
</tr>
<tr>
<td>FSW</td>
<td>0.265</td>
<td>1,058 Million</td>
</tr>
<tr>
<td>ASW</td>
<td>0.244</td>
<td>8,698 Million</td>
</tr>
<tr>
<td>SASW$_{Min}$</td>
<td>0.244</td>
<td>1,286 Million</td>
</tr>
<tr>
<td>SASW$_{Max}$</td>
<td>0.244</td>
<td>3,273 Million</td>
</tr>
<tr>
<td>NSFS</td>
<td>0.265</td>
<td>450 Million</td>
</tr>
<tr>
<td>PF-ASW$_{10%}$</td>
<td>0.243</td>
<td>214 Million</td>
</tr>
<tr>
<td>PF-ASW$_{20%}$</td>
<td>0.244</td>
<td>356 Million</td>
</tr>
<tr>
<td>PF-ASW$_{30%}$</td>
<td>0.247</td>
<td>623 Million</td>
</tr>
</tbody>
</table>

4.3.8 Results (Super Mario and Thermometers)

This subsection describes the average task reward for each feature selection method used, for various time windows. Since even small time windows (of size $k > 1$) can significantly increase runtime, many of the feature selection methods are not computationally feasible to run in all domains. A dash (—) in the table indicates that the test did not terminate after 1 full day, and was therefore considered infeasible. Note that since these domains use a propositional representation, Attribute Structured Wrapper (ASW) and Functor Structured Wrapper (FSW) are not applicable for them, and have therefore been left out.

Simple Thermometers

Table 4.7 shows the average task reward for each feature selection method used and for various time windows in the Simple Thermometers domain. Unlike in the Minecraft domain, in this domain adding more time windows does not in and of itself increase results. However, it can be seen that increasing the time window does allow feature selection to be more effective up to a point (the three pure wrappers become more effective overall up to time windows $k = 3$, and the three variants of PF-ASW become more effective up to time window $k = 4$). Many of the feature selection methods for this domain improve task reward results over using all of the features. Most notably, SASW$_{min}$ and SASW$_{max}$ increase results considerably (earning 47.39 task reward compared to 30.38) without using time windows of size $k > 1$, and using PF-ASW$_{20\%}$ or PF-ASW$_{30\%}$ in combination with a time window of size $k = 4$ garners the highest reward, nearly doubling the task reward of using all the features and a time window of size $k = 1$ (39.58 reward compared to 30.38).

Table 4.8 shows the loss achieved by NSFS for varying numbers of iterations. We can see that,
Table 4.7: Average task reward for various feature selection methods for various time window sizes \((k)\), for the Simple Thermometers domain. Experiments were done using 1,000 training instances.

<table>
<thead>
<tr>
<th>Feat. Sel. Method</th>
<th>(k = 1)</th>
<th>(k = 2)</th>
<th>(k = 3)</th>
<th>(k = 4)</th>
<th>(k = 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFP</td>
<td>30.38</td>
<td>25.61</td>
<td>25.01</td>
<td>26.89</td>
<td>30.18</td>
</tr>
<tr>
<td>PW</td>
<td>42.02</td>
<td>46.50</td>
<td>58.81</td>
<td>36.54</td>
<td>31.57</td>
</tr>
<tr>
<td>SASW(_{\text{min}})</td>
<td>47.39</td>
<td>52.60</td>
<td>58.81</td>
<td>39.30</td>
<td>32.20</td>
</tr>
<tr>
<td>SASW(_{\text{max}})</td>
<td>47.39</td>
<td>45.63</td>
<td>58.81</td>
<td>23.02</td>
<td>32.20</td>
</tr>
<tr>
<td>PF-ASW(_{10%})</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>48.33</td>
<td>48.33</td>
</tr>
<tr>
<td>PF-ASW(_{20%})</td>
<td>0.00</td>
<td>48.33</td>
<td>48.33</td>
<td>59.58</td>
<td>40.54</td>
</tr>
<tr>
<td>PF-ASW(_{30%})</td>
<td>0.00</td>
<td>48.33</td>
<td>48.33</td>
<td>59.58</td>
<td>40.54</td>
</tr>
</tbody>
</table>

Table 4.8: Average task reward of NSFS for time windows of size \(k = 1\) and for varying numbers of iterations (lower is better), within the Simple Thermometers domain. Experiments were done using 1,000 training instances.

<table>
<thead>
<tr>
<th>NSFS Iterations</th>
<th>Task Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>10.88</td>
</tr>
<tr>
<td>2,000</td>
<td>10.88</td>
</tr>
<tr>
<td>5,000</td>
<td>26.90</td>
</tr>
<tr>
<td>10,000</td>
<td>26.90</td>
</tr>
<tr>
<td>100,000</td>
<td>20.56</td>
</tr>
<tr>
<td>200,000</td>
<td>20.56</td>
</tr>
<tr>
<td>500,000</td>
<td>20.56</td>
</tr>
<tr>
<td>1,000,000</td>
<td>18.58</td>
</tr>
</tbody>
</table>

in this domain, NSFS performs poorly, never performing as well as just using all of the features. We theorize that this is because we test and evaluate using a full playthrough of one board, NSFS might be prone to over-fitting on the training boards, reducing its performance on the boards used to evaluate the results of the feature selection process.

Complex Thermometers

Table 4.9 shows the average task reward for each feature selection method used and for various time windows in the Complex Thermometers domain. Unlike in the Simple Therms domain, increasing the size of the time window does increase the task reward. Also, no feature selection method significantly improves over using all of the features when only dealing with time windows of size \(k = 1\). In fact, only one feature selection method (PW with a time window of size \(k = 2\)) considerably improves over using all of the features. It is also interesting to note that PS-ASW performs very, very poorly in this domain. We believe that this is because no individual feature provides enough information to make decisions on its own, so the starting filter is not able to choose a good set of features from which to perform the wrapper. For both Thermometers domains, feature selection experiments were done using 1,000 training instances, which would require a human demonstrator to make 1,000 moves - too many to be provided from human demonstrators. Table 4.10 shows the loss achieved by NSFS for varying numbers of iterations. Similar to the Simple Thermometers domain, NSFS performs poorly, theoretically for the same reason.

Super Mario

Table 4.11 shows the average task reward for various time windows in the Super Mario domain, the domain with the most features. For this domain, increasing the size of the time window actually considerably decreases results. We theorize that there are already so many features that adding more
Table 4.9: Average task reward for various feature selection methods for various time window sizes (\(k\)), for the Complex Thermometers domain. Experiments were done using 1,000 training instances.

<table>
<thead>
<tr>
<th>Feat. Sel. Method</th>
<th>(k = 1)</th>
<th>(k = 2)</th>
<th>(k = 3)</th>
<th>(k = 4)</th>
<th>(k = 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFP</td>
<td>42.67</td>
<td>47.12</td>
<td>49.87</td>
<td>49.87</td>
<td>49.87</td>
</tr>
<tr>
<td>PW</td>
<td>41.57</td>
<td>55.58</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SASW_{min}</td>
<td>42.96</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SASW_{max}</td>
<td>23.22</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>PF-ASW_{10%}</td>
<td>1.02</td>
<td>3.84</td>
<td>0.53</td>
<td>2.73</td>
<td>7.55</td>
</tr>
<tr>
<td>PF-ASW_{20%}</td>
<td>10.36</td>
<td>3.17</td>
<td>4.28</td>
<td>11.30</td>
<td>—</td>
</tr>
<tr>
<td>PF-ASW_{30%}</td>
<td>12.97</td>
<td>9.51</td>
<td>9.22</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 4.10: Average task reward of NSFS for time windows of size \(k = 1\) and for varying numbers of iterations (lower is better), within the Complex Thermometers domain. Experiments were done using 1,000 training instances.

<table>
<thead>
<tr>
<th>NSFS Iterations</th>
<th>Task Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>4.26</td>
</tr>
<tr>
<td>2,000</td>
<td>11.16</td>
</tr>
<tr>
<td>5,000</td>
<td>11.16</td>
</tr>
<tr>
<td>10,000</td>
<td>11.16</td>
</tr>
<tr>
<td>100,000</td>
<td>11.16</td>
</tr>
<tr>
<td>200,000</td>
<td>2.20</td>
</tr>
<tr>
<td>500,000</td>
<td>2.20</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20.55</td>
</tr>
</tbody>
</table>

Table 4.11: Average task reward for various feature selection methods for various time window sizes (\(k\)), for the Super Mario Domain. Experiments were done using 1,000 training instances.

<table>
<thead>
<tr>
<th>Feat. Sel. Method</th>
<th>(k = 1)</th>
<th>(k = 2)</th>
<th>(k = 3)</th>
<th>(k = 4)</th>
<th>(k = 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFP</td>
<td>1144.30</td>
<td>1014.70</td>
<td>473.28</td>
<td>473.28</td>
<td>612.48</td>
</tr>
</tbody>
</table>

Table 4.12: Average task reward of NSFS for time windows of size \(k = 1\) and for varying numbers of iterations (lower is better), within the Super Mario domain. Experiments were done using 1,000 training instances.

<table>
<thead>
<tr>
<th>NSFS Iterations</th>
<th>Task Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>542.2</td>
</tr>
<tr>
<td>2,000</td>
<td>542.2</td>
</tr>
<tr>
<td>5,000</td>
<td>913.83</td>
</tr>
<tr>
<td>10,000</td>
<td>486.20</td>
</tr>
<tr>
<td>100,000</td>
<td>486.20</td>
</tr>
<tr>
<td>200,000</td>
<td>486.20</td>
</tr>
<tr>
<td>500,000</td>
<td>1100.60</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1100.60</td>
</tr>
</tbody>
</table>

significantly increases the amount of noise. Also note that none of the feature selection methods are included in this table because NSFS was the only method that was able to run in less than the 1-day cutoff, due to the large number of features in the Super Mario domain. For these tests, 1,000 training instances were used – which is still too many to be provided from human demonstrators. Table 4.12 shows the loss achieved by NSFS for varying numbers of iterations. When running NSFS
for a large number of iterations, it does not significantly decrease the task reward, but also does not increase it. Overall, none of the feature selection methods presented here are feasible to run in the Super Mario domain, when considering both time required and benefits.

4.3.9 Feature Selection Conclusions

Section 4.3 presented a collection of feature selection strategies for structured, sequential data, including methods inspired by wrappers, filters, Monte Carlo sampling, and time windows. Our results indicate that, for the Minecraft domain, filter/wrapper hybrid methods such as PF-ASW\textsubscript{10\%} achieve the best tradeoff of performance and computation time – they obtained roughly the same results as the other methods, but required many less calls to the similarity method. It is also interesting to note that although running NSFS for many iterations offered competitive performance, we did not observe significant gains by moving from greedy search strategies to it. Additionally, it can readily be seen that for this domain that including previous states by using time windows does reduce the incurred loss, even without feature selection (although it does increase runtime dramatically).

For the two Thermometers domains, the results for feature selection are much more widely varied. Most of the feature selection methods perform well for the Simple Thermometers domain for some value of $k$, but only PW with a time window of size $k = 2$ performs considerably above just using all of the available features. All of the methods were feasible to run in Simple Thermometers, but several of them were infeasible to run for time windows of larger than size $k = 2$ for the Complex Thermometers domain.

For the Super Mario domain, almost all of the presented methods were infeasible to run due to the domain’s large number of features. The only method that terminated within the 1-day cutoff was NSFS, which roughly matched the performance of using all the features, but did not increase it. In conclusion, although feature selection has the potential to reduce the amount of training data needed (as clearly shown in our experiments), the amount of training data that is needed to perform feature selection is often infeasible for human demonstrators to provide.

4.4 Conclusions

This section provided a comparison of pre-existing approaches (and a couple of novel ones). First, state representation was discussed, including symbolic versus propositional and modifying traces before performing learning. This was followed by similarity assessment, with its description of how to compare two world states. Next various feature selection methods were discussed, including one novel one, NSFS, which offers competitive performance in our domains. Finally, one method for handling sequentiality of data, time windows, was presented and shown to have various effects on learning performance depending on the domain and feature selection method used.

For state representation, we found that modifying the traces does not seem to increase results overall, and in fact tends to decrease them. We also found that the symbolic representation (specifically with Levenshtein similarity) garners the best results for unmodified traces, but once they have been discretized the propositional representation performs better. For feature selection, the filter/wrapper hybrids (PF-ASW\textsubscript{10\%}, PF-ASW\textsubscript{20\%}, and PF-ASW\textsubscript{30\%}) can be said to perform the best in Minecraft and Simple Thermometers (depending on time window size), providing competitive results while requiring many less calls to the similarity method in order to do so. These methods did not perform well in the other two domains, but only one method/time window configuration was able to increase results in Complex Thermometers at all over using all the features, and most methods were infeasible to run in Super Mario (and the one that did, NSFS, roughly tied using all the features). This shows that feature selection is worthwhile in some domains but not all of them. Finally, time window results were quite varied over the domains. Having a larger time window increased results in Minecraft, stayed the same or performed erratically in Simple Thermometers, increased results for most methods in Complex Thermometers, and decreased results considerably in Super Mario. Having now examined the performance of LfD approaches, we will move on to exploring a single class of approaches, known as Active Learning approaches, in depth.
Chapter 5: Active Learning

One of the major open problems in LfD is that when a learning agent moves out of the space for which it has demonstration data, the error starts to compound quadratically (See SubSection 2.1.1 for a more detailed explanation of this issue). One way to account for this is to attempt to give the learning agent demonstration data for situations in which it has none. Active learning is one technique for attempting to do so, by having the agent determine when it needs more data and for where in the state space this data is needed, and then requesting that data from the demonstrator.

One of the state-of-the-art algorithms for this is DAgger. DAgger is an iterative algorithm that works by having the demonstrator control 100% of the time for the first iteration, then exponentially less and less over future iterations, and stores the demonstrator’s actions for each world state (For a more detailed description of DAgger, the reader is referred to SubSection 2.4.3). However, DAgger requires a very high amount of training data and was not created with human demonstrators in mind. Therefore, we propose an algorithm loosely inspired by DAgger that reduces the amount of training data needed and is otherwise more feasible for humans. This algorithm, Selective Active Learning from Traces (or SALT for short), is the main contribution of our work.

5.1 Overview

Let us call \( D_l \) the distribution of states in the training set from where the agent has learned (learning distribution), and \( D_t \) the distribution of states the agent would encounter when executing the learned policy (testing distribution). As shown by Ross and Bagnell, small errors in the trained policy can compound during testing (each small error takes the learning agent further and further from the learning distribution, leading to an increased chance of error, due to LfD’s violation of the i.i.d. principle), making \( D_t \) potentially very different from \( D_l \). Intuitively, SALT is an iterative algorithm inspired by DAgger with the goal of collecting training data so that \( D_t \) and \( D_l \) are as similar as possible. New training data is collected in each iteration, and a new policy is trained with the increased training set. In the first iteration, the demonstrator is asked to provide one or more sample traces, and in subsequent iterations, the learning agent is given control. SALT constantly monitors whether the state the learning agent is in falls outside of \( D_l \), and if that’s the case, the demonstrator is given control until the state is back inside of \( D_l \). Each time the demonstrator takes control, more training data is generated and added to the training set. In this way no stochastic mixing of demonstrator and learning agent is required, nor is there any need for asking the demonstrator to provide actions for all the states visited by the learning agent (unlike DAgger), thus minimizing human demonstrator cognitive burden.

The key problem is determining when to give control to the demonstrator, and when to give control back to the learning agent. Three strategies are used for these decisions (A visualization of how these strategies work can be seen in figure 5.1):

- \( \rho_s \): determines when the learner has moved out of \( D_l \).
- \( \rho_b \): when control is given to the demonstrator, it might be interesting to back-up the world for a few time instants, to collect training data on the sequence of states that led to the learning agent falling outside of \( D_l \). This strategy determines how far the world state should be backed up before allowing the demonstrator to perform the task.
- \( \rho_d \): determines how long the demonstrator should perform the task before the learner is given back control.
Protocol 1 ($\rho_s$)
Protocol 2 ($\rho_b$)
Protocol 3 ($\rho_d$)

SALT Strategies

**SALT basic flow:**
1. Run the demonstrator for the first iteration, to get training data
2. Let the learner determine when it has moved out of the demonstrated space (active learning)
3. Backup the world state for a few time steps
4. Let the demonstrator then take over for a while
5. Switch back to the learner and repeat

---

**Figure 5.1:** Illustration of how the three strategies used in SALT govern its behavior.

### 5.2 The SALT Algorithm

*SALT* (Selective Active Learning from Traces) starts with an empty training set (see Algorithm 11 for the detailed procedure). *SALT* performs $N$ iterations, and at each iteration it uses the learner to generate $C$ traces. The first iteration of the algorithm has the demonstrator control, and those traces become the training data for the second iteration. For each iteration excluding the first, the learner performs the task until strategy $\rho_s$ determines that the learner has moved out of $D_l$. When it is determined that this has happened, the state is backed up a certain number of ticks, as determined by $\rho_b$, and then the demonstrator is given control until the state goes back into $D_l$, as determined by strategy $\rho_d$. The states that the demonstrator encounters during its control and the actions that it takes are added to the training data for the next iteration.

**Algorithm 11** $SALT(\rho_s, \rho_b, \rho_d, C, N)$

1: Sample $C$-step traces using $\pi^*$ (the demonstrator’s policy)
2: Initialize $D \leftarrow \{(s, \pi^*(s))\}$ ∕ all states visited by the demonstrator and the actions it took
3: Train classifier $\pi_1$ on $D$
4: for $i = 1$ to $N$
5: Initialize $D_i \leftarrow \emptyset$
6: for $j = 1$ to $C$
7: $D_i = D_i \cup \text{runOneTrace}(\pi_i, \rho_s, \rho_b, \rho_d)$
8: end for
9: Aggregate datasets: $D \leftarrow D \cup D_i$
10: Train classifier $\pi_{i+1}$ on $D$
11: end for
12: return best $\pi_i$ on validation data

---

### 5.3 Strategies

We experimented with several versions for some of the three strategies that make key decisions in *SALT*. There are two versions of these strategies: reward-requiring strategies and reward-free strategies. This section focuses on reward-requiring strategies, with reward-free strategies being discussed in section 5.6. For reward-requiring strategies, we assume the learning agent has access
Algorithm 12 runOneTrace(π, ρs, ρb, ρd)

if not outside of $D_l$ according to $ρ_s$ then
    sample using $π$
else
    back up world according to $ρ_b$
    sample using $π^*$ according to $ρ_d$
end if
return $\{(s, π^*(s)) | s \in S^*\}$, where $S^*$ is the set of all states where $π^*$ was used.

to a reward function $r$, which assigns a numeric reward $r(s)$ to any world state $s$. Given a trace $T_i = [(s_1, y_1), ..., (s_n, y_n)]$, we define the accumulated reward at a time step $j$ as:

$$R(T_i, j) = \sum_{k=1}^{j} r(s_k)$$

Now, given training data $T = \{T_1, ..., T_m\}$, composed of a collection of traces collected from the demonstrator, we can estimate the expected accumulated reward that the demonstrator obtains after having performed the task for $j$ time steps as:

$$ER(T, j) = \frac{1}{m} \sum_{i=1}^{m} R(T_i, j)$$

The reward-requiring strategies use the difference between the expected accumulated reward of the demonstrator and that achieved by the learning agent as a proxy for whether or not the learner has moved out of $D_l$. We experimented with different alternatives for each strategy, listed below.

5.3.1 $ρ_s$ Strategy

For $ρ_s$, which determines when the learner has moved out of $D_l$, we experimented with the following strategies:

- $ρ_s^{SD}$ (Simple Deterministic): signal that the learner has moved out of $D_l$ when $ER(j) - R(j) > κ$, where $j$ is the current time step, and $ER(j)$ and $R(s)$ are defined as above, and $κ$ is a predefined constant.
- $ρ_s^{SS}$ (Simple Stochastic): signal that the learner has moved out of $D_l$ with probability $P = \frac{|ER(j) - R(j)|}{max}$, where $j$ is the current time step and $max$ is the maximum possible reward for the domain.
- $ρ_s^\leq$ (Reward Decreases): signal that the learner has moved out of $D_l$ if the learner’s reward has decreased compared to the previous time step.
- $ρ_s^\geq$ (Reward Doesn’t Increase): signal that the learner has moved out of $D_l$ if the learner’s reward has not increased compared to the previous time step.
- $ρ_s^{DCA}$ (Directly Compare Actions): signal that the learner has moved out of $D_l$ if the learner’s action differs from the demonstrator’s action more than a threshold value $α$ according to a modified edit distance.

5.3.2 $ρ_d$ Strategy

For $ρ_d$, we experimented with one strategy:

- $ρ_d^{RG}$ (Reward-Goal), which signals to give control back to the learner when the accumulated reward reaches the expected accumulated demonstrator reward $ER(j)$, where $j$ is the time step at which the demonstrator took control.
$\rho_b$ Strategy

For $\rho_b$, which determines how far to back up the world state, we experimented with one strategy:

- $\rho^b_n$ (Back-$n$), which backs up the world state $n$ time steps. We have experimented with different values for $n$, eventually settling on $n = 0$ for most of our experiments.

5.4 Experimental Setup

We experimented with SALT, using both J48, a modified version of C4.5$^{44}$, and $k$-nn$^{15}$ with $k = 9$, an instance-based classifier, as underlying learners. For the results detailed in this chapter, $k$-nn was used. For the results in future chapters, J48 was used as the underlying learner unless otherwise noted. We used Jaccard similarity as the underlying similarity measure. However, Levenshtein similarity yields a more accurate comparison of states. In order to ensure that SALT was not being limited by the choice of similarity measure, we also experimented with using the Levenshtein similarity (or rather, the matrix approximation discussed in Section 4.2) as the similarity measure.

Experiments for Super Mario were ran for $N = 10$ iterations of $C = 10$ traces each and experiments for Complex Thermometers were ran for $N = 20$ iterations of $C = 20$ traces each, and each Thermometers board had a move limit of 50 moves (and therefore a maximum length of 50 training instances per trace). Furthermore, we compared the performance of SALT to two baselines:

- *Supervised*: we evaluated the performance of a base supervised learner as the training set increases in size proportionately to the other methods. This data is generated by having the demonstrator complete the task until we have roughly the same amount of training data as for the other methods, and then training the learner using only that.
- *DAgger*: we also evaluate the performance of DAgger$^{95}$ in our domains for the sake of comparison.

5.5 Results

The plots in Figure 5.2 show the average reward achieved by the different methods we tested (vertical axis), as a function of the number of training instances in their training sets (horizontal axis). The left-hand side plot shows results for Super Mario, and the right-hand side plot shows results for Thermometers.

Concerning varying the underlying learning method (using Jaccard vs Edit similarity), we found that in one domain changing the similarity measure to our version of edit distance made SALT perform better, and in the other it made SALT perform worse. For the Mario domain (Figure 5.2.a), both Simple Stochastic and Simple Deterministic benefited by using edit similarity (note that the Edit Stochastic line stops at a considerably higher reward than the Simple Stochastic line, and likewise with the Deterministic lines). For the Thermometers domain (Figure 5.2.b) The exact opposite trend can be noticed – The Edit Stochastic line stops at a considerably lower reward than the Simple Stochastic line, and likewise with the Deterministic lines. Two things can be concluded from this. First, Jaccard distance was not a limiting factor for SALT, and second, it was not worth using Levenshtein similarity as the main similarity measure for $k$-nn (recall that Levenshtein similarity is much, much slower that Jaccard similarity).

The different plots in Figure 5.3 also show the average reward achieved by the different methods we tested (vertical axis), as a function of the number of training instances in their training sets (horizontal axis). The left-hand side plots (a, b) show results for Super Mario, and the right-hand side plots show results for the original symbolic representation of the Complex Thermometers domain.

When examining the strategies in the Mario domain, the one that boasts the highest performance is the $\rho^b_S$ strategy. As can be seen in Figure 5.3.a, This strategy trains faster than $\rho^S_S$ or $\rho^S_D$ and levels off a little higher (around 600 reward at 2,000 training instances compared to 550 and 430 reward for the other two polices). The next best strategy is $\rho^S_S$, which outperforms the other SALT
strategies, followed by $p_s^{DCA}$ (specifically with a threshold of .21), meaning that the agent can only differ from the demonstrator on whether a single action is performed or not (recall that the Mario domain has 5 actions, any subset of which can be performed simultaneously). Figure 5.3.b shows that this strategy offers performance in between our strategies of $p_s^{SD}$ and $p_s^{SS}$, leveling off at about 500 reward compared to 550 for $p_s^{SS}$ and around 475 for $p_s^{SD}$. It is also important to note that all of the SALT strategies perform much better than the standard supervised approach, but most of them perform significantly worse than DAgger (except $p_s^{≤}$ which comes close to tying it).

Similar results can be seen for the Thermometers domain. Figure 5.3.c shows that the Does Not Increase strategy performs better than the other strategies, leveling out at about 21% of constraints satisfied compared to 18% for Simple Stochastic and 16% for $p_s^{SS}$ and $p_s^{DCA}$ (Figure 5.3.d) with a threshold of 0.35 (meaning that the learner can differ from the demonstrator either on what change to make to the tile or one of the two coordinates of the tile to change, but not more) approximately tie, outperforming $p_s^{SD}$ with a reward of 18% of constraints satisfied at 5,000 training instances compared to 15%. In this domain, Standard Supervised performs the best out of the tests we ran, although at least one SALT strategy ($p_s^{≥}$) comes close to tying it. However, DAgger performs poorly in this domain. Our hypothesis for the low performance of DAgger in this domain is that the current world state representation of the domain makes this a very hard problem for LfD to solve, making DAgger struggle. We verified that in a simpler setting of the domain (3x3 boards) DAgger would eventually start increasing performance after collecting enough training data.

### 5.6 Reward-Free SALT Strategies

In addition to strategies which rely on the existence of a reward function in order to estimate when the learning agent had moved to an area of the state space not covered by $D_l$, we also derived strategies which do not. In this section we cover our work trying to lift this restriction on the algorithm by defining strategies which do not require a reward function.

#### 5.6.1 Reward-free SALT strategies

Specifically, we examined two reward-free variants of $p_s$, which determines when the learner has moved out of $D_l$.

---

Figure 5.2: The left graph shows the amount of reward gained in Super Mario (vertical axis) as a function of the amount of training data (horizontal axis) for various SALT strategies and baselines, and the right graph shows the percentage of constraints satisfied (vertical axis) as a function of the amount of training data (horizontal axis) for 5x5 Thermometers grids, for various SALT strategies and baselines. Note the lines are different lengths because all methods are run for the same number of iterations, but some query the demonstrator more and therefore gather more training data.
Figure 5.3: The left column of graphs shows the amount of reward gained in Super Mario (vertical axis) as a function of the amount of training data (horizontal axis) for various SALT strategies and baselines, and the right column of graphs shows the percentage of constraints satisfied (vertical axis) as a function of the amount of training data (horizontal axis) for 5x5 Thermometers grids, for various SALT strategies and baselines. Note that for graph c, some of the strategies gave identical results, so some of the lines perfectly overlap.

- $\rho_{QBC-m}$ (Query By Committee): Train $m$ smaller learners, each on 20% of the training data selected at random. Have each smaller learner predict an action; if these actions are not all the same, signal that the learner has exited $D_l$. In this paper, we set $m = 5$.

- $\rho_{MD}^m$ (Minimum Distance): using Euclidean distance, the most similar world state $s^*$ from the set of states the demonstrator visited during the first iteration of SALT to the current world state $s_j$ is found. If the distance between $s_j$ and $s^*$ is higher than a threshold value $\alpha$, signal the learner has exited $D_l$.

Although this strategy itself was not new, we further expanded this concept into 5 variants. For all variants, let $c(s)$ represent the minimum distance from state $s$ in $D_l$ to any other state in $D_l$. Then let $E$ represent the set of all $c(s) \mid s \in D_l$.

- $\rho_{MinMD}^s$: Set $\alpha$ to the lowest value in $E$, $\alpha = \min(E)$
- $\rho_{LowestQuartileMD}^s$: Set $\alpha$ to the first quartile value in $E$.
- $\rho_{AverageMD}^s$: Set $\alpha$ to the average value in $E$.
- $\rho_{HighestQuartileMD}^s$: Set $\alpha$ to the third quartile value in $E$.
- $\rho_{MaxMD}^s$: Set $\alpha$ to the highest value in $E$, $\alpha = \max(E)$

Note that only $\rho_{MD}^s$ attempts to directly measure whether or not the learner is in $D_l$. The other methods are proxies which attempt to determine this without direct measurement.
We also examined two possible variants of \( \rho_d \), which determines how long to give control to the demonstrator (we did not experiment with strategies that depend on a reward function for \( \rho_d \)):

- \( \rho_d^{MD} \) (Minimum Distance): Signal to give control back to the learner when the minimum distance from the current state to the closest state in the demonstrator’s training data (the first iteration of SALT) is less than a threshold value \( \alpha \). Analogous variants to the 5 variants of \( \rho_s^{MD} \) above were tested.

- \( \rho_d^{BC-m} \) (Query By Committee): Train \( m \) classifiers, each on 20% of the training data, selected at random (just as in \( \rho_s^{BC-m} \)). Have each smaller learner predict an action, and signal to give control back to the learner if they all agree on a move. Just as in \( \rho_s^{BC-m} \), we set \( m = 5 \).

Finally, only one variant of \( \rho_b \) was used:

- \( \rho_b^0 \) (Back-0): Does not back up the world state when the demonstrator is given control.

### 5.6.2 Reward-Free Strategies Versus Reward-Requirements

The purpose of the experiments in Figure 5.4 (left) were to compare SALT methods which require a reward function for the domain to those which do not. Comparisons of SALT to the baselines are performed later in this section, and are shown in 5.4 (right).

The results of these experiments for the Simple Thermometers domain are shown in Figure 5.4.a. These graphs show the task reward gained (y-axis) for the amount of training data gathered (x-axis). Note that different strategies will query the demonstrator a different number of times, leading to some lines being shorter than others, since all were run for the same number of iterations but some obtain more training data than others. For example, \( \rho_d^{MaxMD} \) with \( \rho_s^{MaxMD} \) queries the demonstrator very little, resulting in a very short line. For this domain, SALT with \( \rho_s^{MinMD} \) and \( \rho_d^{MinMD} \) performs best, reaching about 37% of constraints satisfied at 300 training instances and roughly 48% of constraints satisfied by about 950 training instances. Comparing the SALT policies that require a reward function (the dotted lines) with those that do not, we can see that most of the latter perform the same or better than the former — only SALT with \( \rho_s^{MinMD} \) and \( \rho_d^{MinMD} \) is significantly outperformed by at least one of those methods. This shows that, for this domain, SALT learns just as well without a reward function as it does with one.

Results for the Complex Thermometers domain are shown in Figure 5.4.b. For this domain, all of the variants perform roughly the same. Some variants, such as \( \rho_s^C \) and \( \rho_d^{AverageMD} \), dip above and below the general trend (5% at just over 200 training instances and 30% at 400 training instances, compared to 9% and 27% for most of the other variants), but no method performs significantly better than any other. We believe this is because two blank Complex Thermometers boards are very far from each other in the state space. Many of these strategies directly use the distance between states — meaning every time a new board is presented it is so far away from the previous boards in the state space that the differences in the distance based strategies (\( \rho_s^{MD} \) and \( \rho_d^{MD} \)) are very small in comparison. All of the variants performing roughly the same provides further evidence that SALT can learn just as well without a reward function as with one.

Results for the Super Mario domain are shown in Figure 5.4.c. It is interesting to note that unlike in the previous domains, every variant except one which does not require a reward function performs significantly worse than those that do. We believe this is because of the differences in the reward function between this and the thermometers domains. In the thermometers domain, reward is not gained until a constraint is met — this means it often takes multiple correct actions to actually see an increase in reward. In the Super Mario domain, however, moving to the right even one pixel will increase the amount of gained reward. Therefore, the Super Mario reward function is more fine-grained, so the methods which use it are better able to leverage it, making them more effective in this domain. That being said, one method that does not require a reward function does perform competitively: \( \rho_s^{QBC-5} \) and \( \rho_d^{QBC-5} \). This method provides similar if not better performance, reaching 800 reward at about 800 training instances compared to only about 700 reward at the same number of training instances.
Figure 5.4: Task reward (y-axis) plotted against amount of training data (x-axis), for various learning methods. Note both \( \rho_s \) and \( \rho_d \) are stated for each SALT variant, and that dotted lines represent SALT variants that require a reward function. The left side (a,b,c) compares various SALT variants against each other, and the right side (d,e,f) compares SALT against our baselines.
Overall, $SALT$ with $\rho_{QBC}^{QBC-5}$ and $\rho_{QBC}^{QBC-5}$ garners the best cross-domain results, performing among the best in all three domains. More importantly, it does so while also removing the restriction of needing a reward function for the domains – effectively removing a restriction from the $SALT$ algorithm itself.

### 5.6.3 Reward-Free Strategies Versus Baselines

Now that we have presented a new $SALT$ variant which does not require a reward function that performs competitively with those which do, we need to compare this new variant to our baselines. Figures 5.4.d, 5.4.e, and 5.4.f show the results of comparing $SALT$ with $\rho_{QBC}^{QBC-5}$ and $\rho_{QBC}^{QBC-5}$ against other state-of-the-art methods for the Simple Thermometers, Complex Thermometers, and Super Mario domains, respectively. For the two thermometers domains, our new $SALT$ variant performs by far the best, followed by Standard Supervised and then $DAgger$ / $SafeDAgger$. Note that $DAgger$ and $SafeDAgger$ appear to perform very poorly in these domains. As discussed in Section 6.1, $DAgger$ and $SafeDAgger$ both add many duplicates to the training data, and methods which add less duplicates to the training data tend to perform much better in these domains.

For the Super Mario domain (Figure 5.4.f), $SALT$ performs competitively with (and arguably better than) $DAgger$. Notice that the plot corresponding to $SALT$ ends considerably earlier: all algorithms are ran for the same number of iterations, but different algorithms will query the demonstrator different amounts and therefore obtain different amounts of training data. However, the data that $SALT$ gets is more effective, leading to it training faster than $DAgger$ does given the same amount of training data. $SafeDAgger$ is significantly worse than either of them, with Standard Supervised garnering the least task reward. Therefore, since $SALT$ performs much better in two domains and competitively in the third, we can conclude that $SALT$ offers the best overall performance in our domains. Note that we also ran these experiments with $k$-nn, but the graphs are omitted because no differences in trends were exhibited.

### 5.7 Conclusions

In this chapter we presented the $SALT$ algorithm, an Active Learning from Demonstration algorithm which determines when and how long to query the demonstrator for more data based on a collection of strategies. We provided a collection of these strategies and provided an empirical evaluation of $SALT$’s performance across strategy variations and against baselines. In particular, two types of strategies were discussed: those which require a reward function and those which do not. It was found that at one configuration of $SALT$ using only reward-free strategies was able to perform competitively with or outperform all of the reward-requiring strategies while offering the additional benefit of not needing a reward function for the domain. In addition to comparing various $SALT$ strategies against each other, we have also compared them against several baselines (Standard Supervised, $DAgger$, and $SafeDAgger$) and found that the reward-free configuration of $SALT$ is able to considerably outperform every baseline in every domain except one, performing competitively with $DAgger$ in Super Mario. We now set out to understand how $SALT$ works to be able to improve upon its viability for human demonstrators.
Chapter 6: Understanding the Behavior of SALT

In this chapter, we seek to understand how SALT works, and why its performance differs from other related state-of-the-art-methods. This is important because understanding how SALT works behind the scenes will help us to improve upon the algorithm, making it even more effective at learning as well as more feasible for human demonstrators. In order to do this, we empirically analyze two aspects of the learners trained by each algorithm:

- Examination of Duplicated Data: Given that SALT is an active learning algorithm, more and more data is acquired from the demonstrator at each iteration. However, nothing in the algorithm ensures that the data being collected is new. We noticed that during training, the same state would often get visited many times, which would create duplicates in the training data. The most egregious case of this was DAgger in the Complex Thermometers domain, which revisited states so often that 89.6% of its training data was duplicates by the end of training. As this is a significant amount of training data, it might affect learning as well as the behavior of the algorithm in general. Therefore, understanding 1. how many duplicates in the training data each algorithm accrues and 2. how those duplicates actually affect learning are vital for understanding the behavior of SALT compared to other state-of-the-art methods.

Specifically, we examined how much duplicated data SALT and 3 baselines acquire for 3 domains. Furthermore, we directly compare these results to the general task reward results of each algorithm to see if there is any correlation between the number of duplicates and the actual task reward. We then examine how the performance changes when all duplicate instances are removed from the training data, to see if this improves results.

- Examination of Training Set and Test Set Distributions: As explained in Section 5.1, one of the key problems in LfD is that the set of states encountered during learning, $D_l$, tends to differ significantly from the set of states encountered during testing ($D_t$). In order to further understand the differences between SALT and DAgger, we examined the distribution of states encountered by each during both learning and testing.

Specifically, we examined which algorithms are most effective at making $D_l$ and $D_t$ match in two domains (Simple Thermometers and Complex Thermometers), as this is one of the goals Active LfD approaches try to accomplish. We also examine how this affects their performance, as there are theoretical results in the literature which state that when $D_l$ and $D_t$ differ, error compounds much faster than when they are the same.94

6.1 Examination of Duplicated Data

During our experiments, we noticed that some methods would visit the same world state (or a handful of world states) many times. Since this has the potential to add a significant amount of duplicate training data, which in turn has the potential to significantly affect learning, we decided to analyze the amount of duplicate training instances that each method adds to the training data during training. This section will examine the amount of duplicates that each algorithm generates versus the performance of those algorithms, and then examine what happens when duplicates are removed from the training data before evaluation.

6.1.1 Experimental Setup

We examined the amount of duplicate data accrued for 4 methods (SALT DAgger PseudoSafeDAgger and Standard Supervised) across three domains (Simple Thermometers, Complex Thermometers, and Super Mario – as defined in Chapter 3). This was done to see if there were any cross-domain
Table 6.1: This table shows the average proportion of unique instances in the training data (using J48 as an underlying learner) for each method and each domain (Simple Thermometers, Complex Thermometers, Super Mario, and averaged across all domains, in that column order). Lower numbers indicate that there are more duplicated instances in the training data.

<table>
<thead>
<tr>
<th>Method</th>
<th>ST</th>
<th>CT</th>
<th>SM</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Supervised</td>
<td>0.436</td>
<td>0.988</td>
<td>0.677</td>
<td>0.700</td>
</tr>
<tr>
<td>DAgger</td>
<td>0.290</td>
<td>0.104</td>
<td>0.724</td>
<td>0.373</td>
</tr>
<tr>
<td>PseudoSafeDAgger</td>
<td>0.304</td>
<td>0.154</td>
<td>0.736</td>
<td>0.398</td>
</tr>
<tr>
<td>SALT with $\rho_s^{QBC-m}$ and $\rho_d^{QBC-m}$</td>
<td>0.792</td>
<td>0.984</td>
<td>0.734</td>
<td>0.836</td>
</tr>
</tbody>
</table>

Trends related to the amount of duplicates in the training data. These experiments were also done with two different underlying learners, J48 and $k$-nn. The results reported represent the proportion of instances that were unique for each method and domain — that is to say, if we removed all but one of every duplicate instance in the training data, what proportion of the original amount of training data is the new amount of training data.

World states in Super Mario are represented by a collection of 1083 features representing the surroundings of Mario (whether or not there is a wall, where the enemies are, etc.). The action space $Y$ of Super Mario consists of 5 boolean features corresponding to what buttons the agent is pressing in the game (left, right, down, fire/speed, and jump). Complex-Thermometers represents the board and the constraints imposed on each row and column of the board as a vector of 245 features. The action space $Y$ consists of 75 actions, 3 actions for each tile on the grid (set to “full”, set to “empty”, and “clear”). For the second, Simple-Thermometers, states are a vector of 14 features which represent a single row or column of the board, and the actions are to fill a tile, empty a tile, move to the next row/column, or set all tiles in the row/column to undetermined and move to the next one.

Experiments for Super Mario were performed using $N = 1$ traces and $C = 25$ iterations. Each trace has a time limit of 30 seconds, and data is every tick (where the game runs at 20 ticks per second), leading to a maximum trace length of 450 training instances. The reward function used is simply Mario’s X-coordinate in pixels plus a bonus of 500 if he is Fire Mario, and only 250 if he is Large Mario (Mario starts off as Fire Mario, and drops to Large Mario and then Small Mario when he takes one or two hits from enemies, respectively).

For both Thermometers domains, experiments were performed using $N = 1$ traces and $C = 25$ iterations. Both Thermometers domains have a limit of 100 moves per board, and therefore a maximum length of 100 training states per trace. The reward function for both Thermometers domains is simply the overall percentage of constraints satisfied for the current board, where there are two kinds of constraints: The number of filled pieces in a row or column matches the number for that row or column, and that each thermometer has a legal configuration (meaning that it is filled starting from the circularly shaped bulb, and all filled pieces are adjacent).

6.1.2 Results - J48

Table 6.1 shows the average proportion of unique instances in the training data compared to the total amount of training data, for each learning method and domain (using J48 as the underlying learner). SALT has the least duplicates, getting or roughly tying for the highest proportion of unique instances in each domain, and having the highest average proportion of unique instances across domains. Standard Supervised is next, having the second highest average proportion of unique instances. This is because of its high proportion of duplicates in the Simple Thermometers domain, likely because solving puzzles without errors (as Standard Supervised does) naturally leads to encountering the same state many times (for example, a row where 5 tiles need to be filled and all 5 tiles are currently filled will be encountered at least once for every time a 5 is present in a puzzle, and can be encountered more than once as the move() action is used to work through the puzzle).
Table 6.2: This table shows the average proportion of unique instances in the training data (using k-nn as an underlying learner) for each method and each domain (Simple Thermometers, Complex Thermometers, Super Mario, and averaged across all domains, in that column order). Lower numbers indicate that there are more duplicated instances in the training data.

<table>
<thead>
<tr>
<th>Method</th>
<th>ST</th>
<th>CT</th>
<th>SM</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Supervised</td>
<td>0.436</td>
<td>0.995</td>
<td>0.676</td>
<td>0.704</td>
</tr>
<tr>
<td>DAgger</td>
<td>0.204</td>
<td>0.09</td>
<td>0.714</td>
<td>0.336</td>
</tr>
<tr>
<td>PseudoSafeDAgger</td>
<td>0.274</td>
<td>0.163</td>
<td>0.631</td>
<td>0.356</td>
</tr>
<tr>
<td>SALT with $\rho^QBC_m$ and $\rho^QBC_d$</td>
<td>0.691</td>
<td>0.747</td>
<td>0.695</td>
<td>0.711</td>
</tr>
</tbody>
</table>

DAgger and PseudoSafeDAgger gather the most duplicates in two of the domains, and have many more duplicates (a much lower proportion of unique instances) on average across the domains than the other two methods. As the results for Super Mario are fairly close for all 4 methods, it is clear DAgger and SafeDAgger’s high proportion of duplicates comes from the Thermometers domains. This is because in those domains, it is a perfectly valid action to fill a tile that’s already filled or empty a tile that’s already emptied. If such an action is made, the world state is not changed. If the demonstrator is then asked to provide an action for that world state, they will then provide the same action again (at least for the synthetic demonstrators used in these experiments), resulting in a duplicate instance being placed in the training data. Since after the first few iterations of DAgger the learner is almost always in control but an action is always requested from the demonstrator, this results in the same instance being added many times as the learner keeps re-making the same action that does not change the state. For SafeDAgger, what tends to happen in practice in the Thermometers domains is that control “oscillates” – the demonstrator is put in control for a single action for state $X_i$, then the learner takes over in the next state $X_{i+1}$ and chooses an action which moves back into state $X_i$ and this process is repeated, leading to many duplicates of state $X_i$ and the demonstrators action for it in the training data.

Furthermore, if we compare these results to Figures 5.4.d, 5.4.e, and 5.4.f, which show the task reward versus training data for these same tests, we can notice an interesting trend. SALT, which has the least duplicates in the training set, performs the best in all three domains, training faster or getting a higher amount of reward. Furthermore, for the Simple Thermometers domain (Figure 5.4.d), the list of best performing methods matches the list of lowest duplicates (SALT followed by Standard Supervised followed by DAgger and PseudoSafeDAgger). For Complex Thermometers (Figure 5.4.e), the two methods which get the lowest number of duplicates by far (SALT and Standard Supervised) perform significantly better than those which a much highest number of duplicates. For Super Mario (Figure 5.4.f), most methods gained around the same number of duplicates. The only method that got more duplicates was Standard Supervised, which also performs by far the worst in that domain in terms of task reward. These trends provide evidence that adding more duplicates to the training data decreases performance.

6.1.3 Results - k-nn

Table 6.2 shows the average proportion of unique instances in the training data compared to the total amount of training data using k-nn as the underlying learner. As can readily be seen, the amount of duplicate data accrued is quite similar to that with J48. Just as when using J48, SALT and Standard Supervised garner the least duplicates, and DAgger and SafeDAgger garner the most. The main difference is that instead of being tied for the least duplicates in Super Mario, when using k-nn DAgger has the least duplicates (albeit by a small margin). When comparing these results to Figure 6.1, what we see is the same as before – methods which have less duplicates tend to perform better. The major exception to this is Standard Supervised, which accrues the least duplicates in the Complex Thermometers domain by far, but is outperformed by SALT in terms of task reward. This provides evidence that although less duplicates is highly correlated with better performance,
Figure 6.1: Task reward (y-axis) plotted against amount of training data (x-axis), for various learning methods, using $k$-nn as an underlying learner. Note both $\rho_s$ and $\rho_d$ are stated for SALT and that all methods were ran for the same amount of iterations. Since some methods gain much less training data than others, this results in some lines being much shorter than others.
Figure 6.2: Reward gained in the Simple Thermometers domain with J48 as an underlying learner (vertical axis) as a function of the amount of training data (horizontal axis), when keeping in or removing duplicates from the training data for various learning methods.

6.1.4 Removing Duplicate Data

In order to further examine how duplicated data affects task reward, we compared the Keep Duplicates results (where any duplicated data is not removed from the training data) to the Remove Duplicates results (when all duplicates are removed from the training data). This was done with all 3 domains and with 2 different underlying learners (J48 and k-nn) The results of these comparisons are in Figures 6.2 to 6.7.

J48

When using J48 in the Simple Thermometers domain with Standard Supervised as the learner (Figure 6.2.a), tests where the duplicates have been removed gain a higher task reward than tests which keep the duplicates in the training set. The same is true for PseudoSafeDAgger and SALT (Figures 6.2.c and 6.2.d, respectively). For DAgger (Figure 6.2.b), the results with duplicates are roughly the same as those where the duplicates have been removed.

For the Complex Thermometers domain, PseudoSafeDAgger and Standard Supervised (Figures 6.3.a and 6.3.c) both perform better when the duplicates are removed, SALT (Figure 6.3.d) performs slightly better when duplicates are removed, and DAgger (Figure 6.3.b) performs better when duplicates are kept in the training data.

For the Super Mario domain, SALT, PseudoSafeDAgger, and Standard Supervised (Figures 6.4.d, 6.4.c, and 6.4.a, respectively) all perform better when the duplicates are removed, and DAgger (Figure 6.4.b) performs slightly better when duplicates are removed.

For SALT, PseudoSafeDAgger, and Standard Supervised using J48, removing duplicate instances performs better or slightly better in all 3 domains (Figures 6.2.d, 6.3.d, and 6.4.d for SALT, Figures 6.2.c, 6.3.c, and 6.4.c for PseudoSafeDAgger, and Figures 6.2.a, 6.3.a, and 6.4.a for Standard...
Figure 6.3: Reward gained in the Complex Thermometers domain with J48 as an underlying learner (vertical axis) as a function of the amount of training data (horizontal axis), when keeping in or removing duplicates from the training data for various learning methods.

Figure 6.4: Reward gained in the Super Mario domain with J48 as an underlying learner (vertical axis) as a function of the amount of training data (horizontal axis), when keeping in or removing duplicates from the training data for various learning methods.
Figure 6.5: Reward gained in the Simple Thermometers domain with $k$-nn as an underlying learner (vertical axis) as a function of the amount of training data (horizontal axis), when keeping in or removing duplicates from the training data for various learning methods.

Figure 6.6: Reward gained in the Complex Thermometers domain with $k$-nn as an underlying learner (vertical axis) as a function of the amount of training data (horizontal axis), when keeping in or removing duplicates from the training data for various learning methods.
Figure 6.7: Reward gained in the Super Mario domain with $k$-nn as an underlying learner (vertical axis) as a function of the amount of training data (horizontal axis), when keeping in or removing duplicates from the training data for various learning methods.

Supervised). For DAgger, removing duplicates performs roughly the same for Simple Thermometers (Figure 6.2.b), worse for Complex Thermometers (Figure 6.3.b), and slightly better for Super Mario (Figure 6.4.b).

$k$-nn

For using $k$-nn in the Simple Thermometers domain, all four learning methods perform better when duplicates are removed from the training data (Figures 6.5.a, 6.5.b, 6.5.c, and 6.5.d).

For using $k$-nn in the Complex Thermometers domain, DAgger performs better when duplicates are left in the training data (Figure 6.6.b) and Standard Supervised performs slightly better (Figure 6.6.a), but the other two learning methods perform best when duplicates are removed (Figures 6.6.c and 6.6.d).

For using $k$-nn in the Super Mario domain, the results are a bit more diverse. One method performs better when duplicates are removed from the training data (PseudoSafeDAgger, Figure 6.7.c), and one method performs equally (Standard Supervised, Figure 6.7.a). The remaining two methods, DAgger and SALT, both perform worse when duplicates are removed from the training data (Figures 6.7.b and 6.7.d, respectively).

For SafeDAgger, removing duplicates from the training data improves results in all 3 domains (Figures 6.5.b, 6.6.b, and 6.7.b). For Standard Supervised, removing duplicates from the training data improves results in Simple Thermometers (Figure 6.5.a), slightly decreases results in Complex Thermometers (Figure 6.6.a), and does not make a significant difference in the Super Mario domain (Figure 6.7.a). For SALT, removing duplicates increases results in the Thermometers domains (Figures 6.5.d and 6.6.d), but decreases results in the Super Mario domain (Figure 6.7.d). Finally, for DAgger, removing duplicates from the training data increases results for the Simple Thermometers domain (Figure 6.5.b), but decreases it for the other domains (Figures 6.6.b and 6.7.b).
6.1.5 Summary

We can conclude a few things from these results. First, removing duplicates does increase task reward in most cases, and only decreases it in 6 out of 24 cases. This, together with the strong inverse correlation between proportion of duplicates in the training data and task reward, indicates that duplicates in the training set are harmful for our domains. However, it is also interesting to note that, overall, removing duplicates seems to increase results for SafeDAgger and SALT, but not for DAgger. This, coupled with the fact that both SafeDAgger and DAgger garner a very high number of duplicates, indicates that the large amount of duplicates in the training data is a symptom of an underlying difference between DAgger and SALT, not the cause of the difference in performance. Additionally, since the trends for J48 and k-nn are very similar, we have further evidence that this is a cross-learner trend that is not due to our choice of underlying learner. More examination of SALT and DAgger is needed to pinpoint this underlying difference, but one possibility is that SALT’s method of choosing who is in control of the task leads to the demonstrator more effectively getting the learner out of states it is unable to get out of itself, which would also reduce the amount of duplicate data (since the learner would be unable to get itself out of that state, with DAgger it would get many duplicates of that same state in the training data).

6.2 Comparison of Training Set and Test Set Distributions

As explained in Section 5.1, one of the key problems in LfD is that the set of states encountered during learning, \( D_l \), tends to differ significantly from the set of states encountered during testing (\( D_t \)). In order to further understand the differences between SALT and DAgger, we examined the distribution of states encountered by each during both learning and testing.

6.2.1 Experimental Setup

We examined the state distributions for 3 methods (SALT, DAgger, and Standard Supervised) across two domains (Simple Thermometers and Complex Thermometers, as defined in Chapter 3), to see if there were any cross-domain trends. All experiments were done with \( N = 1 \) traces and \( C = 25 \) iterations, with J48 as the underlying learner. The results reported represent the following procedure:

Step 1: Take the training set gathered after each iteration of SALT or DAgger (or the equivalent amount of Standard Supervised data) and use it to create a distribution of states with a Product Kernel Density Emulator via the JSAT library\(^86\).

Step 2: Take each state that is encountered by the learner trained on that training set and calculate the probability that it is in the distribution.

Step 3: Average these values over all states encountered during testing the learner to get the average expected probability that a state encountered by the learner while testing is in the training distribution.

Complex-Thermometers represents the board and the constraints imposed on each row and column of the board as a vector of 245 features. The action space \( Y \) consists of 75 actions, 3 actions for each tile on the grid (set to “full”, set to “empty”, and “clear”). For the second, Simple-Thermometers, states are a vector of 14 features which represent a single row or column of the board, and the actions are to fill a tile, empty a tile, move to the next row/column, or set all tiles in the row/column to undetermined and move to the next one.

For both Thermometers domains, experiments were performed using \( N = 1 \) traces and \( C = 25 \) iterations. Both Thermometers domains have a limit of 100 moves per board, and therefore a maximum length of 100 training states per trace. The reward function for both Thermometers domains is simply the overall percentage of constraints satisfied for the current board, where there are two kinds of constraints: The number of filled pieces in a row or column matches the number for that row or column, and that each thermometer has a legal configuration (meaning that it is filled starting from the circularly shaped bulb, and all filled pieces are adjacent).
6.2.2 Results

Figure 6.8a shows the average expected probability plotted versus training data for SALT, DAgger, and Standard Supervised in the Simple Thermometers domain. Since we plot this over time (in the form of iterations), lines growing higher from left to right means that the probability an encountered state was sampled from the training distribution grows after each iteration of the method. What we can see is that the Standard Supervised line grows a little (since the learner gets a little better as more data is added), but not as fast as SALT or DAgger. SALT increases the probability at roughly double the rate of Standard Supervised, but DAgger performs significantly better than either of them.

From these results, we can see that DAgger performs the best at making $D_l$ and $D_t$ closer together. This is an interesting result, as SALT performs much better in terms of task reward in this domain. On the other hand, if we look at Figure 6.8b, the same experiment but for the Complex Thermometers domain, we see basically the exact opposite trend. Standard Supervised grows the fastest, followed by SALT and then DAgger, respectively. This seems to indicate that Standard Supervised is better than either active learning method at making $D_l$ and $D_t$ be equivalent in this domain, which is counterintuitive, as that is precisely what active-learning from demonstration algorithms try to accomplish. Taking these two graphs together, we can reasonably conclude that making $D_l$ and $D_t$ match is not on its own a sufficient condition for performing well in these domains, and that there must be other factors helping SALT’s performance. This is an interesting result, as it seems to contradict theoretical results in the literature, which states that when $D_l$ and $D_t$ differ, error compounds much faster than when they are the same. To examine why this may be the case, let us look at the Complex Thermometers domain. A two dimensional projection of traces on 4 boards in this domain is shown in Figure 6.9. Each colored circle shows one training instance, and the projection was generated using a force-based method to find the 2d coordinates that best preserve the distances between the training instances in the original high-dimensional space (in a manner very similar to t-SNE). The traces start relatively close together (the bottom right of the image), but spread out as they proceed. This shows that, in this domain, data from different a given starting board is not very useful in helping to perform well in other boards, specially as the board starts filling up. This means that in domains, like Complex Thermometers, where different starting configurations might lead the learning agent to very different areas of the state space, attempting to make $D_l$ and $D_t$ converge might not be possible, or might not be the best strategy, when focusing on low-training data scenarios.

Please note that running these experiments on Super Mario was infeasible due to the speed at which the Super Mario simulator executes. Also note that removing duplicates from the training data also did not change the observed trends.
Figure 6.9: An illustration of four boards in the Complex Thermometers state space. Each colored set of circles represents one trace, with each circle being one training instance. The traces start in the bottom right and expand roughly towards the top left as time proceeds.

6.3 Conclusions

In this chapter, we examined and empirically tested two aspects of the learners trained by each algorithm. First, we examined the amount of duplicates accrued by each method and the effects of removing these duplicates, finding that SALT creates less duplicates in the training data. We also found that, in our domains, having less duplicates increases performance overall, although this appears to be a symptom of an underlying difference rather than directly responsible for the difference in performance. We also compared the algorithms in terms of making $D_t$ and $D_l$ match, showing that this is not on its own a sufficient condition for performing well in a domain when focusing on scenarios where the amount of training data is limited. With this knowledge, we now move onto the main focus of SALT: improving its feasibility for human demonstrators.
Chapter 7: Working Towards Human Demonstrators

One of the current problems in the field of LfD is that existing LfD algorithms often require too much training data to be practical for human demonstrators, as they tend to require the demonstrator to provide a large number of training instances in order to have enough training data to effectively train a learner (As discussed in Chapter 1 and Section 2.1.1). Active LfD approaches tend to exacerbate the problem further - not only does the demonstrator need to provide demonstrations, but they also need to be able to respond to queries that the learner makes. One example of this is DAgger, which can learn behavior very accurately, but requires the demonstrator to relabel all of the training data gained while the learner was in control. Little work has been done on trying to reduce how much training data in the context of LfD. Some exceptions include SafeDAgger, which reduces how much training data is required from the demonstrator, but does not focus on being suitable for human demonstrators. This is because the amount of training data is not the only factor to have in mind when using a human demonstrator (for example, requesting data from the demonstrator many times for very short intervals can be difficult for a human demonstrator).

In this chapter, we discuss changes made to SALT in order to make it more feasible for human demonstrators. As many LfD methods require a very large amount of training data and are not created with human demonstrators in mind, specialized algorithms which can learn effectively from single human demonstrators are needed for domains where humans demonstrators are needed or the amount of available training data is very low. Modifying SALT to be more feasible for human demonstrators helps to address this issue. In order to do this, we examine one major modification (introducing a demonstrator budget of how often the learner can query the demonstrator for more data) and one new evaluation tool (cognitive burden). The metrics of cognitive burden are important because they attempt to specifically capture whether or not a method is feasible for being trained by a human demonstrator, instead of just how much training data is needed or how well the learner can perform the task. Additionally, adding a demonstrator query budget puts a cap on how much training data can be requested from the demonstrator (after all, humans can only provide so much training data) and could also be leveraged by learning algorithms to help more effectively query the demonstrator.

7.1 Demonstrator Budget

In order to reduce the amount of training data that a human demonstrator would have to provide, we introduce the idea of a demonstrator query budget. The operation of active learning from demonstration algorithms (such as SALT or DAgger) can be divided into two separate steps: in a first step the demonstrator just performs the task at hand to generate an initial set of training data (this would be the first iteration of SALT or DAgger). During the second step (the remaining iterations of SALT or DAgger), the learning agent is the one performing the task, and can query the demonstrator for additional training data. We define the demonstrator query budget, $B$, as the limit of the number of training instances that the learning agent can request from the demonstrator during this second step of the algorithms. Once this budget is reached, the learner cannot obtain any more data from the demonstrator. This reduces the amount of training data as well as rewards methods that query the demonstrator most effectively.

We experimented with two different scenarios: (a) global demonstrator query budget, where the budget $B$ is given to the learner at once, and it is up to the learner to decide how this budget should be split over iterations; and (b) per-iteration demonstrator query budget, where the learning agent is given a fixed budget $B/(N-1)$ (recall that $N$ is the number of iterations for which SALT runs) in each of the iterations of SALT (except for the first).
7.1.1 Budget-Aware Strategies

To this test $SALT$ when using a global demonstrator query budget, we examined seven possible variants of $\rho_d$, which determines when the learner has moved out of $D_t$, and thus the demonstrator is asked to take over (several of these strategies have been previously introduced, but are repeated here for clarity):

- $\rho_d^{\text{SS}}$ (Simple Stochastic – not Budget-Aware): signal the learner has exited $D_t$ with probability $P = \frac{|ER_t - R_l|}{\max}$, where $ER_t$ is the expected demonstrator reward at the current time $t$ (estimated from the data collected during the first iteration of $SALT$). $R_l$ is the learner’s reward at the current time, and $\max$ is the maximum possible reward for the domain.

- $\rho_d^{\text{RD}}$ (Reward Doesn’t Increase – not Budget-Aware): signal the learner has exited $D_t$ if the learner’s reward has not increased compared to the previous time step (i.e., if $R_t \leq R_{t-1}$).

- $\rho_d^{\text{MD}}$ (Minimum Distance – not Budget-Aware): using Euclidean distance, the most similar world state $s^*$ from the set of states the demonstrator visited during the first iteration of $SALT$ to the current world state $s_j$ is found. If the distance between $s_j$ and $s^*$ is higher than a threshold value $\alpha$, signal the learner has exited $D_t$. If we let $\epsilon(s)$ represent the minimum distance from state $s$ in $D_t$ to any other state in $D_t$ and let $E$ represent the set of all $\epsilon(s) \mid s \in D_t$, $\alpha$ was set to the average value in $E$ in our experiments.

- $\rho_d^{\text{R pseudo}}$ (PseudoDAgger – Budget Aware): signal the learner has exited $D_t$ with probability $\frac{\text{RemainingBudget}}{B}$, where $\text{RemainingBudget}$ is the query budget left.

- $\rho_d^{\text{WSS}}, \rho_d^{\text{WLE}},$ and $\rho_d^{\text{WMD}}$ (all Budget-Aware): These strategies are the same as $\rho_d^{\text{SS}}, \rho_d^{\text{RD}},$ and $\rho_d^{\text{MD}}$, respectively, but with their chance to signal multiplied by $\frac{\text{RemainingBudget}}{B}$.

We experimented on one possible variant of $\rho_d$ (one of which has been previously introduced), which determines for how long to give control to the demonstrator:

- $\rho_d^{\text{RG}}$ (Reward Goal – not Budget-Aware): Signals to give control back to the learner when the learner’s accumulated reward reaches the demonstrator’s expected accumulated reward for the time step at which the demonstrator took control.

7.2 Experimental Evaluation

We used the Super Mario domain and both Thermometers domains (as defined earlier in Chapter 3) in order to evaluate this work. World states in Super Mario are represented by a collection of 1083 features representing the surroundings of Mario (whether or not there is a wall, where the enemies are, etc.). The action space $Y$ of Super Mario consists of 5 boolean features corresponding to what buttons the agent is pressing in the game (left, right, down, fire/speed, and jump). For Thermometers, both versions were used. The first, Complex-Thermometers, represents the board and the constraints imposed on each row and column of the board as a vector of 245 features. The action space $Y$ consists of 75 actions, 3 actions for each tile on the grid (set to “full”, set to “empty”, and “clear”). For the second, Simple-Thermometers, states are a vector of 14 features which represent a single row or column of the board, and the actions are to fill a tile, empty a tile, move to the next row/column, or set all tiles in the row/column to undetermined and move to the next one.

Experiments for Super Mario were performed using $N = 1$ traces and $C = 25$ iterations. Each trace has a time limit of 30 seconds, and data is every tick (where the game runs at 20 ticks per second), leading to a maximum trace length of 450 training instances. Strategy $\rho_d^{\text{RG}}$ was used to give control back to the learner. The reward function used is simply Mario’s X-coordinate in pixels plus a bonus of 500 if he is Fire Mario, and only 250 if he is Large Mario (Mario starts off as Fire Mario, and drops to Large Mario and then Small Mario when he takes one or two hits from enemies, respectively).

For the Complex-Thermometers domain, experiments were performed using $N = 1$ traces and $C = 50$ iterations, also using strategy $\rho_d^{\text{RG}}$. For the Simple-Thermometers domain, experiments were...
performed using $N = 1$ traces and $C = 25$ iterations, and used strategy $\rho_d^{EOT}$ (note that we used a different $\rho_d$ here due to it obtaining much better results in this domain than $\rho_d^{RG}$ based on empirical observation). Both Thermometers domains have a limit of 100 moves per board, and therefore a maximum length of 100 training states per trace. The reward function for both Thermometers domains is simply the overall percentage of constraints satisfied for the current board, where there are two kinds of constraints: The number of filled pieces in a row or column matches the number for that row or column, and that each thermometer has a legal configuration (meaning that it is filled starting from the circularly shaped bulb, and all filled pieces are adjacent).

Three factors were used to compare approaches, the first two of which are task reward ($R$) and similarity between the learner and demonstrator (defined as the percentage of times the learner predicted the same move as the demonstrator in a set of validation levels). In order to further analyze the performance of $SALT$ versus other state-of-the-art ActiveLfD methods, we also developed metrics which would capture how much cognitive burden is imposed upon the demonstrator while learning. We then analyze different variants of $SALT$ using different strategies against other state-of-the-art methods, over three different domains and with two underlying learners for the task reward, demonstrator similarity, and cognitive burden.

Specifically, two measures of cognitive burden were employed:

- **PSRA** – The proportion of states for which the demonstrator had to provide actions (measured as the total number of states that the demonstrator had to control or relabel over the total length of the trace – lower values mean fewer actions provided). This comes from the notion that having the demonstrator provide more actions will impose a higher cognitive burden.

- **Context-Switches** – The number of times that the demonstrator had to start from a new state. This is made to capture that having a human demonstrator provide a sequence of actions will require many less context switches than having the demonstrator provide the same number of actions but from a different state each time – and as Rogers and Monsell\(^91\) show across 6 cognitive tasks, introducing context switching increases reaction time and error rate.

### 7.2.1 Results

We report the performance of $SALT$ and four baselines:

#### Baselines

- **Standard Supervised**: we evaluate the performance of the base supervised learner as the training set increases in size proportionately to the other methods. This data is generated by having the demonstrator complete the task until we have the same amount of training data as for the other methods, then training the learner.

- **DAGger**: we evaluate the performance of $DAGger^{95}$ in our domains for the sake of comparison.

- **PseudoSafeDAGger\(^{120}\)**: in order to emulate the behavior of $SafeDAGger$, we use strategy $\rho_s^{DCA}$, which signals that the learner has moved out of $D_i$ if the learner’s action differs from the demonstrator’s action more than a threshold value $\beta$ according to a modified edit distance, but only let the demonstrator control for a single frame (as $SafeDAGger$ would). Note that our emulation of $SafeDAGger$ needs to call the demonstrator at each time step in order to compare the actions, while $SafeDAGger$ would attempt to learn a function to compare them without needing to query the demonstrator.

- **PseudoDAGger**: $PseudoDAGger$ is a $DAGger$ inspired algorithm which queries the demonstrator with probability of $\frac{RemainingBudget}{TotalBudget}$. Note that this is the same as using $\rho_s^{Pseudo}$ in $SALT$, but having the demonstrator control for a single frame instead of using $\rho_d$. 

---

**Chapter 7: Towards Human Demonstrators** 7.2 Experimental Evaluation
Figure 7.1: Reward gained in Super Mario and each Thermometers domain as a function of the amount of training data (horizontal axis) for various strategies and baselines. The vertical axis represents the amount of reward gained (in the case of Super Mario) or the % of constraints satisfied (in the case of both Thermometers domains). The top row are results gained using a global budget, and the bottom row are results gained splitting the budget up evenly over iterations.

Task Reward

Figure 7.1 shows the accumulated reward obtained by all of the methods tested in our experiments. It can be readily seen that, in all three domains, methods using an iterative budget tend to train faster than those which use a global budget. Additionally, in two of the domains (Simple Thermometers and Super Mario), methods using an iterative budget also tend to receive a higher reward. For example, in the Simple Thermometers domain, methods that use an iterative budget instead of a global budget generally reach a higher reward and train faster than those that do not (the best method using an iterative budget hits 50% of constraints satisfied compared to about 47.5%, and the fastest training methods reach 45% of constraints satisfied at about 600 training instances instead of around 800). It is also interesting to note that using the budget-aware versions of SALT strategies does not seem to improve results, with most strategies performing the same or better than their budget-aware variants (perhaps the most glaring example of this being \( \rho_S^W \) compared to \( \rho_S^{W,S} \) when using an iterative budget in the Super Mario domain).

We can also observe that SALT outperforms the baselines in every domain – in both Thermometers domains, DAGGER and SafeDAGGER perform the most poorly of all the examined methods, only reaching a very low reward for the amounts of training data examined. In the Super Mario domain, these two methods offer moderate performance, performing approximately the same as many of the other examined methods, but significantly below the best SALT variants. As for PseudoDAGGER, it performs worse than most of the SALT methods in both Thermometers domains, and is not a contender for the best method in either domain. In the Super Mario domain, PseudoDAGGER with a global budget is actually the second best method, but is much worse than the best method, SALT with \( \rho_S^W \).

Finally, many active learning methods perform better than just using supervised learning overall.
In the Simple Thermometers domain, there are many active learning methods that both train faster and gain higher reward than Standard Supervised. In the Complex Thermometers domain there are two active learning methods which train much faster than Standard Supervised and are comparable in terms of reward ($\rho_{s}^{W,SS}$ and $\rho_{s}^{SS}$), and in the Super Mario domain there are a couple methods that reach a much higher reward ($\rho_{s}^{C}$ and PseudoDAgger).

**Demonstrator Similarity**

Figure 7.2 shows the demonstrator similarity obtained by all of the methods tested in our experiments. The first thing that we can observe is that most of the methods using an iteration-based budget train faster to receive the same or higher reward than those using a global budget (the most notable example of this is SALT with $\rho_{s}^{SS}$ in the Super Mario domain although it can also easily be seen via the larger gap between Standard Supervised and the SALT variants when using an iterative budget in the Simple Thermometers domain). Finally, just as in task reward, having budget-aware versions of SALT strategies does not seem to boost results overall, with budget-aware strategies mostly performing the same or slightly worse than their non-budget-aware versions.

We can also see that SALT performs very well compared to the baselines in terms of demonstrator similarity. In both Thermometers domains, SALT variants train faster and receive an overall equal or higher similarity than any of the four baselines, with PseudoDAgger in the Simple Thermometers domain being the only competitive baseline (it trains almost as fast when using an iterative budget, and receives the highest reward). SALT also outperforms three of the baselines in the Super Mario domain, with the exception of Standard Supervised.

Finally, we observe that active learning performs better than just using supervised learning in two of the three domains. In both Thermometers domains, there are multiple active learning methods that both train faster and receive a higher similarity score than Standard Supervised. In the Super Mario domain, Standard Supervised actually receives the highest similarity, and does so...
Table 7.1: Average number of demonstrator calls (“Context-Switches”), and the proportion of states in which the demonstrator either has to control or relabel for various methods and domains (“Proportion of States Requiring Actions”, or “PSRA”), averaged over both underlying learners.

<table>
<thead>
<tr>
<th></th>
<th>S-Therms</th>
<th>C-Therms</th>
<th>Mario</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Context-Switches</td>
<td>PSRA</td>
<td>Context-Switches</td>
</tr>
<tr>
<td>DAgger</td>
<td>29.23</td>
<td>1.000</td>
<td>28.27</td>
</tr>
<tr>
<td>SafeDAgger</td>
<td>813.89</td>
<td>0.350</td>
<td>963.94</td>
</tr>
<tr>
<td>PseudoDAgger</td>
<td>390.00</td>
<td>0.405</td>
<td>296.9</td>
</tr>
<tr>
<td>SALT,  (\rho_{SS})</td>
<td>375.17</td>
<td>0.437</td>
<td>292.03</td>
</tr>
<tr>
<td>SALT,  (\rho_{W,SS})</td>
<td>24.20</td>
<td>0.477</td>
<td>52.26</td>
</tr>
<tr>
<td>SALT,  (\rho_{S}^{W} )</td>
<td>25.00</td>
<td>0.690</td>
<td>305.57</td>
</tr>
<tr>
<td>SALT,  (\rho_{W}^{S} )</td>
<td>25.00</td>
<td>0.690</td>
<td>282.48</td>
</tr>
<tr>
<td>SALT,  (\rho_{MD}^{W})</td>
<td>24.95</td>
<td>0.735</td>
<td>306.82</td>
</tr>
<tr>
<td>SALT,  (\rho_{W,MD})</td>
<td>24.95</td>
<td>0.735</td>
<td>292.03</td>
</tr>
</tbody>
</table>

very quickly, but then drastically drops as more training data is added. We theorize this is due to the agent learning to press “right” and “speed” almost every frame at the start, which yields a higher similarity because the demonstrator presses these buttons a large proportion of the time. As the agent continues to learn it performs these actions less so its similarity to the demonstrator falls, and then eventually rises back up as it begins to more closely emulate the demonstrator. As a result, many of the active learning methods have closer similarity to the demonstrator at about the halfway mark, but Standard Supervised performs the best in the beginning.

Cognitive Burden

From Table 7.1, we see that a SALT variant has the least number of context switches in all 3 domains (SALT with \(\rho_{SS}\) for Simple Thermometers, DAgger for Complex Thermometers, and \(\rho_{W,SS}\) for Super Mario). We also see that SafeDAgger has by far the highest number of context switches in all 3 domains. In terms of the demonstrator having to provide actions for states, SafeDAgger performs the best in Simple Thermometers and Super Mario and SALT with \(\rho_{W,SS}\) performs the best in Complex Thermometers. The methods which performed the worst in this metric was DAgger, as it requires the demonstrator to relabel all states for which it was not in control.

In summary, both SafeDAgger and DAgger perform fairly well in one measure, but very poorly in the other measure, but most SALT policies perform decently in both metrics, with SALT with \(\rho_{SS}\) and SALT with \(\rho_{W,SS}\) performing the overall best across domains. Therefore, it can accurately be said that SALT would require less cognitive burden on the demonstrator for these domains according to these metrics. We present a user study confirming the notion of SALT’s reduced cognitive burden over DAgger for human demonstrators in Chapter 8.

7.3 Conclusions

This chapter discussed the process of making SALT more feasible for human demonstrators. Specifically, one major modification was discussed: imposing a demonstrator query budget. The idea of budget-aware SALT policies was also presented and empirically evaluated, showing that adding a by-iteration demonstrator budget increases the effectiveness of SALT over having a global budget (and by extension over having no budget at all, since no budget is just an infinite global budget). Additionally, two new metrics designed to capture the cognitive burden of the demonstrator were presented and used to evaluate SALT against other algorithms, with promising results indicating SALT incurs less cognitive burden overall. Now that we have modified SALT to potentially be more feasible for humans, let us move on to comparing SALT against DAgger on human demonstrators to see just how feasible SALT really is.
Chapter 8: Learning from Humans using SALT

This chapter covers the results of comparing SALT to another state-of-the-art algorithm, DAgger\textsuperscript{95} (chosen for a fair comparison, as it is another iterative ActiveLfD algorithm with effective results in the literature\textsuperscript{95}), when learning from human demonstrators in the Simple Thermometers domain. This is important as it demonstrates SALT’s potential as an effective LfD algorithm for human demonstrators (a problem discussed in detail in Chapter 1 and Section 2.1.1). Additionally, as most LfD algorithms still assume a very large amount of data, this is also invaluable for situations where data is limited (such as learning from a single human demonstrator). In order to facilitate this comparison, we perform a user study where participants train two learners and comment upon various aspects of the learning process intended to capture how much effort each algorithm was to train as well as which algorithm human demonstrators prefer training. This chapter is used to confirm the results from Chapter 7, which shows on synthetic demonstrators that SALT should require less cognitive burden on human demonstrators according to two cognitive burden metrics. We compare SALT’s performance to DAgger’s on 6 factors, showing its relatively high feasibility for learning from human demonstrators.

8.1 Study Setup

The study has two phases (with a researcher overseeing both): In the first phase, the human demonstrator practices the puzzles without training a learner, to get used to the game and interface. Once both the demonstrator and the overseer feel confident in the demonstrator’s puzzle solving ability, they move onto phase two. In phase two, the human demonstrator trains a learner using SALT and a learner using DAgger, one after the other. The algorithms are anonymized, so they are not aware of which algorithm is which. A total of 21 participants were gathered: 10 trained SALT then DAgger, 10 trained DAgger then SALT, and one trained SALT then DAgger then SALT again (this user felt that they had made many more errors when training SALT than when training DAgger, and wished to train it again making less errors). We removed that user’s results from the study since they were an outlier (being the only one to have trained a method twice). The minimum age was 18, and the maximum age was 62, with the median age being 20. 75% of the 20 included users identified as male, and 25% identified as female. The amount of programming experience that users had is shown in Figure 8.1: on a scale of 0-5, the average was 3.2 and the median was 3.5. Training for each algorithm continued until the user decided they were done - no minimum number of boards was enforced, but there was a maximum of 25 boards per algorithm. After training the first algorithm, the participant is asked to fill out a survey which asks for the following information:

- Their tester ID number. This was used purely to determine which algorithm they trained first.
- Programming Experience: The amount of programming experience they have, where 0 is none at all and 5 means they consider themselves an expert.
- Mental Effort: Participants were asked to rate the mental effort required to train each learner. Ratings were gathered on a scale of 1 to 5, where 1 was very little mental effort required and 5 was a lot of mental effort required.
- Perceived Learning: Participants were asked to rate how well they believed each algorithm learned from the training process and the final task reward on validation boards. Ratings were gathered on a scale of 1 to 5, where 1 means the algorithm learned to perform the task very poorly and 5 means the algorithm learned to perform the task very well.
- The Reason They Ended Training: Three default values were provided (“I got bored”, “I felt like it had learned the task”, and “I felt like it wasn’t learning”), but users could also enter their own response.
• Enjoyability: Participants were asked to rate how enjoyable the algorithm was to train comparing to programming their own puzzle solver by hand. Ratings were gathered on a scale of 1 to 5, where 1 was far less enjoyable and 5 was far more enjoyable.

• Any other comments they may have.

After training the second algorithm, the participant is asked to fill out a survey which contains the same information for the second algorithm, and also re-evaluate the first algorithm now that they have experienced both. The results reported use their final scores for each algorithm (the ones taken from the second survey).

In addition to this, we also captured two pieces of data from the training process:

• Number of Boards: Participants were able to train for as many (up to 25) or as few boards as they wanted, with each board being one iteration of the algorithm. We record the quantity of boards on which they trained each algorithm, to see which algorithm they were willing to train longer.

• Actual Learning: The amount of training data and the average task reward gained on 10 validation levels with the training data that was provided by each participant for each learner.

It is important to note that there are theoretically two ways of implementing DAgger for human demonstrators. The first is to have the demonstrator provide actions for when it is in control, and then have them relabel any states for which they were not in control after the iteration is over. The second is to have the demonstrator provide a move for every board state as it happens, but not always make the move the demonstrator is providing. The former, however, would require the demonstrator going back and relabeling potentially many states after they decide that they are done training the algorithm (which is not realistic to expect from human demonstrators). Because of this, combined with the intuition that it is easier on humans to see the entire solving of the board in context (based on the work of Rogers and Monsell\textsuperscript{91}, who show that introducing context

\textbf{Figure 8.1:} A bar plot showing the reported programming experience of users participating in the study. A rating of 0 means the user had no programming knowledge at all, and a rating of 5 means they believed they were an expert programmer.
Table 8.1: Aggregate results from the post-study questionnaire. This table shows how many people stopped training each algorithm, categorized by the general reason they stopped.

<table>
<thead>
<tr>
<th>Reason</th>
<th>SALT</th>
<th>DAgger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Felt the algorithm had learned the task</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Felt the algorithm wasn’t learning</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>Got bored training the algorithm</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Got frustrated training the algorithm</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Ran out of time for the study</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Hit the 25 board cap while training the algorithm</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Other reasons</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 8.2: Task reward gained in the Simple Thermometers domain (vertical axis) as a function of the amount of training data (horizontal axis) for SALT and DAgger. As different demonstrators played for different numbers of iterations, a 2-value simple moving average was taken to smooth the curves.

switching increases reaction time and error rate), the second implementation was deemed a more fair comparison, and is the one used in this study.

8.2 Study Results

This section details the results of the study, examining the six pieces of data that help capture feasibility for human demonstrators:

- Mental Effort
- Enjoyability
- Perceived Learning
- The Reason They Ended Training
- Number of Boards
- Actual Learning
Table 8.2: Aggregate results from the post-study questionnaire. This table shows the average user rating for each algorithm for each category, along with the results of a 2-tailed paired t-test statistical significance test.

<table>
<thead>
<tr>
<th>Category</th>
<th>SALT</th>
<th>DAgger</th>
<th>Statistical Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mental Effort</td>
<td>2.75</td>
<td>4.65</td>
<td>( p = 8.53 \times 10^{-6} )</td>
</tr>
<tr>
<td>Enjoyability</td>
<td>3.75</td>
<td>1.75</td>
<td>( p = 1.40 \times 10^{-6} )</td>
</tr>
<tr>
<td>Perceived Learning</td>
<td>3.55</td>
<td>1.40</td>
<td>( p = 5.81 \times 10^{-7} )</td>
</tr>
</tbody>
</table>
| Number of Boards       | 12.70| 7.65   | \( p = 2.85 \times 10^{-4} \)

8.2.1 Mental Effort

First, let us examine the metric of Mental Effort. As can be seen from Table 8.2, users rated SALT a 2.75 on the 1 to 5 scale and DAgger a 4.65. This means that users placed SALT almost directly in the middle for this metric, but rated DAgger as requiring a very high amount of mental effort, providing strong evidence that SALT imposes less mental burden on human demonstrators than DAgger. From the comments made during the study, our hypothesis for this is that users found it jarring to make a move and then be in a different state than the one that they expected (this happens with DAgger since the user is not always in control, but never with SALT).

8.2.2 Enjoyability

Next, let us examine the second metric recorded directly from the survey: how enjoyable the algorithm was to train. As can be seen from Table 8.2, users rated SALT much higher in this metric than DAgger (a 3.75 compared to a 1.75 out of 5). This means that they rated SALT as somewhat enjoyable and DAgger as very unenjoyable, providing further evidence of SALT’s feasibility for human demonstrators. We believe this is also due to the users ending up in a state different than they expected. Because of this, many of them commented that they felt like DAgger wasn’t listening, even though they knew it was.

8.2.3 Perceived Learning

The third metric recorded directly from the survey was that of perceived learning. Table 8.2 shows us that users rated SALT as a 3.55 and DAgger as a 1.40, again on a scale of 1-5. Therefore, although users didn’t find SALT to learn extremely well, they perceived DAgger to learn very poorly. We hypothesize that this is because while sharing control with the learner during training, DAgger made more moves that seemed ‘odd’ or illogical’ to the user than SALT. Since it doesn’t matter how effortless or enjoyable the users find an algorithm unless it can also learn to perform the necessary task, SALT being perceived to perform better provides evidence for its feasibility for learning from human demonstrators. Furthermore, given the very small amount of training data in this domain, SALT is not just perceived to learn better, but actually does learn better (this is discussed in detail in Section 8.2.6).

8.2.4 Reason for Ending Training

The fourth and final metric recorded directly from the survey was the reason the user stopped training each algorithm. Table 8.1 shows the aggregated results for this question. As can be readily seen, users stopped training SALT for a variety of reasons. Three people stopped because they believed the algorithm had learned the task and 3 people stopped because they felt that it wasn’t learning. The biggest reason users stopped training SALT was because they got bored, and 5 people only stopped training SALT because they either hit the cap of 25 training boards or ran out of time and had to leave. For DAgger, however, almost 75% of users stopped training because they believed that the algorithm wasn’t learning the task, and 0 stopped because they believed the algorithm successfully performed the task. It is also interesting to note that 4 users explicitly stated they
stopped training DAGGER because they got too frustrated with it, where none did so for SALT. This reasoning provides more evidence that users experienced less mental burden for and perceived better learning with SALT than with DAGGER.

8.2.5 Boards Trained

This metric was not directly placed on the survey, but rather gathered from the users’ training data. Table 8.2 also shows us that users trained SALT for around 5 boards longer on average than DAGGER (12.70 boards compared to 7.65 boards). This provides implicit evidence that SALT was more feasible for the users, as they were willing to train it for much longer than they were with DAGGER.

8.2.6 Actual Learning

Finally, let us examine how well each algorithm actually learned from the users. Figure 8.2 shows the average task reward (with a 2-value simple moving average) for SALT and DAGGER when learning off of the human demonstrators. It can readily be seen that SALT initially trains much faster than DAGGER, and then continues to (slowly) grow overall. DAGGER, however, stays pretty flat overall and even seems to lose performance as training progresses. In our previous work\(^77\) we have also found this to be the case with synthetic demonstrators, where SALT outperforms DAGGER in this domain for significantly larger amounts of training data, and DAGGER seems to lose performance. We believe this is because the current world state representation of the domain makes this a very hard problem for LfD to solve, which makes DAGGER struggle. We also verified that in a simpler setting of the domain (using 3x3 boards instead of 5x5 boards) that DAGGER’s performance eventually starts increasing after collecting enough training data from synthetic demonstrators\(^76\).

8.3 Conclusions

The major result is that SALT performs more desirably than DAGGER in six metrics, including amount of mental effort required to train and learning performance, providing strong evidence that it is more feasible for human demonstrators than DAGGER. The insights uncovered through this study on why SALT appears to impose less mental effort on human demonstrators than DAGGER can help improve SALT and other human-centric algorithms in the future (for example, ensuring that when a demonstrator provides an action, they do not end up in a state other than what they expect due to the learning algorithm). We also found that human demonstrators are not perfect (as is to be expected). Not only do human demonstrators make mistakes, but they also makes moves that are logical but inconsistent (for example, filling an entire row from left to right one time, and then right to left another time). Therefore, learners on human demonstrators could be improved not only by reducing noise caused by errors, but also being able to take these inconsistencies and use them to learn a more general or higher level action (such as filling an entire row regardless of order). Areas of future work in this area include running more user studies to compare SALT against other state-of-the-art baselines or in other domains, to further explore its effectiveness for human demonstrators and gain further insights about the difference in SALT’s performance versus other state-of-the-art algorithms.
Chapter 9: Conclusions

Being able to create AI by simply demonstrating the desired behaviors is an important application area, but Learning from Demonstration is a growing field of research that has widespread applications both inside and out of games. Much work has been done in recent years inLfD, spanning from video games to robotics. The main issue is that most of these methods assume a very large amount of training data, and little work has been done in reducing the amount of training data required or otherwise making LfD methods that can learn from human demonstrators. In this document we described a domain-independent Active LfD algorithm, making progress towards using human demonstrators for LfD.

In this document, we explored the open problems (the need for specialized LfD algorithms, learning from human demonstrators, the sequential nature of data, capturing the internal state of the demonstrator, and the need for specialized evaluation metrics), in LfD and the State-of-the-Art work in addressing each of them. This document also gave some background on the field of LfD as a whole before discussing the domains used to empirically evaluate our work. We then described our initial investigation into standard supervised techniques, specifically data representations and techniques for dealing with sequentiality and feature selection. Finally, we then described our main contribution to the field, the specialized-for-LfD algorithm $SALT$, and examined some differences between it and $DAgger$. Finally, we examined a modification to $SALT$ intended to make it more feasible for learning from human demonstrators and then detailed a user study where we tested $SALT$ on actual human demonstrators.

The main contribution of my work is an Active LfD algorithm which has been shown to learn better from human demonstrators than a state-of-the-art method. The $SALT$ algorithm, discussed in detail in Chapters 5 and 7, has been shown to not only learn off of humans better than a state-of-the-art method in a user study, but to be more enjoyable and less mentally burdensome as well.

Having LfD algorithms which can effectively learn from human demonstrators would be a major boon for the field. Being able to learn well from humans in currently used LfD domains would allow for the expansion of LfD into new and important fields, such as having robots perform surgery. In addition, it would aid game designers by allowing them to simply demonstrate the behavior they want a character or enemy to have instead of needing to code it manually. It would even make LfD agents more feasible for everyday use, aiding in areas such as Programming by Demonstration and improving the average person’s computing experience. The results garnered from the user study provide strong evidence that $SALT$ is indeed a step in this direction.

9.1 Contributions

There are 5 major contributions to Learning from Demonstration from our work:

- Algorithms which can learn from small amounts of training data
- Algorithms which are (relatively) feasible for human demonstrators
- The idea of a demonstrator budget for LfD domains
- Cognitive burden metrics
- A study of LfD via similarity-based methods

I will now describe each of these contributions in detail. The first of my two major contributions is deriving an algorithm which can learn from relatively small amounts of training data. The $SALT$ algorithm, described throughout the document but detailed in Chapter 5, does precisely this. As previously discussed, many LfD algorithms assume access to a huge amount of training data, but
this is not always feasible. In some domains you simply cannot gather a sufficiently large amount of training data (such as teaching robots to complete a task by physically demonstrating that task\(^3\)). Furthermore, human demonstrators can only provide so much training data, which needs to be accounted for when deriving algorithms which are feasible for human demonstrators. The SALT algorithm is an Active LfD algorithm which has been shown to learn tasks more effectively and with less training data than other state-of-the-art methods in multiple domains (Chapters 5 and 7).

The other key contribution of my work is deriving an algorithm which is more feasible for human demonstrators than other State-of-the-Art algorithms. Most LfD algorithms are simply not designed with human demonstrators in mind, but rather for use with a synthetic demonstrator. However, there are factors that apply to human demonstrators that do not bother synthetic demonstrators (such as context-switching\(^3\)). For example, DAGger stochastically switches between whether the demonstrator or the learning agent is controlling, which is fine for a synthetic demonstrator but may be jarring to a human demonstrator. As such, algorithms are needed which impose less cognitive burden on humans, or are otherwise more feasible for learning from them. The SALT algorithm, in addition to the previous benefit of learning from relatively little training data, has also been shown in a user study to not only learn better from human demonstrators than DAGger, but also improve upon DAGger in other areas such as the amount of effort it takes to train, how enjoyable it is to train, and how well human demonstrators perceived the algorithms to learn (The study and results are detailed in Chapter 8).

A more minor, but still important, contribution is the idea of a “demonstrator budget” for LfD domains. This makes it so that the learner has a limit of how much information it can request from the demonstrator, and therefore rewards methods which choose when to call the demonstrator more effectively. This idea, specifically the idea of a budget which is split up by iteration, has been shown to improve results in our domains over letting the learner request as much information from the demonstrator as it wants (this idea and the results of applying it are detailed in Chapter 7).

We also derived two metrics for determining how much cognitive burden is imposed on a demonstrator when training a learner. These are important because they can give an idea of how much cognitive burden an algorithm may impose on humans using synthetic demonstrators, before performing the much more time-intensive process of a user study. We provide detailed descriptions of these two metrics and a comparison of SALT and other state-of-the-art relative to these metrics in Chapter 7.

Finally, we have provided a study of LfD via similarity based methods, which helped to guide our work in LfD. Specifically, we examined trace representation, feature selection (including a novel feature selection method, NSFS), distance functions, and time windows within LfD domains. We examine these concepts in detail, how they affect learning performance in these domains, and which strategies help to more effectively address LfD problems in Chapter 4.

### 9.2 Future Work

One interesting area of future work is farther comparison between SALT and other state-of-the-art methods (extending the work in Chapter 6). So far we have been able to examine some of the differences between SALT and DAGger, but have been unable to pinpoint exactly why they perform so differently. A deeper investigation into the differences between SALT and DAGger/SafeDAGger and a more thorough understanding of why they differ in performance will help to formulate better LfD methods in the future.

Another important area is to improve upon SALT by continuing to derive new policies, focusing on those which can learn from very small amounts of training data or are less cognitively burdensome for humans. Furthermore, performing more user studies in different domains and with different methods may give valuable insights into exactly what human demonstrators like and don’t like about each method, and how effective each method is at learning from human demonstrators – insights which would help drive Learning from Human Demonstration methods in the future.

Similarly, there is a need for new, specialized LfD algorithms which are designed to learn directly from humans. In the user study detailed in Chapter 8, we found that humans can perform an action correctly yet inconsistently (for example, filling the row of a board from left to right one time and
then from right to left the next time). Algorithms which are able to account for this and learn a
more general process from these inconsistent-but-correct activities (such as “fill an entire row”) might
be able to more effectively learn from human demonstrators.

Finally, a more minor but also important area of future work is to derive more LfD domains. 
Currently, there is not as diverse a plethora of LfD domains as there is for other areas (for example, 
standard supervised learning), so more domains are needed as testbeds. Furthermore, by creating 
a variety of LfD domains that illustrate different idiosyncrasies of LfD problems, it would be easier 
to examine how generalizable LfD approaches are. Additionally, by creating data repositories of 
humans performing a task, it would allow for methods to be tested on human data before moving 
on to user studies, which are much more time intensive.
Chapter 10: Appendix A - List of Publications

10.1 Accepted Peer-Reviewed Papers

- Brandon Packard and Santiago Ontaño (2017) Policies for Active Learning from Demonstration, in AAAI Spring Symposium on Learning from Observation of Humans
- Brandon Packard (2016) Similarity-based Approaches to Learning from Demonstration, in ICCBR Workshops

10.2 Pending Peer-Reviewed Papers

- Brandon Packard and Santiago Ontaño (Pending Review) The SALT Algorithm for Active Learning from Demonstration, Machine Learning Journal
Bibliography


[16] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete
data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages


[20] Fifa 18 fans are using terrible players to "exploit" the ai. https://www.eurogamer.net/

[21] Tesca Fitzgerald and Andrea L. Thomaz. Skill demonstration transfer for learning from demonstra-
tion. In *Proceedings of the Tenth Annual ACM/IEEE International Conference on Human-
Robot Interaction Extended Abstracts*, HRI’15 Extended Abstracts, pages 187–188, New York,
NY, USA, 2015. ACM.

on Case-Based Reasoning for Computer Games: 19th international conference on Case-Based
Reasoning (Figure 1)*, pages 37–41, 2011.

[23] Michael W Floyd and Babak Esfandiari. Learning state-based behaviour using temporally


[28] Daniel Fu and Ryan Houlette. Putting ai in entertainment: An ai authoring tool for simulation


Player Behavior in a" First Person Shooter" Video Game*. PhD thesis, University of Wisconsin-
Madison, 2002.

[31] Norman Haas and Gary G Hendrix. An approach to acquiring and applying knowledge. In


[33] Ioannis Havoutis, A Tanwani, and Sylvain Calinon. Online incremental learning of manipulation


Bibliography


