The Thirty-First International Florida
Artificial Intelligence Research Society Conference (FLAIRS-31)

Learning Behavior from Limited Demonstrations in the Context of Games

Brandon Packard,Santiago Ontañón
Drexel University
Philadelphia, PA, USA
{btp36,so367}@drexel.edu

Abstract

A significant amount of work has advocated that Learning from Demonstration (LfD) is a promising approach to allow end-users to create behaviors for in-game characters without requiring programming. However, one major problem with this approach is that many LfD algorithms require large amounts of training data, and thus are not practical. In this paper, we focus on LfD with limited training data, and specifically on the problem of Active Learning from Demonstration in settings where the amount of data that can be queried from the demonstrator is limited by a predefined budget. We extend our novel Active Learning from Demonstration approach, $SALT$, and compare it to related LfD algorithms in both task performance (reward) and similarity to the demonstrator's behavior, when used with relatively small amounts of training data. We use *Super Mario Bros.* and two variations of the *Thermometers* puzzle game as our evaluation domains.

Introduction

This paper focuses on Learning from Demonstration (LfD), also known as Learning from Observation, Behavioral Cloning, Imitation Learning or Apprenticeship Learning. LfD is the problem an agent faces when learning to perform a task by watching the performance of a demonstrator. LfD has been proposed many times as a solution to the problem of behavior authoring (and often within the context of games, employing tactics such as Inverse Reinforcement Learning (Tastan and Sukthankar 2011), Neural Networks (Stanley *et al.* 2005), or C4.5 decision trees (Young and Hawes 2014), to name a few). However, most current LfD approaches assume 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.

In order to address this problem, this paper presents a new LfD setting for $SALT$: LfD with a maximum demonstrator query budget. We extend the $SALT$ framework introduced in our previous work (Packard and Ontañón 2017) to this setting via defining a collection of new $SALT$ strategies which accept a predefined budget. In this context, we define "demonstrator query budget" as the amount of training data the learning agent is allowed to request from the demonstrator. We compare $SALT$ to existing LfD algorithms (including $DAgger$ (Ross *et al.* 2010), $SafeDAgger$ (Zhang and Cho 2016), and a standard supervised approach).

The rest of this paper is organized as follows. After briefly describing some background, our algorithm and experimental setup are described. Our experimental results are then detailed, followed by related work. The paper then concludes with conclusions and future work.

Background

Learning from Demonstration (LfD) is a type of machine learning that focuses on learning to perform a task by observing the actions of a demonstrator. LfD is very common in humans (Schaal 1997; Heyes and Foster 2002), who often look to a teacher for information on how to perform a task. The overall goal of LfD is, given training data consisting of a set of trajectories gotten from a demonstrator, derive a policy which allows the learner to choose an action based on a current observed world state (and which will match the demonstrator's policy as closely as possible). The reader is referred to Argall et al. (2009) for a formal definition of LfD.

Many LfD approaches in the literature today are based on supervised learning, and therefore ignore the fact that LfD violates the i.i.d. assumption[1]. Some existing algorithms attempt to account for this violation, such as $DAgger$ (Ross *et al.* 2010) and SMILe (Ross and Bagnell 2010). However, these have limitations for human demonstrators.

One of those limitations 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. 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. For example, a human demonstrator might have to play or relabel as many as 660 levels of Super Mario in order to effectively train an agent using $DAgger$ (Ross *et al.* 2010). $DAgger$, proposed by Ross *et al.*, is another active Learning from Demonstration algorithm that at-

[1]That instances from the training and test set are *independently and identically distributed*.

tempts to account for the violation of the i.i.d. assumption (Ross *et al.* 2010). 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 in validation is chosen as the final policy.

Selective Active Learning from Traces

This section briefly introduces $SALT$ (Selective Active Learning from Traces), designed to reduce the cognitive burden of the demonstrator in LfD.The reader is referred to (Packard and Ontañón 2017) for a more in-depth description of $SALT$.

The overall idea of $SALT$ is the following: Let D_l be the distribution of states in the training set from which the agent has learned and D_t the distribution of states the agent would encounter when executing the learned policy. Due to LfD's violation of the i.i.d. principle, small errors in trained strategies can compound when testing (Ross and Bagnell 2010). This can cause D_t to be potentially very different from D_l. $SALT$ is an iterative algorithm (like $DAgger$) which attempts to collect training data which will make D_t and D_l as similar as possible by letting the demonstrator and the learning agent take turns in performing the task. The main difference with respect to $DAgger$ is in how the learner and the demonstrator share control. $SALT$ monitors whether the current state the learner is in is within D_l. If the learner leaves D_l, the demonstrator is given control until the state is back in D_l. This means that control between the learner and demonstrator is not done stochastically, as in $DAgger$. Training data is generated only when the demonstrator is in control, so the demonstrator does not need to provide actions for states encountered when not in control. These qualities help reduce the cognitive burden on human demonstrators.

The key to make $SALT$ work is determining when to give control to the demonstrator, and when to give it back to the learner. Three strategies are used to make these decisions:

- ρ_s: determines when the learner has moved out of D_l.
- ρ_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.
- ρ_d: determines how long the demonstrator should perform the task before the learner is given back control.

In the experiments reported in this paper, we tested several different variants for ρ_s, which offer different ways of detecting when the learner has moved out of D_l based on how well the task is being performed or the actions the demonstrator would have made. We experimented with a single variant for ρ_b, which never backs up the world state. Finally, two variants were used for ρ_d, which determine whether control should be given back to the learner based on either how far along performing the task the learner should have been when it moved out of D_l or the completion of a single trajectory. Detailed descriptions of the variants of these three strategies used in our experiments are provided below.

The training data used by $SALT$ is in the form of demonstrations or *trajectories*, where a "trajectory" is defined as one run of the game (playing one level, solving one board, etc). Moreover, $SALT$ is an iterative algorithm where at each iteration a set of C trajectories are collected. At the end of each iteration, $SALT$ re-trains the learning agent with all the training data collected so far. In the experiments below, we executed $SALT$ for a fixed number of iterations, N. For each of the N iterations excluding the first, the learner performs the task until strategy ρ_s determines that the learner has moved out of D_t. The state is then backed up a number of ticks, as determined by ρ_b, and then the demonstrator is given control until ρ_d determines the state is back in D_t. States the demonstrator encounters while controlling and its actions are added to the training data for the next iteration (See Algorithms 1 and 2 for the detailed procedure).

Moreover, several variants of $SALT$ strategies used in this work assume the existence of a *reward function* R. R is assumed to be a function that gives a numeric accumulated reward to the agent at each time step representing how well the agent is doing overall, not just in the current state or last action (for example, in the context of video games the scoring mechanism of the game could be the reward function).

LfD with a Demonstrator Query Budget

In order to reduce the amount of training data that a human demonstrator would have to provide, this paper introduces the idea of a *demonstrator query budget*. As described above, 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 (the first iteration of $SALT$ or $DAgger$). During the second step (the remaining iterations), the learning agent is the one performing the task, but 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. Once the budget is reached, the learner cannot obtain any more data from the demonstrator. This reduces the amount of training data and 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, with the learner deciding how to split the budget over iterations; and (b) *per-iteration demonstrator query budget*, where the learning agent is given a budget $B/(N-1)$ for each iteration of $SALT$ (except for the first).

Budget-Aware $SALT$ Strategies

Specifically, in this work we examined seven possible variants of ρ_s, which determines when the learner has moved out of D_l, and thus the demonstrator is asked to take over:

- ρ_s^{SS} (*Simple Stochastic*): signal the learner has exited D_l with probability $P = \frac{|ER_t - R_t|}{max}$, 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_t is the learner's reward at the current time, and max is the maximum possible reward for the domain.
- $\rho_s^{\leq}$ (*Reward Doesn't Increase*): signal the learner has exited D_l if the learner's reward has not increased compared to the previous time step (i.e., if $R_t \leq R_{t-1}$).
- ρ_s^{MD} (*Minimum Distance*): using Euclidean distance, the most similar game state s^* from the set of states the demonstrator visited during the first iteration of $SALT$ to the current game state s_t is found. If the distance between s_t and s^* is higher than a threshold value α, signal the learner has exited D_l. In our experiments, α was calculated by averaging the minimum distances from each state the demonstrator visited during the first iteration of $SALT$ to any other state visited during that iteration.
- ρ_s^{Pseudo}($PseudoDAgger$): signal the learner has exited D_l with probability $\frac{RemainingBudget}{B}$, where $RemainingBudget$ is the query budget left.
- $\rho_s^{W.SS}$, $\rho_s^{W.\leq}$, and $\rho_s^{W.MD}$: These strategies are the same as ρ_s^{SS}, $\rho_s^{\leq}$, and ρ_s^{MD} respectively, but with their chance to signal multiplied by $\frac{RemainingBudget}{B}$.

We also examined two possible variants of ρ_d, which determines for how long to give control to the demonstrator:

- ρ_d^{RG} (*Reward Goal*): 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.
- ρ_d^{EOT} (*End of Trajectory*): Signals to not give control back to the learner until the end of the current trajectory.

Finally, only one variant of ρ_b was used:

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

Algorithm 1 $SALT(\rho_s, \rho_b, \rho_d, C, N)$

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

Algorithm 2 runOneTrajectory(π, ρ_s, ρ_b, ρ_d)

```
while The task has not stopped nor been completed do
    if not outside of $D_l$ according to $\rho_s$ then
        sample using $\pi$
    else
        back up world according to $\rho_b$
        sample using $\pi^*$ according to $\rho_d$
    end if
end while
return $\{(s, \pi^*(s)) | s \in S^*\}$, where $S^*$ is the set of all states where $\pi^*$ was used.
```

Experimental Evaluation

In order to evaluate our approach, we used two application domains. The first is the classic Super Mario platform game (Figure 2 left), a platformer game where the player has to reach the end of a level while avoiding enemies. The second is a puzzle game known as Thermometers (Figure 2 right), where the player sees a board with overlaid with thermometers of different lengths and orientations, and needs to determine how full or empty each thermometer is based on a set of row/column constraints. Boards in the Thermometers domain were of size 5x5. Any supervised learning method could be used as an underlying learner for $SALT$, but in our experiments WEKA's J48 was used (Witten *et al.* 2016).

For the Super Mario domain, we used the Super Mario implementation from the 2012 Mario AI Championship[2]. World states in Super Mario are represented by a collection of 1083 features representing the surroundings of Mario (walls, enemies, 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, two versions were used. 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"). States in Simple-Thermometers are a vector of 14 features representing 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 $N = 1$ trajectories and $C = 25$ iterations. Each trajectory has a time limit of 30 seconds, and data is taken every tick (where the game runs at 20 ticks per second), with a maximum trajectory length of 450 training instances. Strategy ρ_d^{RG} was used to give control back to the learner. The reward function used is simply Mario's X-coordinate in pixels plus 500 if he is Fire Mario or 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 $N = 1$ trajectories and $C = 50$ iterations, also with strategy ρ_d^{RG}. For the Simple-Thermometers

[2]http://www.marioai.org/home

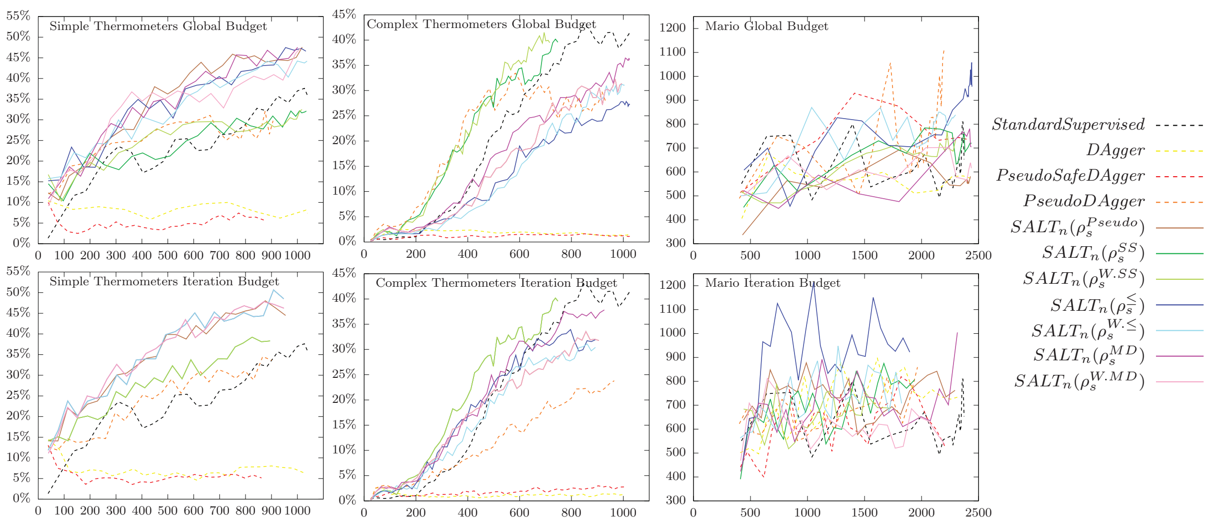

Figure 1: Reward gained in each domain as a function of the amount of training data, for various strategies and baselines. The vertical axis represents the amount of reward gained and the horizontal axis represents the amount of training data. The top row are results gained using a global budget, and the bottom row are results gained splitting the budget up evenly over iterations.

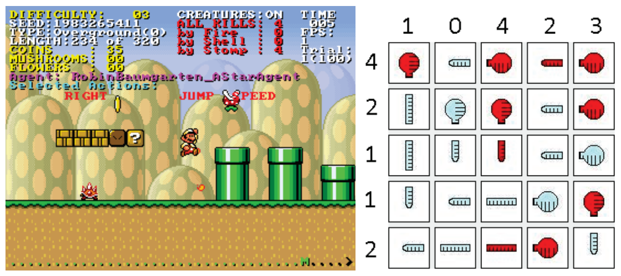

Figure 2: A screenshot of Super Mario (left) and the Thermometers puzzle game (right).

domain, experiments were performed using $N = 1$ trajectories and $C = 25$ iterations, using strategy ρ_d^{EOT} (note that we used a different ρ_d here due to it obtaining much better results in this domain 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 trajectory. The reward function for both is the overall percentage of constraints satisfied for the puzzle, 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 each thermometer has a legal configuration (filled starting from the round bulb, with all filled pieces adjacent).

Two metrics were used in this paper: task reward (R), and similarity between the learner and demonstrator (the percentage of times the learner predicted the same move as the demonstrator in a set of validation levels).

Baselines

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

- *Supervised*: we evaluate the performance of the base supervised learner. This is done by having the demonstrator perform the task until we have the same amount of training data as for the other methods, then training the learner.
- $DAgger$ (Ross *et al.* 2010).
- $PseudoSafeDAgger$ (Zhang and Cho 2016): to emulate the behavior of $SafeDAgger$, we use ρ_s^{DCA} (Packard and Ontañón 2017), which signals 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 β according to a modified edit distance, having the demonstrator control for a single frame as $SafeDAgger$ would. Note our emulation of $SafeDAgger$ calls the demonstrator at each time step, while $SafeDAgger$ would try to learn a function to compare them without calling the demonstrator.
- $PseudoDAgger$: a $DAgger$ inspired algorithm – queries the demonstrator with probability of $\frac{RemainingBudget}{TotalBudget}$. This is the same as using ρ_s^{Pseudo} in $SALT$, but only having the demonstrator control for a single frame.

Results

Task Reward

Figure 1 shows the accumulated reward (as a function of the amount of training data) obtained by all of the methods tested in our experiments. Notice not all lines are of the same length, since some methods collect more data than others. The first thing that we can observe is that many active learning performs 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 gain comparable reward ($\rho_s^{W.SS}$ and ρ_s^{SS}), and in the Super

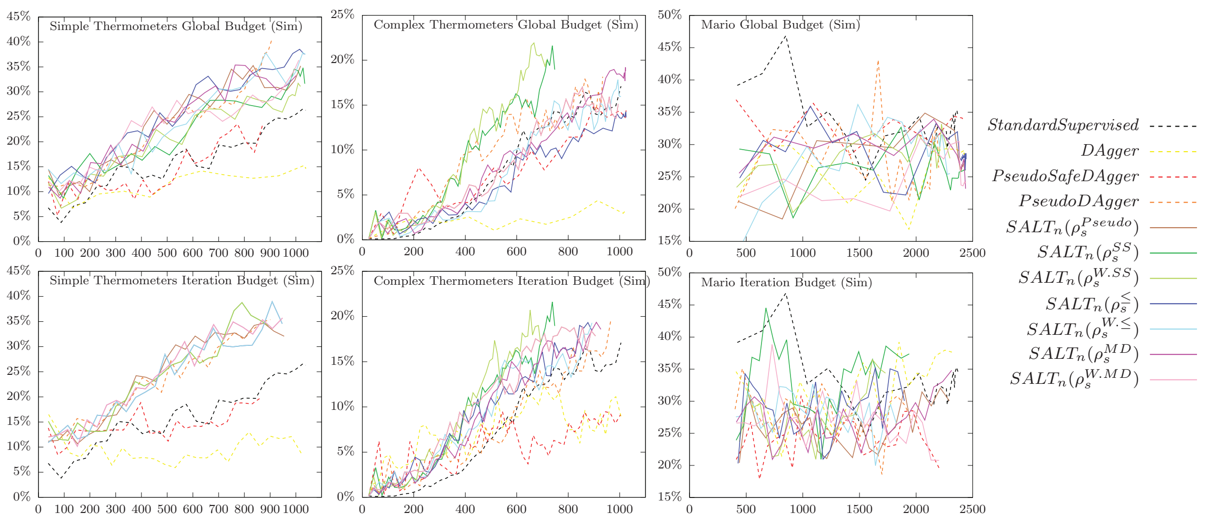

Figure 3: Percentage of time that the learner's chosen move matched the move the demonstrator would of made during validation (vertical axis) as a function of the amount of training data (horizontal axis) for various strategies and baselines.

Mario domain there are a couple methods that reach a much higher reward ($\rho_s^{\leq}$ and $PseudoDAgger$).

Similarly, $SALT$ outperforms the baselines in every domain – in both Thermometers domains, $DAgger$ and $SafeDAgger$ perform the most poorly of all of the examined methods, reaching a very low reward for the amounts of training data examined. In the Super Mario domain, these methods offer moderate performance, but are below that of 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 $\rho_s^{\leq}$.

It can also 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), the former also tend to receive a higher reward. For example, in the Simple Thermometers domain, methods using an iterative budget instead of a global budget generally reach a higher reward and train faster than those that do not (for example, 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 instances instead of around 800). We also see that weighted versions of $SALT$ strategies do not seem to improve results (a glaring example of this being $\rho_s^{\leq}$ compared to $\rho_s^{W.\leq}$ when using an iterative budget in Super Mario).

Demonstrator Similarity

Figure 3 shows the demonstrator similarity obtained (as a function of the amount of training data) by all of the methods tested in our experiments. We can easily 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 quickly receives the highest similarity, but decreases 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 high similarity because the demonstrator presses these buttons a large proportion of the time. As the agent learns it performs these actions less so its similarity falls, and then eventually rises again as it begins to more closely emulate the demonstrator.

$SALT$ also 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, (all but Standard Supervised).

Finally, we can see again that most of the methods using an iteration-based budget train faster and receive the same or higher reward than those using a global budget (the most notable example of this is $SALT$ with ρ_s^{SS} in Super Mario, 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 weighted versions of $SALT$ strategies does not seem to boost results overall.

Related Work

Recent work on trying to reduce training data required by $DAgger$ includes $SafeDAgger$ (Zhang and Cho 2016) and Shiv (Laskey *et al.* 2016). Another active LfD algorithm which attempts to reduce the amount of needed training data is RAIL (Judah *et al.* 2014). Like $SALT$, RAIL seeks to help account for the i.i.d. violation via demonstrator queries after the initial learner has been training. However, RAIL assumes that the learner has access to a simulator of the domain, which $SALT$ does not require. More general methods for reducing training data include novelty reduction and uncertainty reduction (Silver *et al.* 2012), which seek to ask the demonstrator for more demonstrations using sampled problems that are considered to be too different from previously seen states or for which the learner is too uncertain about what to do in them. For their robot navigation domain, they find this requires less demonstrator interaction while getting improved results.

An approach to performing LfD with a demonstrator query budget comes from Floyd and Esfandiari (2011), who use a mixed-initiative system for case acquisition in case-based reasoning. Their system allows the learner to cede control to the demonstrator when it cannot produce an action for the current state, the demonstrator to seize control, and has the demonstrator cede control after a single "turn" of the Tetris game. They found that this increased the number of pieces the learner placed before getting a game over compared to a passive approach. Another example of mixed control is MABLE (Freed *et al.* 2011), a Learning by Instruction system. Like $SALT$, MABLE allows the demonstrator and learner to work together to improve learning. However, in MABLE, it is the human demonstrators who determine there is an issue, not the learner.

Conclusions

This paper has studied learning from demonstration in the context of video games, and specifically on the problem of learning with a limited amount of training data. This is motivated by the fact that if learning from demonstration is expected to be a useful and practical means of defining game AI, it has to work with the amount of data that humans can realistically provide. We addressed this problem by introducing the concept of *demonstrator query budget*, and introducing a new set of strategies to extend the $SALT$ learning from demonstration framework to account for this budget.

The results of our experiments show that adding the notion of a demonstrator budget increases the effectiveness of $SALT$ and improves the performance with respect to not having the budget when training on the same amount of training data. For each domain examined, using a per-iteration budget either trains faster, reaches a higher reward/similarity, or both than using a single global budget.

As future work, we would like to explore a larger collection of the strategies that govern the behavior of $SALT$, focusing on those that can improve learning performance for even lower amounts of training data to make it even more plausible for human demonstrators. Additionally, we would like to perform user studies to obtain further insight into the amount of effort $SALT$ requires from human demonstrators.

References

Brenna Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

Michael W Floyd and Babak Esfandiari. Supplemental case acquisition using mixed-initiative control. In *FLAIRS Conference*, 2011.

Michael Freed, Daniel Bryce, Jiaying Shen, and Ciaran O'Reilly. Interactive bootstrapped learning for end-user programming. *Artificial Intelligence and Smarter Living*, 11:07, 2011.

CM Heyes and CL Foster. Motor learning by observation: Evidence from a serial reaction time task. *The Quarterly Journal of Experimental Psychology: Section A*, 55(2):593–607, 2002.

Kshitij Judah, Alan P Fern, Thomas G Dietterich, et al. Active lmitation learning: formal and practical reductions to iid learning. *The Journal of Machine Learning Research*, 15(1):3925–3963, 2014.

Michael Laskey, Sam Staszak, Wesley Yu-Shu Hsieh, Jeffrey Mahler, Florian T Pokorny, Anca D Dragan, and Ken Goldberg. Shiv: Reducing supervisor burden in dagger using support vectors for efficient learning from demonstrations in high dimensional state spaces. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 462–469. IEEE, 2016.

Brandon Packard and Santiago Ontañón. Policies for active learning from demonstration. In *Proceedings of AAAI 2017 Spring Symposium on Learning from Observation of Humans*, pages 513–519, 2017.

Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. In *International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, pages 661–668, 2010.

Stéphane Ross, Geoffrey J Gordon, and J Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. *arXiv:1011.0686*, 2010.

Stefan Schaal. Learning from demonstration. *Advances in Neural Information Processing Systems (NIPS 1997)*, pages 1040–1046, 1997.

David Silver, J Andrew Bagnell, and Anthony Stentz. Active learning from demonstration for robust autonomous navigation. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 200–207. IEEE, 2012.

Kenneth O Stanley, Ryan Cornelius, Risto Miikkulainen, Thomas DSilva, and Aliza Gold. Real-time learning in the nero video game. In *AIIDE*, pages 159–160, 2005.

Bulent Tastan and Gita Reese Sukthankar. Learning policies for first person shooter games using inverse reinforcement learning. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2011)*, 2011.

I.H. Witten, E. Frank, M.A. Hall, and C.J. Pal. *Data Mining: Practical Machine Learning Tools and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2016.

Jay Young and Nick Hawes. Learning micro-management skills in RTS games by imitating experts. In *AIIDE*, 2014.

Jiakai Zhang and Kyunghyun Cho. Query-efficient imitation learning for end-to-end autonomous driving. *arXiv:1605.06450*, 2016.