Policies for Active Learning from Demonstration

Brandon Packard
Drexel University
Philadelphia, PA, USA
btp36@drexel.edu

Santiago Ontañón
Drexel University
Philadelphia, PA, USA
so367@drexel.edu

Abstract
This paper focuses on Learning from Demonstration, and in particular, on the problem of Active Learning from Demonstration in settings where the amount of data that can be acquired from the demonstrator is limited. In this paper we propose a novel Active Learning from Demonstration approach, SALT, as well as a brief exploration of using a different underlying learning algorithm. We evaluate this approach in two domains: the classic platform game Super Mario Bros. and a grid-based puzzle game called Thermometers.

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 an expert. Given data about what the environment state was like over time, and the actions the expert performed, the goal of LfD is to learn the function that determines the expert actions given the observed environment states.

Many LfD approaches in the literature are based on supervised learning, and thus ignore the fact that LfD violates the i.i.d. assumption¹ (this is elaborated upon below). Some existing algorithms attempt to account for this violation, such as DAgger (Ross, Gordon, and Bagnell 2010) and SMILe (Ross and Bagnell 2010). However, these have limitations when the demonstrator is human.

This paper presents an Active LfD algorithm, which we call Selective Active Learning from Traces (or SALT for short), that does not make the i.i.d. assumption and is focused on being more feasible for human experts. The basic idea for SALT is to allow the learner to perform a task, and when it is determined that the learner has moved out of the space for which it has training data, the expert then takes over and shows the learning agent how to get back into this space. This extra expert data is then added to the training data for the next iteration of the learner. The main idea here is to collect training data for the set of states expected to be found during performance, and thus make the distribution of states found during testing follow a similar distribution to that in the training set (i.e., the goal is to turn the learning task into an i.i.d. task, so that supervised learning algorithms can be used). This is the same idea that algorithms like DAgger follow, with the key difference that SALT focuses on the scenario where the demonstrator is a human by trying to reduce the cognitive load on the expert, since humans cannot provide as many demonstrations or react as quickly as an artificial intelligence.

The rest of this paper is organized as follows. After we briefly describe some background and the differences between LfD and standard supervised learning, our algorithm SALT is detailed. Next, our experimental setup and the results of our experiments are presented. The paper concludes with discussion/conclusions and ideas for 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 an expert. 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.

Argall et al. (Argall et al. 2009) formally define LfD as follows. Let \( S \) be the set of states the world can be in, \( Y \) the set of actions the agent can perform, and that the mapping between states by way of actions is then defined by a probabilistic transition function \( T(s'|s, y) : S \times Y \times S \rightarrow [0,1] \). It is assumed that the state is not fully observable, with the learner instead having access to the set of possible observed states \( Z \) through the mapping \( M : S \rightarrow Z \). A policy \( \pi : Z \rightarrow Y \) selects actions based on observations of the world state. A trace \( T := [(z_1, y_1), ..., (z_n, y_n)] \) is defined as a sequence of observation-action pairs (also known as training instances), where \( z_i \in Z \), and \( y_i \in Y \). The goal of LfD is then, given training data consisting of a set of traces \( T = \{T_1, ..., T_n\} \), derive a policy which allows the learner to choose an action based on the current observed world state. Moreover, for simplicity, in this paper we will ignore the observation function. LfD differs from standard supervised learning in three key ways:

1. **Violation of the i.i.d Assumption**: standard supervised approaches assume that instances of data are drawn independently and that the training and test data are identically

¹That instances from the training and test set are independently and identically distributed.
distributed. However, this is not the case for LfD, as the training set is generated according to the policy of the expert, while the states encountered during testing depend on the learned policy.

2. Sequential Nature: data in LfD is sequential in nature, since past states and actions might affect future actions. Therefore, LfD is related to sequential machine learning (Dietterich 2002).

3. Difficulty of Evaluation: for most supervised machine learning applications, simply measuring how a learner performs on the desired task is a sufficient performance metric. However, this is not the case for LfD – it is not enough to measure how well a learner performs a task, it is also important to measure how similarly to the expert the learner performs the task. As such, while standardized supervised machine learning has many well-established performance metrics, such as precision and recall, not many metrics which evaluate both aspects of success in LfD have been proposed in the literature.

Due to these differences, applying standard supervised approaches to LfD meets with limited success (Bagnell 2015), and therefore there is a need for specialized approaches that account for these differences.

Related Work

The problem of LfD has received significant attention in the literature. For example, Ross, Godron, and Bagnell, study LfD in the context of the Super Tux Cart and Super Mario Bros games (Ross, Gordon, and Bagnell 2011). Other approaches to LfD include Inverse Reinforcement Learning (Abbeel and Ng 2004; Tastan and Sukthankar 2011), Dynamic Bayesian Networks (such as HMMs) (Dereszynski et al. 2011; Ontañón, Montaña, and Gonzalez 2014), and supervised learning techniques (Sammut et al. 1992) (including relational approaches (Natarajan et al. 2011)). The reader is referred to (Argall et al. 2009) and (Ontañón, Montaña, and Gonzalez 2014), for recent surveys on LfD.

LfD is often studied in the context of reinforcement learning (Schaal 1997). For example, a recent example is the work of Kim et al. (2013), where they integrate policy iteration with LfD. Another common approach is that of Inverse Reinforcement Learning (IRL) (Ng, Russell, and others 2000), where a reward function is inferred from demonstrations, and then reinforcement learning is used to derive a policy. A recent example of the application of IRL is the work of Boularias, Kober, and Peters (2011), where they focus on generalization from a small number of samples using an extension of maximum entropy IRL.

Most algorithms for LfD in the literature, however, still use standard supervised machine learning approaches. For example, Sammut et al. propose the use of decision trees built from expert traces to teach a learner how to fly a plane (Sammut et al. 1992). They attained good results by making several adjustments to their problem/data (ideally, of course, no adjustments would be needed, but the learner did accomplish the flight task fairly well.). One such adjustment is that they split the data into seven segments based on the seven maneuvers required during the flight task that they were trying to learn, and trained a decision tree for each action for each stage, switching what policy (which decision trees) the learner uses depending on where in the flight it is.

One algorithm that was derived to attempt to account for the violation of the i.i.d. assumption is SMILe, by Ross and Bagnell (2010). 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, Gordon, and Bagnell, is another active Learning from Demonstration algorithm that attempts to account for the violation of the i.i.d. assumption (Ross, Gordon, and Bagnell 2010). The idea behind DAGger is to take learning data from a series of traces of the expert performing a task, and train a learner on that data. The algorithm then repeats the process, but reduces how often the expert 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 expert 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. A related approach is that of Floyd and Esfandiari (2009), who focus on the problem of how to create sequences of problems to show to the expert in interactive learning by observation.

Selective Active Learning from Traces

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 (2010), 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 $D_l$ potentially very different from $D_t$. Intuitively, SALT is an iterative algorithm inspired by DAGger with the goal of collecting training data so that $D_l$ and $D_t$ 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 expert 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 expert is given control until the state is back inside of $D_l$. Each time the expert takes control, more training data is generated and added to the training set. In this way no stochastic mixing of expert and learning agent is required, nor is there any need for asking the expert 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 expert, and when to give control back to the learning agent. Three policies are used for these decisions:
• $\rho_s$: determines when the learner has moved out of $D_t$.
• $\rho_b$: when control is given to the expert, 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 $D_t$. This policy determines how far the world state should be backed up before allowing the expert to perform the task.
• $\rho_d$: determines how long the expert should perform the task before the learner is given back control.

**SALT**

*SALT* (Selective Active Learning from Traces) starts with an empty training set (see Algorithm 1 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 expert control, and those traces become the training data for the second iteration. For each iteration excluding the first, the learner performs the task until policy $\rho_s$ determines that the learner has moved out of $D_t$. 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 expert is given control until the state goes back into $D_t$, as determined by policy $\rho_d$. The states that the expert encounters during its control and the actions that it takes are added to the training data for the next iteration.

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

1: Sample $C$-step traces using $\pi^*$ (the expert’s policy)
2: Initialize $D \leftarrow \{(s, \pi^*(s))\}$ - all states visited by the expert and the actions it took
3: Train classifier $\pi_1$ on $D$
4: for $i = 1$ to $N$ do
5: Initialize $D_i \leftarrow \emptyset$
6: for $j = 1$ to $C$ do
7: $D_i = D_i \cup \text{runOneTrial}(\pi_i, \rho_s, \rho_b, \rho_d)$
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

**Algorithm 2** $\text{runOneTrial}(\pi, \rho_s, \rho_b, \rho_d)$

if not outside of $D_t$ 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
return $\{(s, \pi^*(s))|s \in S^*\}$, where $S^*$ is the set of all states where $\pi^*$ was used.

**Policies**

We experimented with several versions for some of the three policies that make key decisions in *SALT*.

We assume the learning agent has access to a reward function $r$, which assigns a numeric reward $r(s)$ to any world state $s$. Given a trace $T_t = [(s_1, y_1), \ldots, (s_n, y_n)]$, we define the accumulated reward at a time step $j$ as:

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

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

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

In this paper, we propose to use the difference between the expected accumulated reward of the expert and that achieved by the learning agent as a proxy for whether or not the learner has moved out of $D_t$. We experimented with different alternatives for each policy, listed below.

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

- For $\rho_b$, we experimented with one policy, $\rho_b^{B}$ (Back-n), which backs up the world state $n$ time steps. We experimented with different values for $n$, but in the experiments reported below, we settled for $n = 25$ and $n = 0$ for the Super Mario and Thermometers domains respectively, as they produced the best results.

- For $\rho_d$, we experimented with one policy, $\rho_d^{RG}$ (Reward-Goal), which signals to give control back to the learner when the accumulated reward reaches the expected accumulated expert reward $ER(j)$, where $j$ is the time step at which the expert took control.

**Underlying Learning Method**

For *SALT*, any supervised learning method could be used as the underlying learning method, but for our experiments we used $k$-nn (Cover and Hart 1967). World states were represented using logical clauses, and thus we employed the
Algorithm 3 stateSimilarity($X_1, X_2$)

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

Jaccard distance between clauses as our distance measure for k-nn. Given two world states, the Jaccard distance is defined as one minus the number of predicates shared between the world states, divided by the number of different predicates appearing in both world states:

$$J(z_1, z_2) = 1 - \frac{|z_1 \cap z_2|}{|z_1 \cup z_2|}$$

where $z_1 \cap z_2$ is a clause containing only predicates in both $z_1$ and in $z_2$, and $z_1 \cup z_2$ is a clause that contains the union of predicates in $z_1$ and in $z_2$. $|\cdot|$ represents the number of predicates in a clause. Two predicates are considered equal when both their functor and all their attributes are identical.

We also experimented with the Levenshtein (edit) distance, to examine whether the underlying learning method is becoming a limitation for SALT. Since the edit distance between large clauses (representing the world state) might have a prohibitive cost, we employed an approximation algorithm (Algorithm 3). This algorithm works by first computing a matrix with as many rows as predicates in $X_1$ and as many columns as predicates in $X_2$. Each position of this matrix contains the edit distance between the corresponding row and column predicates. When computing the edit distance between two predicates, we normalize it by the size of the larger predicate, in order to bound the distance to be in the interval $[0, 1]$. Then, the distance between the two clauses is computed by adding the maximum distance value in each column of this matrix, and dividing by the number of predicates in $X_1$ (the larger of the two clauses). This approximation algorithm thus has polynomial cost with respect to the number of predicates in the clauses, instead of the exponential cost required to compute the exact edit distance.

**Experimental Setup**

In order to evaluate our approach, we used two application domains. The first is the classic Super Mario platform game (Figure 1), where the player has to reach the end of a level while avoiding enemies and picking up coins. The second is a puzzle game known as Thermometers (Figure 2), 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. Every square starting as “undeclared”, and the player needs to determine which squares are filled and which squares are empty. For the Thermometers puzzle, we tested with boards of size 5x5.

World states in Super Mario are represented by a collection of 40 logical predicates representing the surroundings of Mario (whether there is a wall or not, where are the enemies, etc.). For example, if a state has the predicate $coin(1, 0)$, it means that there is a coin 1 tile immediately to the right of the main character (discretized coordinates are used which correspond to the tiles of the game environment, rather than exact coordinates). Note that there can be multiple instances of a predicate in a world state - if there are 3 goombas near Mario, the predicate “goomba” will appear in the world state 3 times. 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, the states are represented as a set of 15 logical predicates representing the board and the constraints imposed on each row and column of the board. The action space $Y$ consists of 75 actions, 3 actions for each tile on the grid (set to “full”, set to “empty”, and “clear”).

Experiments for Super Mario used $N = 10$ iterations and $C = 10$ traces with an $A^*$ agent as the demonstrator. For the Simple Deterministic policy, $\kappa$ was set to 64. Each trace has a time limit of 30 seconds. Data is recorded 3 out of every

![Figure 1: A screenshot of Super Mario.](image)

![Figure 2: A screenshot of the Thermometers puzzle game.](image)
4 ticks (the game runs at 20 ticks per second), leading to a maximum trace length of 450 instances.

For the Thermometers domain, experiments were performed using $N = 20$ iterations and $C = 20$ traces, with the demonstrator being a logic-based solver. For the Simple Deterministic policy, $\kappa$ was set to 10. Each 5x5 board has a limit of 50 moves, and therefore a maximum length of 50 instances per trace. In our experiments we found that the performance was not very sensitive to having “optimal” SALT parameters, reasonable parameters were sufficient.

**Baselines**

We report the performance of SALT and two baselines:

- **Supervised**: we evaluate 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 expert 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 (Ross, Gordon, and Bagnell 2010) in our domains for the sake of comparison.

**Results**

The different plots in Figure 3 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, c) show results for Super Mario, and the right-hand side plots show results for Thermometers.

Concerning varying the underlying learning method (using Jaccard vs Edit distance), we found that in one domain changing the similarity metric to our version of edit distance made SALT perform better, and in the other it made SALT perform worse. For the Mario domain (Figure 3.a), both Simple Stochastic and Simple Deterministic benefited by using edit distance (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 3.d) 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. It is also interesting that for the Mario domain, SALT with edit distance and the deterministic policy boasts roughly the same performance as DAgger does with Jaccard Distance.

When examining the policies in the Mario domain, the one that boasts the highest performance is the Reward Does Not Increase policy. As can be seen in Figure 3.b. This policy trains faster than Simple Stochastic or Simple Deterministic and levels off a little higher (around 600 reward at 2,000 training instances compared to 550 and 450 reward for the other two policies). The next best policy is Simple Stochastic, which outperforms the other SALT policies, followed by the Directly Compare Actions policy (specifically with a threshold of .21), meaning that the agent can differ from the expert 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 3.c shows that this policy offers performance in between our policies of Simple Deterministic and Simple Stochastic, leveling off at about 500 reward compared to 550 for Simple Stochastic and around 475 for Simple Deterministic. It is also important to note that all of the SALT policies perform much better than the standard supervised approach, but most of them perform significantly worse than DAgger (except Reward Does Not Increase, which comes close to tying it).

Similar results can be seen for the Thermometers domain. Figure 3.e shows that the Does Not Increase policy performs better than the other policies, leveling out at about 21% of constraints satisfied compared to 18% for Simple Stochastic and 16% for Simple Deterministic. Simple Stochastic and Compare Actions Directly (Figure 3.f) with a threshold of 0.35 (meaning that the learner can differ from the expert 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 Simple Deterministic with a reward of 18% of constraints satisfied at 5,000 training instances compared to 15%. In this domain, standard Supervised performs the best of the tests we ran, although at least one SALT policy (Reward Does Not Increase) 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 LiD 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.

**Conclusions**

In this paper, we presented a new algorithm for Active Learning from Demonstration, named SALT. SALT is inspired by the existing algorithm DAgger, but focuses on reducing the interaction with the demonstrator, in order to make these algorithms useable with human demonstrators. Although experiments with human demonstrators are part of our future work, we showed that SALT can almost match the performance of DAgger in Super Mario, and is robust enough for handling domains like the Thermometers domain, where DAgger performed very poorly. Moreover, these results were obtained not requiring the demonstrator to relabel traces after each iteration, nor using any stochastic mixing of learning agent and expert, thus reducing the cognitive effort on the demonstrator.

In the Mario and Thermometers domains, the Directly Compare Actions and Reward Does Not Increase policies both performed very well, with the later getting a higher reward than both old policies in both domains, allowing the learner to better learn the demonstrated task. In terms of the underlying learning method, it was found that in one domain (Thermometers), using the edit distance instead of Jaccard distance performs considerably worse, while for the other domain (Mario) the edit distance performs much better. Our hypotheses for the low performance of the edit distance on the Thermometers domain is that the current state representation does not afford good generalizations.
Figure 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 policies 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 policies and baselines. Note that for graph c, some of the policies gave identical results, so some of the lines perfectly overlap.

In the future, we would like to continue improving upon SALT, exploring a larger collection of variants for the three policies that govern the behavior of SALT and studying their performance in different domains and using different base learning algorithms. Additionally, we would like to perform user studies to obtain further insights into the amount of effort required by human experts to train a learning agent using SALT. Finally, we would like to expand our set of domains, in order to test the generality of our results.

Acknowledgements. This project was partially funded by NSF grant SCH-1521943.
References


