In *IEEE 7th International Conference on Development and Learning* (ICDL 08), Monterey, California, USA,, August 2008.

TAMER: Training an Agent Manually via Evaluative Reinforcement

W. Bradley Knox and Peter Stone
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712–0233
{bradknox,pstone}@cs.utexas.edu

***Abstract*—Though computers have surpassed humans at many tasks, especially computationally intensive ones, there are many tasks for which human expertise remains necessary and/or useful. For such tasks, it is desirable for a human to be able to transmit knowledge to a learning agent as quickly and effortlessly as possible, and, ideally, without any knowledge of the details of the agent's learning process. This paper proposes a general framework called Training an Agent Manually via Evaluative Reinforcement (TAMER) that allows a human to train a learning agent to perform a common class of complex tasks simply by giving scalar reward signals in response to the agent's observed actions. Specifically, in sequential decision making tasks, an agent models the human's reward function and chooses actions that it predicts will receive the most reward. Our novel algorithm is fully implemented and tested on the game Tetris. Leveraging the human trainers' feedback, the agent learns to clear an average of more than 50 lines by its third game, an order of magnitude faster than the best autonomous learning agents.**

I. Introduction

Reinforcement learning has recently made great strides in terms of applicability to complex sequential decision making tasks [1], [2], [3]. Nonetheless, there remain many real world applications in which tabula rasa learning either is intractable or takes too long for practical purposes. For instance, when learning on physical robots, extensive learning can subject the robot to physical wear [4]; or when learning in "high stakes" environments, suboptimal performance during learning can lead to significant financial losses. In some cases, there may be no alternative to learning from scratch. But in domains in which humans have some intuition or expertise, it can be necessary and/or useful to transfer knowledge about the task at hand to learning agents so as to reduce learning time.

Currently, the vast majority of knowledge transfer from humans to learning agents occurs through a programming language interface. This method, unfortunately, is slow and can only be harnessed by a small, technically trained subset of the population. Work has been done to create systems that allow humans to give advice to agents [5] or to demonstrate the task for the agent [2]. But because of the complexity of these two valuable methods, implementing them in a way that is accessible to a user without technical training is still challenging. Additionally, these methods respectively require that the human be able to articulate advice about the task or to perform the task herself.

In this paper, we develop a method by which the human trainer can merely give positive and negative reinforcement signals (called "reward" in the learning agent community) to the agent. It only requires that a person can observe the agent's behavior, judge its quality, and send a feedback signal that can be mapped to a scalar value (e.g. by button press or verbal feedback of "good" and "bad"). We assert that the minimal level of human expertise needed to train within our system is less than the level required to give advice or demonstrate correct behavior to the agent.

We extend early work [6] in which agents exploit human-given reward signals to aid task learning. We propose a general agent-trainer framework, called Training an Agent Manually via Evaluative Reinforcement or simply TAMER, that makes use of established supervised learning techniques to model a human's reward function and uses the learned model to choose actions that are projected to receive the most reward. Furthermore, we describe a specific, fully implemented algorithm that fits within the human training framework. We apply the novel algorithm to the game of Tetris under the guidance of human trainers. The agent learns a decent policy within 3 games, more than an order of magnitude faster than the best autonomous learning agents.

II. The General TAMER Framework

Sequential decision making tasks are commonly modeled as Markov decision processes (MDPs). Though MDPs are often addressed with reinforcement learning algorithms [7], in this paper we focus on learning the reward function (R) via supervised learning. A finite MDP is specified by the tuple (S, A, T, γ, D, R). S and A are, respectively, the sets of possible states and actions. T is a transition function, $T : S \times A \times S \rightarrow \mathbb{R}$, which gives the probability, given a state and an action, of transitioning to another state on the next time step. γ, the discount factor, exponentially decreases the value of a future reward. D is the distribution of start states. R is a reward function, $R : S \times S \rightarrow \mathbb{R}$, where the reward is a function of states s_t and s_{t+1}. Figure 1 shows a diagram of traditional agent-environment interaction in an MDP.

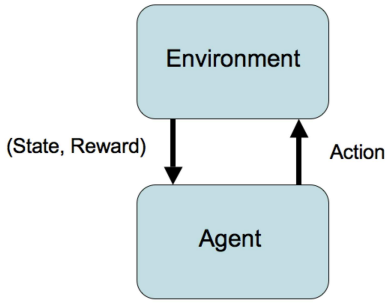

Fig. 1. Traditional scheme of interaction between agent and environment in a Markov Decision Process.

Typically an agent learns autonomously via environmental interaction [7]. In this work, we allow a human trainer to give feedback as shown in Figure 2. The agent's interaction with the environment differs from the

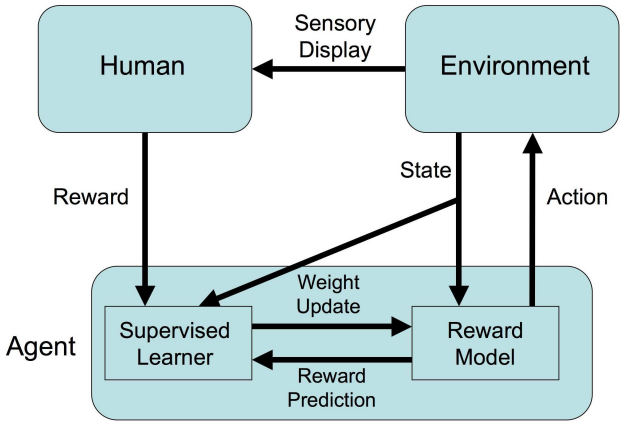

Fig. 2. Framework for Training an Agent Manually via Evaluative Reinforcement (TAMER).

usual framework because the reward R function has has been removed from the task specification, creating an MDP\R. Instead, reward comes from a human trainer who receives information about the current state, most likely via a visual representation. Based on his or her evaluation of the agent's recent performance, the trainer can choose to give reward in any form of expression that can be mapped to a scalar value.

Given the current state description, the agent's goal is to choose the action that will receive the most reward from the human. To do this, the agent models the human's "reward function" and greedily chooses actions that it expects to earn the most reward. After learning an accurate model of the human's reward, the agent can continue to perform the task in the absence of the human, choosing actions that are predicted to maximize the received reward if the human were present. Note that the agent tries to maximize immediate reward, not expected return (i.e. a discounted sum of all future reward), under the assumption that the human trainer is already taking each action's long-term implications into account when providing feedback.

This problem naturally lends itself to a supervised learning approach. We treat each action as a training sample. Specifically, for an action chosen at time t, the two states s_t and s_{t+1} are considered the attributes of the sample and the human trainer's reward for that action is considered the label. In more complex domains, state feature vectors $\overrightarrow{f_t}$ and $\overrightarrow{f_{t+1}}$, derivable from s_t and s_{t+1}, can be used as attributes instead.

Inconveniently for such an approach, reinforcement learning research [8] has found that humans do not have a consistent reward function. In fact, in the animal learning literature, training by reinforcement (i.e. shaping) is defined as rewarding successive approximations of the goal behavior [9]. In other words, as performance increases, the trainer's standards for what are "good" actions and increase as well. Thus, human reward functions are a moving targets. Therefore, function approximators (i.e. supervised learning algorithms) that are recency-weighted and allow incremental updates are neeeded.

A primary difference between our algorithm and most related work (exceptions are noted in Section III-C) is that the reward function comes only from the human, and can change according to the human's changing desires, perhaps as it seeks to fine-tune the agent's performance. The environment's specification carries no information that determines "correct" behavior. Behavior is correct if and only if the human trainer deems it correct.

In principle, the agent could learn autonomously as well, using both the trainer's feedback and the environmental reward, by combining it with an existing RL algorithm. Though we consider this extension in Section VI, learning only the human's reward function is a challenging and important subproblem of creating such an agent that is worth examining in isolation (by having the agent act greedily in accordance with the learned reward function). Additionally, there are times when the environmental reward function cannot be easily defined, such as when several goals are being sought at once (e.g. driving while staying on the road, avoiding obstacles, and with high speed). Furthermore, for agents that serve humans, sometimes the correct policy depends on the specific human's preferences. In these cases as well, learning from a specified environmental reward function could be problematic.

III. Comparison to other methods of human training

We are not the first to attempt to learn from human knowledge through natural interaction. In this section, we compare TAMER, our training framework, to others that allow humans to impart task knowledge to a learning agent. In comparison, our framework is among the easiest to implement. It also places a relatively small cognitive load and low demand of task expertise on the human trainer. We argue that the relative strengths and weaknesses of our training system put it in a unique space that is not currently occupied by any other approach. Furthermore, we believe that many of the approaches we review are complementary to ours, and that an ideal learning agent would blend elements of several of them.

A. Advice-taking agents

Advice, in the context of MDPs, is defined as suggesting an action when a certain condition is true. Maclin and Shavlik pioneered the approach of giving advice to reinforcement learners[5].

Although the informational richness of advice is powerful, it also places a larger burden, both attentional and temporal, on the human than a simple expression of approval or disapproval. Additionally, many current advice-taking systems [5], [10] require that the human encode her advice into a scripting or programming language, making it inaccessible to non-technical users.

Giving advice via a natural language interface would reduce the burden on a human. Though general natural language recognition is yet unsolved, Kuhlmann et al. [11] created a domain-specific natural language interface for giving advice to a reinforcement learner. However, their natural language unit requires manually labeled training samples, and work still remains on embedding advice into established learning algorithms.

In contrast, our system is relatively simple to implement. Additionally, there will likely be times when the trainer knows that the agent has performed well or poorly, but cannot determine exactly why. In these cases, advice will be much more difficult to give than positive or negative feedback.

B. Learning by human example

Another approach is for a human to give the agent an example of desired behavior, and then, from its observation, the agent learns to copy the human's example or even improve on it. This promising approach goes by a number of names, including imitation learning, learning by example, and learning by demonstration. The human teacher can give a demonstration either by performing the task with his or her own body [12] or by controlling a device similarly to how the agent does [2] (e.g. driving a robotic car). In a fully realized form, learning by example will be a natural and effective means of transferring knowledge from a human to a learning agent (or between agents), allowing the agent to learn a competent policy before taking a single action.

In Apprenticeship Learning [2], a subtype of learning by example, the algorithm begins with an MDP\R (as does the TAMER framework). A human temporarily controls the agent (or something similar). From the human's period of control, the algorithm learns a reward function, and then the agent trains on the MDP.

Both advice giving and learning by human example seem to place a relatively high cognitive load on the human, decreasing his or her ability to perform other tasks. This follows from the inuitive assertion that evaluating a performance takes less cognition than performing it oneself. Our interface only requires such an evaluation and a feedback signal that can be mapped to a scalar.

Additionally, considering the demonstration type in which a human controls the machine as the agent would, there are some tasks that are too difficult for a human trainer. This might be because the agent has more actuators than can be put in a simple interface (e.g. many robots) or because the task requires that the human be an expert before being able to control the agent (e.g. helicopter piloting in simulation). In these cases, a demonstration is infeasible. But as long as the human can judge the overall quality of the agent's behavior, then he or she should be able to provide feedback via our system, regardless of the task's difficulty.

C. Extracting Reward Signal from a Human

TAMER learns from a human's reward signal. Another name for training by human reward is clicker training. The name comes from a form of animal training in which a audible clicking device is previously associated with a reward and then used as a reward itself to train the animal.

Most clicker training has involved teaching tricks to simulated or robot dogs. Kaplan et al. [13] and Blumberg et al. [14] respectively implement clicker training on a robotic and a simulated dog. Blumberg et al.'s system is especially interesting, allowing the dog to learn multi-action sequences and associate them with verbal cues. While interesting as novel techniques of teaching pose sequences to their respective platforms, neither is evaluated using an explicit performance metric, and it remains unclear if and how these methods can be generalized to other, possibly more complex MDP settings.

Thomaz and Breazeal [6] interfaced a human trainer with a table-based, Q-learning agent in a relatively simple environment. Their agent seeks to maximize its discounted total reward, which for any time step is the sum of human reward and environmental reward. This approach is a form of what is called shaping in the reinforcement learning literature, which is adding a supplementary reward function to the environmental reward function. The TAMER framework, in contrast, does not rely on shaping or temporal difference updates. Additionally, while their system was in a simple environment, the TAMER framework is designed with complex tasks (and thus function approximation) specifically in mind.

In another example of mixing human reward with on-going reinforcement learning, Isbell et al. [8] enable a social software agent, Cobot, to learn to model human preferences in LambdaMOO. Cobot "uses reinforcement learning to proactively take action in this complex social environment, and adapts his behavior based on multiple sources of human reward." Like Thomaz and Brazeal, the agent doesn't explicitly learn to model the human reward function, but rather uses the human reward as a reward signal in a standard RL framework.

The TAMER system is distinguished from previous work on human-delivered reward in that it is designed to work in complex domains through function approximation. It also uniquely forms a model of human reinforcement and uses that for greedy action selection.

IV. Implemented Algorithm

In this section we describe the learning algorithm. From a high level, the algorithm uses supervised learning to model the human's reward function and then acts greedily according to this model. It should not be seen as a reinforcement learning (RL) algorithm as described by Sutton and Barto [7], though it may be possible to use it in conjunction with RL algorithms.

As currently implemented, our algorithm relies on two assumptions: 1) the task is deterministic, and 2) there is enough time between actions for a human trainer to provide feedback. We discuss how to relax them in Subsection V-C.

The learning algorithm consists of an overarching function that loops once per time step, RunAgent() (shown in Algorithm 1), and the two functions that are called by RunAgent(), UpdateRewardModel() and ChooseAction(). UpdateRewardModel(), Algorithm 2, updates its model of the human's reward pattern based on feedback on a previous action. ChooseAction(), Algorithm 3, chooses an action based on the current model of human reward. The core of the learning algorithm is a linear function approximator (i.e. a perceptron), used in line 6 of ChooseAction() and updated in line 6 of UpdateRewardModel(). Unlike most other learning algorithms, only one hand-tuned parameter is used, the update step-size parameter α.

For line 3 of ChooseAction(), we assume that the transition function is known. Agents that do not begin with such knowledge of the transition function can model it using a number of established techniques [7]. Since the model is only used to look ahead a single step, we expect the algorithm to be tolerant to small errors.

Like Abeel and Ng [2], who also work with MDP\Rs, in our implemented algorithm we assume the reward function can be reasonably modeled as a linear combination of state feature values and learnable weights. However, our more general algorithm, described in Section II, does not restrict the representation of the reward function.

Algorithm 1 RunAgent()

Require: $Input$: α

1: $t \leftarrow 0$
2: $\overrightarrow{w} \leftarrow \overrightarrow{0}$
3: $\overrightarrow{f_{t-2}} \leftarrow \overrightarrow{0}$
4: $\overrightarrow{f_{t-1}} \leftarrow \overrightarrow{0}$
5: $a \leftarrow ChooseAction(s_t, \overrightarrow{w})$
6: $takeAction(a)$
7: **while** $true$
8: $\quad t \leftarrow t + 1$
9: $\quad$ **if** $t \geq 2$
10: $\quad\quad r_{t-2} \leftarrow getHumanFeedback()$
11: $\quad\quad$ **if** $r_{t-2} \neq 0$
12: $\quad\quad\quad \overrightarrow{w} \leftarrow UpdateRewModel(r_{t-2}, \overrightarrow{f_{t-1}}, \overrightarrow{f_{t-2}}, \overrightarrow{w}, \alpha)$
13: $\quad a \leftarrow ChooseAction(s_t, \overrightarrow{w})$
14: $\quad takeAction(a)$
15: $\quad s_t \leftarrow getState()$
16: $\quad f_{t-2} \leftarrow f_{t-1}$
17: $\quad f_{t-1} \leftarrow getFeatureVec(s_t)$

Algorithm 2 UpdateRewardModel()

Require: $Input$: $r_{t-2}, \overrightarrow{f_{t-2}}, \overrightarrow{f_{t-1}}, \overrightarrow{w}, \alpha)$

1: Set α as a parameter.
2: $\overrightarrow{\Delta f_{t-1,t-2}} \leftarrow \overrightarrow{f_{t-1}} - \overrightarrow{f_{t-2}}$
3: $projectedRew_{t-2} \leftarrow \sum_i (w_i \times \Delta f_{t-1,t-2,i})$
4: $error \leftarrow r_{t-2} - projectedRew_{t-2}$
5: **for** i in $range(0, length(\overrightarrow{w}))$
6: $\quad w_i \leftarrow w_i + \alpha \times error \times \Delta f_{t-1,t-2,i})$
7: **return** $\overrightarrow{w}$

Algorithm 3 ChooseAction()

Require: $Input$: $s_t, \overrightarrow{w}$

1: $\overrightarrow{f_t} \leftarrow getFeatureVec(s_t)$
2: **for** each $a \in getActions(s_t)$
3: $\quad s_{t+1,a} \leftarrow T(s_t, a)$
4: $\quad \overrightarrow{f_{t+1,a}} \leftarrow getFeatureVec(s_{t+1,a})$
5: $\quad \overrightarrow{\Delta f_{t+1,t}} \leftarrow \overrightarrow{f_{t+1,a}} - \overrightarrow{f_t}$
6: $\quad projectedRew_a \leftarrow \sum_i (w_i \times \Delta f_{t+1,t,i})$
7: **return** $argmax_a(projectedRew_a)$

RunAgent() begins by initializing the time t, the weights $\overrightarrow{w}$ for the reward model, and feature vectors $\overrightarrow{f_{t-2}}$ and $\overrightarrow{f_{t-1}}$. It then calls ChooseAction() and takes its first action. At that point, RunAgent() begins an endless loop that iterates once per time step. Like many agents within MDPs, our agent receives a full state description s_t and a reward signal r_{t-2} each time step and chooses an action a. However, its reward signal comes exclusively from a human trainer, not from the task enviroment.

The action chosen at time $t-2$ is observed by the human as it is enacted between time $t-2$ and time $t-1$, and then the human can give feedback between time $t-1$ and time t, resulting in that action's feedback only being processed two turns later. Note that this one-step time delay can result in a single repitition of an undesired move before the trainer can negatively reinforce the first instance of the move. A human reward signal of 0 is not considered feedback, since the human could have left the agent to repeatedly perform its learned policy. In the case of zero reward, the action is considered an unlabeled sample and UpdateRewardModel() is not called, as line 11 of Algorithm 1 shows. Before the loop ends, ChooseAction() is called and the chosen action is taken.

A distinction of this algorithm is the it represents the action in terms of how it will change the state. To represent an action at time t, a vector of what we will call delta-features $\overrightarrow{\Delta f_{t+1,t}}$ are calculated by subtracting state feature vector $\overrightarrow{f_{t+1}}$ from $\overrightarrow{f_t}$. These delta-features are the only action representation used. Using the change in state as the features for each training sample, the linear function approximator can richly capture the effects of a continuous or discrete action, which, it might be argued, are equivalent to the action itself.

UpdateRewardModel(), Algorithm 2, uses gradient descent to adjust the weights of the linear funtion approximator. The inputs are the human reward amount, r_{t-2}, state feature vectors $\overrightarrow{f_{t-2}}$ and $\overrightarrow{f_{t-1}}$ (which represent the action as its effect on the state from s_{t-2} to s_{t-1}), the current weight vector $\overrightarrow{w}$, and the step-size parameter αThe $error$ is then calculated as the difference between the projected reward (from inputting $\overrightarrow{\Delta f_{t-1,t-2}}$ into the model) and the given reward for the action chosen at time $t-2$. For each delta-feature $\Delta f_{t-1,t-2,i}$, the new weight is the product of the step-size parameter α, the $error$, and the delta-feature. In other words, the weights are moved towards those that would output the reward that the human gave.

In the function ChooseAction(), the agent evaluates the effects of each potential action and chooses the one it deems most valuable. The inputs for a call of ChooseAction() at time t are the state value s_t and the weights $\overrightarrow{w}$. For each potential action a in the set of legal actions, the agent looks ahead one step, using the transition function T, and determines $\overrightarrow{f_{t+1}}$ for the next state (at time $t+1$). The delta-features $\overrightarrow{\Delta f_{t+1,t}}$ are calculated and input into the model to predict the reward that each action would receive from the human trainer. The action with the largest expected reward is chosen. If more than one action maximizes the projected reward, one of the maximizing actions is chosen randomly.

V. Experimental Evaluation

In this section we describe Tetris, the domain in which we have tested our algorithm, and the Tetris-specific aspects of our algorithm. We then discuss how our results compare to previous results.

A. The Experimental Domain: Tetris

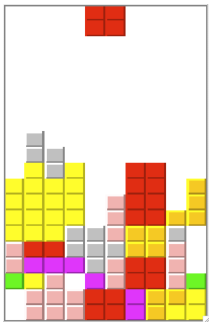

Fig. 3. A screenshot of RL-Library Tetris.

Tetris is a game played on a $w \times h$ grid in which "tetrominoes," different shapes of four blocks, fall one at a time from the top of the grid and stack upon the grid's base or any blocks below. If the blocks fall such that there is a row completely filled with blocks, then that line (i.e. row) is "cleared" – all of the blocks in that row disappear and all of the blocks in higher rows shift down a row. When the blocks stack up beyond the top of the grid, the game ends. The goal of a Tetris player is to maneuver the falling blocks in such a way as to clear as

TABLE I
RESULTS OF VARIOUS TETRIS AGENTS.

Method	Mean Lines Cleared		Training Games
	at Game 3	at Peak	for Peak
TAMER	**65.89**	**65.89**	**3**
RRL-KBR [17]	5	50	120
Policy Iteration [15]	0 (random until game 100)	3183	1500
Genetic Algorithm [18]	0 (random until game 500)	586,103	3000
CE+RL, Decr. Noise [16]	0 (random until game 100)	348,895	5000

many lines as possible before the game ends. Since clearing a line moves blocks down, away from the top, clearing lines allows a player to play longer, likely clearing more lines. Thus we measure success in Tetris by the number of lines per game.

Most Tetris implementations, including the one we use, are Markov Decision Processes. The particular implementation of Tetris used within this project is RL-Library Tetris[1]. In the version played by humans, the player's choices are to move the tetromino right or left, to rotate it clockwise or counterclockwise, or to not move it all. Also, the game forces a downward movement at each time step that is performed after the player's action. After a tetromino is placed, a new one is randomly drawn from an uniform distribution of the seven possible tetromino types and begins to fall from the top of the grid. Unlike human players, all known Tetris-learning algorithms, including ours, instead choose from among legal tetromino placements. In other words, the agent does not make independent decisions to move the tetromino left, right, etc., but rather chooses its final location from a list given by a subroutine.

Even with the aid of such a list, Tetris remains a complex and interesting challenge. Tetris is highly stochastic – RL-Library Tetris draws tetromino types randomly from a uniform distribution. Tetris also has an immense state space. For a board size of $m = w \times h$, it is greater than 2^m.

B. Tetris-Specific Aspects of the Algorithm

We use 22 features to describe the Tetris state space of a 10×20 board. Ten of these are the ten column heights. One is the height of the tallest column. Nine are the absolute values of the height differences between adjacent columns. One is the number of holes, defined as empty grid cells with at least one block above in the same column. These state features are taken from [15], who also used them with a linear function approximator. A linear combination of these features has been used successfully by a number of other researchers since and described in papers including [16] (results shown in Table I).

Recall that Algorithm 3, ChooseAction(), requires a deterministic transition function. In Tetris, though, new tetrominoes are determined stochastically. Therefore, in our Tetris-specific algorithm, the transition function is replaced with a function that returns the deterministic afterstate, which is the board configuration once the tetromino has been placed and before the next tetromino appears. It does not allow the agent to know what the next tetromino will be.

C. Discussion of algorithm and results

Using learning rate as a metric for comparison, our learning algorithm outperforms previous autonomous learners by more than an order of magnitude. Figure 4 shows results from a group of 5 trainers who are well-versed in artificial intelligence, 4 trainers who have no computer science background, and the complete group of 9 trainers. The trainers were given limited instructions[2] and were not told anything about the agent's features or learning method other what was required to use the interface. Separating the trainers into two groups was done merely to show that the training method is accessible to non-technical users. An investigation of group differences is beyond the scope of this paper and would probably require significantly more data and formal controls on subjects' background to yield conclusive results.

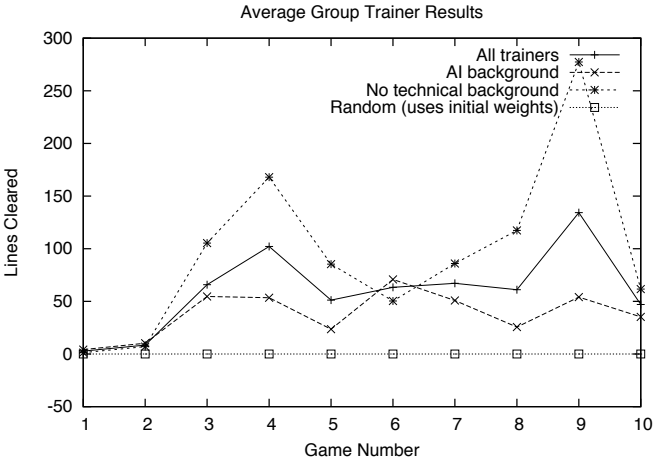

Fig. 4. The average amount of lines cleared per game by experimental group: all subjects, subjects who have an AI background, subjects who have no computer science background, and the untrained agent using its initial weights, which cause it to choose randomly.

By the third game, on average, performance reached an approximate peak of 65.88 lines cleared per game. Compared to autonomous agents, this is incredibly fast. Ramon et al. RRL agent reached somewhat lower performance after 120 games. Others trained for a hundred or more games (see Table I) before even changing from their initial policy. We should note that Ramon et. al. rejected a form of their algorithm that reached about 42 lines cleared on the third game. They deemed it unsatisfactory because it unlearned by the fifth game and never improved again, eventually performing worse than randomly. Ramon et al.'s agent is the only one we found that approaches the performance of our system after 3 games.

Most trainers quit giving feedback by the end of the fifth game, stating that they did not think they could train the agent to play any better. Therefore most agents are operating with a static policy by the sixth game. Score variations come from the stochasticity inherit in Tetris, including the highest scoring game of all trainers (809 lines cleared), which noticeably brings the average score of game 9 above that of the other games. Figure 5 shows, in rough terms, how much feedback was given each game. Specifically, each data point is the sum of the absolute value of reward given for the corresponding game, averaged across all trainers. It shows that the amount of feedback per game increases with the lengths of the games until the third game. After this, the amount of feedback plummets, despite the games lasting approximately as long.

[1] code.google.com/p/rl-library/

[2] Exact instructions can be seen at www.cs.utexas.edu/∼bradknox/icdl08.html

The peak performance of our Tetris agent, however, is significantly lower than other top learning agents. Since it uses the same function approximator as some of the higher performing algorithms, it seems that it should also perform equally well. From our informal analysis, it seems that the low peak performance is likely a limitation of applying incremental updates with a constant step size α to the function approximator. Tetris has repeatedly frustrated attempts to train a classical Q-learning or SARSA agent (e.g. [15] and from our experience), which also use incremental updates. It also might be that the features and linear function approximator are expressive enough to describe a policy that can clear hundreds of thousands of lines per game, but it is not quite expressive enough to model a value function (as in SARSA) or a reward function like we do.

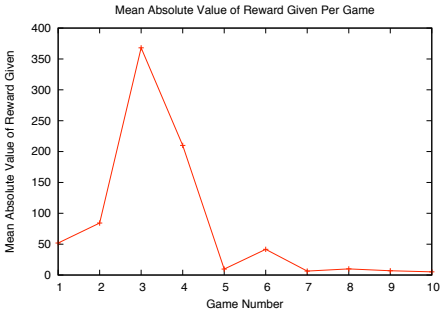

Fig. 5. The average amount of reward given per game (absolute values of feedback were used for the calculation).

Despite the low (but precedented) peak performance, the agent's learning speed represents a significant step forward. For agents that incur a real cost (material wear, endangerment, etc.) from time spent training or failures at the task, using our algorithm is a vastly more appealing option than learning autonomously.

As mentioned in Section IV, for our training algorithm to work as implemented, the task environment must satisfy two assumptions. For each assumption, we describe a straightforward extension that allows the assumption to be relaxed.

First, the algorithm assumes that the task is deterministic (or in the Tetris case, that there are deterministic afterstates). For a non-deterministic enviroment, the transition function would return a set that holds the probabilities of transitioning to each possible state. To extend our algorithm to handle this, when f_{t+1} is calculated for any action, it is the weighted sum of the features of all possible states.

Second, this algorithm also requires that there is enough time between actions for a human to give feedback on the previous action. Many common MDP tasks, including mountain car, pole balancing, and acrobat, violate this assumption. For tasks with more frequent actions, our algorithm could be generalized by using eligibility traces [7], applying the feedback to all previous actions under a factor that exponentially decays the effect of the feedback as actions are farther back in time from the feedback signal.

VI. Conclusion and Future Work

We have presented a general framework, called Training an Agent Manually via Evaluative Reinforcement (TAMER), and a fully implemented algorithm that allows a human to train a learning agent to perform a task via a scalar reward signal. The algorithm was applied to the complex domain of Tetris, and empirical results show that it is effective in the hands of people with and without technical backgrounds, increasing learning speed by more than an order of magnitude. The algorithm has a simple interface and is relatively easy to implement, having only one parameter (α).

There are numerous directions to proceed from the work reported in this paper. In future work we intend to apply this to more varied tasks to test extensions including having the agent learn the transition function, using a weighted sum of next state features in a nondeterministic environment, and using eligibility traces to apply feedback over more all previous actions. We are also investigating how to combine the human training framework with an autonomous learner to allow the agent to learn both with the human's feedback and in its absense.

Acknowledgments

This research is supported in part by NSF CAREER award IIS-0237699 and the DARPA IPTO Bootstrap Learning program.

References

[1] G. Tesauro, "TD-Gammon, a self-teaching backgammon program, achieves master-level play," *Neural Computation*, vol. 6, no. 2, pp. 215–219, 1994.

[2] P. Abbeel and A. Ng, "Apprenticeship learning via inverse reinforcement learning," *ACM International Conference Proceeding Series*, 2004.

[3] P. Stone, R. S. Sutton, and G. Kuhlmann, "Reinforcement learning for RoboCup-soccer keepaway," *Adaptive Behavior*, vol. 13, no. 3, pp. 165–188, 2005.

[4] N. Kohl and P. Stone, "Machine learning for fast quadrupedal locomotion," in *The Nineteenth National Conference on Artificial Intelligence*, July 2004, pp. 611–616.

[5] R. Maclin and J. W. Shavlik, "Creating advice-taking reinforcement learners," *Machine Learning*, vol. 22, pp. 251–282, 1996.

[6] A. Thomaz and C. Breazeal, "Reinforcement Learning with Human Teachers: Evidence of Feedback and Guidance with Implications for Learning Performance," *AAAI-2006*.

[7] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[8] C. L. Isbell, Jr., M. Kearns, S. Singh, C. Shelton, P. Stone, and D. Kormann, "Cobot in LambdaMOO: An adaptive social statistics agent," *Autonomous Agents and Multiagent Systems*, vol. 13, no. 3, November 2006.

[9] M. Bouton, *Learning and Behavior: A Contemporary Synthesis*. Sinauer Associates, 2007.

[10] D. Moreno, C. Regueiro, R. Iglesias, and S. Barro, "Using prior knowledge to improve reinforcement learning in mobile robotics," *Proc. Towards Autonomous Robotics Systems. Univ. of Essex, UK*, 2004.

[11] G. Kuhlmann, P. Stone, R. Mooney, and J. Shavlik, "Guiding a reinforcement learner with natural language advice: Initial results in RoboCup soccer," in *The AAAI-2004 Workshop on Supervisory Control of Learning and Adaptive Systems*, July 2004.

[12] S. Schaal, "Is imitation learning the route to humanoid robots?" *Trends in Cognitive Sciences*, vol. 3, no. 6, 1999.

[13] F. Kaplan, P. Oudeyer, E. Kubinyi, and A. Miklósi, "Robotic clicker training," *Robotics and Autonomous Systems*, vol. 38, no. 3-4, 2002.

[14] B. Blumberg, M. Downie, Y. Ivanov, M. Berlin, M. Johnson, and B. Tomlinson, "Integrated learning for interactive synthetic characters," *Proc. of the 29th annual conference on Computer graphics and interactive techniques*, 2002.

[15] D. Bertsekas and J. Tsitsiklis, *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[16] I. Szita and A. Lorincz, "Learning Tetris Using the Noisy Cross-Entropy Method," *Neural Computation*, vol. 18, no. 12, 2006.

[17] J. Ramon and K. Driessens, "On the numeric stability of gaussian processes regression for relational reinforcement learning," *ICML-2004 Workshop on Relational Reinforcement Learning*, pp. 10–14, 2004.

[18] N. Bohm, G. Kokai, and S. Mandl, "Evolving a heuristic function for the game of Tetris," *Proc. Lernen, Wissensentdeckung und Adaptivitat LWA*, 2004.