JMLR: Workshop and Conference Proceedings 39:299–314, 2014 ACML 2014

Reinforcement Learning with Value Advice

Mayank Daswani MAYANK.DASWANI@ANU.EDU.AU
Peter Sunehag PETER.SUNEHAG@ANU.EDU.AU
Marcus Hutter MARCUS.HUTTER@ANU.EDU.AU
Research School of Computer Science
Australian National University,
Canberra, ACT-2601, Australia.

Editor: Dinh Phung and Hang Li

Abstract

The problem we consider in this paper is reinforcement learning with value advice. In this setting, the agent is given limited access to an oracle that can tell it the expected return (value) of any state-action pair with respect to the optimal policy. The agent must use this value to learn an explicit policy that performs well in the environment. We provide an algorithm called RLAdvice, based on the imitation learning algorithm DAgger. We illustrate the effectiveness of this method in the Arcade Learning Environment on three different games, using value estimates from UCT as advice.

1. Introduction

Reinforcement learning (RL) agents (Sutton and Barto, 1998) learn how to act well via trial-and-error interactions with an environment that provides a real-valued reward signal. An RL agent has less information than in supervised learning, since the reward signal provides only partial feedback. Additionally, the agent needs to make choices about which parts of the environment to explore, leading to the famous exploration-exploitation problem. There has been research into reducing the difficulty of the reinforcement learning problem, particularly for large environments, by providing additional information to the agent in various forms. The related fields of imitation learning, learning from demonstration, reinforcement learning with policy advice, inverse reinforcement learning and transfer learning all fall into this category.

This paper is an attempt to answer the following problem. We are given some class of function approximators $\mathcal{Q}$ of our value function, and an oracle that provides the expected return of any state-action pair under the optimal policy. We assume that we cannot always use the oracle because of some constraints, for example, the oracle costs a lot of computation time/memory or we lose access to the oracle at some point e.g. separate training/testing stages. Thus we want to use the oracle information to find a policy defined by $\hat{Q} \in \mathcal{Q}$ that performs well in the environment.

Motivation. The primary motivation of this approach comes from the need to extract explicit policies from anytime algorithms such as UCT (Kocsis and Szepesvári, 2006) represented by a value function approximation in some feature space. The reasons for this need are two-fold. Consider a problem that has two stages, training and testing. In the training

© 2014 M. Daswani, P. Sunehag & M. Hutter.

stage, the algorithm may use UCT on the simulator for control, however in the testing stage there may not be the computing resources or the time to run UCT. In such cases it is useful to extract a reactive policy from UCT that will perform well without need for further simulation. While UCT does not use features of the environment, an explicit policy will rely on such features. A second use of an explicit reactive policy is to judge the usefulness of a particular class of function approximators in representing a good (or optimal) policy for a problem. If the learned policy extracted from UCT by our approach performs well, it indicates that the approximators being used are capable of representing such a policy. Thus, this method also provides a tool for evaluating classes of function approximators, although there is no guarantee that it finds the best performing policy in the class.

A related problem is to imitate the oracle policy as closely as possible. Focusing on this leads to a solution to our original problem. For example, we may simply treat it as a regression problem with training samples being the features and the oracle return following the oracle's policy. Unfortunately, the regression model from this dataset results in a policy that does not necessarily perform well. The intuition behind this failure is that the class of function approximators cannot represent the value function of the oracle's policy, and the agent is learning according to the oracle's state distribution rather than its own. In imitation learning, an algorithm known as Dataset Aggregation (DAgger) (Ross and Bagnell, 2010) deals with this problem, while still retaining the goal of imitating the oracle. Our proposed solution to the original problem is a modification of DAgger to suit our setting. In practice, the oracle may not be perfect, and one might want to continue learning after the training phase is over.

Main contribution. The main contribution of this work is learning how to act well in a reinforcement learning problem given access to an oracle that can provide the value of any state-action pair in a training stage. We provide an algorithmic contribution in the form of a modification of an existing imitation learning algorithm for this task, along with a comparison of various methods. Our testing suite is the Arcade Learning Environment which shows the scalability of our method. Using just a few episodes of data, we obtain much better results than SARSA after 5000 episodes, although the settings are not directly comparable.

This paper is organised as follows. Section 2 explains the background necessary for the rest of the paper. Section 3 provides information on related work, primarily imitation learning which while close in some senses, has a different objective. In Section 4 we formally introduce our objective and our algorithmic solution. Section 5 describes the experimental setup, methodology, and displays our results. In Section 6 we discuss the results. We conclude in Section 7 and talk about some possible future work.

2. Background

Agent-Environment Framework. An agent acts in an Environment Env by choosing from actions $a \in \mathcal{A}$. It receives observations $o \in \mathcal{O}$ and real-valued rewards $r \in \mathcal{R}$ where $\mathcal{A}, \mathcal{O}$ and $\mathcal{R}$ are all finite. This observation-reward-action sequence happens in cycles indexed by $t = 0, 1, 2, ...$ The history of an agent at time t is h_t which contains the sequence of observation-reward-action tuples up to time t.

Markov Decision Process (MDP). If $Pr(o_t r_t | h_t, a_t) = Pr(o_t r_t | o_{t-1} a_t)$, the environment is said to be a discrete MDP (Puterman, 1994). In this case, the observations form the state space of the MDP. Formally an MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \gamma, R \rangle$ where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions and $R : \mathcal{S} \times \mathcal{A} \rightsquigarrow \mathcal{R}$ is the (possibly stochastic) reward function which gives the (real-valued) reward gained by the agent after taking action a in state s. $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the *state-transition function*. The agent's goal is to maximise its expected future discounted reward, where a geometric discount function with rate γ is used. In an episodic setting, the discounted sum is truncated at the end of an episode. The value of a state-action pair according to a policy (π) is given by $Q^\pi(s, a) = E_\pi\{R_t | s_t = s, a_t = a\}$ where $R_t = \sum_{k=0}^{t_{end}} \gamma^k r_{t+k+1}$ is the *return* and t_{end} indicates the end of the episode containing time t. In the non-episodic setting $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$. We want to find the optimal action-value function Q^* such that $Q^*(s, a) = \max_\pi Q^\pi(s, a)$, since then the greedy policy with respect to Q^* is optimal, $\pi^*(s) = \arg\max_a Q^*(s, a)$.

Reinforcement learning algorithms. RL agents come in several flavours. One of the biggest distinctions is between model-based and model-free agents. Model-based algorithms explicitly learn a transition function and reward function, whereas model-free algorithms learn the value function directly. This paper focuses on a model-free algorithm which uses linear function approximation to scale to large state spaces. Commonly used model-free methods include SARSA and Q-learning which are both *temporal difference* methods. SARSA is an on-policy algorithm that uses the update rule $Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha\delta$ where $\delta = r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) - Q^\pi(s_t, a_t)$ is called the temporal difference and α is known as the learning rate. The *tabular setting* describes the case where we represent the value function by a table with entries corresponding to the value of each state-action pair. However, when the state space is very large we need to represent the value function more compactly via a suitable class of parameterized function approximators. A linear function approximator over d-dimensional features $\phi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^d$ approximates the value $Q(s, a)$ by $w^\top \phi(s, a)$ where $w \in \mathbb{R}^d$ is a weight vector we learn. The SARSA update rule is then $w_i \leftarrow w_i + \alpha\delta\phi(s_t, a_t)$ for all $w_i \in w$.

UCT. Upper Confidence Bounds for Trees (UCT) by Kocsis and Szepesvári (2006) is a Monte-Carlo Tree Search (MCTS) algorithm that uses UCB from the bandit setting for exploration in the forward model setting. MCTS algorithms are expectimax searches using a forward model with some heuristic for selecting actions to avoid expanding the full tree. They run to some depth either determined by time or as a fixed constant, after which they play out a predefined (often random) policy to estimate the remainder of the return. UCT is used by us in the form of an oracle. We provide pseudo-code for it in Algorithm 1. Later on, we also provide a modification to UCT in which it behaves as a reinforcement learning algorithm in a deterministic environment, with the same great performance but terrible computational efficiency.

Arcade Learning Environment (ALE). The reinforcement learning community has lacked a set of general environments that can be used for testing new algorithms in a robust manner. In Veness et al. (2011) a set of small challenging problems were provided, but several algorithms (Daswani et al., 2013; Nguyen et al., 2012) can no longer be differentiated based on them. The recently introduced ALE by Bellemare et al. (2013) attempts to address this big gap in the field by utilising games made for the ATARI 2600 as a test bed for

reinforcement learning algorithms. The environments in this setting are games made for humans which can be relatively complex, but due to the space/processing limits of the ATARI console, still computationally feasible for current RL techniques. The ALE consists of an interface to Stella which is an open-source Atari 2600 games emulator.

This gives access to hundreds of games of this format, which range from side-scrollers, to arcade games, shooters and puzzles. The interface provides access to the screen pixel matrix and the internal state representation of the ATARI games themselves. This allows for both reinforcement learning and planning algorithms to be tested, since the ability to reset to a particular state is crucial for some planning algorithms like UCT.

3. Related Work

The related problem of imitating the oracle with full-information is briefly considered by Ross (2013) as the full-information cost-to-go setting. However, they only look at the partial information cost-to-go setting in detail. In the full information tabular setting, there is no need for exploration since we know that the policy can be perfectly represented, and we simply have a regression problem. However in our setting, due to the large state space, we need function approximation which may mean that the value function of the policy cannot be fully represented in the function approximation class. This reintroduces the exploration problem, but in a different way. We have access to the oracle's value function at any instant, but this function may not be in the approximation class, so we need to explore in this space and use the oracle's value function as an optimistic guide.

Imitation learning. Traditionally an imitation learning problem is framed as a loss minimisation problem, rather than a reward maximisation problem. In the standard setting we have an expert which provides the correct action $a_s^* = \arg\max_{a \in A} Q^*(s,a)$. The learner has no access to the return, and simply sees the action the oracle prescribes. The task is now normally framed as minimising a *surrogate loss* based on the oracle's policy, where the surrogate loss is defined as a distance between the each action and the optimal one with the simplest being a 0-1 loss.

In the partial information setting, the agent has some access to the expected return but not for every action. Ross (2013) considers the case where the learner can have cost information about only one of the actions by playing out a (agent or oracle) policy after choosing an action. The decision of which action to learn about is made uniformly, but this is suboptimal as they point out.

In the full information setting, where the agent has access to the value of every action in a given state, it aims to imitate the oracle by choosing a policy $\arg\max_\pi \sum_{t=1}^m Q(s_t, \pi(s_t))$, where we have m samples of the form $\{s_t, Q(s_t, \cdot)\}$ and $Q_t(s_t, \cdot)$ is a vector of expected returns for each action provided by an oracle (Ross, 2013). This above ideal policy imitates the oracle within the approximation class. In practise, solving this problem for interesting policy classes is computationally hard, and one approximates it by reducing it to a different, preferably convex, optimisation problem.

Previous work on reinforcement learning with advice. There has been interest in various formulations of the reinforcement learning with advice problem. Maclin and Shavlik (1996) define a learner that can accept advice in the form of instructions in a

simple imperative programming language. Wiewiora et al. (2003) define potential-based advice which uses shaping functions defined over states and actions to give the agent hints about whether a state-action pair is good or bad. Maclin et al. (2005) construct agents that can accept advice in the form of bounds on the Q-value in certain parts of the state-action space. Azar et al. (2013) look at regret bounds for the case where the agent is given advice in the form of some set of (hopefully good) input policies. Most recently, Taylor et al. (2014) define a teacher-student framework where both teacher and student are reinforcement learning agents, and the teacher must choose when to give advice in the form of recommended actions to the student. The advice is assumed to be budgeted. This work is of interest to us, since the experiments are also performed on video games (Starcraft and Pacman). However, none of these various advice settings address our particular problem.

Previous work on the ALE. Also of interest to us is previous work on the ALE. The initial paper by Bellemare et al. (2013) extensively described the performance of a vanilla SARSA implementation using a few classes of function approximators based features on the pixel matrix and linear function approximation. These agents do not perform so well. However, later papers use other function approximators that perform much better. For example, Hausknecht et al. (2013) use neuro-evolutionary techniques and Mnih et al. (2013) use convolution neural nets in a deep learning style, to learn features of the matrix. There has also been work in the model-based setting by Bellemare et al. (2013) but these agents have so far only been used for prediction rather than control, since it is still quite computationally difficult to find a good policy using a model in such a large space. In our work, we focus on trying to improve the performance on a particular feature set (BASS) using only linear function approximation.

4. Algorithm

Ultimately we want the best policy within our approximation space. While this may not be achievable in practice, it gives us a goal to aim for. We have d-dimensional features $\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$ over states and actions. Let us define a set of policies parameterised by a weight vector $v \in \mathbb{R}^d$ as $\pi_v(s) = \arg\max_a v^\top \phi(s,a)$. Then the best policy within our approximation space is given by

$$\pi_v^* = \arg\max_{\pi_v} Q^{\pi_v}(s, \pi_v(s)) \tag{1}$$

In practise, we instead look for the best approximation to the optimal value function and take the greedy policy with respect to that function, i.e.

$$w = \arg\min_v \sum_{s,a} d(s,a) |v^\top \phi(s,a) - Q^*(s,a)|^2 \tag{2}$$

where $d : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a distribution over the state-action pairs induced by the behaviour policy. We then choose $\hat{\pi}^*(s) = \arg\max_a w^\top \phi(s,a)$.

The algorithm (RLAdvice, Algorithm 2) we define is a modification of the Dataset Aggregation algorithm (DAgger) (Ross and Bagnell, 2010) used in imitation learning in various forms. RLAdvice retains the core idea from DAgger of aggregating samples derived from the agent's own policies. However, it collects different information and uses a different

Algorithm 1: UCT (Kocsis and Szepesvári, 2006), pseudocode adapted from Bellemare et al. (2013)

```
Input: search horizon m, simulations per step k, Environment Env with reset to state
       ability.
Input: search tree Ψ, current state s.

search(s)
if Ψ is ∅ or root(Ψ) ≠ s then
    Ψ ← empty search tree.
    Ψ.root ← s.
end
repeat
    sample (Ψ, m)
until Ψ.root.visits = k;
a ← bestAction (Ψ).
prune (Ψ, a).
return (a)

Function sample(Ψ, m)
    n ← Ψ.root.
    while n is not a leaf, m >depth(n) do
        if action a has not yet been taken in node n then
            reward ← emulate (n,a).
            Create child node c_a of n.
            immediate-return(c_a) ←reward.
            Change the current node to c_a, i.e. n ← c_a.
        end
        else
            a ← selectAction (n).
            n ← child(n,a).
        end
    end

Function emulate(n, a)
    Input: Node n containing environment state, Action a
    Env.resetToState(n.state).
    Execute action a in Env and store reward in r.
    if End of game then
        Set node to leaf node.
    end
    return r
```

objective function. In our setting, we use $\phi(s_t, a) := \phi(s_t)$ and learn a separate weight vector for each a. We add tuples of the form $(\phi(s_t), Q(s_t, a))$ to the dataset D_a for all a and for each timestep t in the episode i, where $Q(s_t, a)$ is provided by the oracle. In the first episode, the agent follows the policy provided by the oracle, i.e. $\arg\max_a Q(s, a)$. For every following episode, the agent acts based on its own prediction which it obtains from the current (action) value function approximation ${w_i^a}^\top \phi(s)$, and adds the corresponding tuples to D_a. The intuition behind using the agent's own predicted policy over the oracle's is that the oracle policy might not be representable in the agent's approximation space and the agent might make mistakes that the oracle never makes. Due to the representation issue, the agent might fail to learn not to end up in certain situations, which the oracle would not visit, but knows how to act in. In order to learn about these situations, we use the agent's policy. Once an episode is over, the agent learns a new set of regression weights (w_i^a) for each dataset D_a, using the following regularised ϵ-insensitive objective optimised with LIBLINEAR (Fan et al., 2008),

$$w_i^a = \arg\min_v \left(\frac{1}{2} v^\top v + C \sum_{(\phi(s), Q(s,a)) \in D_a} \max\{0, |v^\top \phi(s) - Q(s,a)| - \epsilon\}^2 \right) \tag{3}$$

LIBLINEAR solves this as a support vector regression problem. ϵ specifies the accuracy to which we minimise the loss for each data point and C is the regulariser. For the next episode $i+1$, the agent follows the policy $\pi_i(s) = \arg\max_a {w_i^a}^\top \phi(s)$.

Algorithm 2: Reinforcement learning with value advice

```
Initialise D ← ∅.
Initialise π_1 (= π*).
t = 0
for i = 1 to N do
    while not end of episode do
        foreach action a do
            Obtain feature φ(s_t) and oracle expected return Q*(s_t, a).
            Add training sample {φ(s_t), Q*(s_t, a)} to D_a.
        end
        Act according to π_i.
    end
    foreach action a do
        Learn new model Q̂_i^a := w_i^a⊤ φ from D_a using regression.
    end
    π_i(·) = argmax_a Q̂_i^a(·) .
end
```

We believe the algorithm satisfies similar regret bounds to DAgger. The primary difference between RLAdvice and DAgger is that we generate a sequence of weights, for each action, over our function approximation class, whereas DAgger directly generates a sequence of policies. We can perform a similar analysis as by Ross and Bagnell (2010), viewing the

Table 1: Experimental Parameters

Setting	Parameter	Value
ALE	Environment Distribution	False
	Frame skip	5
UCT	Exploration constant	0
	Number of simulations	36
	Horizon	50
BASS features	Grid width	16
	Grid height	14
	Colours	8
RLAdvice	Pong Regulariser	0.1
	Space Invaders Regulariser	1.0
	Atlantis Regulariser	1.0
SARSA-RLA	Discount factor	0.999
	Learning rate	0.5
	Exploration constant	0.05
RLAdvice-SDCA	Pong Regulariser	10
	Pong stopping gap	0.0001
	Space Invaders Regulariser	1.0
	Space Invaders stopping gap	10
	Atlantis Regulariser	100
	Atlantis stopping gap	5000

data collected from each iteration of RLAdvice along with the corresponding loss over our parameters as a single instance of an online learning problem. For $\epsilon = 0$, the loss we use is strongly convex, so the no-regret bounds for follow-the-leader apply. Note that such bounds include a term which is the minimum loss we can achieve in our function approximation class. The bound does not mean that we converge, the algorithm could conceivably oscillate between approximations. If the class of approximators can approximate the optimal value function well, then the oscillations will be between good approximators and therefore not a problem. A problem occurs only if the class of approximators is weak.

5. Experiments

We have attempted to test various configurations of the algorithm to show what works and what does not. Firstly, we show that using the oracle's own trajectory does not work. Our modification to DAgger using the oracle that provides the return works very well, providing the best known results for Pong and Atlantis using this class of function approximators. This result demonstrates that the class is capable of representing a good Pong playing policy. On the other hand, we see no improvement over SARSA in the Space Invader results, which might indicate that the problem here is the feature representation. We can thus see the diagnostic use of this approach.

The Oracle. As pointed out previously, our primary motivation is in extracting efficient reactive policies from slow MCTS algorithms such as UCT, which can be computationally expensive to use outside of a training phase. The oracle that we use is therefore UCT with a specified horizon and number of simulations. As an aside, note that UCT itself can be viewed as a reinforcement learning algorithm in a deterministic environment in the following sense. It is possible to mimic the ability to reset to a particular state, by saving in each node, the action sequence that led to that node. Then in order to evaluate a child of a particular node, the agent simply has to reset to the start of the game either by completing the current episode or by using an *end game* action, and then play out the saved actions, followed by the action of the child it wishes to evaluate. We provide this replaced emulate function in Algorithm 3. Given that the ALE is a deterministic environment, the impressive UCT results of Bellemare et al. (2013) can be said to be reinforcement learning results (albeit with a much higher number of trajectories used) rather than planning results and this brings into critical light the weak SARSA results provided in that paper. It is important to remember that even in this "reinforcement learning" mode, UCT still does not learn either an explicit or complete policy, but just an action sequence. By testing our algorithm on the ALE with UCT as our oracle in similar configurations to that used in Bellemare et al. (2013) we are also trying to discover whether the fault lies with SARSA or with the function approximation class.

Algorithm 3: The modified **emulate** function for UCT as an RL agent

Input: MCTS Node containing action sequence $a_{1:l-1}$, Environment Env, Action a
Output: Reward for the execution of action a, after sequence $a_{1:l-1}$

```
// The following loop can be replace by a call to reset_game() if
   available.
```

while *not end of game* **do**
    Execute random action in Env.
end
Start new episode.
Execute all actions in Node action sequence ($a_{1:l-1}$).
Execute latest action a in Env and store reward in r.
if *End of game* **then**
    Set node to leaf node.
end
return r

Features. We use the feature class described in Bellemare et al. (2013) as Basic Abstraction of Screen Shots (BASS). It consists of a tiling of the screen into blocks, with each block containing indicator functions for each SECAM colour i.e. the function is on if a particular colour was present in the block. BASS consists of these features along with the pairwise AND of all those features. In the default setting of a 16x14 grid this results in a feature space that contains 1,606,528 features.

Games. The legal action set for a game in the ALE contains all 18 actions that could physically be pressed on an ATARI2600 controller. The minimal action set contains only

the actions that are needed for a particular game. For computational reasons, we selected games that have a minimal action set with size less than the number of legal actions (18). We also wanted games that showed better than random performance on SARSA (on the BASS feature set), as some indication of a linear function approximator being successful. Another criterion was that games did not have to last the full 18,000 frames to end, since that would increase both computation time and memory storage. Given those constraints, we chose Pong and Atlantis. We also chose one game, Space Invaders, where picking a constant action with some ϵ-random actions performed better than SARSA. There is some indication here that the function approximator cannot represent the value function, and we attempt to confirm that with our experiments.

Pong is a game with two paddles and a ball, a 2D version of table tennis. The aim is to hit the ball back such that the opponent cannot reply. Our agent plays against the hardcoded ATARI 2600 agent, which is pretty hard to beat even for a human. If the agent scores it receives 1 point, and -1 point when the opponent scores. Thus the total score is the difference between the agent's score and the opponent's score. The minimal action set contains 6 actions.

Space Invaders involves shooting down columns of alien spaceships, while avoiding their return fire. The enemy spaceships are arranged in columns, with lower rows worth lesser points. The columns move from left to right and then back, with each movement to the end advancing the spaceships further down the screen. The agent also has the option of occasionally shooting down a special purple fighter for an extra 200 points. The game ends when the agent loses 3 lives, or when the moving columns of spaceships get to the bottom row. The minimal action set contains 6 actions.

Atlantis is also a shooter, but here the enemy spacecraft fly across the sky very quickly. The agent is in charge of three fixed guns, a primary central one and two secondary guns on the sides and is tasked with protecting the city of Atlantis. The agent must shoot down as many enemy spacecraft as it can. The enemy spacecraft also occasionally attempt to use lasers to take out the agent's 3 guns and 4 other structures of Atlantis. Lost structures, including the guns, can be regained by destroying enough enemy spaceships. The agent loses when all structures are destroyed. The minimal action set contains 4 actions.

Methodology. A trial consists of running RLAdvice for $N = 30$ episodes. For each environment we do 5 trials, and our graphs show these results with error bars suppressed for clarity. We use LIBLINEAR to learn linear regression models after each iteration based on the data accumulated so far. We used the default ϵ for LIBLINEAR, and chose regularisation constant C based on trials on a small set of episodes in each environment, selecting from $\{0.001, 0.01, 0.1, 1.0, 10.0\}$. We use the L2-regularised support vector regression algorithm to solve our regression problem.

We also ran further experiments using RLAdvice with our own implementation of Stochastic Dual Coordinate Ascent (SDCA) (Shalev-Shwartz and Zhang, 2013) minimising the L2-regularised square loss ($\epsilon = 0$) for 100 episodes on Pong and Space Invaders, and 50 episodes on Atlantis, to examine the performance improvement given more data. We used our own implementation in order to provide a warm-start to the regression routine using the agent's previous solution to speed-up the procedure for longer runs. Unfortunately LIBLINEAR does not offer such an option. We found a large speed-up on our most demanding domain, Atlantis, where the regression algorithm dominates computation time.

The total time taken to learn weights for all actions on the final episode goes from 3 hours to approximately 30 minutes, and our overall training time (for 30 episodes) goes from approximately 74 hours to 40 hours.

Comparisons. We compare the following algorithms on Pong, Space Invaders and Atlantis within the ALE framework. Note that we use the minimal action set for each environment. This gets rid of superfluous actions for each environment. It also means that our experiments are computationally less demanding both in memory and time, since we learn a model for each action independently. The SARSA results are taken from Bellemare et al. (2013). Note that RLAdvice-best and UCTRLA-best show the best total reward in any episode so far for RLAdvice and UCTRL.

- SARSA (traditional reinforcement learning) [SARSA].
- RLAdvice using regression. We have the following variations,
  - Training starting with a UCT policy and then iterating the model [RLAdvice].
  - Training only with the UCT policies [UCTRLA]. We note that the UCTRLA result on Atlantis is averaged over 2 trials rather than 5, due to the large computation time required per trial (approximately 5 days).
- SARSA trained using the regression weights obtained from RLAdvice [SARSA-RLA]. The weight vector from the RLAdvice algorithm can be further improved on (in theory) by using SARSA with these weights as the initialisation.

6. Discussion

The results obtained on Pong, Atlantis and Space Invaders demonstrate our two objectives. It is intuitively clear that the class of function approximators we are using should be able to represent the value functions of good Pong policies. We have pairwise features of grids of SECAM colours. There are only three objects in the Pong domain, the two paddles and the ball. The value function of states where the ball is approaching the agent can be represented by the pairwise function of the agent's paddle and the ball position. On Pong we perform better with 30 episodes than the SARSA results over 5000 episodes (Figure 1(a)), noting that we have more information through the oracle than SARSA does. This confirms our intuition that good policies are representable within the class of function approximators, but SARSA does not find them. The result using the oracle policy for training (UCTRLA) confirms that using the oracle policy to train the RLAdvice agent does not work.

A further examination of the Pong behaviour requires observation of the video of an RLAdvice agent playing Pong. http://mdaswani.me/rlavideos/ contains a playlist of videos of the agent playing various games, including Pong at different stages during training. The first thing one notices is the jitteriness of the agent (green, on the right). This can be explained by the behaviour of the oracle UCT. Before the ball is very close to the agent, all actions are nearly equal in reward, since the agent can always reach the position it needs quite quickly. So the oracle acts randomly. Thus the agent inherits this jittery behaviour.

The agent makes mistakes when the data collected from previous episodes is insufficient to learn a good value function approximation. As it makes mistakes, it collects data about

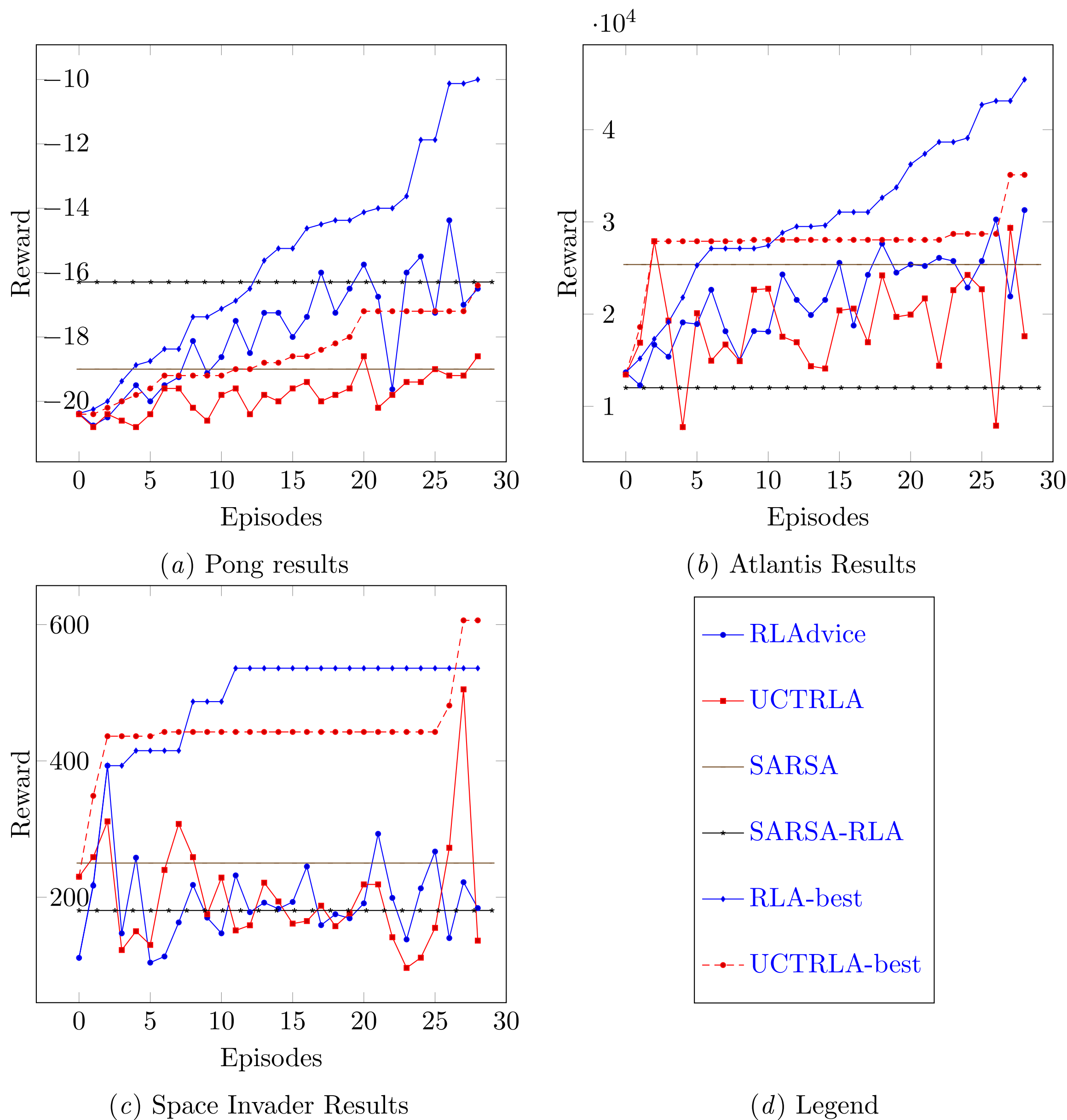


Figure 1: Comparisons against RLAdvice using LIBLINEAR for 30 episodes

the values for the actions in those situations which improves the approximation in those areas. This analysis also illustrates why learning from the oracle policy alone is not satisfactory. If one looks at the video for a UCT agent playing Pong, the agent wins nearly all the time, so the opponent does not even get to serve, or gets to serve only a few times. Thus learning from this policy does not provide the agent with any data on how to return a serve, and so UCTRLA fails on this task.

Our longer runs show much improvement (Figure 2(a)) with the best Pong result being approximately -5 on average, a score of 21-16, comparable to a good human player.

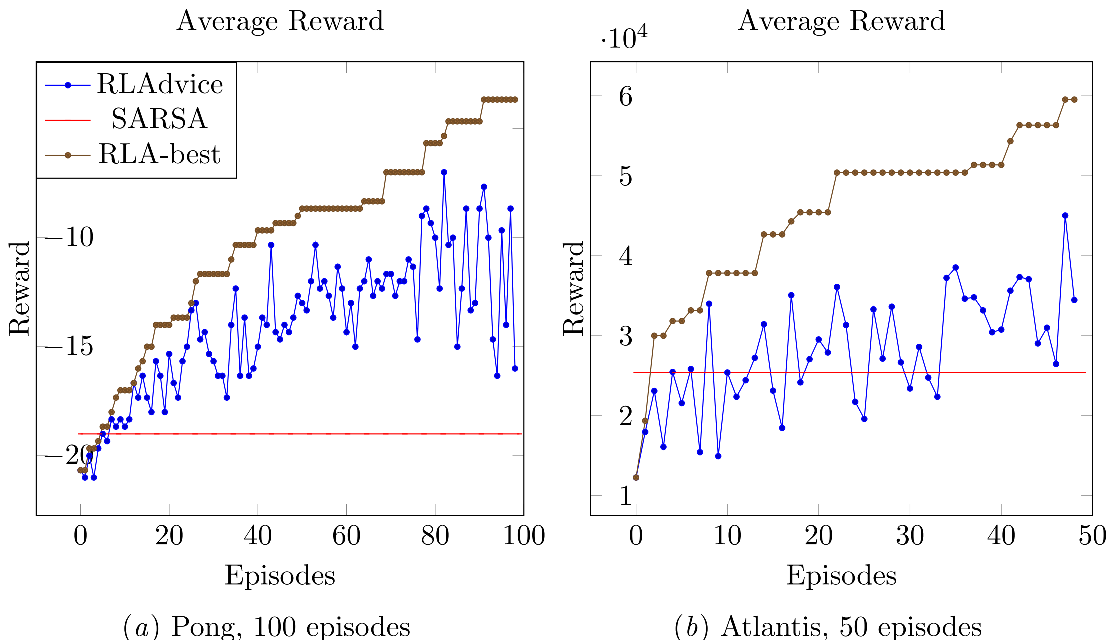
On Space Invaders, RLAdvice performs on average slightly worse than the SARSA result. Here, our result suggests that perhaps the function approximation class we consider is insufficient for doing much better than SARSA, but this is uncertain. An alternative hypothesis is that both SARSA and RLAdvice are unable to find a good policy for an unexplained reason. However, we have some intuition for why the value function may not be representable by a linear combination of the pairwise BASS features. Space Invaders is fairly chaotic, and has the same colour for shots fired by the enemy spaceships and for the agent's own. This makes it hard to distinguish between the two, even though the ALE provides a colour averaging between every two frames. The much higher density of objects in the domain compared to Pong, also makes it harder for the objects to be clearly defined, resulting in feature vectors that look very similar for fairly different situations (such as being hit and being missed by a laser beam). Longer runs on Space invaders show equally poor results.

Atlantis is a more stationary game than the other two, with the agent controlling fixed guns and trying to prevent the guns from being hit by enemy fire, while shooting down enemy spaceships. The RLAdvice agent does quite well on the domain. This domain has only four actions which means that UCT has better estimates of the action values since we keep the number of simulations constant. The average best reward (Figure 1(b)) shows a constant improvement in the best reward with no plateau. The longer run in Figure 2(b) shows that the trend continues and the agent is now consistently better than SARSA.

An issue that we need to take into account, is the accuracy of the estimates of the action-values provided by the oracle (UCT). There are two sources of error. One stems from the number of simulations that we use to sample. The other is the UCB formula itself, which does not select actions which seem to not have a high value within a few iterations. While this exploration-exploitation strategy is useful when acting well in an environment (and indeed solves it in the bandit setting), in our setting we need estimates for all actions, not just the good ones. Pruning the amount of simulations spent on bad actions for the oracle policy means that we have good value estimates for the good actions. The UCB formula means that we will underestimate the value of the bad actions, which is better than overestimation since it gives us a better margin for error.

The SARSA-RLA results are surprising. The agent is given a good model (for Pong and Atlantis) to start with, and ends up playing worse after training using SARSA. We tried various parameter settings, but apart from one trial of a good Pong result (-13.68) the results are disappointing. The average SARSA-RLA Pong result at -16.29 is still better than the average SARSA results, but the Atlantis results are substantially worse at 12015. It seems that temporal difference methods (at the very least SARSA) might not work very well on this task.

Computational issues. We can look at the computation time taken in various stages of the algorithm. The experiments were primarily performed on a Intel Xeon X5650 (2.67 Ghz) with 12 cores and 141GB of RAM. Consider Space Invaders, in which the worst time to perform regression for a given action is 30 minutes, and the average is not much better around 20 minutes. Thus in 30 runs, using 6 actions the regression part cumulatively contributes 5400 minutes to the running time. UCT on the other hand takes 40 minutes cumulatively per episode (approximately 3 seconds for one call to UCT), meaning over 30 episodes this is around 1200 minutes. Other games have similar computational profiles. The



(a) Pong, 100 episodes (b) Atlantis, 50 episodes

Figure 2: Longer results using SDCA

Table 2: Timing data on Atlantis, the most computationally expensive game, using LIBLINEAR for RLAdvice and UCTRLA. The time for RLAdvice includes generating the advice using UCT. The time for SARSA-RLA here is post-RLAdvice training and would be comparable to time taken by standard SARSA.

Algorithm	Training episodes	Training time	Testing time (per episode)
SARSA-RLA	5000	22.01 hours	1 second
RLAdvice	30	73.99 hours	1 second
UCTRLA	30	118.38 hours	1 second
UCT	N/A	N/A	3600 seconds

learning scales linearly with the number of possible actions, which adds a severe constraint on the number of actions we can learn the value of using this method. On the other hand, if we can speed up the regression process, we can significantly improve the running time of the procedure.

In terms of memory, the usage is substantial. A single run of 30 iterations close to a little over 10 GB of RAM on average. This memory is used in storing the feature vectors for every screen visited, along with the current model for each action. The models are approximately 200MB each, which is a small fraction of the total. The models essentially contain a weight for each feature stored in a sparse format. Even though the sparsity of feature vectors is high (around 1%), given 1.6 million features and trajectories in the order of a few thousand frames is enough to add up to a large amount of RAM for each iteration.

Stochastic Environments. Though the experiments were performed on a class of deterministic environments, RLAdvice has no dependence on the determinism of the environment. In the case of randomised initial state in ALE, we can also use the policy trained on the deterministic environment since it generalises according to the linear function approximator. Thus it is much more versatile than simply learning a fixed trajectory.

7. Conclusion

We introduced a modification of the DAgger agent for the reinforcement learning with advice problem. RLAdvice can be used to find explicit policies for anytime algorithms such as UCT, and for checking the usefulness of a function approximation class. It shows improved performance on the Pong and Atlantis domains in the Arcade Learning Environment indicating that value functions of good policies are representable in the class, and similar bad performance to SARSA on Space Invaders which suggests a problem with the function approximation class.

Future Work. As pointed out in the paragraph on computational issues, the computation time is dominated by the model learning. A suitable next step would be to consider a budgeted advice setting where the agent must limit the number of calls it makes to the oracle. This would save time via less UCT computations as well as a smaller number of samples which results in quicker model learning.

Further work on the ALE should use different classes of function approximators, and indeed this is being done (see Section 3). Ultimately, we would like algorithms that are able to automatically find useful feature sets, and deep learning approaches (Mnih et al., 2013) are promising.

The current work treats the oracle as a black box. However, instead of the oracle-learner framework that we have here, it might be better to examine a teacher-learner setup as in Taylor et al. (2014), where the teacher is attempting to optimise the policy it shows to the agent. RLAdvice corrects flaws in its own policy, however this can still fail as seen in the Pong example, where not enough data about a serve from one side of the screen results in suboptimal behaviour. This could be prevented if the teacher was able to predict this failure and show the learner the right data for the learner's function approximation class.

Acknowledgements. We thank the Australian Research Council for support under grant DP120100950 and J.E. Brand for doing the voice-overs on the videos.

References

M. G. Azar, A. Lazaric, and E. Brunskill. Regret bounds for reinforcement learning with policy advice. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Zelezný, editors, *ECML/PKDD (1)*, volume 8188 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2013. ISBN 978-3-642-40987-5.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, 06 2013.

M.G. Bellemare, J. Veness, and M. Bowling. Bayesian learning of recursively factored environments. In Sanjoy Dasgupta and David Mcallester, editors, *Proceedings of the 30th International*

Conference on Machine Learning (ICML-13), volume 28, pages 1211–1219. JMLR Workshop and Conference Proceedings, May 2013.

M Daswani, P Sunehag, and M Hutter. Q-learning for history-based reinforcement learning. In *Asian Conference on Machine Learning*, pages 213–228, 2013.

R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

M. Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general Atari game playing. In *IEEE Transactions on Computational Intelligence and AI in Games*, 2013.

L. Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *In* The 17th European Conference on Machine Learning, pages 99–134, 2006.

R. Maclin and J. W. Shavlik. Creating advice-taking reinforcement learners. *Machine Learning*, 22 (1-3):251–281, 1996.

R. Maclin, J. W. Shavlik, L. Torrey, T. Walker, and E. W. Wild. Giving advice about preferred actions to reinforcement learners via knowledge-based kernel regression. In M. M. Veloso and S. Kambhampati, editors, *AAAI*, pages 819–824. AAAI Press / The MIT Press, 2005. ISBN 1-57735-236-X.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D., and M. Riedmiller. Playing Atari with deep reinforcement learning. 2013.

P. M. Nguyen, P. Sunehag, and M. Hutter. Context Tree Maximizing. In Jörg Hoffmann and Bart Selman, editors, *AAAI*. AAAI Press, 2012.

M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1994. ISBN 0471619779.

S. Ross. Interactive learning for sequential decisions and predictions. Ph.D. thesis, Carnegie Mellon University. 2013.

S. Ross and D. Bagnell. Efficient reductions for imitation learning. In Yee Whye Teh and D. Mike Titterington, editors, *AISTATS*, volume 9 of *JMLR Proceedings*, pages 661–668. JMLR.org, 2010.

Shai Shalev-Shwartz and Tong Zhang. Stochastic dual coordinate ascent methods for regularized loss. *J. Mach. Learn. Res.*, 14(1):567–599, February 2013. ISSN 1532-4435.

R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

M. E. Taylor, N. Carboni, A. Fachantidis, I. P. Vlahavas, and L. Torrey. Reinforcement learning agents providing advice in complex video games. *Connect. Sci.*, 26(1):45–63, 2014.

J. Veness, K. S. Ng, M. Hutter, W. Uther, and D. Silver. A Monte Carlo AIXI Approximation. *Journal of Artificial Intelligence Research*, 40:95–142, 2011. ISSN 1076-9757. doi: 10.1613/jair.3125.

E. Wiewiora, G. W. Cottrell, and C. Elkan. Principled methods for advising reinforcement learning agents. In Tom Fawcett and Nina Mishra, editors, *ICML*, pages 792–799. AAAI Press, 2003. ISBN 1-57735-189-4.