In *Proceedings of the 6th International Conference on Autonomous Agents and Multiagent Systems*, pp. 650--657, ACM, New York, NY, USA, 2007.

Batch Reinforcement Learning in a Complex Domain

Shivaram Kalyanakrishnan and Peter Stone
Department of Computer Sciences
The University of Texas at Austin
{shivaram, pstone}@cs.utexas.edu

ABSTRACT

Temporal difference reinforcement learning algorithms are perfectly suited to autonomous agents because they learn directly from an agent's experience based on sequential actions in the environment. However, their most common algorithmic variants are relatively inefficient in their use of experience data, which in many agent-based settings can be scarce. In particular, they make just one learning "update" for each atomic experience. Batch reinforcement learning algorithms, on the other hand, aim to achieve greater data efficiency by saving experience data and using it in aggregate to make updates to the learned policy. Their success has been demonstrated in the past on simple domains like grid worlds and low-dimensional control applications like pole balancing. In this paper, we compare and contrast batch reinforcement learning algorithms with on-line algorithms based on their empirical performance in a complex, continuous, noisy, multiagent domain, namely RoboCup soccer Keepaway. We find that the two batch methods we consider, Experience Replay and Fitted Q Iteration, both yield significant gains in sample complexity, while achieving high asymptotic performance.

Categories and Subject Descriptors

I.2.6 [**Computing Methods**]: Artificial Intelligence—*Learning*

General Terms

Algorithms, Experimentation.

Keywords

Learning, Evolution and Adaptation, Perception and Action.

1. INTRODUCTION

Reinforcement learning problems are those in which an agent takes sequential decisions in its environment in an effort to maximize its long-term reward. Temporal difference (TD) algorithms, such as Sarsa [9] and Q-learning [12], are *on-line* methods for addressing such problems. We refer to them as "on-line" because they incrementally update the agent's policy based on each individual experience. An experience is defined as a transition from one state to another, coupled with an immediate reward signal, resulting from the agent's action.

Rather than representing a policy directly, temporal difference algorithms learn an intermediate structure known as a *value function* that represents the expected long-term reward from a given state if an agent takes a particular action. If the value function is known, the optimal policy follows immediately: from any given state, the agent selects the action with the maximum value.

Q-learning has been shown to provably converge to the optimal value function in discrete domains [16]. However, in so doing, it is very wasteful of its experience data. In fact, the convergence proof relies on the agent visiting each state infinitely often. This reliance is due, in large part, to the fact that the algorithm only processes experience incrementally. That is, each state transition is used to update the agent's value function immediately, and is then discarded.

Incremental updates have the advantage of requiring little computation and memory, in exchange requiring a lot of data. However, in many complex agent domains, the opposite constraints are the reality. For example, the time between training examples on a physical robot or in a complex real-time simulation environment leaves a lot of time for processing on a modern computer; and even at a (relatively fast) rate of one state transition per second, memory does not become constrained for a considerable length of time. In such domains, the sheer time, tedium, or labor involved in gathering training experiences could be the overriding concern.

Batch reinforcement learning algorithms are designed explicitly to reduce the amount of experiential data needed by leveraging available computation and memory. They do so by saving experience data and using it in aggregate to make updates to the learned policy. In this paper, we compare and contrast two model-free batch reinforcement learning algorithms, Experience Replay [5] and Fitted Q Iteration [2], with on-line TD methods. These batch algorithms have been shown in the past to speed up learning in simple domains such as grid worlds and in low-dimensional control problems such as pole-balancing. We consider the design issues in scaling them to a complex, continuous, multiagent task, namely RoboCup soccer Keepaway [11]. Despite being a simulated environment, Keepaway incorporates the attributes of a real-world task: for instance, agents are provided noisy perceptions, and must cope with imperfect actuators. The task itself models a realistic scenario from soccer in which opposing teams of agents vie for possession of the ball. On-line training in Keepaway takes hours or even days, so making any reduction in its sample complexity can have a significant impact on the feasibility to learn in such a domain. Furthermore, due to the large state space, value function approximation is a necessity, violating the assumptions for guaranteed convergence and thus leaving room for asymptotic performance gains as well.

The remainder of this paper is organized as follows. Section 2 specifies the batch and on-line algorithms implemented in this paper, and provides a detailed comparison within a common algorithmic framework. Section 3 introduces the details of our test-bed domain. Section 4 reports our main experimental results and Section 5 presents a deeper analysis of our experiments. Section 6 summarizes related work and Section 7 concludes.

2. ON-LINE AND BATCH REINFORCEMENT LEARNING

Reinforcement learning involves trial and error. Typically, an agent in some state s will choose to execute an action a, on whose execution it will receive a reward r from the environment and be transported to a new state s'. The sample (s, a, r, s') constitutes an *experience*, which provides it information to improve its behavior.

On-line reinforcement learning is the paradigm where the agent only uses its latest experience to update its policy. For example, consider an on-line Q-learning agent that begins with an initial action value function Q_0, and goes through a sequence of t experiences $D_t = d_1, d_2, d_3, \ldots, d_t$, where $d_i = (s_i, a_i, r_i, s_{i+1}), i = 1, 2, 3, \ldots, t$. On gathering experience d_t, the agent makes an update to its Q-function as follows: $Q_t(s_t, a_t) \leftarrow Q_{t-1}(s_t, a_t) + \alpha(r_t + \gamma \max_a Q_{t-1}(s_{t+1}, a) - Q_{t-1}(s_t, a_t))$, using some learning rate α and discount rate γ. Notice that to compute the updated action value function Q_t, the agent only needs its previous value Q_{t-1} and the last experience d_t.

In discrete domains, on-line Q-learning is guaranteed to converge to the optimal value function, so long as each state is visited infinitely often, and some other minor conditions are met [16]. The convergence guarantee does not extend to continuous domains; in fact, even with discrete domains, the time taken to achieve convergence may exceed the limits imposed by practical applications. It is usually the case that faster learning can be achieved by not just using the last experience d_t in the update, but by also using past experiences encountered in the sequence D_t. **Batch** reinforcement learning algorithms have the precise motivation that gathered experience samples can be used more efficiently by simultaneously using all of them in making an update to the policy. Such an update would look like: $Q_t = trainBatch(Q_{t-1}, D_t)$. In fact, even Q_{t-1} is not necessary for the update since the experience sequence D_t and the initial action value function Q_0 contain all the information necessary to compute Q_t. Nevertheless, knowledge of Q_{t-1} can help speed up the update.

Batch reinforcement algorithms can potentially extract more information from a given sequence of experiences than on-line algorithms, since they are not constrained to make only a single learning update based on each experience[1]. This can lead to faster learning, with fewer samples being needed to learn good policies. In practice, it is usually the case that batch learning updates are not made after each experience, but only after intervals, or *batches*, of several experiences. Algorithm 1 outlines a general framework for batch reinforcement learning. The agent begins with some initial policy represented through the action value function Q_0 (line 1). The policy is interpreted through the *selectAction()* function, which, for instance, may implement ϵ-greedy action selection (line 14). It follows this policy for m episodes, saving the experiences encountered to memory (lines 14–19). The sequence of experiences D thus gathered is then used as a batch to compute a revised action value function Q, which defines its new policy (line 21). This policy is again used to generate a new batch of experiences, and the cycle continues until Q has converged. The size of the memory D can be controlled by choosing a suitable value for m.[2] Note that in Algorithm 1, we have omitted the subscripts for D and Q, treating them as implicit.

Algorithm 1 Batch Reinforcement Learning

1: $Q \leftarrow Q_0$. // Initialize action value function.
2: // Generate batches of experiences, and update policy after each generation.
3: **repeat**
4: $\quad D \leftarrow \emptyset$. // Initialize sequence of experiences D.
5: $\quad i \leftarrow 0$. // Initialize size of D.
6: $\quad$// Collect experiences for m episodes.
7: $\quad episodes \leftarrow 0$.
8: $\quad$**repeat**
9: $\qquad i \leftarrow i + 1$.
10: $\qquad$**if** new episode **then**
11: $\qquad\quad s_i \leftarrow getStateFromEnvironment()$.
12: $\qquad\quad episodes \leftarrow episodes + 1$.
13: $\qquad$**end if**
14: $\qquad a_i \leftarrow selectAction(Q, s_i)$.
15: $\qquad$Execute a_i.
16: $\qquad r_i \leftarrow getRewardFromEnvironment()$.
17: $\qquad s_{i+1} \leftarrow getStateFromEnvironment()$.
18: $\qquad d_i \leftarrow (s_i, a_i, r_i, s_{i+1})$.
19: $\qquad D.append(d_i)$.
20: $\quad$**until** $episodes = m$.
21: $\quad Q \leftarrow trainBatch(D)$. //Update policy.
22: **until** Q has converged

In this paper, we examine two different methods to implement the trainBatch() function in Algorithm 1, namely

[1] In this presentation, as also in our experiments, we only consider algorithms that do not make use of a model of the underlying task for the purpose of learning. Model-based methods have been used in the past to learn good control based on a small sample of training experiences [7].

[2] In our experiments, we store *all* past experiences in D, not just the experiences generated during the current batch. We cite reasons for doing so in Section 5.

Experience Replay [5] and **Fitted Q Iteration** [2]. Experience replay can be viewed as a direct extension of on-line learning to the batch case. With on-line learning, a single update is made based on an experience and then that experience is discarded. Rather than discarding each experience after a single update, with experience replay, experiences are stored and the batch update replays them repeatedly. As shown in Algorithm 2, k passes are made through the collected set of experiences D, and within every pass a Q-learning update is made for each experience.

Algorithm 2 *trainBatchExperienceReplay*(D)

$Q \leftarrow Q_0$. // Initialize action value function.
// Train for k iterations.
for *iteration* $= 1$ to k **do**
    // Replay each experience.
    for all $i \in [1..|D|]$ **do**
        // $d_i = (s_i, a_i, r_i, s_{i+1})$.
        $Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha(r_i + \gamma \max_a Q(s_{i+1}, a) - Q(s_i, a_i))$.
    end for
end for
Return Q

Fitted Q Iteration, presented in Algorithm 3, adopts the approach of computing the action value function through a series of successive approximations, starting with some initial approximation Q_0. The training data set D is processed through several epochs (line 2). At the beginning of every epoch, a target Q-value T_i is fixed for each experience d_i in D (lines 4-7), based on the current approximation $Q_{epoch-1}$. The next approximation, Q_{epoch}, is obtained by training the function approximator in a supervised fashion to fit these targets (lines 10-15). The targets get revised again based on the new approximation, and the cycle continues for a number of epochs E until the approximations begin to converge.

Algorithm 3 *trainBatchFittedQIteration*(D)

1: // Starting with initial value Q_0, compute a sequence of approximations of Q until it converges.
2: **for** *epoch* $= 1$ to E **do**
3:     // Fix targets for each d_i based on $Q_{epoch-1}$.
4:     **for all** $i \in [1..|D|]$ **do**
5:         // $d_i = (s_i, a_i, r_i, s_{i+1})$
6:         $T_i \leftarrow r_i + \gamma \max_a Q_{epoch-1}(s_{i+1}, a)$
7:     **end for**
8:     // Fit the targets using supervised learning.
9:     $Q_{epoch} \leftarrow Q_0$.
10:     **for** *iteration* $= 1$ to k **do**
11:         **for all** $i \in [1..|D|]$ **do**
12:             // $d_i = (s_i, a_i, r_i, s_{i+1})$.
13:             $Q_{epoch}(s_i, a_i) \leftarrow Q_{epoch}(s_i, a_i) + \alpha_{supervised}(T_i - Q_{epoch}(s_i, a_i))$.
14:         **end for**
15:     **end for**
16: **end for**
17: Return Q_E

Before empirically comparing the on-line and batch algorithms that we use in our experiments, i.e., simple on-line learning (OL), Experience Replay (ER), and Fitted Q Iteration (FQI), we first compare and contrast their general properties. ER and FQI are both batch methods that make updates based on saved data. However, the "batch" update of ER is simply a series of repeated Q-learning updates based on individual saved experiences. Consider the (unusual) case when k is set to 1 in Algorithm 2; this models an ER algorithm that replays experiences from a batch exactly once. Learning from the same batch of experiences, and visiting them in the same order, this ER algorithm would make the same updates and learn exactly the same action value function as an on-line algorithm. In contrast, the updates made to the function approximator using FQI are akin to supervised learning updates: target Q-values are frozen *prior* to making the updates, which are themselves gradient descent steps to minimize the error function defined by the targets. Since the targets for *all* the experiences in the batch are fixed before making updates to any state-action pair, we can think of FQI's update as *multi-experience*. Table 1 summarizes the relationship between OL, ER, and FQI.

Table 1: Comparison of Methods

Update	Experiences not saved	Experiences saved (Batch)
Single-experience	On-Line	Experience Replay
Multi-experience	n/a	Fitted Q Iteration

Our test domain for comparing OL, FQI, and ER is Keepaway, which is described in the next section.

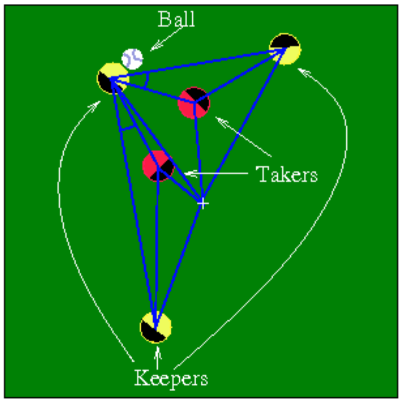
3. TEST DOMAIN: KEEPAWAY SOCCER

Keepaway soccer [11] is a challenging benchmark problem for reinforcement learning. Keepaway is a subtask of simulated RoboCup soccer [6], a domain in which agents have both teammates and adversaries, are provided partial and noisy sensory perceptions, have imperfect actuators, and must act in real time. In Keepaway, a team of m *keepers* faces the task of keeping possession of the ball[3] within a rectangular region of play, resisting attempts by the opposing team of n *takers* to wrest possession. The task is episodic; each episode begins with one of the keepers having possession, and ends when any of the takers gets the ball or the ball goes outside the rectangular region. The keepers are evaluated simply based on the length of the episode, also called the **hold time**. Figure 1 shows a screen-shot from Keepaway with 3 keepers and 2 takers (3 versus 2).

The keeper who is closest to the ball, denoted K_1, can choose to execute one of m high-level actions: **HoldBall()**, by which it keeps the ball within kick-able distance, but away from any approaching taker, or **PassBall(**k**)**, $k = 2, 3, 4 \ldots, m$, which is a direct pass to its $k-th$ teammate, the teammates always being ordered based on their distances to K_1. In this paper, we treat treat **HoldBall()** and **PassBall(**k**)** as *actions*; nonetheless, they are actually high-level *skills* or *options* [13] implemented through low-level actions like **Turn** and **Kick**. **HoldBall()** usually lasts $1-2$ cycles of simulation time, a cycle being 100 milliseconds in real time. **PassBall(**k**)** typically lasts between 4

[3] *possession* of the ball means having it close enough for it to be kicked.

Figure 1: 3 versus 2 Keepaway.



and 12 cycles, depending on the distance the pass has to travel. The learning problem is to decide which action K_1 must execute from any given state, in order that episode length be maximized. The behaviors of keepers other than K_1 and the takers are fixed. Since the state space is continuous, it is represented using a set of state variables. For the 3 versus 2 version of the task, Stone *et al.* [11] use 13 such state variables, involving distances and angles among the players. They also provide a solution to the Keepaway problem that involves on-line Sarsa reinforcement learning. A function approximator, whose inputs are the 13 state variables, is used to store the action value function Q for every action. The action a that the learned policy π chooses from any state s, is given by $\pi(s) = \text{argmax}_a Q(s, a)$. Since the task is episodic, no discounting is used for computing the Q-function.

The main focus of this paper is to study the performance of batch algorithms on Keepaway, and especially how they compare with the Sarsa based on-line algorithm of Stone *et al.*. Our experiments use a slightly modified version of the benchmark version of Keepaway [10, 11]. The modifications preserve the main challenges posed by Keepaway, including its continuous and high-dimensional state space that necessitates the use of function approximation, and noise both in perception and action. In aggregate, they make the task itself slightly more difficult, providing the scope to differentiate among the learning methods. Informal testing shows that they do not adversely affect the on-line learning by a significant amount; also, all the experiments reported in this paper are performed on this same version of the task, so comparisons among the algorithms remain fair. The modifications are as follows.

- Agent Communication: Unlike Stone *et al.*'s algorithm, in which the keepers all learn autonomously, we allow them to communicate messages in order to share their experiences, effectively simulating a centralized learning paradigm. In the autonomous learning case, an agent's updates depend upon the policy followed by other agents, making the learning problem non-stationary. In the centralized paradigm, there is effectively only a single learner, and the learning task becomes stationary. The latter approach has been shown in the past to speed up learning in Half Field Offense [3], a task that extends Keepaway to a goal shooting scenario.

- Action Value function: In our implementation, batch learning algorithms often have to make updates based on samples generated by very different policies, so it is more convenient to learn the *optimal* action value function Q^*, by making Q-learning updates, as opposed to learning the *on-policy* action value function using Sarsa updates. We verified that Q-learning and Sarsa have similar performance with on-line learning in our task variant.

- State Variables: The state variables adopted by Stone *et al.* [11] (depicted in Figure 1) involve distances and angles between the other players and K_1. We find that though K_1 is typically very close to the ball, better generalization is achieved by actually using the position of the ball instead of the position of K_1 while computing state variables. Thus, we would use $dist(Ball, T_1)$ as a state variable in place of $dist(K_1, T_1)$.

- Taker Possession time: In the benchmark version [10, 11], a taker is deemed to have acquired possession only if it keeps the ball for 5 cycles of simulation time. Therefore, it often happens that a taker gets the ball, but some keeper is able to steal it before 5 cycles have elapsed. The dynamics of the learning problem are made more straightforward by avoiding such a scenario, so we terminate an episode as soon as a taker is in a position to kick the ball (for 1 cycle). Note, however, that doing so makes the keepers' task more difficult: episodes that would not have ended in the benchmark version, do end in our version.

4. EXPERIMENTS AND RESULTS

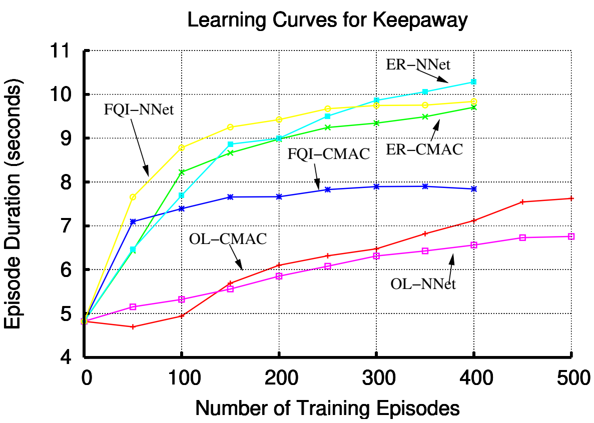
We implemented each of the three learning algorithms (OL, ER, and FQI) on Keepaway with two different function approximation schemes: CMAC tile coding (CMAC) and neural network (NNet). Under both schemes, we maintain a separate function approximator for each action, which takes as input the 13 state variables, and computes as output the Q value for that action.

CMAC [1] is a linear function approximation scheme that partitions the input space into axis-aligned regions called *tiles*. Several different partitions can be overlaid, and each is called a *tiling*. The function approximator associates a weight with each tile, and the function value for any input point is computed as the sum of the weights of the tiles (one from each tiling) within which that point falls. In our experiments, we found CMAC to be robust to parameters like the number of tilings and the width of tiles, and we used the same values used by Stone *et al.* [11], i.e., 32 one-dimensional tilings along each state variable, with tile widths of $3m$ for state variables that represent distances, and 10° for angles. CMAC weights were initially set to 0. Learning updates involved simple gradient descent, with a learning rate of $\alpha = 0.125$.

We found that NNet, which can represent complex, non-linear functions, is more sensitive than CMAC to the actual parameter settings. We experimented informally with networks having one and two hidden layers, with the number of hidden nodes ranging from 20 to 100. With these configurations, we tried multiple values of the learning rate α ranging from 10^{-6} to 10^{-1}. We found that the best performance was achieved by a network with one hidden layer comprising 40 nodes, using sigmoid activation functions. The network weights were initialized to random values in the range $[-0.5, 0.5]$. It was trained using back-propagation. Unlike CMAC, we found it necessary to tune the learning rate to the specific algorithm employing NNet; we used $\alpha = 5\times10^{-4}$ with OL, $\alpha = 10^{-4}$ with ER, and $\alpha = 10^{-5}$ with FQI.

Figure 2 shows the performance of these algorithms on the Keepaway task. The x axis marks the number of episodes of training. At intervals of 50 episodes, we freeze the policy learned at that time and evaluate it by running it off-line (without learning) for 1000 episodes. The y axis represents the average hold time per episode during the evaluation period. Notice that initially, i.e., after 0 episodes of training, all policies register a hold time of about 4.8 seconds, which corresponds to the performance achieved by a random policy. Each curve in the graph represents an average of at least 30 independent trials.

Figure 2: Performance: 500 episodes



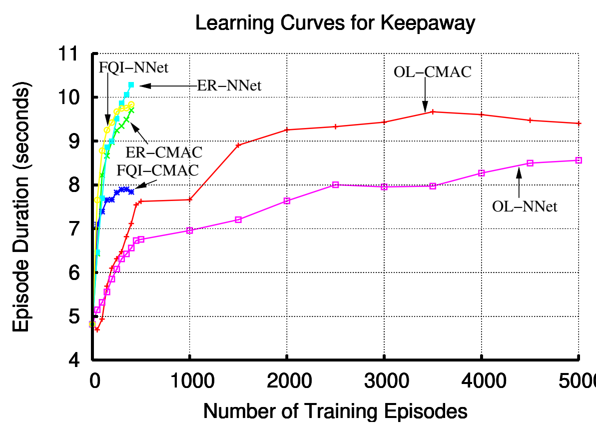
From the graph, it is apparent that there is a stark difference in the learning speed of the batch and on-line methods. After just the first 50 episodes of training, ER and FQI are able to achieve hold times that are between 1 and 3 seconds higher than those achieved by OL, irrespective of the function approximator being used. In fact, most of their learning occurs within the the first 300 episodes. ER and FQI achieve a hold time of 8 seconds in fewer episodes than OL, with a p-value of $p < 10^{-13}$, both under CMAC and NNet. This is in spite of rounding *down* the number of episodes taken by the OL–CMAC and OL–NNet to the nearest multiple of 500, past the first 500 episodes of training. Table 2 displays the average number of episodes taken by the methods to achieve hold times of 6, 7, 8, 9, and 10 seconds. Each entry also provides the number of runs amongst the total conducted for each algorithm that reach the specified hold time. Thus, all the runs of all the algorithms reach 6, 7, and 8 seconds of hold time, but only 32 of the 36 runs conducted for ER–CMAC reach 9 seconds. Incidentally, FQI–CMAC is never able to learn policies having a hold time of 9 seconds or more. Note that the number of samples on average needed by FQI–NNet and ER–NNet to reach 10 seconds of hold time, which is close to the best performance achieved by *any* algorithm on this task, improves the number of samples required by OL–CMAC and OL–NNet to achieve the same hold time by over an order of magnitude.

Due to the high computation time needed by the batch algorithms (see Section 5), we were unable to run experiments with them beyond 400 episodes. Nevertheless, we were able to train using the on-line algorithms for up to 5000 episodes. Figure 2 shows that past the first 150 episodes, OL–CMAC registers a better performance than OL–NNet; from Figure 3 we see that by 1500 episodes, it has also overtaken FQI-CMAC. It is not clear if the performance of ER–NNet has begun to stabilize after 400 episodes, when it has achieved a hold time of 10.28 seconds. OL–CMAC appears to have reached its highest performance, a hold time of 9.67 seconds, at 3500 episodes. A single tailed t-test comparing their best performance values reveals that ER–NNet outperforms OL-CMAC with p-value $p < 2.5 \times 10^{-3}$, establishing its superior asymptotic performance. FQI–NNet, on the other hand, registers 9.84 seconds after 400 episodes, and exceeds OL–CMAC's performance with p-value $p < 0.14$; thus, their performances are at best comparable.

Table 2: Sample Complexity

Algorithm	6s	7s	8s	9s	10s
OL–CMAC 34	**262** 34	**325** 34	**777** 34	**1497** 34	**2761** 23
ER–CMAC 36	**61** 36	**89** 36	**132** 36	**217** 32	**297** 16
FQI–CMAC 39	**50** 39	**69** 39	**248** 32	– 0	– 0
OL–NNet 36	**222** 36	**486** 36	**2235** 36	**3262** 21	**3583** 6
ER–NNet 37	**62** 37	**104** 37	**157** 37	**199** 35	**263** 30
FQI–NNet 39	**50** 39	**63** 39	**82** 39	**151** 39	**246** 27

Figure 3: Performance: 5000 episodes



5. DISCUSSION

Our results show that batch algorithms can yield significant gains in both sample complexity and asymptotic performance when compared to on-line algorithms. Their superiority is demonstrated on a complex task. In this section, we proceed to investigate their behavior along several dimensions, motivating design issues for extending them to other problems. We also consider possible directions for future research.

5.1 Function Approximation

An attractive feature of linear function approximators like CMAC is that under some conditions, they can be shown to converge. Such a guarantee is not available when making Q-learning updates; nonetheless, our experiments show that both on-line and batch algorithms using CMAC eventually achieve good performance. A very natural extension to the experiments we have carried out would be an implementation of Least-Squares Policy Iteration, a provably convergent batch learning method for linear function approximators that has been proposed by Lagoudakis and Parr [4]. In Section 6, we discuss some issues involved in doing so.

Despite having no guarantees of convergence, we find in our experiments that that both ER and FQI show higher asymptotic performance with NNet than with CMAC. NNet is indeed capable of representing far more complex functions than linear function approximators, but due to the broad and "global" nature of its generalization, it can be very sensitive to parameter settings. Nonetheless, our experiments demonstrate that its potential can be realized through stable batch algorithms like FQI, extending the findings of Riedmiller [8], which were limited to low-dimensional control tasks, to the more complex Keepaway domain.

5.2 Training Experiences

For all our batch algorithms, we gathered training experiences in batches of 50 episodes, an episode typically comprising $10 - 20$ experiences. At the end of each batch, we used *all* the experiences so far gathered to compute a new policy. Thus, the first policy update would use experiences from the first 50 episodes, the second from the first 100 (including the first 50), the third from the first 150, and so on. Our scheme contrasts with the more common approach followed by batch learning and policy iteration algorithms, which only use "recent" experiences for making a batch policy update (as outlined in Algorithm 1). For instance, in Lin's experience replay algorithm [5], only samples from the 100 most recent experiences are used for replaying.

There are advantages to using only recent samples for updates. Bootstrapping methods are likely to converge only when updates are on-policy [12]. With on-policy updates, the learned policy is likely to be sub-optimal if the updates involve samples from early in the training phase, when behavior is close to random. In fact, it becomes necessary in non-stationary environments to only use recent experiences, as past ones may have ceased to reflect the current dynamics of the task. Interestingly, though, our experiments in the Keepaway domain revealed that the best performance is always achieved by using *all* the training samples gathered prior to an update for making that update. A possible explanation for this observation is as follows. As successive batch updates start yielding better policies, then by following them, fewer and fewer sub-optimal actions are taken. Hence, for some state s, the most recent experiences are likely to contain only information about taking the optimal action from s; by making the next update to the policy based on these experiences alone, the function approximator receives little information about the consequences of taking *other* actions from s. But since the new policy implicitly represents the action to choose from s through Q-functions for all actions, it is possible that in the absence of training data, non-optimal actions have higher Q-values for s than the optimal action. This could lead to non-optimal actions being chosen, affecting the performance of the policy. However, in samples obtained early in training, all actions are taken with reasonable likelihood; so these result in a better estimate of the Q-function for non-optimal actions as well. Note that in order to accommodate for the use of samples generated by different policies, with both ER and FQI, we perform *off-policy* updates that directly approximate the optimal action value function Q^*. Also, our variant of the Keepaway task is stationary; so samples obtained from early in training continue to reflect the true dynamics of the environment.

Both with ER and FQI, the batch update involves a number of atomic updates to the function approximator making use of state-action pairs from the individual samples. The order in which these updates are made can have an impact on the learning. Lin [5] suggests that it is useful to replay experiences in *backward* order, i.e., replay more recent experiences first, in order to propagate rewards faster. We followed the same procedure in our experiments with ER. For FQI, we randomized the order of updates within every supervised training iteration.

The ability to re-use experiences entails the cost of storing them. In general, our approach of storing all previous experiences may not be feasible. Ideally, we would only like to store at any point, a small, bounded set of experiences. It is an avenue for future research to identify which experiences are likely to be "useful", and to only save a manageable number of useful experiences.

5.3 Computational Complexity

An important concern with batch algorithms is the computational expense involved in making the batch update. An on-line learning algorithm only makes one update to the function approximator for every experience, so the number of such updates it makes while encountering n training samples is just $\theta(n)$. A batch of n examples that has to be visited over k iterations by ER, however, takes $\theta(nk)$ updates. If FQI sweeps through the batch for E epochs, and within each epoch takes k iterations to train the function approximator to convergence, its complexity is $\theta(nEk)$. The best results were achieved in our experiments by setting: $k = 10$ for ER–CMAC; $E = 5, k = 100$ for FQI–CMAC; $k = 10$ for ER–NNet; and $E = 10, k = 500$ for FQI–NNet. With these values, it took FQI–NNet roughly a day on a 3.0 GHz dual CPU processor for doing a batch update using 400 episodes of experience. It was indeed the large computation time taken by batch algorithms that prevented us from training them with more than 400 episodes of experience for our experiments.

The main motivation for employing batch algorithms, despite the computational overhead involved, is their economy in sample complexity. In many domains, it can be laborious and time consuming to collect experiences; and the most efficient use needs to be made of the small sample collected. Though Keepaway is a simulated domain, in which this is not the case, it is easy to imagine a similar task involving real robots that would take much longer. Possibly, a human would have to be present while gathering data from the environment. It could be impractical to learn such a task using OL, but the thousands of episodes saved by ER and FQI could in fact make it feasible.

Based on the settings used for our experiments, we find that ER–CMAC and FQI–CMAC roughly make about 10 and 500 times as many updates as OL–CMAC, respectively, for training a single batch of samples. FQI–CMAC's asymptotic performance compares poorly with those of ER–CMAC and OL–CMAC, despite making far more updates per episode than either. It seems unlikely, then, that computational complexity *alone* accounts for the superior performance of batch algorithms. Lin [5] examines the issue of *over-training*, which occurs when the same experiences are used far too many times. We noticed too, that increasing the number of iterations k of ER–CMAC beyond 10 sometimes causes the CMAC weights to diverge.

5.4 Underlying Algorithm

The two batch algorithms we used in our experiments, ER and FQI, both show high initial learning speed. After 400 episodes, we find that ER has learned a better policy than FQI under both CMAC and NNet function approximation schemes. It should be mentioned, though, that ER requires more careful tuning than FQI, which seems to operate reasonably well under a wider range of parameter values. A possible explanation is that the supervised learning approach employed by FQI lends greater stability while training the function approximator. It emerges that the performance of a batch algorithm rests on the interplay of several parameters: the number of samples used for the batch update, their distribution, the complexity of the update, and the properties of the function approximator. The main contribution of this paper is a demonstration that with careful experimentation, it is possible to realize batch algorithms that yield significant gains over on-line algorithms in complex domains.

ER and FQI are both *model-free* methods that make updates to the function approximator solely based on the transitions observed from the environment. If a model of the environment is available, or if it can be learned, then it can be used to simulate transitions that mimic the dynamics of the environment. This can greatly reduce sample complexity, as updates can be made based on the simulated transitions. Of course, the quality of the solution will depend on the accuracy of the model. To the best of our knowledge, a model-based approach has not been considered before for Keepaway. The domain poses challenges in the form of a continuous, partially observable state space, and noisy, high-level actions. To be able to learn a model for this domain and use it to speed up learning promises to be an interesting problem for future research.

6. RELATED WORK

This paper presents empirical evidence that two existing batch algorithms, Experience Replay and Fitted Q Iteration, can significantly improve upon the performance of on-line methods in a complex task for reinforcement learning: Keepaway. Experience Replay is due to Lin [5], who introduces it as a technique to speed up on-line learning. His test domain is a 25×25 grid world in which an agent must reach cells containing food, while evading enemies pursuing it. It is shown in this domain that ER learns faster than OL. Incidentally, the function approximator employed is a neural net; in our experiments too, ER–NNet shows the best asymptotic performance. In addition, our results demonstrate the feasibility of using ER in conjunction with CMAC for function approximation.

Ernst *et al.* introduce Fitted Q Iteration [2] as a batch reinforcement learning method, and prove convergence properties for certain classes of function approximators called kernel-based methods. They consider the special case of tree-based function approximators, and provide empirical support for the guarantees of the algorithm on classical control problems like Acrobot and Mountain Car. Riedmiller [8] expressly studies the use of neural networks as function approximators with FQI. He reports that successful policies can be learned using relatively few training samples on three control tasks: Pole Balancing, Mountain Car, and Cart-pole. These tasks have continuous state spaces, with perfect state information, but none of them involve more than 4 state variables. Our test domain, 3 versus 2 Keepaway, employs as many as 13 state variables; further, state information is noisy. Another key difference between the control tasks Riedmiller considers and Keepaway is that in the former, it is possible to generalize *across* actions. In Pole Balancing, for instance, the two available actions are a left force and right force of equal magnitude; these are given to the neural network through a single *input*. No such generalization is possible in Keepaway, since the actions are composite, high-level commands **HoldBall()** and **PassBall(k)**. Since we use a separate function approximator for each action, learning becomes all the more difficult.

Lagoudakis and Parr's Least-Squares Policy Iteration (LSPI) [4] is a model-free batch learning method applicable exclusively to problems where the function approximator computes the action value function as some linear combination of basis functions defined over the state variables. If there are k such functions, LSPI's batch update directly computes their weights through a linear operation involving a $k \times k$ matrix that has been populated based on sample data from the batch. While it is guaranteed that LSPI will converge, the quality of the solution hinges on the particular choice of basis functions and the distribution of collected samples, both of which become important concerns if the method is to be applied to Keepaway. We note that for the tasks they consider in their experiments, Lagoudakis and Parr only have to employ a few tens of basis functions to represent the Q-function. With our current CMAC scheme, a few thousand basis functions are used in order to achieve good resolution for the Q-function: this could result in a very large matrix. Also, in their experiments, just a single batch of experiences collected using a random policy seems to suffice for learning a good policy; our results indicate that in Keepaway, samples collected by following updated versions of the policy contribute significantly to the learned performance. For these reasons, a direct comparison of our methods against LSPI is not straightforward, and falls beyond the scope of this paper. Nonetheless, implementing LSPI for Keepaway holds promise as future work.

Keepaway is a challenging benchmark problem for reinforcement learning. Accompanying their introduction of the task, Stone *et al.* [11] present a Sarsa-based on-line algorithm for Keepaway, using CMAC for function approximation. On-line learning has subsequently been extended to other function approximators like neural networks and radial basis functions [10]. Problems related to reinforcement learning, such as behavior transfer [15] have been studied in the context of Keepaway. Taylor *et al.* [14] have also applied evolutionary methods to this domain, which, like the batch methods we have considered, use a significant amount of off-line processing time to improve the learned policy. But to the best of our knowledge, none of the other algorithms applied to Keepaway give qualitative results of a reduction in sample complexity comparable with the order of magnitude reduction seen in our experiments.

7. CONCLUSION

In this paper, we have compared batch reinforcement learning algorithms with on-line algorithms. Our particular concern is economy in the sample complexity of algorithms, which is a key issue for learning in complex domains. We have adopted Keepaway, one such complex, continuous, multiagent domain to carry out our experiments. On this task, we demonstrate the superior sample complexity achieved by two existing batch algorithms, Experience Replay and Fitted Q Iteration. In addition to comparing and contrasting batch algorithms an on-line algorithms, this paper presents the first implementations of batch RL algorithms with a CMAC function approximator, and presents results for batch algorithms in a significantly more complex scenario than has been previously studied. Our results indicate that using batch algorithms may be a viable approach to scaling up reinforcement learning to more real-world tasks.

8. ACKNOWLEDGEMENTS

The authors are thankful to Yaxin Liu, Shimon Whiteson, and the anonymous reviewers of this paper for providing useful comments. This research was supported in part by NSF CISE Research Infrastructure Grant EIA-0303609, NSF CAREER award IIS-0237699, and DARPA grant HR0011-04-1-0035.

9. REFERENCES

[1] J. S. Albus. *Brains, Behavior, and Robotics*. BYTE Books, Peterborough, 1981.

[2] D. Ernst, P. Geurts, and L. Wehenkel. Tree-based batch mode reinforcement learning. *J. Mach. Learn. Res.*, 6:503–556, 2005.

[3] S. Kalyanakrishnan, Y. Liu, and P. Stone. Half field offense in RoboCup soccer: A multiagent reinforcement learning case study. *Proceedings of the RoboCup International Symposium 2006*, June 2006.

[4] M. G. Lagoudakis and R. Parr. Least-squares policy iteration. *Journal of Machine Learning Research*, 4:1107–1149, 2003.

[5] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

[6] M.Chen, E.Foroughi, F.Heintz, Z.Huang, S.Kapetanakis, K.Kostiadis, J.Kummeneje, I.Noda, O.Obst, P.Riley, T.Steffens, Y.Wang, and X.Yin. Users manual: RoboCup soccer server — for soccer server version 7.07 and later. *The RoboCup Federation*, August 2002.

[7] A. Y. Ng, H. J. Kim, M. I. Jordan, and S. Sastry. Autonomous helicopter flight via reinforcement learning. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.

[8] M. Riedmiller. Neural fitted q iteration - first experiences with a data efficient neural reinforcement learning method. In J. Gama, R. Camacho, P. Brazdil, A. Jorge, and L. Torgo, editors, *ECML*, volume 3720 of *Lecture Notes in Computer Science*, pages 317–328. Springer, 2005.

[9] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

[10] P. Stone, G. Kuhlmann, M. E. Taylor, and Y. Liu. Keepaway soccer: From machine learning testbed to benchmark. *RoboCup-2005: Robot Soccer World Cup IX*, 4020:93–105, 2006.

[11] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[12] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[13] R. S. Sutton, D. Precup, and S. P. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.

[14] M. Taylor, S. Whiteson, and P. Stone. Comparing evolutionary and temporal difference methods for reinforcement learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1321–28, July 2006.

[15] M. E. Taylor and P. Stone. Behavior transfer for value-function-based reinforcement learning. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, editors, *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 53–59, New York, NY, July 2005. ACM Press.

[16] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.