Published in Deep Reinforcement Learning: Frontiers and Challenges, IJCAI 2016 Workshop

On-Policy vs. Off-Policy Updates for Deep Reinforcement Learning

Matthew Hausknecht and Peter Stone
University of Texas at Austin
{mhauskn, pstone}@cs.utexas.edu

Abstract

Temporal-difference-based deep-reinforcement learning methods have typically been driven by off-policy, bootstrap Q-Learning updates. In this paper, we investigate the effects of using on-policy, Monte Carlo updates. Our empirical results show that for the DDPG algorithm in a continuous action space, mixing on-policy and off-policy update targets exhibits superior performance and stability compared to using exclusively one or the other. The same technique applied to DQN in a discrete action space drastically slows down learning. Our findings raise questions about the nature of on-policy and off-policy bootstrap and Monte Carlo updates and their relationship to deep reinforcement learning methods.

1 Introduction

Temporal-Difference (TD) methods learn online directly from experience, do not require a model of the environment, offer guarantees of convergence to optimal performance, and are straightforward to implement [9]. For all of these reasons, TD learning methods have been widely used since the inception of reinforcement learning. Like TD methods, Monte Carlo methods also learn online directly from experience. However, unlike TD-methods, Monte Carlo methods do not bootstrap value estimates and instead learn directly from returns. Figure 1 shows the relationship between these methods.

In this paper, we focus on two methods: on-policy Monte Carlo [9] and Q-Learning [10]. On-policy MC employs on-policy updates without any bootstrapping, while Q-Learning uses off-policy updates with bootstrapping. Both algorithms seek to estimate the action-value function $Q(s,a)$ directly from experience tuples of the form $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ and both provably converge to optimality so long as all state-value pairs are visited an infinite number of times and the behavior policy eventually becomes greedy. Both methods are driven by temporal difference updates which take the following form, where y is the update target and α is a stepsize:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(y - Q(s_t, a_t))$$

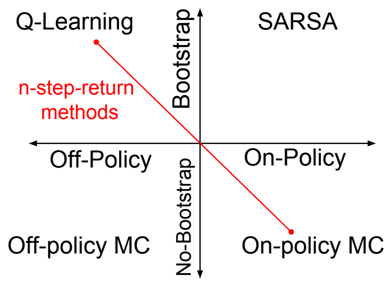


Figure 1: Relationship of RL algorithms according to whether they bootstrap the value function and if they are on or off-policy. This work compares Q-Learning updates with On-Policy Monte-Carlo updates. N-step-reward methods such as n-step Q-Learning bridge the spectrum between Q-Learning and on-policy Monte-Carlo.

The main difference between these methods may be understood by examining their update targets. The update targets for Q-Learning, n-step-Q-learning, and on-policy MC may be expressed as follows:

$$y_{\text{q-learning}} = r_t + \gamma \max_a Q(s_{t+1}, a)$$
$$y_{\text{n-step-q}} = r_t + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \max_a Q(s_{t+n+1}, a)$$
$$y_{\text{on-policy-monte-carlo}} = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$$

As seen in the update, the on-policy MC targets are estimated directly from the rewards received in the experience tuples. In contrast the Q-Learning target truncates the reward sequence with its own value estimate.

One way of relating Q-Learning to on-policy MC is to consider n-step-return methods. These methods make use of multi-step returns and are potentially more efficient at propagating rewards to relevant state-action pairs. On-policy MC is realized when n approaches infinity (or maximum episode length). Recently, multi-step returns have been shown to be useful in the context of deep reinforcement learning [7].

The next section introduces background on deep reinforcement learning before discussing how to efficiently compute and utilize on-policy Monte Carlo updates.

2 Background

Using deep neural networks to approximate the value function is a double-edged sword. Deep networks are powerful function approximators and strongly generalize between similar state inputs. However, generalization can cause divergence in the case of repeated boostrapped temporal-difference updates. Let us consider the case of the same Q-Learning update applied to deep neural network parameterized by θ:

$$Q(s_t, a_t|\theta) = r_{t+1} + \gamma \max_a Q(s_{t+1}, a|\theta)$$

If it is the case that $r_{t+1} > 0$ and s_t is similar to s_{t+1}, then Q-Value estimates will quickly diverge to infinity as this update is repeated. The divergence is because the network's generalization causes the estimate of s_{t+1} to grow with the estimate of s_t, causing update targets to continually grow.

To address this problem a target network is used to make bootstrap updates more stable [8][1]. By updating the target network at a slower rate than the main network, it is possible to limit the generalization from s_{t+1} to s_t, stabilize the update targets, and prevent Q-Value divergence. Using τ (typically .001) to govern the rate that the target network follows the main network, the same update using a target network $\hat{\theta}$ takes the following form:

$$Q(s_t, a_t|\theta) = r_{t+1} + \gamma \max_a Q(s_{t+1}, a|\hat{\theta})$$

$$\hat{\theta} = \tau\theta + (1-\tau)\hat{\theta}$$

On-policy Monte Carlo updates remove the need for a target network since the target is computed directly from the rewards of the trajectory rather than bootstrapped. Such an update makes sense particularly when there is reason to believe that the neural network's estimates of the next state $Q(s_{t+1}, a|\hat{\theta})$ are inaccurate, as is typically the case when learning begins. Additionally, Monte Carlo update targets cannot diverge since they are bounded by the actual rewards received. However, on-policy MC updates suffer from the problem that exploratory actions may negatively skew Q-Value estimates. We now address the issue of how to efficiently compute on-policy MC targets.

3 Computing On-Policy MC Targets

We store on-policy targets y_t in the replay memory by augmenting each transition to include the on-policy target: $(s_t, a_t, r_t, y_t, s_{t+1}, a_{t+1})$. As shown in Algorithm 1, we first accumulate a full episode of experience tuples then work backward to compute on-policy targets and add augmented experiences to the replay memory. Once stored in the replay memory, on-policy MC targets can be accessed directly from the augmented experience tuples without requiring any additional computation.

[1]Another way to address this problem is to not repeatedly update the same experience tuple.

Algorithm 1 Compute On-Policy MC Targets

```
Given: Trajectory $T_{0...n}$, Replay Memory $\mathcal{D}$
$R \leftarrow 0$
for $t \in \{n \dots 0\}$ do
    $R \leftarrow r_t + \gamma R$
    $y_t \leftarrow R$
    $\mathcal{D} \leftarrow (s_t, a_t, r_t, y_t, s_{t+1}, a_{t+1})$
```

4 Mixing Update Targets

Rather than using exclusively on-policy or off-policy targets it is possible, and in many cases desirable, to mix on-policy MC targets with off-policy 1-step Q-Learning targets. Mixing is accomplished using a β parameter in $[0, 1]$. The overall mixed update target is expressed as follows:

$$y = \beta\, y_{\text{on-policy-MC}} + (1-\beta)\, y_{\text{q-learning}}$$

Like n-step-return methods, mixed targets present a way to tradeoff between on-policy MC updates, and off-policy bootstrap updates. The next sections present results using mixed update targets for the cases of discrete action space learning using DQN and continuous action space learning using DDPG.

5 Results in discrete action space

The DQN architecture [8] uses a deep neural network and 1-step Q-Learning updates to estimate Q-Values for each discrete action. Using the Arcade Learning Environment [2], we evaluate the effect of mixed-updates on the Atari games of Beam Rider, Breakout, Pong, QBert, and Space Invaders. Results are presented for a single training run in Figure 2. Mixed updates dramatically slow learning for all the games except Q-Bert.[2] The next section explore results in continuous action space.

6 Results: DDPG

Deep Deterministic Policy Gradient (DDPG) [6] is a deep reinforcement learning method for continuous action space learning. DDPG uses an actor-critic architecture (Figure 3) with the following approximate 1-step Q-Learning update, where $\hat{\theta}_Q$ denotes the parameters of the critic target network and $\hat{\theta}_\mu$ the parameters of the actor target network:

$$y_{\text{q-learning}} = r_t + \gamma Q(s_{t+1}, \mu(s_{t+1}|\hat{\theta}_\mu)|\hat{\theta}_Q)$$

This update is approximate in the sense that it replaces a max over the possible next state actions with with the actor's preferred action $\mu(s_{t+1}|\hat{\theta}_\mu)$. Such an approximation is necessary because computing a max in continuous space is difficult. However, the approximation is only true when the actor is optimal with respect to the critic. As shown in the results, mixed updates may reduce the error in this approximation.

[2]We are unsure why the agent's performance in Q-Bert remains unaffected by on-policy MC updates.

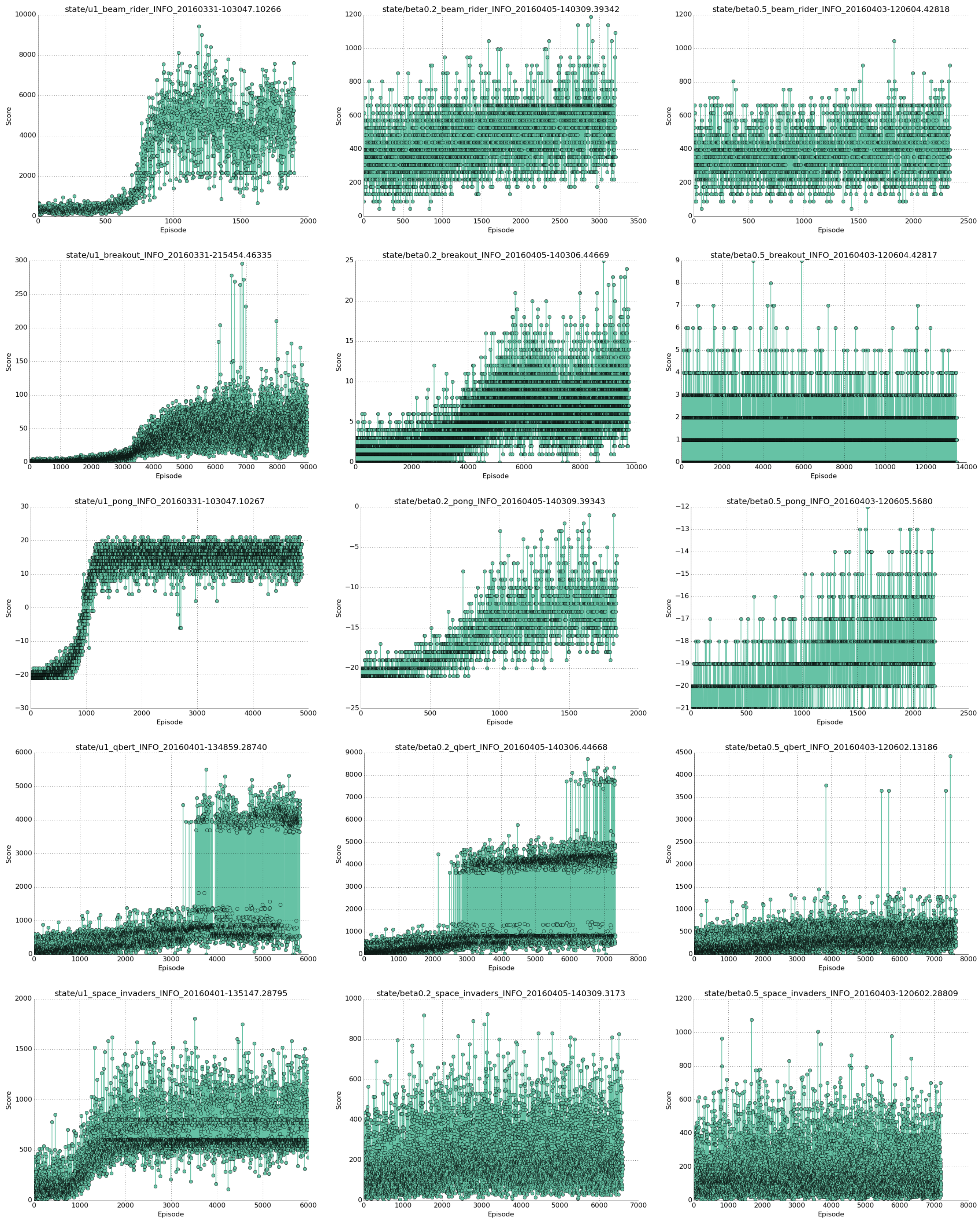


Figure 2: Performance of mixed updates with DQN on various Atari games. The left, middle, and right columns respectively show performance of $\beta = 0$ (Q-Learning), $\beta = 0.2$, and $\beta = 0.5$. Rows correspond to the games Beam Rider, Breakout, Pong, Q-Bert, and Space Invaders. With the possible exception of Q-Bert, mixed updates uniformly slow down DQN's learning. Note that the scale of the y-axis changes.

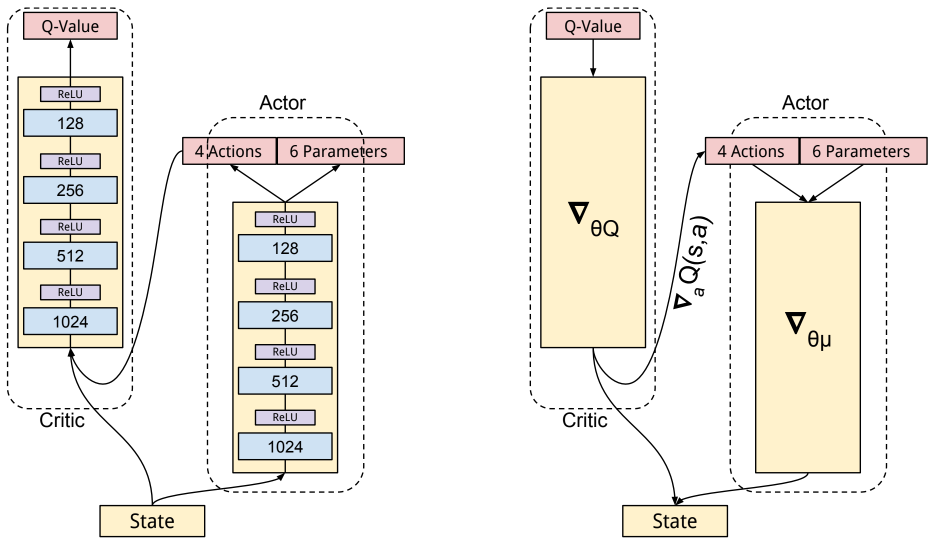


Figure 3: **Actor-Critic architecture (left)**: actor and critic networks may be interlinked, allowing activations to flow forwards from the actor to the critic and gradients to flow backwards from the critic to the actor. The gradients coming from the critic indicate directions of improvement in the continuous action space and are used to train the actor network without explicit targets. **Actor Update (right)**: Backwards pass generates critic gradients $\nabla_a Q(s, a|\theta^Q)$ w.r.t. the action. These gradients are back-propagated through the actor resulting in gradients w.r.t. parameters ∇_{θ^μ} which are used to update the actor. Critic gradients w.r.t. parameters ∇_{θ^Q} are ignored during the actor update.

6.1 Half Field Offense Domain

We evaluate DDPG in the Half Field Offense (HFO) domain `https://github.com/LARG/HFO`. HFO is a simulated 2-D soccer task in which agents attempt to score and defend goals. In HFO, each agent receives its own state sensations and must independently select its own actions. HFO is naturally characterized as an episodic multi-agent POMDP because of the sequential partial observations and actions on the part of the agents. HFO features a continuous state space and a parameterized-continuous action space.

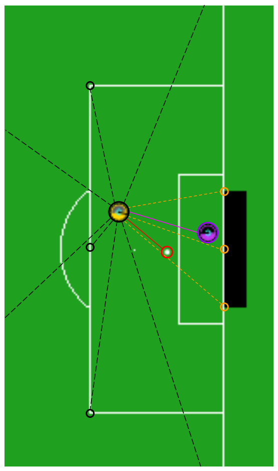


Figure 4: HFO State Representation uses a low-level, egocentric viewpoint providing features such as distances and angles to objects of interest like the ball, goal posts, corners of the field, and opponents.

Half Field Offense features a low-level, parameterized action space. There are four mutually-exclusive discrete actions: Dash, Turn, Tackle, and Kick. At each timestep the agent must select one of these four to execute. Each action has 1-2 continuously-valued parameters which must also be specified. An agent must select both the discrete action it wishes to execute as well as the continuously valued parameters required by that action. The full set of parameterized actions is:

`Dash`(power, direction): Moves in the indicated direction with a scalar power in $[0, 100]$. Movement is faster forward than sideways or backwards. `Turn`(direction): Turns to indicated direction. `Tackle`(direction): Contests the ball by moving in the indicated direction. This action is only useful when playing against an opponent. `Kick`(power, direction): Kicks the ball in the indicated direction with a scalar power in $[0, 100]$. All directions are parameterized in the range of $[-180, 180]$ degrees.

The next sections present two tasks in the HFO umbrella and show that in both cases, mixed updates yield superior performance and stability compared to purely on-policy MC or off-policy Q-Learning updates.

6.2 Experimental Parameters

The following parameters were used throughout the experiments. Both actor and critic employed 5-layers networks with 1024, 512, 256, 128 hidden nodes in each layer. Networks were trained using the Adam Solver [5] with momentum 0.95 and an actor learning rate of 10^-5 and critic learning rate of 10^-3. Gradients were clipped at 10 scaled as they approached parameter bounds [3]. Epsilon greedy exploration with random values for continuous parameters was annealed over 10,000 iterations. Gamma was set to .99 and replay memory capacity was 500,000. Target networks were updated using soft updates [6] with $\tau = .001$. One actor-critic update was performed for every ten new experience tuples. Our implementation uses the Caffe library [4].

6.3 Scoring on an empty goal

The first task we examine is that of scoring on an empty goal. In this task, the agent is initialized at a random position on the offensive half of the field and must learn to approach the ball, dribble the ball towards the goal, and score. Rewards are provided for approaching the ball (e.g. minimizing $d(a,b)$, the distance between the agent and the ball), for getting in range to kick the ball (e.g. an indicator of kickable $\mathbb{I}_t^{kick}$), for moving the ball towards the goal, and for scoring a goal $\mathbb{I}_t^{goal}$:

$$r_t = d_{t-1}(a,b) - d_t(a,b) + \mathbb{I}_t^{kick} + 3\big(d_{t-1}(b,g) - d_t(b,g)\big) + 5\mathbb{I}_t^{goal}$$

Previously, DDPG has been demonstrated to be capable of learning policies for performing this task [3]. However, by incorporating mixed-updates, DDPG becomes more stable across a wide spectrum of β values, as shown in Figure 5. These results are encouraging as learning stability is a crucial aspect for training agents.

6.4 Scoring on a keeper

A far more difficult task is scoring on a goal keeper. The goal keeper's policy was independently programmed by Helios RoboCup 2D team [1] and is highly adept. The keeper continually re-positions itself to prevent easy shots and will charge the striker if it nears the goal. The keeper blocks any shots within its reach, but the size of the goal allows a correctly positioned striker to score with a precise kick.

We modify the task initialization in order to emphasize goal scoring rather than approaching the ball. Specifically, to begin each episode we initialize the agent three fifths of the way down the field and give it possession of the ball. The agent must learn to dribble and position itself as well as learn to precisely kick the ball at open goal angles. Rewards in this task are the same as in the empty goal task: the agent is rewarded for approaching the ball, moving the ball towards the goal, and scoring.

Results in Figure 6 show that mixed updates are not only more stable and higher performing, they are also necessary to learn to reliably score on the keeper. Preferring off-policy targets yields the best performance on this task with $\beta = 0.2$ exhibiting the fastest learning and and a final policy that successfully scores goals every time in an evaluation consisting of 100 episodes. In contrast, the expert hand-coded Helios offense agent scores 81.4% of the time against the keeper. This offense agent was programmed by an independent team of human experts specifically for the task of RoboCup 2D soccer. That it is significantly outperformed by a learned agent is a testament to power of modern deep reinforcement learning methods. A video of the learned policy may be viewed at `https://youtu.be/JEGMKvAoB34`.

7 Future Work

Much remains to be understood about why mixing off-policy bootstrapped updates with on-policy monte-carlo updates provides increased performance and stability for DDPG but slows learning for DQN. DDPG features a pseudo off-policy update in the sense that it does not compute a true max over next state actions. Perhaps this approximate update is responsible for occasional instability (as seen in $\beta = 0$ results). Mixing on-policy targets with approximate off-policy targets may reduce the negative effects of the approximate update. In contrast, DQN implements a true off-policy update in discrete action space and shows no benefit from mixed updates.

We have not explored the performance of DQN or DDPG when using SARSA, off-policy Monte Carlo, or n-step-q-learning.

Bootstrap updates are inaccurate when learning begins since the untrained network's estimate of next-state Q-Values is inaccurate. In this case, non-bootstrapped updates provide more stable targets. As the network's Q-Value estimates become more informed, it makes sense to trend towards off-policy updates so that exploratory actions do not negatively skew estimates. A natural extension would be to anneal β to favor on-policy MC targets in the beginning of learning where bootstrapped estimates are likely to be inaccurate and later favor Q-Learning targets once boostrapped estimates become more accurate.

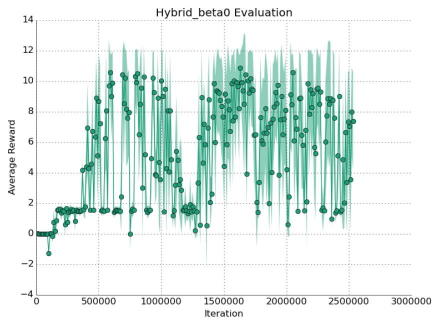
Mixed updates using β likely have a relationship to n-step-q-learning, where $\beta = 1$ is equivalent to infinite-step-q-learning and $\beta = 0$ is equivalent to 1-step-q. The nature of the relationship between the methods has yet to be understood. Additionally, $TD(\lambda)$ methods average all returns from 1-step to n-steps. Such an update target would likely be difficult to efficiently compute.

When learning from an experience database filled with trajectories generated by an expert, it makes sense to prefer on-policy updates since we trust the expert's action selection more than our own. In this case, off-policy updates could easily lead to overestimating Q-values for suboptimal actions and subsequent policy collapse.

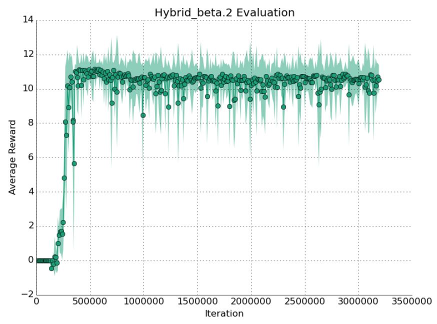
Finally, it would be valuable to see if the benefits of mixed updates with DDPG extend beyond the domain of Half Field Offense to other continuous action domains.

8 Conclusion

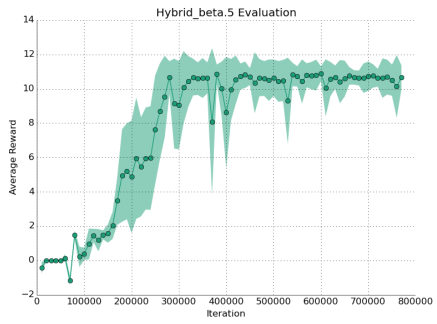
Traditionally, deep reinforcement learning methods such as DQN and DDPG have relied on off-policy bootstrap updates. We examine alternative on-policy monte-carlo updates and present a method for efficiently computing mixed update targets. Our empirical results show that mixed updates increase DDPG's performance and stability in two continuous action tasks and hinder DQN's learning across four out of five Atari games. These results provide evidence that bootstrap off-policy updates are not always the update of choice, and in continuous action space, mixing on and off-policy targets yields superior performance. We hope these results will spark interest in further understanding the relationship between on-policy and off-policy, boostrap and monte-carlo updates in the context of deep reinforcement learning.



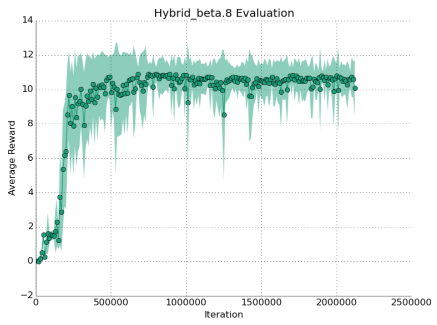
(a) $\beta = 0$



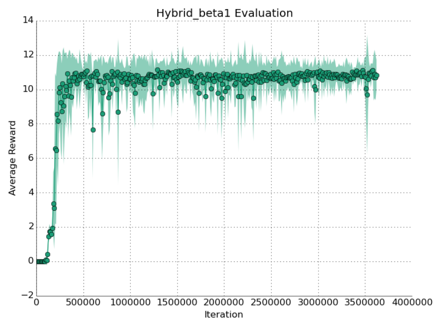
(b) $\beta = 0.2$



(c) $\beta = 0.5$

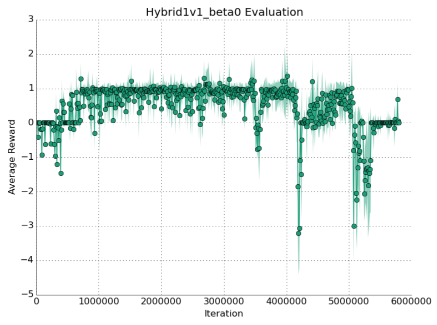


(d) $\beta = 0.8$

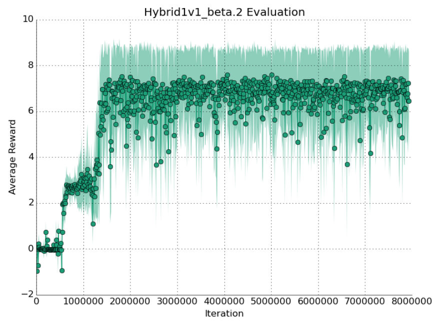


(e) $\beta = 1$

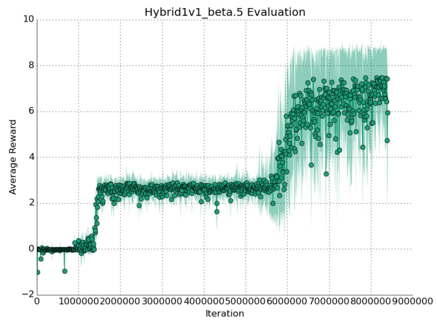
Figure 5: **Performance of mixed-updates on empty goal task**: The maximum possible reward is 11. Purely off-policy updates ($\beta = 0$) achieve this maximum reward but show inconsistent performance. All of the mixed updates achieve the maximum task performance with far greater consistency and with fewer updates. Note that the scale of the y-axis changes between plots.



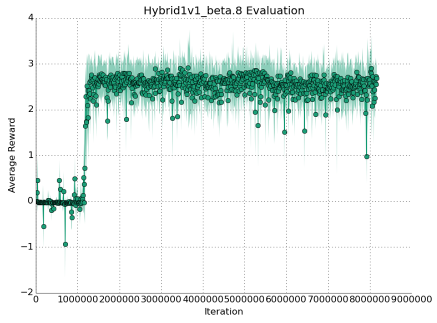
(a) $\beta = 0$



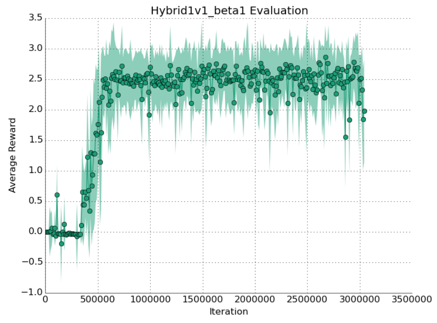
(b) $\beta = 0.2$



(c) $\beta = 0.5$



(d) $\beta = 0.8$



(e) $\beta = 1$

Figure 6: **Performance of mixed-updates on Keeper task**: On this task, only mixed-updates ($\beta = 0.2, 0.5$) achieve the maximum reward of 6 and are able to learn to reliably score on the keeper. Off-policy updates ($\beta = 0$) and on-policy updates ($\beta = 0.8, 1$) never reliably learn to score. Note that the scale of the y-axis changes between plots.

Acknowledgments

Thanks to Tim Lillicrap for insightful discussion. This work has taken place in the Learning Agents Research Group (LARG) at the Artificial Intelligence Laboratory, The University of Texas at Austin. LARG research is supported in part by grants from the National Science Foundation (CNS-1330072, CNS-1305287), ONR (21C184-01), AFRL (FA8750-14-1-0070), AFOSR (FA9550-14-1-0087), and Yujin Robot. Additional support from the Texas Advanced Computing Center, and Nvidia Corporation.

References

[1] Hidehisa Akiyama. Agent2d base code, 2010.

[2] 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, jun 2013.

[3] Matthew J. Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. *CoRR*, abs/1511.04143, 2015.

[4] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[6] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *ArXiv e-prints*, September 2015.

[7] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

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