Ordinary vs. Weighted Importance Sampling

Anne Chel
University of Amsterdam
10727477

Gerson Foks
University of Amsterdam
12589845

Sven Poelmann
University of Amsterdam
11007613

Walter van Rijen
University of Amsterdam
12622273

Abstract—In this report ordinary and weighted importance sampling are compared through analyzing different off-policy learning methods in a selection of different environments, identifying differences in performance and convergence. The results show that weighted importance sampling can outperform its ordinary variant, but that in some environments both exhibit similar performance.

I. Introduction

We compared two different methods for off-policy learning of value functions that are used in reinforcement learning (RL). Our research question was: How much do ordinary and weighted importance sampling differ in practice and which is better? And does this depend on the type of algorithm (e.g. bootstrapping vs. Monte-Carlo)? In this report, we will first give a brief overview of related work and the relevant literature. Then, we discuss the difference between ordinary and weighted importance sampling and why it is relevant to compare their respective performance. Then, we describe our experiments and their results. Finally, in the conclusion, we provide a brief summary of our findings.

II. Reinforcement learning and importance sampling

A. RL basics

In reinforcement learning problems, the agent's goal is to learn a good policy for a sequential decision problem. In contrast to supervised deep learning techniques, the agent is never given any specific training data of the behaviour it is supposed to learn [Sutton and Barto, 2018]. Instead, it has to learn this behaviour itself, by exploring an environment.

We can model RL problems by means of a Markov decision process. The agent and the environment interact at discrete time steps. At each time step t, the agent receives some representation of the environment's state, S_t, and on that basis selects an action, A_t. After completion of the action, the agent receives a numerical reward, R_{t+1}, and finds itself in a new state, S_{t+1}, after which it will choose a new action, A_{t+1}.

The goal, then, is to learn the optimal policy π_*, which maps every state S_t to the best action A_t that can be taken in S_t. That is, π_* selects the actions such that the sum of their return values is maximized [Sutton and Barto, 2018].

B. Exploration vs. Exploitation

One big challenge in RL, that does not arise in other kinds of learning, is the trade-off between exploration and exploitation. At any point, the agent has limited information about its environment. And it can choose actions to either 1) exploit that information (choosing the action that is believed to be best) or 2) explore other actions, perhaps getting new information. To maximize reward, an agent clearly needs to prefer actions that it has tried already and has found to be good. However, to discover such actions the agent has to try out actions that it has never taken before. One way to deal with this trade-off is the epsilon-greedy method. The greedy action in a state is the action that is believed to be optimal in that state. If we use the epsilon-greedy method, the agent takes, in every state, the greedy action with probability 1-epsilon, and a random action with probability epsilon. This ensures that we use the information that we already have (exploit), but also look for even better information (explore) [Sutton and Barto, 2018].

C. Importance sampling

The epsilon-greedy method is a compromise. It does not learn action values for the optimal policy, but for an almost optimal policy that still explores. An alternative approach is to use two policies: one that is learned about and converges to optimality (target policy), and another one that is exploratory and is used to actually choose actions (behaviour policy). This approach is called off-policy learning. Almost all off-policy methods use importance sampling, which is a way of estimating expected values under one distribution (target) given samples from another (behaviour). The importance sampling ratio, that is, the relative probability of a trajectory under the target and the behaviour policy, is given by

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|Sk, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k) p(S_{k+1}|Sk, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{1}$$

We can use this ratio to transform the returns obtained through the behaviour policy into an estimate for the value function under the target policy. To estimate $v_\pi(s)$, we scale the observed returns by the ratios and average the results as follows.

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:\mathcal{T}(t)-1} G_t}{|\mathcal{T}(s)|} \tag{2}$$

Where Tau of s is the set of all time steps in which state s is visited (every-visit) or the set of time steps that were first visits to s within their episodes (first-visit). And where G is the Monte Carlo (MC) return.

An alternative to this 'ordinary' importance sampling is weighted importance sampling, defined as follows

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:\mathcal{T}(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:\mathcal{T}(t)-1}} \tag{3}$$

Note that first-visit ordinary importance sampling is unbiased, but that its variance can be (very) high. Weighted importance sampling is biased (although the bias converges asymptotically to zero), but generally has lower variance. Sutton and Barto [Sutton and Barto, 2018] write that, in practice, the weighted variant usually has much lower variance and is therefore strongly preferred. This claim is the starting point of our experiments. And by now, the relevance of these experiments should be clear: Off-policy methods are more powerful and general than on-policy methods. But they need importance sampling. So, it is relevant to know which kind of importance sampling gives us the best results in practice.

D. Related Work

D. Precup[Precup, 2000] compared four different sampling techniques, namely ordinary importance sampling, weighted importance sampling, per-decision importance sampling and weighted per-decision importance sampling on 100 randomly constructed MDP's, where the action space consisted of two distinct actions. The target policy was choosing the first action with 80 percent probability and the second action with 20 percent. Two different behaviour policy were used: the uniform policy and a policy exactly opposite to that of the target policy. They empirically showed that ordinary importance sampling exhibits high variance, is less stable and converges slower.

E. Compared techniques

To answer our research question Monte Carlo and Temporal Difference methods are implemented. Where in the MC case we distinguish between ordinary and weighted importance sampling. For an in depth discussion of the difference between MC and TD, see [Sutton and Barto, 2018]. For the present research, it is enough to note that MC algorithms update $V(S_t)$ towards the observed return starting at S_t as follows.

$$V(S_t) = V(S_t) + \alpha[G_t - V(S_t)] \tag{4}$$

And the TD(0) algorithm updates $V(S_t)$ towards the TD error δ as follows.

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{5}$$

So we compared the performance of the following three different methods to learn a value function.

- Off-policy MC with ordinary importance sampling
- Off-policy MC with weighted importance sampling
- Off-policy TD with importance sampling

Both every-visit and first-visit MC are included in this research. Notice that no distinction is made between ordinary and weighted importance sampling in the off-policy TD case. The reason for this will be clear after inspecting the n-step TD update, given by

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha \rho_{t:t+n-1}[G_{t:t+n} - V_{t+n-1}(S_t)] \tag{6}$$

where the importance sampling ratio $\rho_{t:t+n-1}$ is the relative probability under the two policies of taking the n actions from A_t to A_{t+n-1} and with $0 \leq t < T$. To make equation 6 match equations 2 and 3, the learning rate α would have to be substituted with the inverse of N or the the inverse of the sum of importance weights so far respectively. However, this would mean that the learning rate will shrink as the number of updates increase. This imposes no problems in the case of MC, since updates are always done towards the real return. However, when bootstrapping, updates in the beginning will be made towards [r + a very bad estimate of v_π] with a high α, and later towards [r + a better estimate of v_π] with a (much) smaller α. This is clearly not desirable.
So, the TD algorithm will not directly help us answer our research question. But we still included it as a baseline. So that we had something to compare our two MC variants to.

III. Methods

We run the three above discussed algorithms on different existing environments and compare their respective performance.

A. Environments

In this research only the tabular case (no function approximation) is considered. Relatively simple environments were chosen:

- Blackjack-v0. A card game where the goal is to obtain cards that sum to as near as possible to 21 without going over. The player can request additional cards (hit=1) until they decide to stop (stick=0) or exceed 21 (bust) [1].
- FrozenLake-v0. The agent moves in a grid world towards a goal state. Some tiles are walkable, others lead to the agent falling into the water (and thereby restarting the episode). In the slippery variant, the agent does not always move in the intended direction [2]. The non slippery variant is included in this research, due to it's deterministic property.
- NChain-v0. The agent moves along a linear chain of states, with two actions. Forward, which moves along the chain but returns no reward. And backward, which returns to the beginning and has a small reward. The end of the chain, however, presents a large reward. At each action, there is a small probability that the agent 'slips' and the opposite transition is instead taken [3].

In this research the possibility of slipping is omitted from experiments; the agent will always move in its intended direction.
For Blackjack a simple strategy that sticks with 20 or 21 points and hits otherwise was chosen. For Frozenlake the shortest path was chosen as a policy and for Nchain an epsilon greedy variant of a policy that always moves to the right was chosen. The policies were chosen for there simplicity. The behaviour policy throughout this research is the random policy.

B. Hyperparameters

For each environment grid search was used to find optimal parameters for TD(n), which included a learning rate α and the value for n. Gridsearch was done by using root mean squared error with respect to the true value function. The combination of parameters with the lowest mean where chosen.

This resulted in the following parameters:

- Frozenlake: $\alpha = 0.01, n = 4$
- Blackjack: $\alpha = 0.01, n = 4$
- 5-chain: $\alpha = 0.001, n = 8$
- 10-chain: $\alpha = 0.001, n = 8$
- 15-chain: $\alpha = 0.1, n = 1$

For the other algorithms no hyperparameters needed to be tuned.

C. Measured quantities

We measured three things: 1) how fast did the value function converge?, 2) how good was the converged value function? 3) how much bias and variance was there? To answer these questions, we made a plot of the amount of episodes (x-axis) and the (root) mean squared error (RMSE) (y-axis). To calculate the RMSE we compared the value function that we learned off-policy, to the true value function (the baseline) v_π for target policy π. (Where we obtained v_π using on-policy MC.) Given this plot, answers to the first two questions are easily read off: when the RMSE does not change anymore, the value function has converged. And the quality of the converged value function is given by its RMSE to v_π. We also measured the bias and variance of the three algorithms.

D. Amount of random runs

For every environment, the different algorithms under investigation were performed on ten random runs. Allowing for the capturing of variance and stability.

IV. Implementation

All implementations were done in Python and available on a publicly available GitHub repository[4]. Environments were provided by Gym [Brockman et al., 2016]. The off-policy algorithms, described in section E, are based on the provided pseudo-code in [Sutton and Barto, 2018].

V. Results

The results of the previously elaborated experiments for Blackjack and Frozen Lake are visible in figure 1 and 2 respectively. For the NChain environment experiments are conducted on different chain lengths, that is 5, 10, 15 and 20. Results with chain length of 5 are visible in figure and 3; for the results of the other lengths we refer you to the appendix. Monte Carlo with ordinary sampling exhibits higher variance and slower convergence than its weighted variant in the NChain environment (for a chain length smaller than 20). This is in line with the findings of D. Precup [Precup, 2000]. Both MC sampling methods

[1]https://gym.openai.com/envs/Blackjack-v0
[2]https://gym.openai.com/envs/FrozenLake-v0
[3]https://gym.openai.com/envs/NChain-v0

[4]https://github.com/jaapjorisjaap/rl2020-project

in the other environments exhibit similar variance and performance after converging. It is also noticeable that Monte Carlo with weighted importance sampling, in all but the Blackjack environment, converges to the true value as its bias is zero. The TD algorithms (with their optimal n) perform the worst out of all the methods in all but one, showing an unstable learning curve with a high spiky RMSE over episodes in 2 and 3. From figure 6 in the Appendix, we can conclude that for long chains TD outperforms the Monte Carlo methods for both ordinary importance sampling and weighted importance sampling.

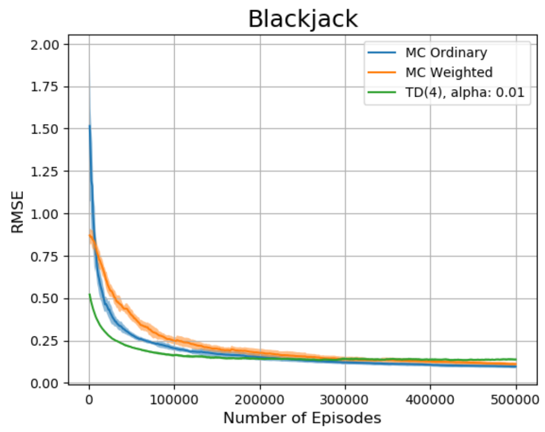

Fig. 1. 10 runs where done with different random seeds. The lines represent the mean RMSE for each algorithm. The shaded area represent the spread, one standard deviation around the mean.

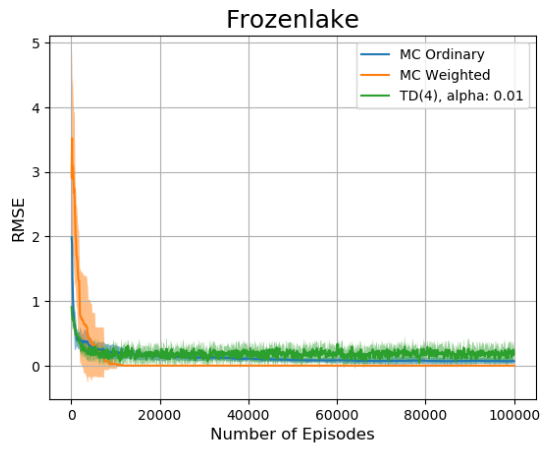

Fig. 2. no slipage, 4×4, $\epsilon = 0.01$, 10 runs where done with different random seeds. The lines represent the mean RMSE for each algorithm. The shaded area represent the spread, one standard deviation around the mean.

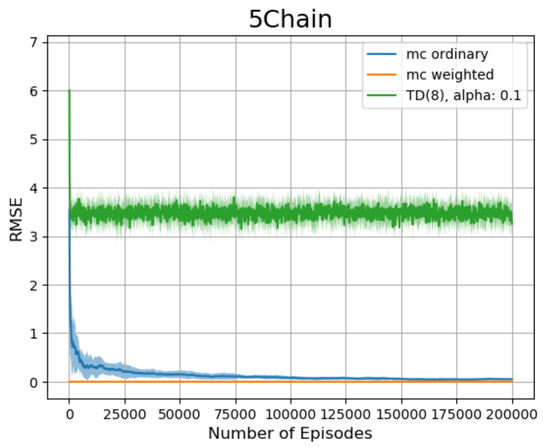

Fig. 3. NChain with termnial states beyond the chain, $N = 5$, $\epsilon = 0.01$. 10 runs where done with different random seeds. The lines represent the mean RMSE for each algorithm. The shaded area represent the spread, one standard deviation around the mean.

VI. Conclusion

In this research we analysed the the impact ordinary importance sampling and weighted importance sampling have on the performance, convergence and variance of every visit Monte Carlo algorithms. TD-learning is used as a baseline, where no distinction is made between ordinary and weighted sampling since this would yield unwanted results (see section E). These algorithms were compared on three different environments: Blackjack, Frozen Lake and NChain. This research emperically shows that weighted importance sampling converges faster, exhibits less variance and lower bias than ordinary importance sampling in the NChain environment for a chain length smaller than 20. In the other environment no such evident conclusions could be made, as they exhibit similar variance and performance. Monte Carlo methods with weighted and ordinary importance sampling outperform TD-learning with importance sampling in almost all environments, in addition to being more stable in their learning curve. When however a reward is delayed over many time steps, as is in the NChain environment ($n = 20$), TD outperforms the Monte Carlo methods.

References

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

[Precup, 2000] Precup, D. (2000). Eligibility traces for off-policy policy evaluation. Computer Science Department Faculty Publication Series, page 80.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

Appendix

A. Performance in the NChain environment for different chain lengths

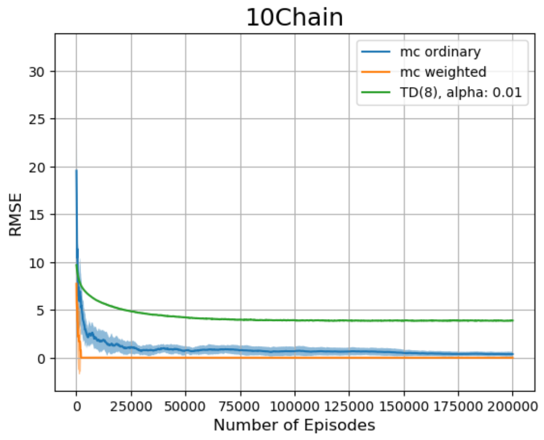

Fig. 4. NChain with termnial states beyond the chain, $N = 10$, $\epsilon = 0.01$. 10 runs where done with different random seeds. The lines represent the mean RMSE for each algorithm. The shaded area represent the spread, one standard deviation around the mean.

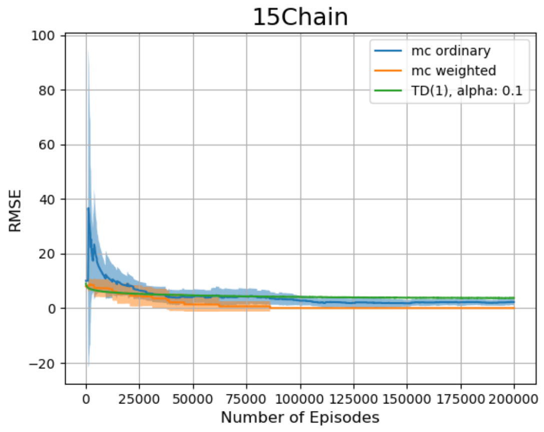

Fig. 5. NChain with termnial states beyond the chain, $N = 15$, $\epsilon = 0.01$. 10 runs where done with different random seeds. The lines represent the mean RMSE for each algorithm. The shaded area represent the spread, one standard deviation around the mean.

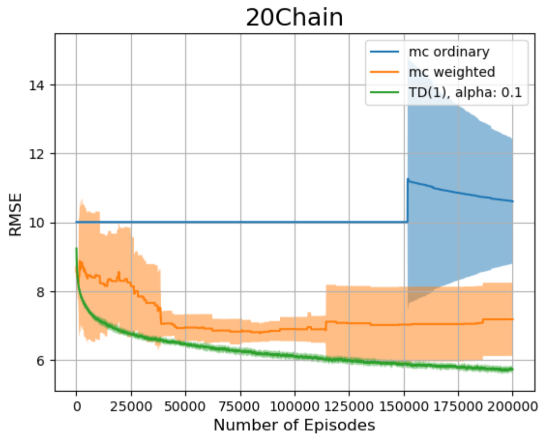

Fig. 6. NChain with termnial states beyond the chain, $N = 20$, $\epsilon = 0.01$. 10 runs where done with different random seeds. The lines represent the mean RMSE for each algorithm. The shaded area represent the spread, one standard deviation around the mean.