Reinforcement Learning for Sports: Cricket

COMP 767:

Julyan Keller-Baruch, julyan.keller-baruch@mail.mcgill.ca, 260388157
Navin Mordani, navin.mordani@mail.mcgill.ca, 260744902
MohammadReza Davari, mohammadreza.davari@mail.mcgill.ca, 260587820

Abstract

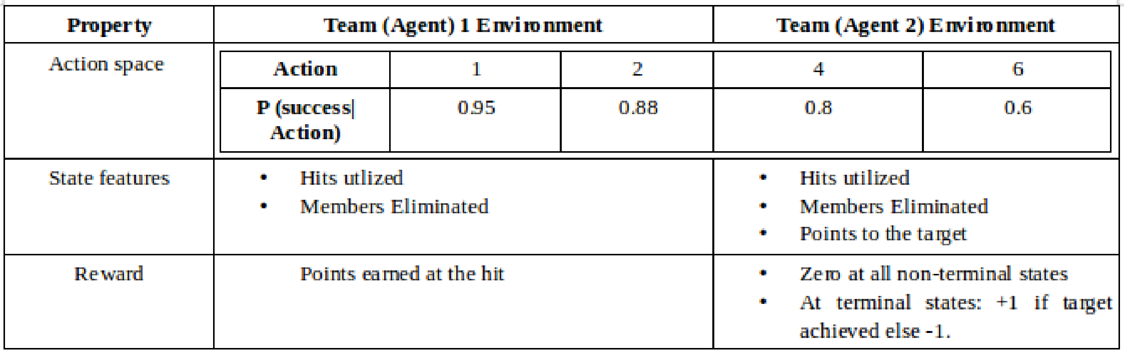
Reinforcement learning has already made its mark of expertise in two player strategy games such as chess and GO that require sequential decision making. The game of cricket is also a game between two high level entities (teams) and involves sequential decision making under uncertainty. It requires customizing one's strategy according to the situation they are facing. At the heart of it, we can view it with similar reinforcement learning problem formulation where the teams can be modeled as agents. The difference however, is in the knowledge that the two agents carry at a point in the game supporting their actions is not exactly the same giving rise to the need for formulating two separate environments for one game and training two separate agents unlike regular minimax or zero-sum game.

1 Introduction

In this project we have used reinforcement learning to model the sequential decision making that goes into the game of cricket.

1.1 Overview of the Game of Cricket

At a high level overview the game of cricket can be seen as played between two entities Team 1 and Team 2. Team 1 goes for it's batting turn first, sets a score, and then Team 2 goes for it's batting turn and tries to beat the score set by Team 1 which in cricketing terms is called as the target set by Team 1 for Team 2 to chase. If Team 2 is successful in beating the score or achieving the target it wins the game otherwise Team 1 is the winner. In a batting turn the team gets a fixed number of opportunities to hit the ball and obtain points. The sum of the points obtained at each of these hits amounts to the score at the end of the turn. At each of these opportunities a member of the team can hit the ball with various levels of aggression. More the aggression, more are the points on success and more is the risk of failure. There are four different levels of aggression giving points of 1, 2, 4, and 6 on success with increasing uncertainty of success. A failure of a hit is bad as it causes the member of the team hitting the ball to be eliminated. If a team has all it's members eliminated then it's batting turn comes to an end even though it may not have utilized it's entire quota of hits. Hence, a batting turn can come to an end for Team 1 if they have played their entire quota of hits or all of it's members have been eliminated. In addition to the above two ways the batting turn for Team 2 can also come to an end if it has achieved the target. One cannot be wrong in saying that Team 1 increases it's chances of winning if it sets very high targets that are difficult for Team 2 to achieve. But, given the uncertainty associated with the different aggression levels with which a member can choose to hit the ball, different situations can arise and Team 1 has to customize it's strategy and decide how aggressively or defensively it needs to play so that it can maximize it's score from the given situation. Team 2 also has to customize it's strategy throughout it's batting turn and play with the right level aggression depending on it's current situation and the target it has to achieve. A team has to decide on the aggression level depending on how much it values points and fears elimination given a situation. For instance if Team 1 is to face it's last hit it may be a good strategy to go for the highest aggression level and aim for 6 points as its batting turn is anyways coming to an end after the hit. Similarly, if Team 2 is in a situation where it needs 1 point to win the game and is only one member away from facing the elimination of all it's members then it is the best strategy to go for the safest hit and look to obtain only 1 point



Property	Team (Agent) 1 Environment			Team (Agent) 2 Environment	
Action space	**Action**	1	2	4	6
	P (success| Action)	0.95	0.88	0.8	0.6
State features	• Hits utlized • Members Eliminated			• Hits utilized • Members Eliminated • Points to the target	
Reward	Points earned at the hit			• Zero at all non-terminal states • At terminal states: +1 if target achieved else -1.	

Figure 1: Outline of the Environments for Team 1 (Agent 1) and Team 2 (Agent 2).

and win the game rather than going for hits that have higher chances of causing elimination and in turn loss of the game. The nature of the sequential decision making is a good indicator that the problem can be well formulated as a reinforcement learning (RL) problem.

2 Methodology

2.1 Environments

The game is played between two teams namely, Team 1 and Team 2, and both these entities have a slightly different information through their batting turns. The Team 1 has information about how many hits it has already utilized and how many members of it's team are eliminated whereas Team 2 along with these two pieces of knowledge also has a third piece of information which tells it how much more points it needs to obtain to achieve the target and win the game. Also, the goal of both these entities is different Team 1 has a goal as mentioned earlier to set targets as high as possible and make it harder for Team 2 to achieve them. On the other hand Team 2 has a fixed target in front of it to achieve throughout it's batting turn and it's sole purpose is to achieve that. Owing to the fact that both teams have different bits of information and goal guiding their decision making through their batting turn, if modeled as RL agents these would need two different environments. Figure 1 shows the outline of both the environments. The action space remains the same for both the environments. Here, to keep the matter simplified we have kept the action numbers or labels to be the same number as the maximum number of points associated with the action. The probability of an action being successful is also mentioned and are in a decreasing trend with the increase in the points they offer on success. Coming to the reward function for Team 1's environment a reward upon an action is the number of points earned and hence the return at the end of an episode in this case a batting turn is the score set by the team. On the other hand for Team 2 all the non-terminal states have a reward of zero. The terminal states give a reward of +1 if the target is achieved otherwise the reward associated with the terminal states is -1.

The important point to note here is that the environments that have been implemented can be used to initialize instances of environments with different values of state features so that one can test their methods. For the purpose of this project we have used the instance of the game with 6 hits per batting turn and 2 members per team to train our agents to play.

2.2 Learning for Team (Agent) 1

As seen in section [2.1] the state space for agent 1's environment is defined by two variables indicating the number of hits utilized and the number of members eliminated prior to the current situation. In our case dealing with a game having 6 hits per turn and 2 members per team as mentioned earlier makes the state space for agent 1 to be tractable enough for the use of tabular methods for learning. We used the methods:

- Q(λ)-learning
- n-step double Q-learning
- n-step double weighted Q-learning [ZPK17]

2.2.1 Q(λ)-learning

Q(λ)-learning is a variant of Q-learning with eligibility traces for better handling of credit assignment and bridging the gap between one -step and Monte-Carlo backup to settle for a better bias variance trade-off. We use the Watkin's Q(λ) [Wat89] version here. After every step of the episode the update rule for Q(λ) as shown in equation [1] below is used to update the Q value for every state action pair(s,a):

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t e_t(s,a) \tag{1}$$

Where:

- $\alpha \in [0,1]$ is the learning rate
- $\delta_t = r_{t+1} + \gamma \max_{a'}(Q_t(s_{t+1}, a')) - Q(s_t, a_t)$ with r_{t+1} being the reward at time $t+1$
- $\gamma \in [0,1]$ is the discount factor
- λ is the trace decay parameter and can range from 0 to 1
- $e_t(s,a)$ is the eligibility trace vector updated in the following manner:

$$e_t(s,a) = \begin{cases} 1 + \gamma\lambda e_{t-1}(s,a) & \text{if } s = s_t, a = a_t, Q_{t-1}(s_t,a_t) = \max_a Q_{t-1}(s_t,a) \\ 0 & \text{if } Q_{t-1}(s_t,a_t) \neq \max_a Q_{t-1}(s_t,a) \\ \gamma\lambda e_{t-1}(s,a) & \textit{otherwise} \end{cases}$$

As seen from the above equation [1] for Watkin's Q(λ) the eligibility trace is set to zero at the first non-greedy action. This is done so that there are no backups for a non-greedy actions.

2.2.2 Double Q-learning

The Q-learning method uses a single estimator to estimate the maximum expected value of $Q(s',a')$ i.e. $\max_{a'} \mathbf{E}\{Q(s',a')\}$. It estimates this by calculating $\max_{a'} Q(s',a')$ which is a sample drawn from a probability distribution with expectation of $\mathbf{E}\{\max_{a'} Q(s',a')\}$ [ZPK17]. As $\mathbf{E}\{\max_{a'} Q(s',a')\} \geq \max_{a'} \mathbf{E}\{Q(s',a')\}$ the Q-learning estimate suffers from a maximization bias or a positive bias as explained in [SB98]. In our setting the reward function is stochastic and Q-learning can suffer from maximization bias and overestimate Q values of state action pairs resulting in the choice of suboptimal actions. Hence, we thought it is worth trying the double Q-learning approach. Double Q-learning algorithm as mentioned in [SB98] unlike Q-learning does not use the same set of samples to find the maximizing action and to estimate it's value. It keeps two Q functions namely Q_1 and Q_2 and learns them using two separate sets of samples. Here we use the n-step version of double Q-learning. In the n-step version the Q value update of a state action pair is based on the target calculated by summing the next n discounted rewards and the accordingly discounted estimated Q value of the state and the maximizing action n steps later. After every step in an episode one of the two Q functions is randomly selected to be updated by this method and the estimate of the state and maximizing action n-steps later is given by the other Q function. The maximizing action n-steps later is determined based on Q function whose value is to be updated. For example if Q_1 is to be updated for state action pair that occurred at time t ($s_t = s, a_t = a$) then the update rule given below in the set of equations [2] needs to be used:

$$\boxed{\begin{array}{l} G_{t:t+n} = \sum_{i=t+1}^{\min(t+n,T)} \gamma^{i-t-1} r_i \\ G_{t:t+n} = G_{t:t+n} + \gamma^n Q_2(s_{t+n}, arg\max_a Q_1(s_{t+n}, a)) \text{ if } (t+n) < T \\ \delta = G_{t:t+n} - Q_1(s_t,a_t) \\ Q_1(s_t,a_t) = Q_1(s_t,a_t) + \alpha\delta \end{array}} \tag{2}$$

Where,

- T is the time-step when the episode terminates
- γ is the discount factor
- r_i is the reward encountered at time step i
- The other symbols have the same meaning as they had for equation [1]

2.2.3 Double Weighted Q-learning(DWQ learning)

DWQ learning introduced recently in [ZPK17] is an intermix of Q-learning and double Q-learning. Double Q-learning deals with the overestimation arising due to the maximization bias in case of Q-learning but suffers from negative bias as shown in [ZPK17]. Double Q learning as shown in section [2.2.2] finds the maximizing action a^* using the Q function that is to be updated, without loss of generality let us say we update Q_1 and the state whose maximum expected value is to be estimated as $s^{'}$. Therefore, $a^* = arg\max_a(Q_1(s^{'}, a))$. However, to update Q_1 the estimate of the Q value of the pair $(s^{'}, a^*)$ $Q_2(s^{'}, a^*)$ is used which is estimated using different set of samples than those used to estimate Q_1. Hence, $Q_2(s^{'}, a^*)$ is a sample from the probability distribution with expectation $\mathbf{E}\left[Q_2(s\prime, a^*)\right] = \mathbf{E}\left[Q(s', a^*)\right]$ and $\mathbf{E}\left[Q_2(s', a^*)\right] \leq \max_a \mathbf{E}\left[(Q(s', a))\right]$ which leads to the estimations suffering from negative biases. To deal with the overestimation of Q-learning and underestimation of double Q-learning DWQ learning comes to the rescue. For DWQ learning, we also have adapted the n-step version as we did in the case of double Q-learning. DWQ learning like double Q-learning stores two separate Q functions Q_1 and Q_2. At a given time step randomly chooses to update any one of them with equal probability. Similar to the other two approaches the target towards which the Q function selected to be updated is determined as a sum of the discounted rewards occurring in the next n-steps and an estimation of the value of the state occurring after n-steps discounted appropriately. What differs is how the value of the state which is n-steps away and whose maximum expected value is to be estimated is derived. Let us say without loss of generality if we want to update the Q1 function then the state update is as follows:

$$\boxed{\begin{array}{l} a^* = arg\max_a Q_1(s,a) \\ a^L = arg\min_a Q_1(s,a) \\ \beta^1 = \frac{|Q_2(s,a^*)-Q_2(s,a^L)|}{c+|Q_2(s,a^*)-Q_2(s,a^L)|} \text{ where } c \geq 0 \\ G_{t:t+n} = \sum_{i=t+1}^{\min(t+n,T)} \gamma^{i-t-1} r_i \\ G_{t:t+n} = G_{t:t+n} + \gamma^n \left[\beta^1 Q_1(s,a^*) + (1-\beta^1) Q_2(s,a^*)\right] \text{ if } (t+n) < T \\ \delta = G_{t:t+n} - Q_1(s_t,a_t) \\ Q_1(s_t,a_t) = Q_1(s_t,a_t) + \alpha\delta \end{array}} \tag{3}$$

Where, the symbols have the same meanings as in equations [1] and [2]. We can see that the update target that DWQ learning estimates involves a weighted average of the two Q functions and this helps to eliminate the problem of overestimation and underestimation caused in the cases of Q-learning and double Q-learning. The weighting is controlled by the parameter c. On the two extremes for $c = 0$ and $c = \infty$ lie the Q-learning and double Q-learning updates. And on changing the parameter c one can settle for an intermix of Q-learning and double Q-learning.

2.3 Learning for Team (Agent) 2

In section [2.1], we have mentioned that the state space for Agent 2 is one dimension larger than that of Agent 1. This extra dimension corresponds to how far the current score is from the target. This results in a much larger number of possible states and makes it difficult to solve the problem using tabular methods. Under our setting, with the agent having 2 members per team and 6 hits, there are more than 350 states in the state space of this agent as in our setting the feature "points to target" can range from 0 to 36(reward of 6 on every hit) both inclusive. It is important to note that the number of states in the state space of this agent will exponentially increase if it is given more members and more hits. This motivated us to explore non-tabular methods for the learning of this agent. We have considered the following two methods:

- Policy Gradient Method
- Kernel-Based Reinforcement Learning Method

2.3.1 Policy Gradient Method

A policy gradient method allows for the need of fewer parameters when learning in a large or continues state space. To learn an optimal behavior for the second agent, we employed and actor-critic method which combines the idea of policy gradient and value function approximation [SB98]. The idea here is that at every step the actor (policy) can evaluate if the action it took was good or

bad according to the critic. Then the parameters will be adjusted to perform that action more or less often depending on whether the action was good or bad respectively. The actor-critic method used here takes advantage of eligibility traces on both actor and critic to assign credit to the most recent and most frequent features. The motivation behind the use of the eligibility traces for this method was to adjust the actor (policy) in the direction that all the critics think is better.
The actor is defined as:

$$\pi(a|s,\theta) = softmax(h(s,a,\theta))$$

Where h is the parameterized numerical preferences for each state-action pair. We define h to be:

$$\theta^T x(s,a)$$

This means that we define our preferences to be linear in features. Where the feature vector x is simply given by:

$$[1, s, 1, 0, 0, 0]$$

$$[1, s, 0, 1, 0, 0]$$

$$[1, s, 0, 0, 1, 0]$$

$$[1, s, 0, 0, 0, 1]$$

For action 1, 2, 3, 4 respectively. The vector s here indicates the state of the agent. Note that under this definition we have:

$$\nabla_\theta \ln\pi(a|s,\theta) = x(s,a) - \sum_b \pi(b|s,\theta)x(s,b)$$

The critic function $\hat{v}(s,w)$ is given by $w^T x(s)$ where $x(s)$ is the degree 1 polynomial feature of state s, therefore we have:

$$x(s) = [1, s]$$

Note that in this definition we have:

$$\nabla_w \hat{v}(s,a) = x(s)$$

The algorithm we used can be formulated as below:

```
For each episode:
    Initialize $S$ (the first state of the episode)
    $z^\theta \leftarrow 0$ (eligibility trace vector)
    $z^w \leftarrow 0$ (eligibility trace vector)
    $I \leftarrow 1$
    Loop until $S$ is the terminal state:
        $A \sim \pi(S,\theta)$
        Take action $A$, observe $S', R$
        $\delta \leftarrow R + \gamma\hat{v}(S',w) - \hat{v}(S,w)$
        $z^w \leftarrow \gamma\lambda^w z^w + I\nabla_w \hat{v}(S,w)$
        $z^\theta \leftarrow \gamma\lambda^\theta z^\theta + I\nabla_\theta \ln\pi(A|S,\theta)$
        $w \leftarrow w + \alpha^w \delta z^w$
        $\theta \leftarrow \theta + \alpha^\theta \delta z^\theta$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
```

where,

- γ is the discount factor
- λ is the trace decay parameter ranging in $[0, 1]$

2.3.2 Kernel-based Reinforcement Learning

In this section we propose the use of kernel-based reinforcement learning [OS02] to model the act of chasing targets for Agent 2.

Kernel-based reinforcement learning is a non-parametric method that assigns value function estimates to states along a trajectory and iteratively updates these estimates until convergence is reached. This estimated value function can then be used to infer the value of unseen states, assuming that the user can define a similarity metric, or a kernel, that effectively maps new observations to this approximated value function. In doing so, an agent can act successfully in an environment by only relying on the data acquired from this sample trajectory.

As mentioned, the user must define a metric by which similarity will be measured across states. For our experiments, we used the additive chi-square kernel [PVG+11] which gives a measure of similarity between two vectors x and y as:

$$k(x, y) = -\sum_i \frac{(x[i] - y[i])^2}{x[i] + y[i]} \tag{4}$$

We begin collecting data by initializing the environment with a random target between 0 and 36 and allow the agent to perform randomly until a set of m transitions and immediate rewards have been collected for each action $\in \{0,1,2,3\}$.

Once collected, we build two matrices with these transitions and rewards. The first, Θ, is of dimension $m \times 4 \times m$ and is composed of stacked matrices of size $m \times m$ each representing the similarity between each starting point and each end point across all sampled transitions. This similarity is computed by using a user-defined kernel. In our case, we used the 'additive chi-square kernel' to define this similarity across states. The second matrix $\hat{R}$ is simply a $m \times 4$ matrix representing rewards for each transition, per action.

Next, a user defines a discount factor and initializes, 'current' $\hat{J}$ and 'subsequent' $\hat{J}'$ value functions to zero matrices of size $m \times 4$ and performs value iteration using the following update rule until convergence is reached.

$$\hat{J}' := \max_{axis=2} \left(\Theta\big[\hat{R} + \gamma \hat{J}\big]\right) \tag{5}$$

Where $\max_{axis=2}$, represents an operator that takes a $m \times 4 \times 4$ matrix maximizes over the second axis and returns a $m \times 4$ matrix, that is the new value function $\hat{J}$ representing a value for each of the four actions across all of the sampled states.

At this point, our value function has been approximated and the action-value of a new unseen sample x can be inferred using the following function:

$$J_a(x) \quad = \sum_{(s,s',r)\in S^a} k(x, s)\big[r + \gamma J(s')\big] \tag{6}$$

Where $(s,\ s')$ and r are transition-reward pairs from the set of observed samples and k is the same kernel function used to generate Θ. This outputs a single value representing the value of taking action a in state x. We then return a list of four values corresponding to each of the actions and the agent selects the action corresponding to the maximum value of this list.

3 Results

3.1 Results for Agent 1

As mentioned earlier in section [2.2] we used Q(λ), n-step double Q-learning and n-step double weighted Q-learning methods for agent 1. The hyper-parameters for the methods were chosen by comparing the root mean square error(RMSE) between the state values $V(s)$ estimated by the methods and the true $V_*(s)$ obtained by solving the Bellman equation. Table [1] shows the set of hyper-parameters for each of the methods that gave the lowest RMSE values. To settle down on the hyper-parameter values the performance measured on the RMSE metric was averaged over 10 runs. The comparison using the RMSE values amongst different set of hyper-parameters for the methods Q(λ) and n-step double Q-learning is shown in Figure 2. Note that such a plot for the n-step double weighted Q-learning method is not presented here as the plot has a very large

Table 1: Tabular methods used for agent 1 along with their best performing parameters according to the RMSE metric

Sr. No	Method	Hyper-parameters
1.	Q(λ)	$\alpha = 0.2$, $\lambda = 0.8$
2.	n-step double Q	$\alpha = 0.2$, $n = 4$
3.	n-step double weighted Q	$\alpha = 0.2$, $n = 3$, $c = 1$

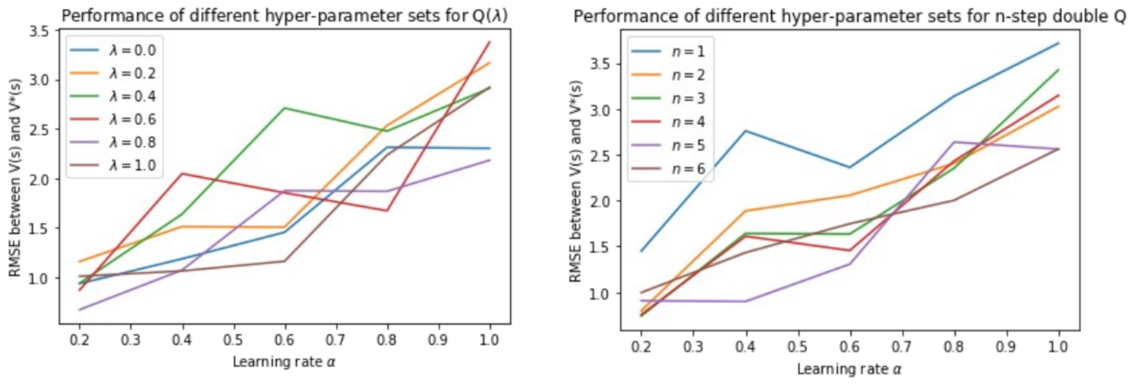


Figure 2: Comparison of performance between different hyper-parameter sets for Q($|lambda$) and n-step double Q-learning.

number of curves as there are more hyper-parameters to fit making it complicated to follow for the viewer.

The performance for the best hyper-parameter sets of the three methods can be seen in Figure [3]. The metric used here is the average score per episode or also simply called as the running average averaged over 10 runs. The bottommost orange dashed line shows the mean score that agent achieves upon playing according to a equiprobable random policy. The topmost blue dashed line gives the mean score that the agent achieves when playing according to the optimal policy derived by solving the Bellman equation. The 3 curves in between these two dashed lines correspond to the our methods. We see that n-step double weighted Q performs the best amongst all the three and Q(λ) performs the worst.

3.2 Results for Agent 2

3.2.1 Policy Gradient Method

The performance of this approach in learning for agent 2 can be investigated in Figure [4]. As seen form the figure, the agent learns to win about 46.25% of the time which in comparison to the random policy with 12.5% of the time wining is about 3 times more improvement. The agent here is almost wining and losing half the time. This indicates that the agent is almost at an equilibrium with respect to the first agent and given the randomness of the environment that is an ideal state to be at.

Note that the Actor-Critic method we used to train the agent has 5 hyper parameters. The set of hyper-parameters used to produce the results for this method, was chosen by an exhaustive grid search mechanism. The set of hyper-parameters which resulted in a highest average (over 10 runs) score (after 1000 episodes) was chosen to proceed the learning with. The chosen values for the hyper-parameters are listed in the table [2]:

Table 2: Policy gradient hyper-parameters which performed best in the learning of the agent 2

γ	λ^θ	λ^w	α^θ	α^w
0.1	0.5	0.5	0.0001	0.001

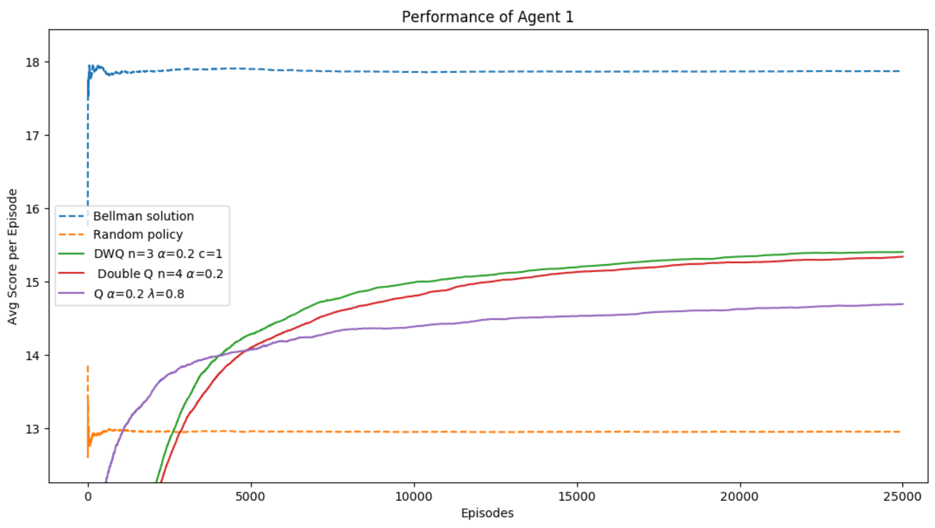


Figure 3: Comparison of performance between Q(λ), n-step double Q-learning, n-step double weighted Q-learning (DWQ), random policy and optimal policy.

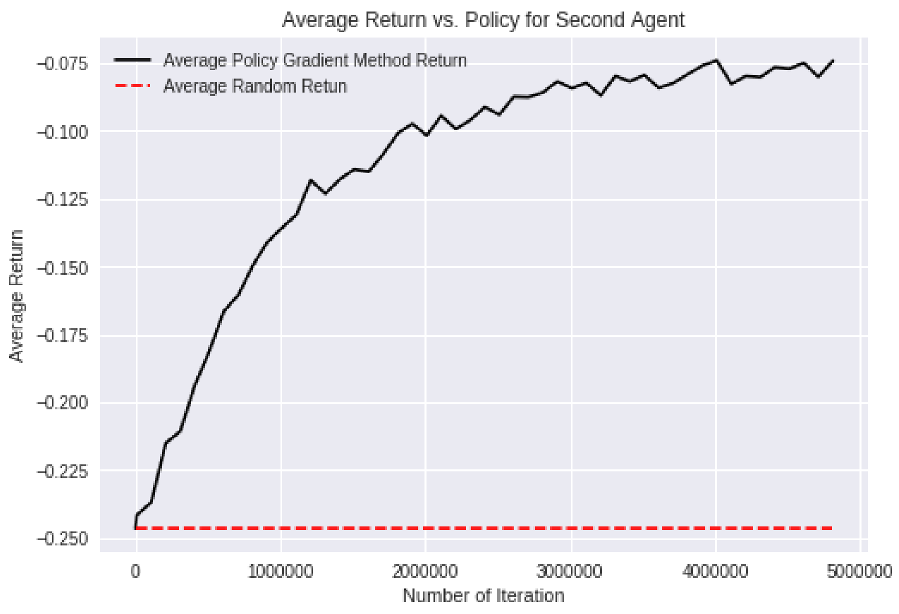


Figure 4: Average performance of Actor-Critic method in comparison to the random policy.

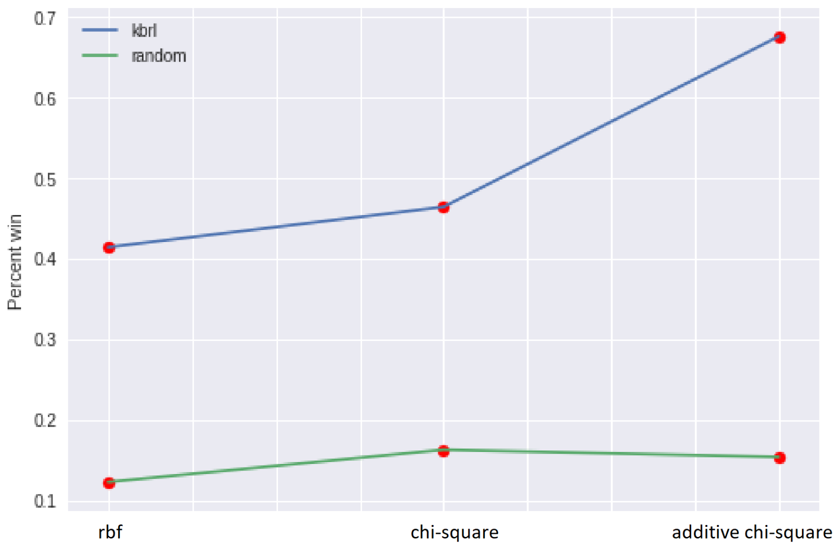


Figure 5: Percent win over 1000 games for three different kernels: radial basis function (rbf), chi-square and additive chi-square.

3.2.2 Kernel-based Reinforcement Learning

In this section, performance is measured as the percentage of wins over 1000 games. First, we defined a set of kernels that were to be tested with our algorithm. Of the kernel functions tested, three resulted in behavior that performed better than the random: radial basis function, chi-square and additive chi-square kernels, yielding percent wins of 0.41, 0.46 and 0.68, respectively. Performance was highest using the additive chi-square (Figure 5) and thus we opted to use this kernel for the remainder of our experiments.

Next, we randomly explore the environment and collect data to iteratively approximate the value function. For each of the four actions in the agent's action space, we randomly sampled m transitions in an environment with a target that is randomly re-initialized, upon termination of episode, to a value between zero and 36. This repeats until all action-transition-lists are of length m. We randomize the target with which we initialize the environment to diversify the state-transitions that we obtain at the sampling step. The effect of varying the number of sampled transitions m is depicted in Figure 6 . Increasing the number of transitions sampled beyond 100 does not seem to improve maximal performance of the agent. In light of this observation, we chose to continue the remainder of our analysis using $m = 100$ for computational efficiency. Although 100 data points seems to be sufficient for learning in most experiments, it appears that in some cases the data gathering does not acquire enough relevant information to converge to a value function and ultimately a policy that will lead the agent to choose winning actions. Although increasing the sample size does not increase the maximal performance, it may remedy the above-mentioned issue by guaranteeing that the approximate value function is based on a larger, more diverse set of samples.

To perform value iteration on the matrices built from sampled transitions, we must define a discount factor γ. We tested a range of γ within 0.1 and 1.0 and did not observe a significant effect of γ on performance (Figure 7). We therefore arbitrarily selected our value of γ to be 0.95 for all other experiments.

Finally we tested the effect of increasing the target score of the agent. We tested targets scores ranging from 0 to 50, of which two target scores, 40 and 50, are unattainable by the agent. That is, given the starting conditions of the environment (a maximum of 6 hits), the maximal score that an agent can attain is 36. Figure 8 depicts the performance of the agent over these target scores. As expected, both the KBRL and random agents perform quite well for target scores near zero. As we increase the target score toward unattainable goals, the performance of the random agent quickly tapers off while that of the KBRL agent does so at a slower rate. For targets 40

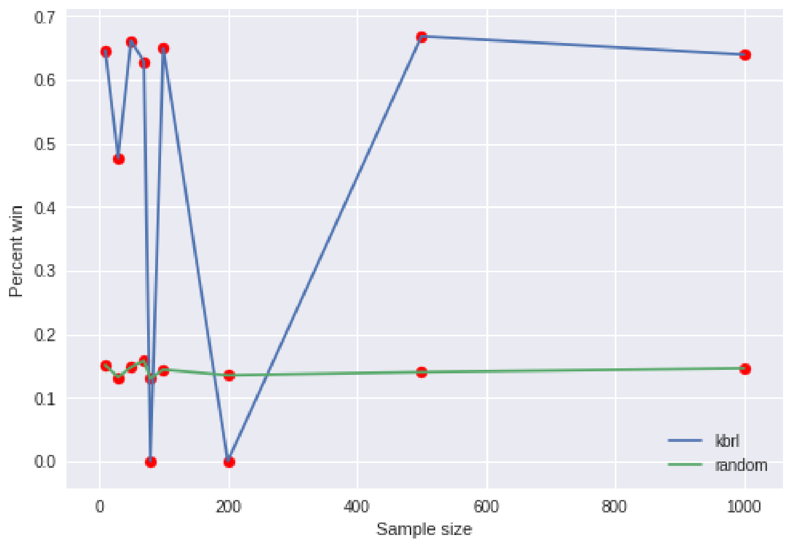


Figure 6: The effect of increasing the number of sampled transitions at the data acquisition step.

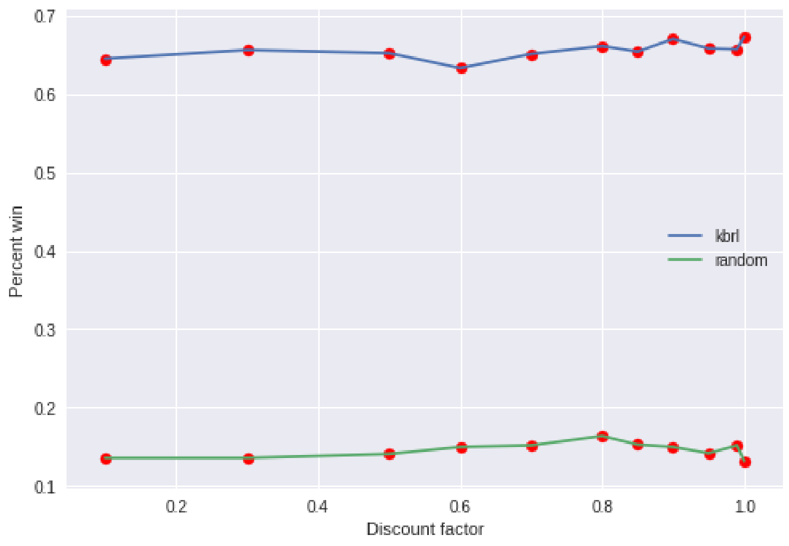


Figure 7: The effect of various discount factors on the performance of the agent over 1000 games.

and 50, both agents achieve a percent win of zero as the score is unachievable. It is also worth noting that the average performance over all valid targets (target ≤ 36) is slightly higher than the performance of our policy gradient method, although it is based on 6 data points and so this increase in performance may not be significant.

4 Discussion

In case of agent 1, as seen from the results in section [3.1] uphold the claims made about double Q-learning improving upon Q-learning, and double weighted Q-learning improving on both double Q-learning and Q-learning in section [2.2] is upheld in case of our task. This is credited to the fact that the reward function in our task is non-deterministic in nature causing the errors due to negative and positive biases. Also, due the non-deterministic nature of the reward function the learning requires a considerably large number of episodes while other tasks with deterministic reward functions like the grid-world with the same amount of state action pairs require about only a tenth of the amount of episodes to learn. This may be a problem if the size of the state space increases and may have to be dealt with techniques such as integrated planning and learning tabular methods or non-tabular approximation methods.

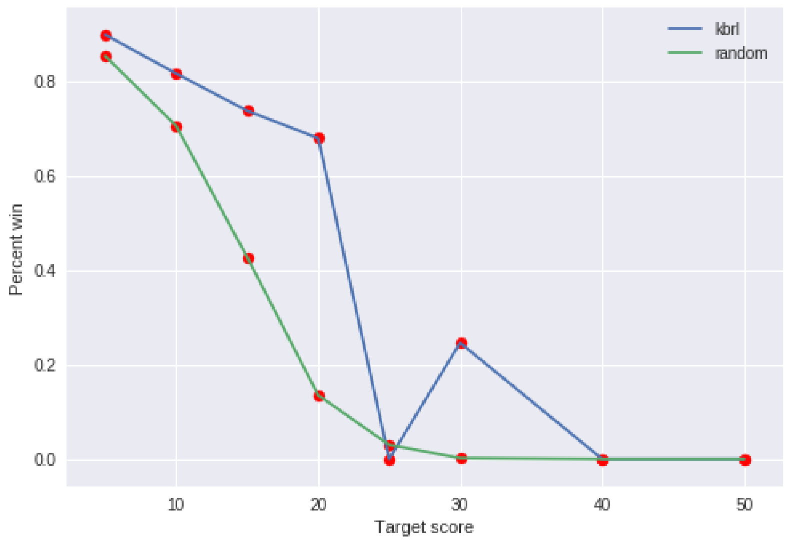


Figure 8: The effect of increasing the target score on performance of the agent over 1000 games.

In the case of the agent 2, as mentioned in section [3.2.1] we observe that both the policy gradient and kernel-based reinforcement learning approaches perform well in our environment. However, the need for future improvements is apparent. For policy-gradient method, we can explore more complex architectures, such as neural networks for our actor and critic functions.

In section [2.3.1] we made a few architectural choices to implement the method, such as linear function approximation and linear features. The function approximation, could possibly benefit from deep learning models. The features used in the method could as well be changed to other feature mappings in hopes of better performance. However since the result we found using linear function approximation and linear features was satisfying we did not explore other models.

The kernel-based reinforcement learning method, can be improved in two different ways. Firstly, we can increase the number of transition acquired at the data acquisition step which would result in a larger and more diverse set of transition samples. However, doing so greatly increase computation time and thus a second improvement can be to use implement parallelized computations of kernel functions to speed up the generation of the matrices required for value iteration.

5 Statement of Contribution

1. **Julyan Keller-Baruch:**
   I implemented kernel-based reinforcement learning method and contributed to the planning and implementation of the environments.

2. **Navin Mordani:**
   I was responsible for implementing the learning methods for agent 1 and designed, planned and implemented the environments.

3. **MohammadReza Davari:** I was responsible for implementing the Policy Gradient method for agent 2 and contributed to the planning and implementation of the environments.

References

[OS02] Dirk Ormoneit and Śaunak Sen. Kernel-based reinforcement learning. *Machine Learning*, 49(2):161–178, Nov 2002.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction.* MIT Press, 1998.

[Wat89] C.J.C.H. Watkins. Learning from delayed rewards. *PhD thesis, Cambridge University, Cambridge, England*, 1989.

[ZPK17] Zongzhang Zhang, Zhiyuan Pan, and Mykel J. Kochenderfer. Weighted double q-learning. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, 2017.