On and Off Policy Approaches to Reinforcement Learning Using Chess

Caleb B Parikh

I. Introduction

Supervised, unsupervised and reinforcement learning are the three types of algorithm concerning the field of machine learning. Reinforcement learning in particular has gained a lot of media attention recently, partly due to impressive achievements such as beating the best players of the ancient board game GO ([1]) and enabling incredible levels of dexterity in robotic applications ([2]).

Reinforcement learning is a very general learning framework for sequential decision making. Agents develop behaviours that maximise some cumulative reward signal. In practice the environment is often represented by a Markov decision process. Reinforcement learning is traditionally implemented using a Q-table. This is simply a look up table containing the expected future rewards for actions from each state. One issue with this approach is that for many problems the number of states is so large that the table would be unfeasible huge with respect to current computational power.

To rectify this problem, reinforcement learning can be implemented in a deep neural network which is known as deep reinforcement learning. This allows us to leverage the powerful representation capabilities of deep learning algorithms. The combination of these techniques provides an excellent method for learning state representations for a wide variety of challenging tasks in the real world.

II. Reinforcement Learning Algorithms

A. Q-learning

Q-learning is an off-policy temporal difference (TD) algorithm ([3]). The update rule for one step Q-learning is shown in equation 1.

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha \left[R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a\right) - Q\left(S_t, A_t\right)\right] \tag{1}$$

Where $Q(S_t, A_t)$ is the Q value for some state and action, α is the learning rate R_{t+1} is the new reward value (in the next state), γ is the discount rate and $\max_a Q\left(S_{t+1}, a\right)$ is the maximum expected future reward (given the new state and actions from that state).

This is known as off policy as Q which functions as the learned action-value function, will directly approximate q_* (the optimal action policy) even if the agent is following another policy. It is important to note that the policy still dictates the actions that the agent takes (and consequently which states are explored and updated). The algorithmic implementation of Q-learning generally following the implementation shown in figure algorithm 1.

B. SARSA

SARSA is similar in form to Q-learning but differs in that SARSA is an on -policy learning algorithm. This means that SARSA learns Q-values based on an action performed by the current policy instead of the greedy policy that Q-learning employs in it's update rule. The general update rule for SARSA is given in equation 2.

Result: Optimal Q-values for each state action combination
Initialise $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$;
repeat
  Initialise S;
  repeat
    Choose A from S using policy derived from Q (e.g. , $\epsilon-$greedy);
    Take action A, observe R, S';
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q\left(S', a\right) - Q(S,A)\right]$;
    $S \leftarrow S'$;
  until *Until S is terminal*;
until *until last episode*;

Algorithm 1: Q-learning Algorithm for Off-policy Control

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha\left[r_{t+1} + \gamma Q\left(s_{t+1}, a_{t+1}\right) - Q\left(s_t, a_t\right)\right] \tag{2}$$

SARSA stands for state action reward state action, which is reflective of the Q-value update being dependent on S_t, A_t, R_t, S_{t+1} and A_{t+1}.

Result: Optimal Q-values for each state action combination
Initialise $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$;
repeat
  Initialise S;
  Choose A from S using policy derived from Q (e.g. , $\epsilon-$greedy);
  repeat
    Take action A, observe R, S';
    Choose A from S using policy derived from Q (e.g. , $\epsilon-$greedy);
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q\left(S', A'\right) - Q(S,A)\right]$
    $S \leftarrow S'; A \leftarrow A'$
  until *Until S is terminal*;
until *until last episode*;

Algorithm 2: SARSA Algorithm for On-policy Control

Note that SARSA uses the policy twice in the algorithmic implementation shown in algorithm 2.

C. Comparison of SARSA and Q-learning

The on or off policy nature of each algorithms described above leads to various advantages and disadvantages.

Q-learning will learn the optimal path in most situations. The optimal path is the one which accrues the largest reward. However, this does not account for it's own exploration which is informed by it's policy. If it's exploration lead to large decreases in reward there were close to the optimal path, Q-learning will still view that path favourably despite the average reward per trial potentially being

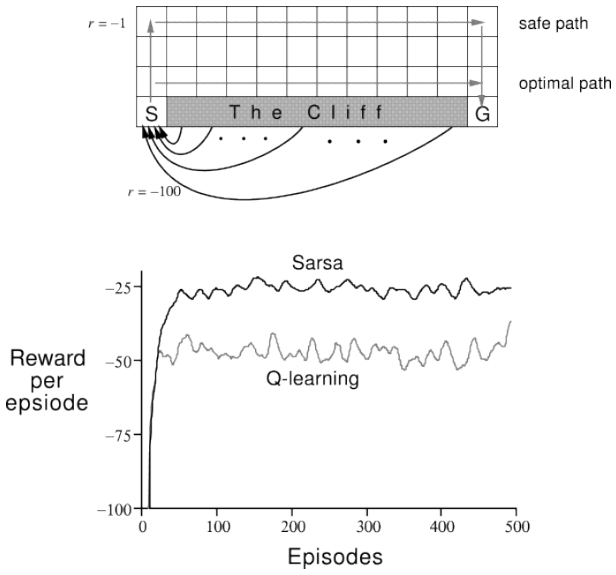

Figure 1. Cliff-walking task [[4]]

lower. SARSA on the other hand, is able to account for the expected cost of it's exploration. This means that SARSA will often favour a safer path over the optimal path (in effect maximising mean reward over the highest possible reward given it's policy). To illustrate this it may be helpful to consider the cliff example shown in figure 1.

As you can see SARSA takes a less optimal but safe path away from the cliff edge whereas Q-learning takes a more direct optimal path resulting in a more optimal but lower mean reward strategy. If a good starting policy is available on-policy may give a good results but may not explore other policies, however, if more exploration is necessary off-policy may be advisable, but could be slower and could get trapped in limit cycles (it's success depends on exploration and therfore the parameters α, ε).

III. Implementation of Q-learning

To test the off policy Q-learning algorithm we implemented Q-learning in a simplified game of chess. We implemented the algorithm and tracked the cumulative reward as a measure of ability. The number of moves per game was also tracked. The results with the default parameters $\eta = 0.0035, \gamma = 0.85$, and $\epsilon_0 = 0.2$ which is probability that the agent chooses to explore (from the epsilon greedy policy). We obtained the graph shown in figure 8. The agent trained over 20,000 games and slowly improved. All games received a reward of either 1 or 0 but the exponentially weighted moving average (EMA) is shown. We found that due to the stochastic nature of the environment and initialisation the performance was not always consistent to the mean score and error metrics are shown on the plot over 3 runs.

IV. Effect of Varying Parameters on Performance

In order to investigate the effect of varying the parameters we varied the values for γ and β and measured their performance with respect to reward accumulated and the number of moves taken per game. As before our results are smoother using the exponential moving average ($\alpha = 0.001$) and the model is ran three times to account for differences between simulations due to changes in initial conditions. The mean, max and min value of each simulation is plotted to give a sense of the distribution.

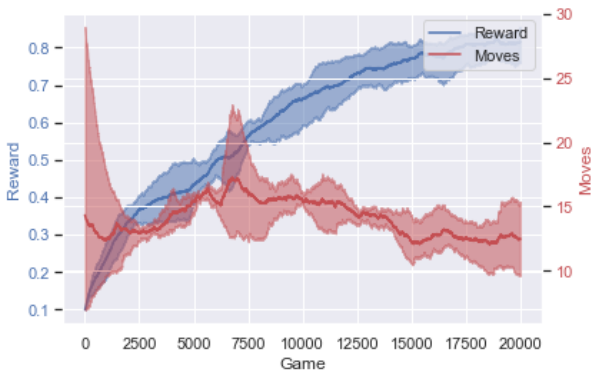

Figure 2. Plot of Average Reward and Number of Moves Using Q-learning for 20000 Games

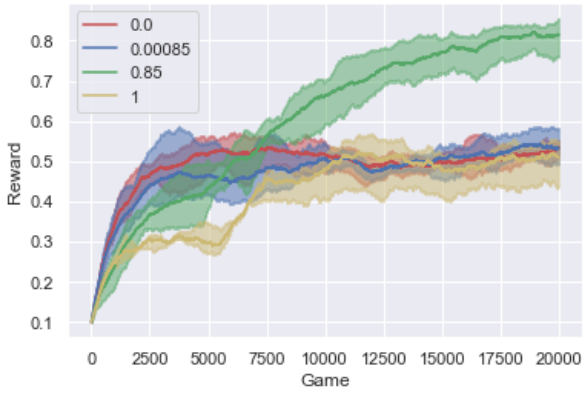

Figure 3. Plot of Average Reward for Various Values of Gamma

A. Varying γ

The discount factor (γ) controls how important rewards in the future are to the agents. If γ were set to 0 the agent will become myopic (lacking in foresight). This means it will only consider immediate rewards. Conversely as $\gamma \to 1$ the agent will only focus on the long term reward. $\gamma \approx 1$ with produce instability and errors will tend to propagate when using an artificial neural network.

As you can see from figure 3 , the average reward is optimised best when γ is around 0.85 and the agent does not learn particularly at other values. It should be noted that the agent seems to learn more quickly for lower values of γ when focusing on shorter term rewards.

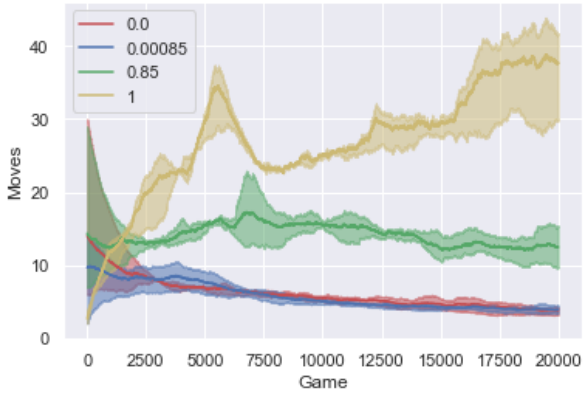

Figure 4. Plot of Average Moves and Number of Moves for Various Values of Gamma

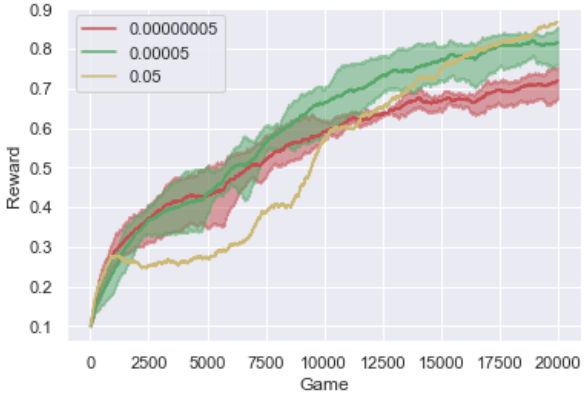

Figure 5. Plot of Average Reward for Various Values of Beta

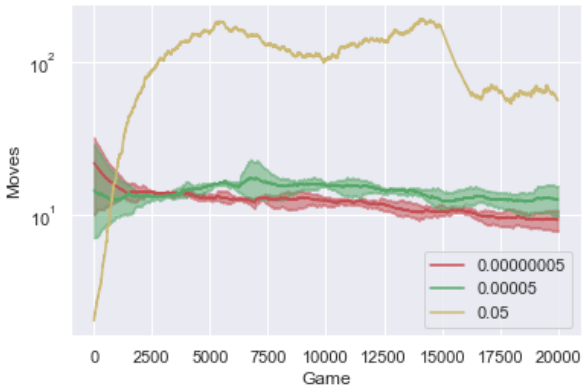

Figure 6. Plot of Average Moves and Number of Moves for Various Values of Beta

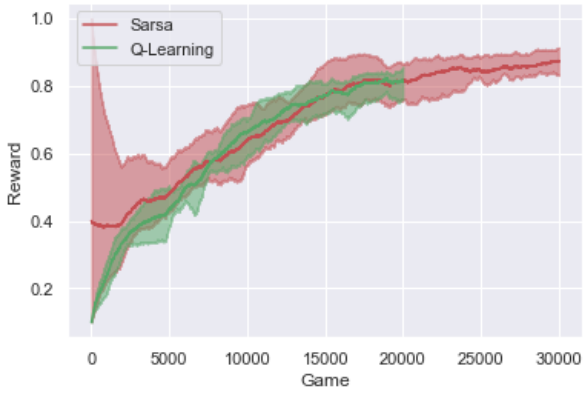

Figure 7. Plot of Average Reward Using On and Off Policy Learning

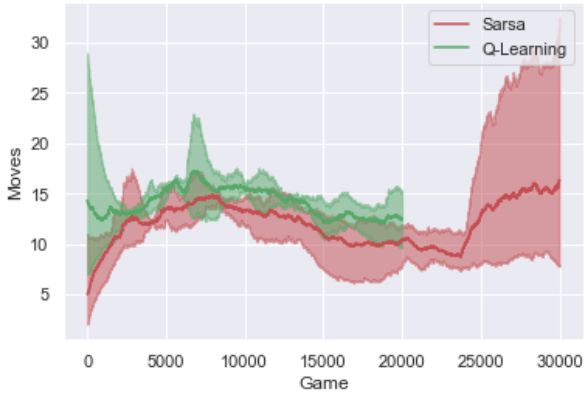

Figure 8. Plot of Average Moves Using On and Off Policy Learning

The difference is more noticeable when considering the number of moves the agent takes (figure 4). For low values of γ the agent the agent wants to accrue reward quickly so takes few moves, but for larger values, the agent does not mind taking more time before see a reward meaning the games last longer (maybe this could be interpreted as the agent being less considerate as it is not penalised w.r.t reward for taking longer).

B. Varying β

The agent is using the epsilon greedy policy, where the best option (the options with the highest Q-value) is given a proportion of $(1-\epsilon)$ and the event is determined probabilistically using the uniform distribution. We have implemented a slight adaptation where epsilon slowly decays so that in later games, when the agent understands it's environment better, the agent will choose to explore less and exploit it's knowledge more. The adaptation uses the rule $\epsilon_f = \frac{\epsilon_0}{1+\beta * n}$ where ϵ_f is the value used for game n and ϵ_0 is the original value.

All values for β give a similar performance regarding reward with marginally poorer performance for smaller values. The difference is far more noticeable when looking at the number of moves. We were only able to run the simulation once due to the large time it took to run (so our results may lack some validity) but the number of moves is far greater for larger values of β. This could be because there is insufficient exploration so the agent travels around the board largely uninformed whilst the opposition makes unpredictable moves. There is a smaller increase in moves in the same direction for the other β values.

V. Implementation of SARSA

We then implemented the on policy SARSA algorithm on the same task. Again the cumulative reward and number of moves was tracked. In order to compare this to the performance with the off policy algorithm we plotted both performances together as shown in figure V and V.

The performance of both algorithms seems very similar. we noticed a slight improvement in average reward (Q-learning is plotted till steady state but Sarsa keeps climbing). This is to be expected particularly as the increase in reward (above Q-learning) seems to happen at the same time as an increase in the number of moves (extrapolated from Q-learning). This is likely to be because the agent is taking a safer route with a better guarantee of reward given that it knows it may explore and does not know when it will explore. Please note that Q-Learning was plotted for a shorter amount of time as it reached steady state on average more quickly and there was a limit placed on compute due to access.

VI. The Problem of Exploding Gradients

Neural networks form very complex functions. When calculating the gradients of these functions, there can be either a vanishing problem or an exploding problem. In the exploding case, weights seem to increase exponentially due to the unstable nature of their gradient. This is because error as error is propagated through the net work it can grow without bound at each layer.

There are a few solutions to this. One could be to use a technique called gradient clipping where we push the gradient values to lie between some specific minimum and maximum value stopping the gradient vanishing or exploding. A more sophisticated solution is

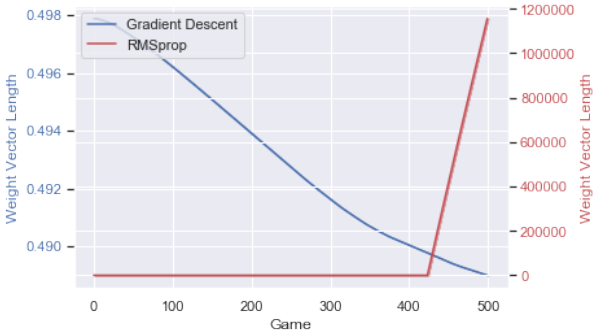

Figure 9. Plot showing RMSprop Stabilising the System

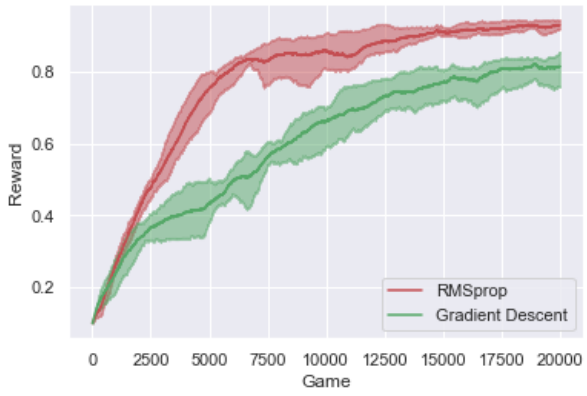

Figure 10. Plot of Reward Under Different Gradient Descent Optimisation Algorithms

to use a technique called RMSprop. RMRSprop effectively reduces the weight changes by smoothing using an exponentially weighted average. The following update rules are used for RMSprop.

$$v_{dw} \leftarrow \beta \cdot v_{dw} + (1 - \beta) \cdot dw^2$$

$$v_{db} \leftarrow \beta \cdot v_{dw} + (1 - \beta) \cdot db^2$$

$$W \leftarrow W - \alpha \cdot \frac{dw}{\sqrt{v_{dw}}}$$

$$b \leftarrow b - \alpha \cdot \frac{db}{\sqrt{v_{db}}}$$

Where v_{dw} is the moving average of δ (that will essentially replace δ in our implementation of backpropagation).

Figure 9 shows the effect of RMS on an unstable system. We see a very sudden increase in gradients as the system explodes. This is rectified by the addition of RMSprop and we see the the weight vector remain at a reasonable value as sudden shifts to large delta values are handled more gradually.

Over three trials we see a higher reward with RMSprop (10) and the agent appears to learn more quickly. Although we did not see exploding gradients in this case the weighted averaging allowed the agent to learn better and more quickly. There is some variation in learning with respect to number of required moves (11), but the average is lower at almost all points and seems to converge more quickly. It is also possible to use larger learning rates with RMSprop, allowing for larger learning rates. It also has a somewhat similar effect to gradient descent with momentum when avoiding local minima but we are unsure about how the

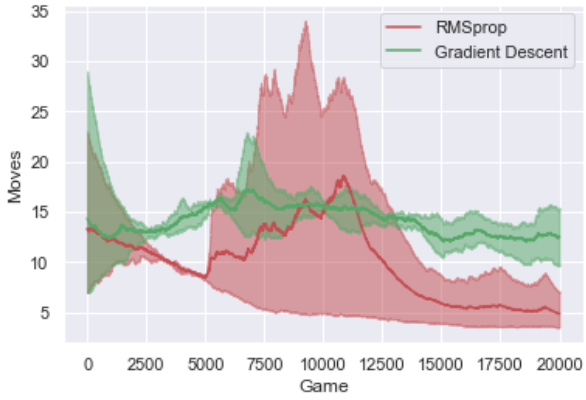

Figure 11. Plot of Moves Under Different Gradient Descent Optimisation Algorithms

performance compares to other techniques such as gradient descent or scheduled learning rates.

VII. Conclusion and Future Work

In this investigation we have shown an on and off policy approach used to tackle a simplified version of chess. A comparison of some of the key hyperparameters have been performed leading and from this heuristics can be developed to inform reinforcement learning strategies. We have also shown a technique to combat the problem of exploding gradients. In the future we would like to apply these techniques to open AI gym problems such as the cart pole game as well as investigating some of the other gradient optimisation techniques mentioned. The investigation would also benefit from repeated runs with more compute or optimisation of the code base.

VIII. Appendix

A. Backpropagation for Q-Learning using RELU

```
import numpy as np

act1 = np.dot(W1, x) + bias_W1
act2 = np.dot(W2, out1) + bias_W2

#cost function (or target value)
cost = R + gamma * np.max(Q_next)

if R == 0:
    dirac = (cost - Q[a_agent]) * np.heaviside(act2[
    a_agent], 1)
else:
    dirac = (R - Q[a_agent]) * np.heaviside(act2[a_agent
    ], 1)

#calculate backprop variables
delta_wo = eta * dirac * out1
delta_bo = eta * dirac

dirac_k1 = dirac * W2[a_agent] * np.heaviside(act1, 1)
delta_wh = eta * np.outer(dirac_k1 , x)
delta_bh = eta * dirac_k1

#update all parameters
W1 += delta_wh
W2[a_agent] += delta_wo
bias_W1 += delta_bh
bias_W2[a_agent] += delta_bo
```

B. Backpropagation for SARSA using RELU

```
import numpy as np

act1 = np.dot(W1, x) + bias_W1
act2 = np.dot(W2, out1) + bias_W2

a = np.concatenate([np.array(a_q1),np.array(a_k1)])
allowed_a = np.where(a > 0)[0]
prob = np.random.rand()

#on policy part for epsilon greedy
if prob>epsilon_f:
    cost = R + gamma * np.max(Q_next)
else:
    cost = R + gamma * Q_next[np.random.choice(allowed_a
    )]


if R == 0:
    dirac = (cost - Q[a_agent]) * np.heaviside(act2[
    a_agent], 1)
else:
    dirac = (R - Q[a_agent]) * np.heaviside(act2[a_agent
    ], 1)

#calculate backprop variables
delta_wo = eta * dirac * out1
delta_bo = eta * dirac

dirac_k1 = dirac * W2[a_agent] * np.heaviside(act1, 1)
delta_wh = eta * np.outer(dirac_k1 , x)
delta_bh = eta * dirac_k1

#update all parameters
W1 += delta_wh
W2[a_agent] += delta_wo
bias_W1 += delta_bh
bias_W2[a_agent] += delta_bo
```

C. RMSprop

```

v_dw = Rbeta * v_dw + (1-Rbeta)*dirac**2

dirac = dirac/np.sqrt(v_dw)
```

D. Epsilon Greedy Policy

```


prob = np.random.rand()

Q_a = Q[allowed_a] #the q values of allowed actions
choice = 0

if prob>epsilon_f:
    a_agent = allowed_a[np.argmax(Q_a)]
else:
    # choose random value
    ranChoice = np.random.randint(0,allowed_a.size)
    a_agent = allowed_a[ranChoice]
```

E. Initialisation of the Network

```

W1=np.random.uniform(0,1,(n_hidden_layer,n_input_layer))

W1=np.divide(W1,np.matlib.repmat(np.sum(W1,1)[:,None],1,
    n_input_layer))

W2=np.random.uniform(0,1,(n_output_layer,n_hidden_layer)
    )

W2=np.divide(W2,np.matlib.repmat(np.sum(W2,1)[:,None],1,
    n_hidden_layer))

bias_W1 = np.zeros((n_hidden_layer,))
bias_W2 = np.zeros((n_output_layer,))
```

All plot data can be found in the provided Jupyter Notebook Plots were produced using seaborn and matplotlib

References

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge", *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017, ISSN: 1476-4687. DOI: 10.1038/nature24270. [Online]. Available: https://www.nature.com/articles/nature24270 (visited on 05/10/2019).

[2] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning dexterous in-hand manipulation", *arXiv:1808.00177 [cs, stat]*, Aug. 1, 2018. arXiv: 1808.00177. [Online]. Available: http://arxiv.org/abs/1808.00177 (visited on 05/10/2019).

[3] (). Watkins: Learning from delayed rewards - google scholar, [Online]. Available: https://scholar.google.com/scholar_lookup?title=Learning%20from%20delayed%20rewards&author=C.J.C.H..%20Watkins&publication_year=1989 (visited on 05/10/2019).

[4] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction", p. 352,