Reinforcement Learning

Department of Computer Science
University of Sheffield

***Abstract*—This report describes how two different reinforcement learning algorithms, Q-learning and SARSA, have different characteristics and produce different outcomes. The aim of this report is to compare two algorithms by compare and contrast the results of implementing them in certain condition. From the results, Q-learning showed advantage on finding optimal path, but risked safety and SARSA showed advantage of safetiness but the performance is slightly poorer and takes more time.**

***Index Terms*—Introduction, Q-learning, SARSA, Implementation, Result, Conclusion**

I. Introduction

Reinforcement learning is one form of machine learning, which takes suitable action to give maximum rewards for the given conditions and situations. The delayed rewards are awarded in the next time step to evaluate its previous trial. Figure 1 shows how the reinforcement learning works.

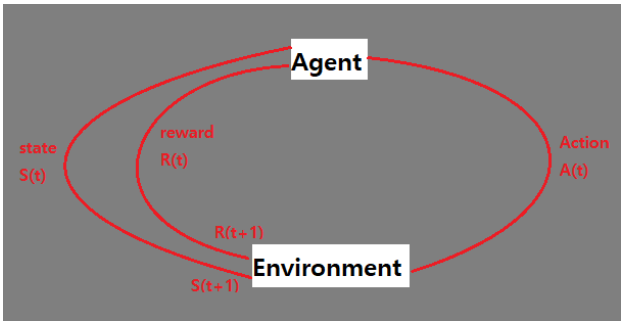

Figure 1. Reinforcement Learning algorithm

The differences between the reinforcement learning and the unsupervised learning and supervised learning are the reinforcement learning chooses the best action to get the biggest reward and gets feedback after complete task, while unsupervised only finds the similarities or differences by analysing the unlabelled data and supervised gets the direct feedback after set of action is done, trained by labelled data and able to forecast the results with new data-set.

For the reinforcement learning, the expected reward for each action are updated after the action is made, thus the equation for the simple update rule is following:

$$\Delta Q(s,a) = \eta(r - Q(s,a))$$

where $\Delta Q(s,a)$ is new Q value, η is learning rate, r is reward and $Q(s,a)$ is current Q value.

Within the reinforcement learning, there are various algorithms with different setup of agent or environment. Q-learning and the State-Action-Reward-State-Action (as known as 'SARSA') are both reinforcement learning with slight difference. SARSA is the on-policy algorithm that Q-values are updated using the Q-value of next state and epsilon-greedy action and Q-learning is the off-policy algorithm that the Q-values are updated using the Q-value of next state but instead of using e-greedy action, it uses current policy's action. In other words, Q-learning updates the estimate from the maximum estimate of possible next action with the highest rewards, while SARSA updates the estimates based on the same action it took previously. The advantages and disadvantages for each algorithm are followed:

1) The advantages and disadvantages of Q-learning.
   - commit to always exploring and try to find the best policy that still explores.
   - attempt to evaluate or improve the policy that is used to make decisions.
   - might stuck in local minima
2) The advantages and disadvantages of SARSA.
   - evaluate one policy while following another.
   - the policy used for behaviour should be sof
   - policies may not be sufficiently similar.
   - can behave poorly in some stochastic environments

II. Q-learning

The update rule for Q-learning is followed:

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

where $Q(s_{t+1},a)$ is Q value of next state, η is learning rate, r is reward, γ is the discount factor and $Q(s_t,a_t)$ is current Q value.

As mentioned previous section, Q-learning algorithm is off-policy algorithm, means instead of following policies such as e-greedy, the next actions are chosen which maximum Q-value is returned. Figure 2 shows how does the flow of Q-learning algorithm like. The advantage of Q-learning is since all the actions brings the maximum rewards, the algorithm ideally choose to take the optimal path. However, optimal only means it's the most efficient, does not mean it always success. The algorithm still contains the risk of failure and sometimes is very dangerous in some simulations. The purpose of this report is to figure out the difference between two algorithms in set experiment with actual results.

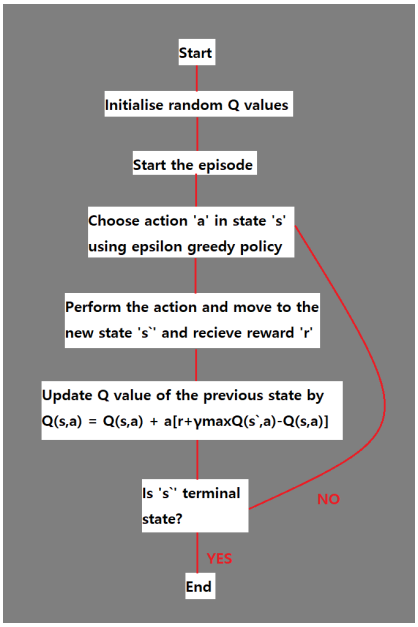

Figure 2. Flow of Q-learning algorithm

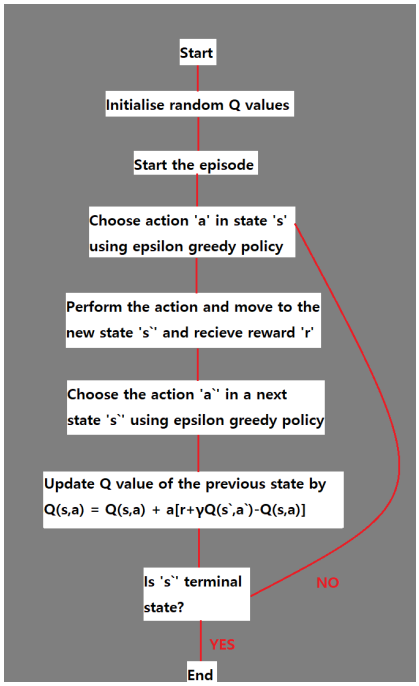

Figure 3. Flow of SARSA algorithm

III. SARSA

The update rule for SARSA is followed:

$$Q(s_t, a_t) = Q(s_t, a_t) + \eta(r_{t+1} - Q(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}))$$

where $Q(s_{t+1}, a_{t+1})$ is new Q value of next state and epsilon-greedy action, η is learning rate, r is reward, γ is the discount factor and $Q(s_t, a_t)$ is current Q value.

Also, as mentioned previous section, SARSA algorithm is on-policy algorithm, means it follows policies such as e-greedy and the next actions are chosen by that. Figure 3 shows how does the flow of SARSA algorithm like. The advantage of SARSA is as the randomness is added by the policy, so it tends to be safer and secure to achieve the goal than Q-learning, but the speed is slower as the safer path need more time to be calculated.

IV. Implementation

In order to implement the algorithms, the set situations are needed. In this paper, the chess game with specific situation is used. The size of the board is 4×4 and use only 3 pieces; one black king, one white king and one white queen. The aim of the agent is to checkmate the opponent by controlling white pieces. Different rewards are given with different results; the checkmate rewards 1 and the draw rewards 0.1. The average rewards and the average moved for each episode will be measured and plotted on the graph. The initial parameters are set as following and some of these will be changed in order to compare the results:

- Number of neurons of the hidden layer = 200
- Number of neurons of the output layer = 32
- Episodes = 100,000
- Initial epsilon for the e-greedy policy = 0.2
- Initial epsilon discount factor = 0.0005
- Initial discount factor = 0.05
- Initial learning rate = 0.0035

V. Results

A. Q-learning VS SARSA with initial condition

Figure 4 and 5 shows the change of average rewards and moves made for the Q-learning algorithm and Figure 6 and 7 shows the change of average rewards and moves made for the SARSA algorithm. Both algorithms resulted with exponential growths for the average rewards and exponential decay for the average moves. These imply both algorithms are working well and learning. The maximum average rewards for Q-learning is around 0.45 and the average moves decreased from around 25 to 10 at episode approximately 40,000. On the other hand, for the SARSA algorithm, the maximum average rewards is around 0.53 and the average moves decreased from around 25 to 9 at episode approximately 40000. SARSA shows better result for average reward because it's safer than Q-learning, which is more dangerous. This can be seen at Figure 4, where at the point of 40,000 episodes, the line decreases compare to Figure 6, where the line constantly grows exponentially. However, when compare with the average moves, SARSA algorithm's average moves increase from the minimum point while Q-learning's average moves decays constantly due to following optimal paths.

B. Setting with different parameters

By changing discount factor(gamma) and epsilon discount factor(beta), the results are compared by the algorithms themselves each. Figure 8,9,10 and 11 shows the change of average

rewards and moves made for each algorithm with different parameters. Blue line has beta of 0.0005 and gamma of 0.05 and the orange line has beta of 0.00005, gamma of 0.005 for Q-learning and gamma of 0.0005 for SARSA. For the results of Q-learning in Figure 8 and 9, orange line wins for the average awards but loses for the average moves. In contrast, for the results of SARSA in Figure 10 and 11, blue line wins for the average awards but loses for the average moves.

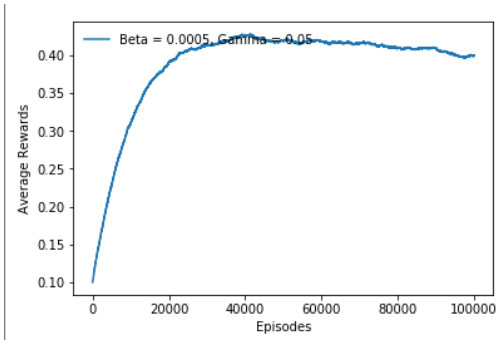

Figure 4. The average rewards made by Q-learning algorithm with set parameters

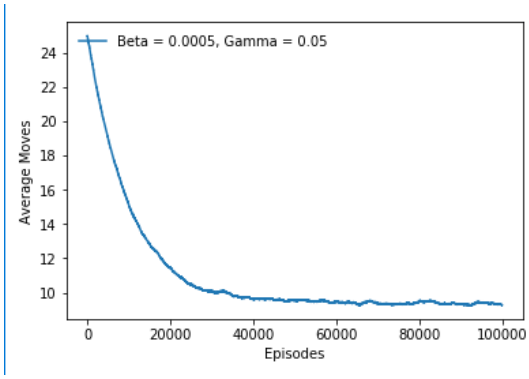

Figure 5. The average moves made by Q-learning algorithm with set parameters

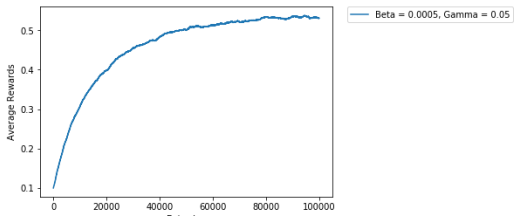

Figure 6. The average rewards made by SARSA algorithm with set parameters

VI. Problem and Solution

One of the problems for implementing such reinforcement learning algorithms is exploding gradients, which it's hard to see from the results in this paper. Exploding gradient occurs by error gradient that calculated during the training of a neural network. These error gradients are used to update the network

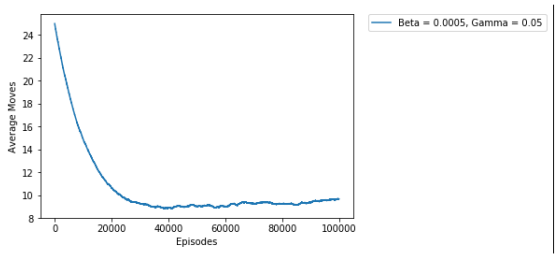

Figure 7. The average moves made by SARSA algorithm with set parameters

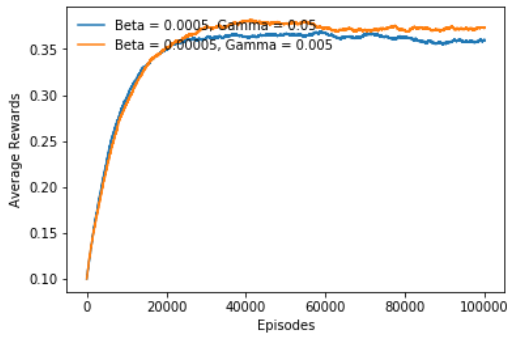

Figure 8. The average rewards made by Q-learning algorithm with different parameters

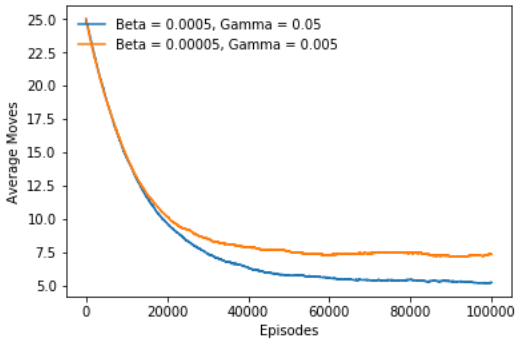

Figure 9. The average moves made by Q-learning algorithm with different parameters

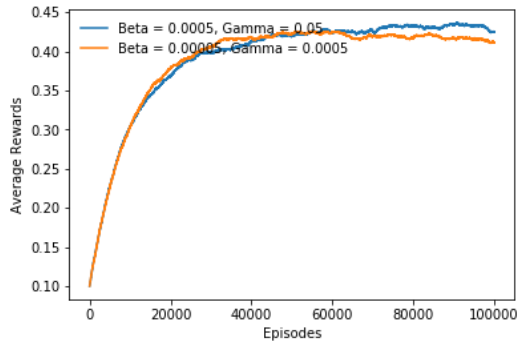

Figure 10. The average rewards made by SARSA algorithm with different parameters

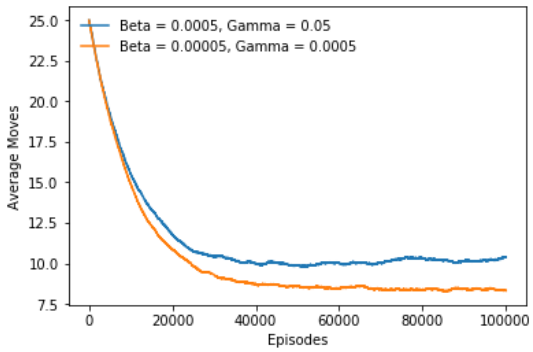

Figure 11. The average moves made by SARSA algorithm with differnet parameters

weights and results a large update to the weights and make the network unstable. If the model has poor loss or shows unstable learning, it means the exploding gradient has occurred.

One of the solutions to solve the issue is using Root Mean Square Propagation(RMSProp). The role of RMSProp is an optimizer that descent the gradient by adjusting the learning rate for each iteration depends on the weights. According to the Geoffrey Hinton, the founder of RMSProp, the way RMSprop works is it divide the learning rate for a weight by a running average of the magnitudes of recent gradient for that weight. The equation is following:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta)(\frac{\delta C}{\delta w})$$

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{E[g^2]_t}} \frac{\delta C}{\delta w}$$

where $E[g]$ is moving average of squared gradients, $\frac{\delta C}{\delta w}$ is gradient of the cost function with respect to the weight, η is learning rate and β moving average parameter.

The result of implementing RMSProp to Q-learning is Figure 12 and 13 with same parameter with Figure 4 and 5. As it can be seen, from episodes 40,000, there are less spikes of line by implementing RMSProp.

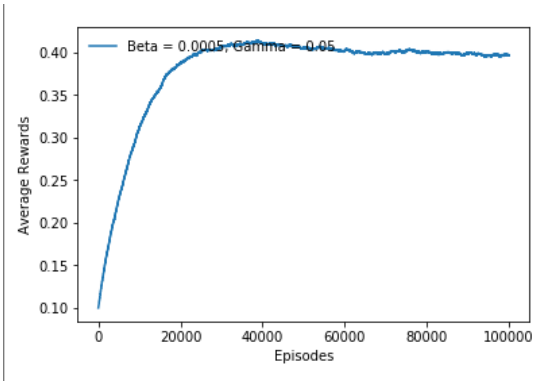

Figure 12. The average rewards made by RMSProp implmented Q-learning algorithm with set parameters

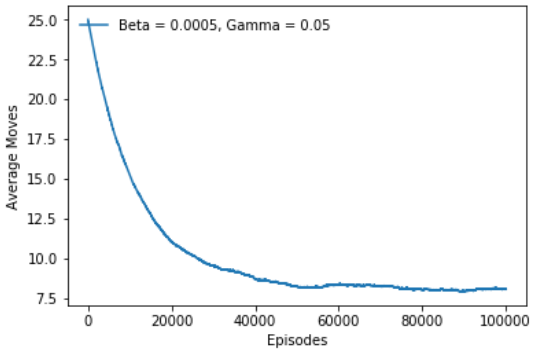

Figure 13. The average moves made by RMSProp implmented Q-learning algorithm with set parameters

VII. Conclusion

The aim of this paper is to find the difference of how two reinforcement algorithms, Q-learning and SARSA, work. From the results above, it's possible to state the algorithms worked well to find the clear difference between them. Q-learning has strength in finding optimal path but has to risk some failure, which is contrast of SARSA. It is also found that different discount factor(gamma) and epsilon discount factor(beta) affects to the algorithms. Adjusting two parameters are depends on the purpose of implementation with ratio between making good result and risking safetiness. One of the solutions for common neural network problem is also stated with some improvement of the results.

Appendix A
Reproduce Results

To reproduce the results, run 'chess_student.py' with using 'Spyder' or by python3 command line. The parameters can be adjusted to get different results.

Appendix B
Code

```
"""
Weights W1 = Input => hidden, W2 = hidden
    => output
"""
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));


"""
```

```
Biases bias_W1 = Input => hidden, bias_W2
    = hidden => output
"""
bias_W1=np.zeros((n_hidden_layer,))
bias_W2=np.zeros((n_output_layer,))
```

Listing 1. Random weights generated in range of 0 and 1. W1 stands for the weight from input layer to hidden layer and W2 stands for the weight from hiddenn layer to output layer. The biases for each weights are generated with same shape but as arrays of zeros.

```
import numpy as np

def Q_values(x, W1, W2, bias_W1, bias_W2)
    :

    # Neural activation: input layer ->
        hidden layer

    act1 = np.dot(W1,x) + bias_W1
    # Apply the sigmoid function
    out1 = 1 / (1 + np.exp(-act1))



    # Neural activation: hidden layer ->
        output layer

    act2 = np.dot(W2, out1) + bias_W2
    # Apply the sigmoid function
    Q = 1 / (1 + np.exp(-act2))

    return Q, out1
```

Listing 2. Function that initialize the Q value and the output by back propagation with given data x and weights W1

```
# Epsilon-greedy parameter
eGreedy = int(np.random.rand() <
    epsilon_f)


if(eGreedy):
    action = np.random.choice(allowed_a)
else:
    argMax = np.argmax(Q[allowed_a])
    action = allowed_a[argMax]


a_agent = action
```

Listing 3. The epsilon greedy function for on-policy algorithms. Randomly choose between the maximum Q value and randomly generated one.

```
# Rectified output
output = np.zeros((n_output_layer,1))
output[a_agent,0] = 1

# Weighted updated for each iteration
dw2 = eta * (rOld - QvalueOld + gamma *
    np.max(Q[allowed_a])) * rectOut.dot(
    hiddenOld.T)
W2 += dw2

dbw2 = eta * (rOld - QvalueOld + gamma *
    np.max(Q[allowed_a])) * rectOut.
    flatten('F')
bias_W2 += dbw2

# Heaviside function
r_Hidden = np.heaviside(hiddenOld, 0)

dw1 = eta * (((rOld - QvalueOld + gamma *
     np.max(Q[allowed_a])) * rectOut).T.
    dot(dw2).T * r_Hidden).dot(inputOld.T)
W1 += dw1

dbw1 = (eta * ((rOld - QvalueOld + gamma
    * np.max(Q[allowed_a])) * rectOut).T.
    dot(dw2).T * r_Hidden).flatten('F')
bias_W1 += dbw1

# Update variables for q-learning
QvalueOld = np.max(Q[allowed_a])
rectOut = output
hiddenOld = nOut1
inputOld = nX
rOld = R

R_save[n,0] = rOld

if n == 0:
     S_ema[n,0] = 0.1
     N_moves_ema[n,0] = 25
S_ema[n+1,0] = (alpha * rOld) + (1-alpha)
    *S_ema[n,0]
N_moves_ema[n+1,0] = (alpha *
    N_moves_save[n,0]) + (1-alpha)*
    N_moves_ema[n,0]
```

Listing 4. Implementation of Q-learning through back-propagation and rectified linear activation to update the weights and biases for every iteration. The average rewards and the average moves are calculated according to given data.

```
# Rectified output
output = np.zeros((n_output_layer,1))
output[a_agent,0] = 1

# Weighted updated for each iteration
dw2 = eta * (rOld - QvalueOld + gamma * Q
    [a_agent]) * rectOut.dot(hiddenOld.T)
W2 += dw2

dbw2 = eta * (rOld - QvalueOld + gamma *
    Q[a_agent]) * rectOut.flatten('F')
```

```
bias_W2 += dbw2

# Heaviside function
r_Hidden = np.heaviside(hiddenOld, 0)

dw1 = eta * (((rOld - QvalueOld + gamma *
    Q[a_agent]) * rectOut).T.dot(dw2).T *
    r_Hidden).dot(inputOld.T)
W1 += dw1

dbw1 = (eta * ((rOld - QvalueOld + gamma
    * Q[a_agent]) * rectOut).T.dot(dw2).T
    * r_Hidden).flatten('F')
bias_W1 += dbw1


# Update variables for sarsa
QvalueOld = Q[a_agent]
rectOut = output
hiddenOld = nOut1
inputOld = nX
rOld = R

R_save[n,0] = rOld

if n == 0:
    S_ema[n,0] = 0.1
    N_moves_ema[n,0] = 25
S_ema[n+1,0] = (alpha * rOld) + (1-alpha)
    *S_ema[n,0]
N_moves_ema[n+1,0] = (alpha *
    N_moves_save[n,0]) + (1-alpha)*
    N_moves_ema[n,0]
```

Listing 5. Implementation of SARSA through back-propagation and rectified linear activation to update the weights and biases for every iteration. The difference from the Q-learning is with the epsilon-greedy policy instead of only using maximum Q value. The average rewards and the average moves are calculated according to given data.

```
grad_squared = 0.0

# Rectified output
output = np.zeros((n_output_layer,1))
output[a_agent,0] = 1

# Weighted updated for each iteration
dw2 = eta * (rOld - QvalueOld + gamma *
    np.max(Q[allowed_a])) * rectOut.dot(
    hiddenOld.T)
gradient = np.gradient(dw2, axis=0)
gradient = np.array(gradient, dtype=float
    )
grad_squared = 0.9 * grad_squared + 0.1
dw2 = dw2 - (eta/np.sqrt(grad_squared)) *
    gradient
W2 += dw2

dbw2 = eta * (rOld - QvalueOld + gamma *
    np.max(Q[allowed_a])) * rectOut.
    flatten('F')
gradient = np.gradient(dbw2, axis=0)
gradient = np.array(gradient, dtype=float
    )
grad_squared = 0.9 * grad_squared + 0.1
dbw2 = dbw2 - (eta/np.sqrt(grad_squared))
    * gradient
bias_W2 += dbw2

# Heaviside function
r_Hidden = np.heaviside(hiddenOld, 0)

dw1 = eta * (((rOld - QvalueOld + gamma *
    np.max(Q[allowed_a])) * rectOut).T.
    dot(dw2).T * r_Hidden).dot(inputOld.T)
gradient = np.gradient(dw1, axis=0)
gradient = np.array(gradient, dtype=float
    )
grad_squared = 0.9 * grad_squared + 0.1
dw1 = dw1 - (eta/np.sqrt(grad_squared)) *
    gradient
W1 += dw1

dbw1 = (eta * ((rOld - QvalueOld + gamma
    * np.max(Q[allowed_a])) * rectOut).T.
    dot(dw2).T * r_Hidden).flatten('F')
gradient = np.gradient(dbw1, axis=0)
gradient = np.array(gradient, dtype=float
    )
grad_squared = 0.9 * grad_squared + 0.1
dbw1 = dbw1 - (eta/np.sqrt(grad_squared))
    * gradient
bias_W1 += dbw1

# Update variables for q-learning
QvalueOld = np.max(Q[allowed_a])
rectOut = output
hiddenOld = nOut1
inputOld = nX
rOld = R

R_save[n,0] = rOld

if n == 0:
    S_ema[n,0] = 0.1
    N_moves_ema[n,0] = 25
S_ema[n+1,0] = (alpha * rOld) + (1-alpha)
    *S_ema[n,0]
N_moves_ema[n+1,0] = (alpha *
    N_moves_save[n,0]) + (1-alpha)*
    N_moves_ema[n,0]
```

Listing 6. Implementation of RMSProp to Q-learning. The gradients are calculated for all weights and biases.