Collective Report B: Reinforcement Learning
UID:160151901

I. Q-learning and SARSA

Q-learning and SARSA (State-Action-Reward-State-Action) are two commonly used algorithms in the field of reinforcement learning which are similar but have subtle differences. Let us first define reinforcement learning. In contrast to supervised learning where we are given labelled training data and learn using features of the data, reinforcement learning consists of a decision-making agent, an environment and possible actions which the agent can take to go from one state to the next. The agent chooses an action, the environment returns a reward which can be positive or negative and then transitions into the next state. Reinforcement learning models learn from the rewards received as oppose to labels provided. Unsupervised learning is also contrasted to reinforcement learning ; in unsupervised learning there are no labels provided at all, where as in reinforcement learning feedback is provided in the form of the reward function. The goal of a reinforcement learning agent is to maximise its total reward.

We also need to introduce the exploration vs exploitation dilemma. Exploitation is where an agent chooses the action which will lead to the greatest reward given current information. However, there may be other actions which the agent has not come across which are better; the agent must explore. As a researcher, one must balance exploration and exploitation which can be done for example using an $\epsilon - greedy$ policy (choose greedy only a certain proportion of the time based on ϵ).

Q-learning and SARSA are off-policy and on-policy respectively. In simple terms, a policy is a mapping from states to actions. In this case, off-policy means that Q-learning updates Q-values based on greedy actions and on-policy means that SARSA updates Q-values based on the policy being used. Depending on the policy this can result in fairly different results but if the policy is a plain greedy policy the two algorithms would actually perform the same.

Q-learning learns the optimal policy whereas SARSA will learn a near-optimal policy due to the exploration aspect. SARSA would take longer to reach the near-optimal policy due to the amount of exploration so Q-learning can be said to learn faster.

II. Q-learning results Achieved

Q-learning was implemented and the results are shown on Figures 1 to 6 on the next two pages of this report. Figure 1 & 2 show results using the suggested parameter values. Figures 3 & 4 show results when running with γ reduced from 0.85 to 0.085. Figures 5 & 6 show results when running with β reduced from 0.00005 to 0.0000025.

We can see that the exponential moving average of reward value tends to 1. The final average is 0.93. This indicates that as we get closer to 100,000 games played the network wins a larger proportion of games. We can conclude from this that the network has learned successfully. The moving average for moves produces an interesting plot. We see that the number of moves starts off low and slowly increases, peaking at around 20,000 games before decreasing again. One possible explanation for this is that at the start our network plays the game poorly and so loses in a small number of moves. As the network learns it is able to get to further points in the match and so the number of moves increases. Further learning then makes the network better at playing the game and so it starts to win games in fewer moves.

By comparing Figure 1 and Figure 3 we can see the effect of changing γ (discount factor) from 0.85 to 0.085 has on the network. The discount factor is used to determine how much future rewards should be valued in the present. This can be likened to inflation in money. Money amounts in the future must be discounted when comparing to current amounts due to the effect the rate of inflation has. We use the discount factor to discount the future reward a certain amount, by reducing the discount factor from 0.85 to 0.085 the agent seeks to maximise immediate reward instead of future reward. We can see this drastically effects the results; the final EMA (Exponential Moving Average) has decreased from around 0.9 to 0.6. As this generally performs worse, the number of moves is also lower as the agent loses quicker.

In addition to varying the discount factor, we also experiment with reducing β (speed) from 0.0005 to 0.0000025. The speed is how fast the network learns. This is controlled by applying a discount rate to ϵ. By reducing this discount rate, the network explores for a greater number of episodes as ϵ takes a longer time to discount to where it would be with the previous speed. The final EMA of reward is 0.65 as seen in Figure 5. We can also see that the number of moves has not decreased as drastically as in Figure 1. This suggests that it is likely the network has not

finished learning which would be expected when reducing speed.

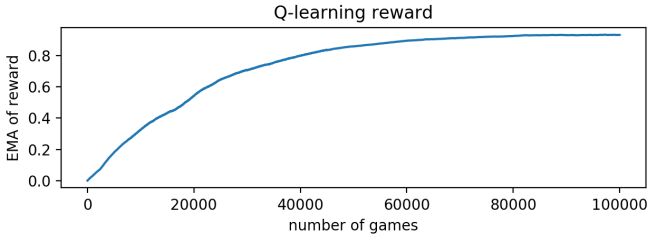

Figure 1: Q-learning reward with the suggested parameter values

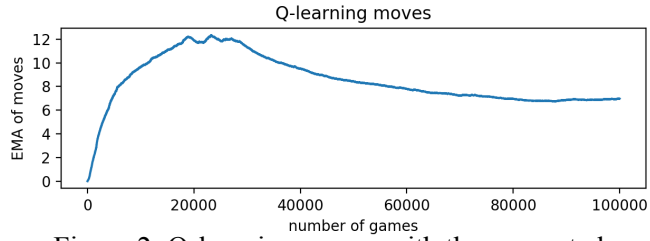

Figure 2: Q-learning moves with the suggested parameter values

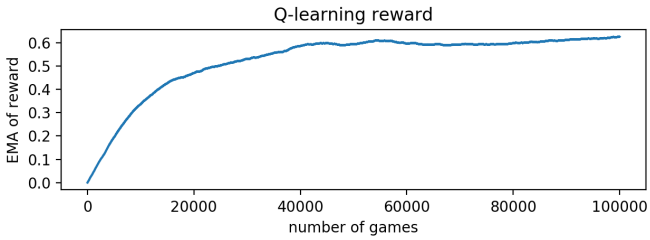

Figure 3: Q-learning reward with discount factor = 0.085

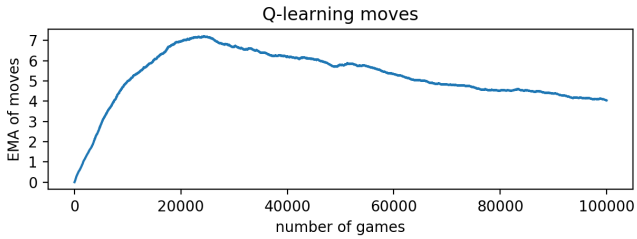

Figure 4: Q-learning moves with discount factor = 0.085

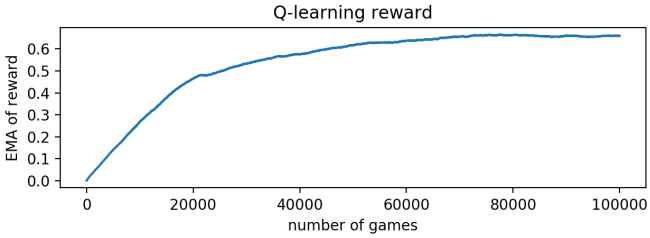

Figure 5: Q-learning reward with speed = 0.0000025

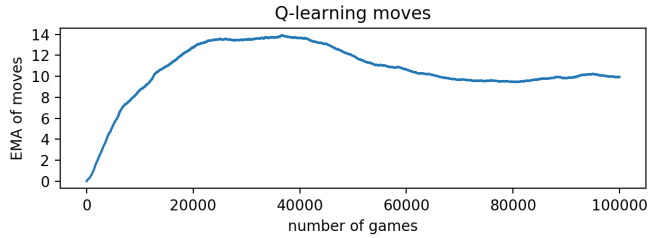

Figure 6: Q-learning moves with speed = 0.0000025

III. Comparing SARSA and Q-learning

Using the original suggested parameter values, we run both Q-learning and SARSA to see how they compare. The results for reward are shown in Figure 7 and for number of moves in Figure 8.

We see that SARSA performs better than Q-learning up until 7,500 games and then Q-learning overtakes it for the remainder. An explanation for this is that by 7,500 games most of the possible moves have been explored and now the best action will be to implement a greedy policy which is what Q-learning does. SARSA on the other hand continues to explore using $\epsilon - greedy$ and so will perform worse as time goes on. Ultimately, the final result is a difference of approximately 0.4 in EMA. Q-learning generally has a greater amount of average moves as I believe SARSA would take exploring actions too much and so end up losing games quicker.

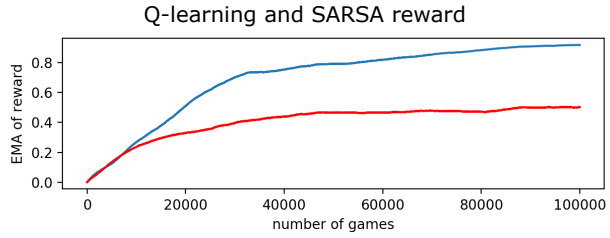

Figure 7: Q-learning (blue) and SARSA (red) reward

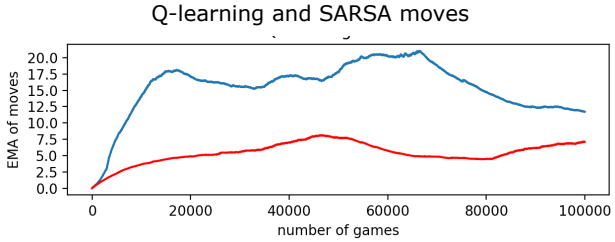

Figure 8: Q-learning (blue) and SARSA (red) moves

IV. Exploding Gradients

RMSprop (Root mean square prop) is an optimisation algorithm for neural networks. RMSprop gives us a different method for updating the weights. With our previous method there can be jumps in either direction when updating weights as the learning rate stays the same during learning. RMSprop keeps an EMA of the squares of the derivatives of the weights and then use the square root of this to update weights in such a way that the weights will be updated in much smaller increments as time goes on. Unfortunately due to time constraints this was not implemented for this report however this is something which could be explored in future work.

V. Reproducing Results

Run *chess_student.py* to reproduce the results in this paper. For figures 1 and 2 the file can be ran as is. For figures 3 and 4 change the gamma value to 0.085 and then run. For figures 5 and 6, change the beta value to 0.000005 and then run. For figures 7 and 8 change the *sarsa* parameter found in the file to 1 and then run. When changing from one experiment to another ensure that any previous changes have been reverted to when the file was first opened.

VI. References

1. Coursera: Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization https://www.coursera.org/lecture/deep-neural-network/rmsprop-BhJlm

2. Reinforcement Learning: An Introduction by Richard S. Sutton and Andrew G. Barto (2018 MIT Press)

APPENDIX (CODE)

$\epsilon - greedy$ POLICY

```
eGreedy = int(np.random.rand() < epsilon_f)
# if egreedy then random, else use optimal move
if eGreedy:
    index = np.random.randint(len(allowed_a))
    a_agent = allowed_a[index]
else:
    # get highest q value for an action which is allowed
    opt_action = max([Q[j] for j in allowed_a])
    a_agent = np.where(Q == opt_action)[0][0]
```

Figure 9: If statement showing if eGreedy then choose a random index from the allowed actions, else get the max Q-value of the possible actions.

DIFFERENT UPDATING FOR Q-LEARNING/SARSA

```
# if statement for next action using sarsa and q learning
if sarsa:
    eGreedy = int(np.random.rand() < epsilon_f)
    #if egreedy then random, else use optimal move
    if eGreedy:
        index = np.random.randint(len(allowed_a))
        a_agent = allowed_a[index]
    else:
        # get highest q value for an action which is allowed
        opt_action = max([Q[j] for j in allowed_a])
        a_agent = np.where(Q == opt_action)[0][0]
    # target according to sarsa
    target = R+(gamma*(Q_next[a_agent]))
    # rectified output for specified action
    rectOutput = np.zeros((n_output_layer,1))
    rectOutput[a_agent,0] = 1
else:
    target = R+(gamma*max(Q_next))
    rectOutput = np.zeros((n_output_layer,1))
    rectOutput[a_agent,0] = 1
```

Figure 11: If SARSA then target is calculated using our policy, else we take the maximum Q-value as should be done in Q-learning.

NETWORK INITIALISATION

```
import numpy.matlib

# weights between input layer and hidden layer
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));
# weights between hidden layer and output 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 for hidden layer
bias_W1 = np.zeros(n_hidden_layer,)
bias_W1 = bias_W1.reshape(n_hidden_layer,1)
# bisa for output layer
bias_W2 = np.zeros(n_output_layer,)
bias_W2 = bias_W2.reshape(n_output_layer,1)
```

Figure 12: Initialising weights and biases at the start.

```
# Backpropagation: output layer -> hidden layer
# update q
Qdelta = (target - Q) * rectOutput
# update weights and biases
W2 = W2 + (eta * np.outer(Qdelta, out1))
bias_W2 = bias_W2 +(eta * Qdelta)

# Backpropagation: hidden -> input layer
#rectified output
rectOutput2 = np.zeros((n_hidden_layer,1))
for j in range(0,len(out1)):
    rectOutput2[int(out1[j][0]),0] = 1

#update q
out1delta = np.dot(W2.T,Qdelta) * rectOutput2
#update weights and biases
W1 = W1 + (eta * np.outer(out1delta,x))
bias_W1 = bias_W1 + (eta * out1delta)
```

Figure 13: Using back propagation we update our Q-values and weights and biases. This code is based on the formulas given in the assignment PDF.

Q-VALUES (Q_VALUES.PY)

```
#reshape x to be usable in calculations
x = x.reshape(50,1)
# Activation calculations using relu
# input layer to hidden layer
act1 = np.dot(W1,x) + bias_W1
out1 = act1 * (act1>0)

# hidden layer to output layer
act2 = np.dot(W2,out1) + bias_W2
Q = act2 * (act2>0)
```

Figure 14: Computing Q-values using rectified linear units.

PLOT FIGURES

```
# plot
plt.subplot(211)
plt.xlabel('number of games')
plt.ylabel('EMA of reward')
plt.title('Q-learning reward')
plt.locator_params(axis='y', nbins=10,tight=True)
plt.plot(R_save)
if R_save_sarsa[0]:
    plt.plot(R_save_sarsa, color='red')

plt.subplot(212)
plt.xlabel('number of games')
plt.ylabel('EMA of moves')
plt.title('Q-learning moves')
plt.locator_params(axis='y', nbins=10,tight=True)
plt.plot(N_moves_save)
if N_moves_save_sarsa[0]:
    plt.plot(N_moves_save_sarsa, color='red')

plt.tight_layout()
plt.savefig('figure.png')
plt.show()
```

Figure 15: Code illustrating how the plots are produced using *matplotlib*.