Temporal Difference with Simplified Chess

Vladyslav Bondarenko

I. Temporal Difference Algorithms

A. Introduction

The main aim of temporal difference (TD) algorithms is efficient calculation of Q-values for state-action pairs. A Q-value $Q(s,a)$ is given by Eq. 1 and is defined as the expected reward for taking action a in state s following policy π. Q-values tell the agent the 'quality' of a given action where higher value implies more optimal action. In theory, determining exact Q-values for every state-action pair should allow the agent to always succeed in the task which solves the problem (always wining in a game of chess in particular).

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t=s,A_t=a] \\ &= \mathbb{E}_\pi[r+\gamma Q_\pi(s_{t+1},a_{t+1})|S_t=s,A_t=a] \end{aligned} \tag{1}$$

In TD learning the Q values are updated according to Eq. 2, where highlighted part is called TD-Target. After each action in each state this learning rule is applied updating current value in proportion to calculated error from TD-Target. Performing this update iteratively over many trials will result in an accurate estimate of $Q(s,a)$. The main difference between two TD algorithms described here is the way in which TD-Target is calculated.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t)+\alpha(\,R_{t+1}+\gamma Q(s_{t+1},a_{t+1})\,-Q(s_t,a_t) \tag{2}$$

B. SARSA

The algorithm described above and shown in Eq. 2 is often referred to as SARSA (State-Action-Reward-State-Action). SARSA is an example of an *on-policy* algorithm. This means that the same policy $\pi(a|s)$ is used to select the action for current state s_t as well future state s_{t+1}. For example, if a greedy policy is adopted than the maximum value is selected for both $Q(s_t,a_t)$ and $Q(s_{t+1},a_{t+1})$. SARSA implies following and improving the same policy at the same time.

C. Q-learning

Q-learning is an example of an off-policy TD algorithm. In an off-policy algorithm there are two different policies for behaviour and target selection. In Q-learning in particular, the behaviour is selected according to a policy of choice (e-greedy for example) and the target policy is always greedy. Following greedy policy provides with an optimal action-values $Q*(s,a)$ The updated equation for this algorithm is shown in Eq. 3. With Q-learning, the agent follows a policy which allows for exploration but optimises according to the greedy policy.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t)+\alpha(R_{t+1}+\gamma\max_{a'}Q(s_{t+1},a_{t+1})-Q(s_t,a_t) \tag{3}$$

D. Comparison

One of the disadvantages of SARSA is that it learns a near-optimal policy where as Q-learning directly learns the optimal policy. In order to learn an optimal policy with SARSA an e-greedy policy can be used with a decaying epsilon parameter. This ensures that as training proceeds, more optimal policy is adopted. However, such an approach may require complex hyper-parameter tuning. On the other hand, Q-learning has higher per-sample variance than SARSA, which could result in problems with convergence. This is particularly true when training an algorithm using a neural network. Another major difference is due to exploratory strategies, where SARSA adopts a more conservative approach. If there is a state with high negative reward near optimal path, Q-learning agent will tend to enter those states when exploring, whereas SARSA algorithm attempts to avoid those states by taking safe route at first and slowly approaching optimal path. However, in reality the last difference is to an extent irrelevant for chess playing and simulations as a whole, and would only be applicable to training robotic agents in dangerous environments. Overall, in theory, Q-learning approach is more optimal for chess playing since only final average reward is important rather than those gained whilst learning. However, since neural network will be used to train the agent, it is important to achieve optimal convergence. [1]

II. Implementation

Initial implementation focused on developing Q-learning algorithm for update of the weights of a 2 layer neural Q-network. The network takes in state values such as positions and degrees of freedom of chess pieces and outputs Q-value for each possible action. The activation function used is ReLU as it is computationally simple, allows for sparsity and is easier to optimise due to linear behaviour. An action is selected out of those that are allowed according to e-greedy policy and then Q-learning and back propagation are used to update the weights in the network.

A. Initialisation

The weights are initialised at random from a uniform distribution and then normalised by the number of total connections between two layers. Usually the biases are initialised to 1, however when using ReLU it is suggested [2] to use a small, positive value such as 0.1. This ensures that almost all units are initially active for most inputs and allow the derivative to pass trough.

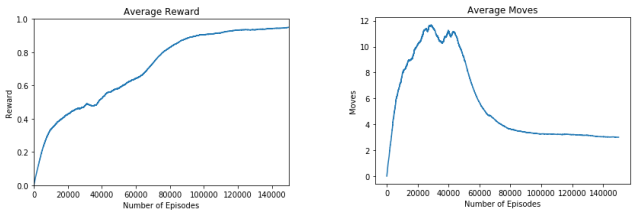

Fig. 1: First implementation results

B. Results

The first implementation of the chess playing agent learned over 150000 games to ensure at least partial convergence. Recommended parameters were used: epsilon = 0.2 and decay beta = 0.00005 such that at the end of learning epsilon reduces to almost 0, gamma decay = 0.85 and learning rate = 0.0035 (those parameters will be referred to as default). Exponential moving average with window of 10000 is used to produce a smooth plot for both rewards and moves per episode, shown in Fig. 1. By the end of training reward has began to converge, reaching the final value of about 0.95. However the gradient of the curve suggests that further training would result in slight improvement. Reward growth fast at initial stages of training due to large error and thus moves fast towards optimal solution. It then slows down between 20000 and 60000 and speeds up again after. This is likely to be due to noise, and if experiment done over multiple epochs with different initial condition the curve should follow a smooth logarithmic growth pattern. The average moves curve rises to almost 12 moves per episode at initial stages and then gradually reduces to about 3. This represents high exploration and low awareness at the initial stages and as epsilon reduces the agent sticks to the optimal found solution which consists of 3-4 moves.

III. Optimization

A. Initialisation

Effective weight initialisation is a key part of successful training of a neural network. This requires understanding the approximate distribution of weights prior any training. If the weights are too small the signal will shrink as it passes through layers and become more noisy as it reaches the output. If the weights are too large the signal will grow as it passes trough each layer. The later is particularly dangerous with ReLU activation function since it does not pose a limit on the output value. Xavier proposed an optimal initialisation algorithm named after him [3] which is now extensively used with *sigmoid* and *tanh* activation functions. Kaiming He has later adapted this approach for ReLU [4]. The results of those algorithms are shown in Fig. 2 comparing with original method used. Both xavier and henormal algorithms performed significantly better at all stages of training resulting in faster and smoother learning rate and much smaller maximum number of moves. A lower number of average moves at all stages halved the run time of the algorithm allowing to complete

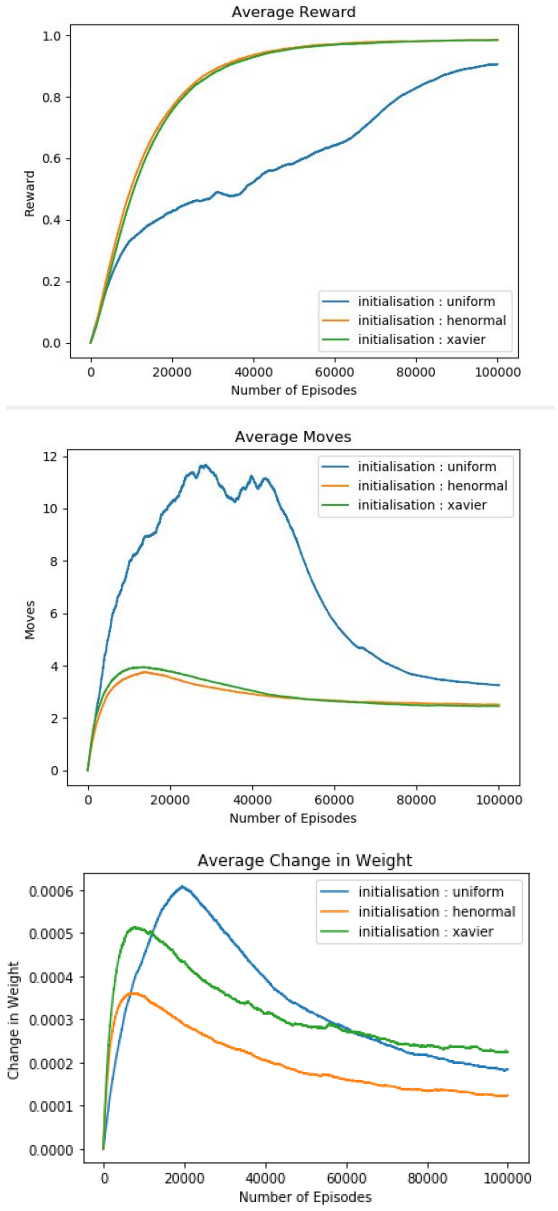

Fig. 2: Testing initialisation approaches

more sophisticated further experiments. The difference between xavier and henormal is clear in average weight change graph, where henormal has smaller change in weights at all stages of training. This implies that henormal estimates the distribution of weights most accurately and thus is chosen as an initialisation approach for all further experiments.

B. Parameters

For each parameter 3 different values were tested where in most cases default was used as the middle values and two additional were picked such that one is below and the other above the suggested one. Each parameter was tested with identical initial condition forced by a set seed for random number generator. Here only results for beta and gamma

parameters are presented and the rest can be found in the Appendix (Fig. 9 and Fig. 10). For gamma and beta parameters the tests were run for 7 epochs and then averaged producing more reliable results with less noise. Only average reward and average movement are included here and average change in weights per episode can again be found in the Appendix (Fig. 8).

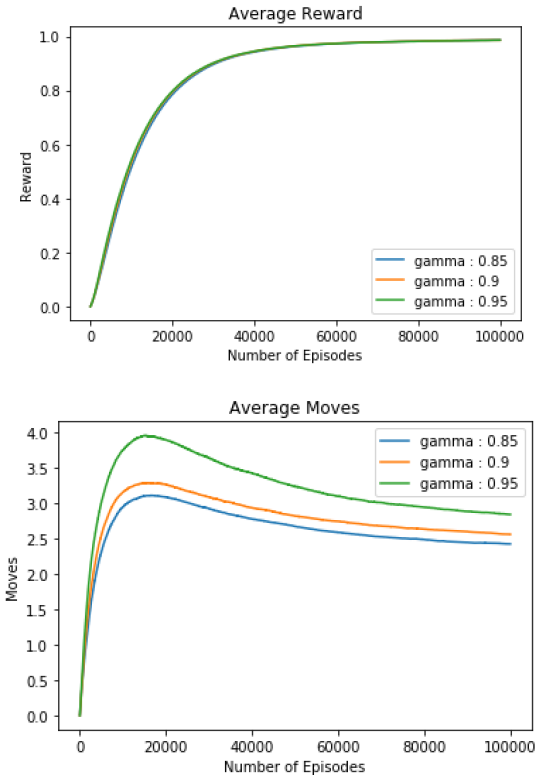

Fig. 3: Testing increasing gamma parameter

1) Gamma: discount factor for reward of future actions available from current state. Its main aim is to prevent summation over infinite series in continuous space, however it also represents the fact that future rewards are less valuable than current. Varying gamma changes how much impact on the error do future available actions have. In chess, long term strategy is particularly valuable since making the most optimal action at current state might result in a lost piece or opponent moving away. However, having too high gamma will give too much value for future rewards. The effect of this is shown in Fig. 3, where although the average reward doesn't vary significantly, the average number of moves has some minor difference. As value of gamma increase further from default the average number of moves at all stages of learning increases, with a proportionally higher increase as gamma gets closer to 1. Further results for decreasing gamma are shown in Appendix (Fig. 7) which shows that as gamma is reduced there is a further shift in average moves curve towards smaller values but again, by increasingly smaller proportion. Moreover, the average reward starts to shift down as well. Thus, the default value of 0.85 was kept as the optimal one.

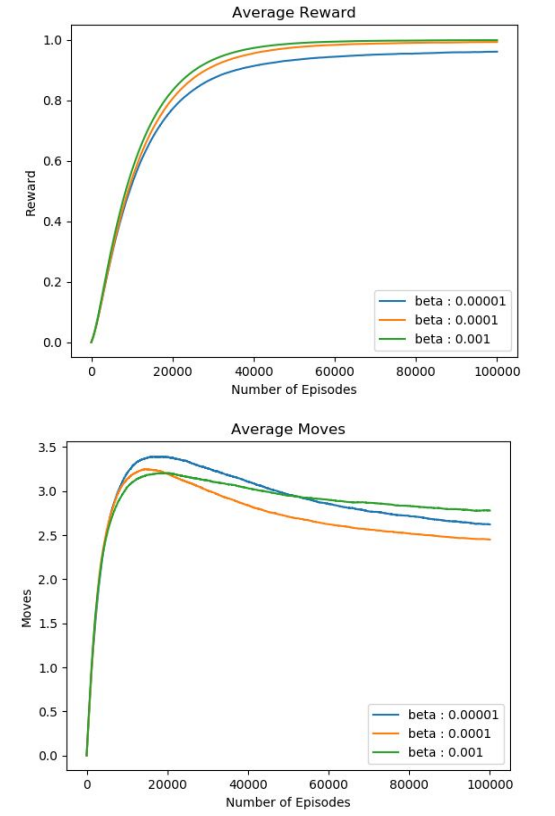

Fig. 4: Testing beta parameter

2) Beta: decay rate of the epsilon parameter which allows to trade off exploitation for exploration as training proceeds. The equation describing the use of beta is shown in Eq. 4 where n is current episode and ϵ_0 is initial value of epsilon which is set by default to 0.2. From the equation we can see that as beta increases the value of epsilon would decrease at a higher rate, resulting in less exploratory actions further down the training. Having high beta value should result in faster convergence but might not guarantee convergence at optimal value since the agent might get stuck in local minimal before exploring all possible options. As was mentioned earlier, this is particularly applicable to SARSA algorithm where both policies used are e-greedy. Small value of beta would have a higher chance of producing optimal Q-values but if not fully converged by the end of the training can cause noise due to continues exploration. This is confirmed by results shown in Fig. 4 where smaller beta converges slower and ends up at an average reward smaller than 1 and larger beta converges faster at average reward of 1. Looking at average moves it can be deduced that beta with value of 0.0001 produces the most optimal solution.

$$\epsilon = \frac{\epsilon_0}{1 + (\beta * n)} \tag{4}$$

IV. SARSA vs. Q-learning

In order to simplify the comparison between two algorithms another version of SARSA was implemented with 0 beta parameter, from now referred to as SARSA_0. After the first implementation of SARSA_0 it was found that in some cases the weights would explode. This problem and solution employed is discussed in further detail in the next section but it is worth mentioning that results presented here (Fig. 5) are integrated with that solution. The solution has caused some minor performance issues and the optimal run for SARSA and Q-learning is shown in Appendix (Fig. 11 converging at a reward of about 0.99. The results were again obtained from 7 runs and then averaged. Fig. 5 shows that all algorithms perform almost identical in terms of average reward with only a minor advantage for Q-learning where it converges slightly faster. However, the difference could be due to noise which is still hard to eliminate fully. The key difference between different algorithms is shown in average moves plot where SARSA_0 is able to complete a game with less number of moves in general. This is likely to be due to more aggressive exploration strategy allowing to find the global minimal faster. On the other hand, both SARSA and Q-learning have almost identical values at all stages. This suggest that with an optimal beta parameter SARSA algorithm essentially generalises to Q-learning. By the end of the training all three algorithms converged at about 2.7 moves per episode. Further experimental results are presented in the Appendix (Fig. 12 and Fig. 13).

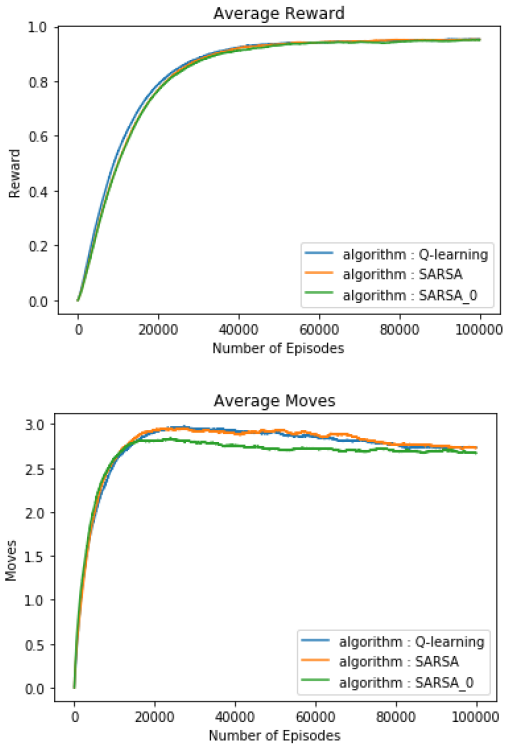

Fig. 5: Comparing SARSA and Q-learning.

V. Exploding Weights

The problem with exploding weights occurs when error accumulates during an update resulting in very large gradients. This causes a large change in weights within the network. Since ReLU doesn't impose a limit on output activation, the error starts to increase, further increasing the weights and causing the recursive explosion. This results in an unstable network which is unable to learn and in extreme cases, causes an overflow with NaN values. There are a number of approaches to eliminate this problem. A direct solution would be to regularise the weights by either bounding them by a maximum value or normalising using L1 or L2 penalty. Those approaches would keep the weights within a certain range and prevent the exploding growth. A more indirect solution is to change the gradient descent algorithm to RMSprop or Adam. Both of those incorporate momentum which is used for computing a separate learning rate for each parameter. Although later could be more optimal for solving the problem, it is likely to be more computationally intensive and requires introduction of multiple new parameters. Thus it was decided to implement L2 normalisation technique results of which are shown in Fig. 6. It consist of an addition of an extra value to an error function forming Eq. 5 with additional value highlighted. In practice this forces weights to decay towards 0 keeping them small and preventing gradient explosion. Parameter γ determines by how much the weights decay and was set to 0.00001 for this application. [5]

$$E = \frac{1}{2}\sum_{i=1}^{n}(t_i - h_\theta(x_i))^2 + \lambda\sum_{i=1}^{n}\theta_i^2 \tag{5}$$

The test was run using a $SARSA_0$ algorithm described earlier, optimal parameters and henormal initialisation. Without normalisation the weights start to increase exponentially where as with L2 they converge similar to previous experiments. If training is continued the weights would continue to increase further causing overflow. This affects both average reward and average number of moves where average reward starts to decrease and average number of moves increases. This is because values passed by the network are starting to get too large producing noisy output. (Appendix Fig. 14)

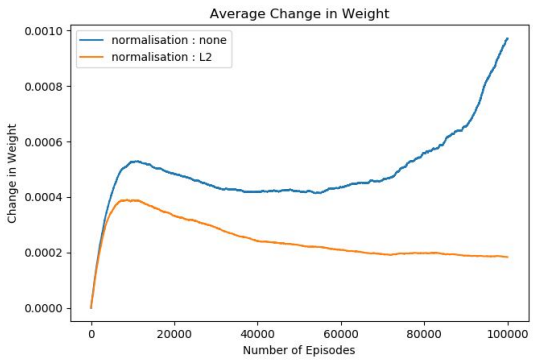

Fig. 6: Solution for exploding gradients with L2 normalisation

References

[1] Richard S. Sutton, Andrew G. Barto *Reinfocment Learning (Adaptive Computation and Machine Learning series)* 2nd. edition
The MIT Press (2018)

[2] Ian Goodfellow, Yoshua Benglo, Aaron Courville *Deep Learning (Adaptive Computation and Machine Learning series)*
The MIT Press (2016)

[3] Xavier Glorot, Yoshua Bengio *Understanding the difficulty of training deep feedforward neural networks* PMLR 9:249-256, 2010.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun *Delving Deep into Rectifiers*
2015

[5] Razvan Pascanu, Tomas Mikolov, Yoshua Bengio *On the difficulty of training recurrent neural networks*
ICML13, Vol 28. pg. 1310-1318, 2013

Appendix

A. Reproducing Results

Simply running *chess_student.py* will run a Q-learning algorithm with optimal parameters and henormal initialisation. With such a set up it takes under 10 minutes to run a single iteration of the code. Running my_test.py will partially reproduce the results shown in Fig. **??**. The number of repetitions is set to 0 meaning the code is run only once, thus producing more noisier version of results. To display graph with zoom on final episodes *line 239* has to be uncommented, but may require installing mpl_toolkits. It is also possible to delete SARSA_0 parameter to just see the comparison between standard SARSA and Q-learning. Lines 236 and 237 if uncomented would reproduce results in Fig. 4 and Fir. 3 respectively. The repetitions are set to 0 again but can be set to 6 in order to reproduce results presented in the figures exactly. (May take long time to run...). It is also possible to change the values of tested parameters by editing the list passed in. Lines 234 was used to test different initialisation techniques shown in Fig. 2 and can be sped up by passing optimal parameters. Line 235 was used to produce Fig. 6 with solution for exploding weights. Most personal code was written in my_code file with few exceptions such as error calculation.

B. Further Results

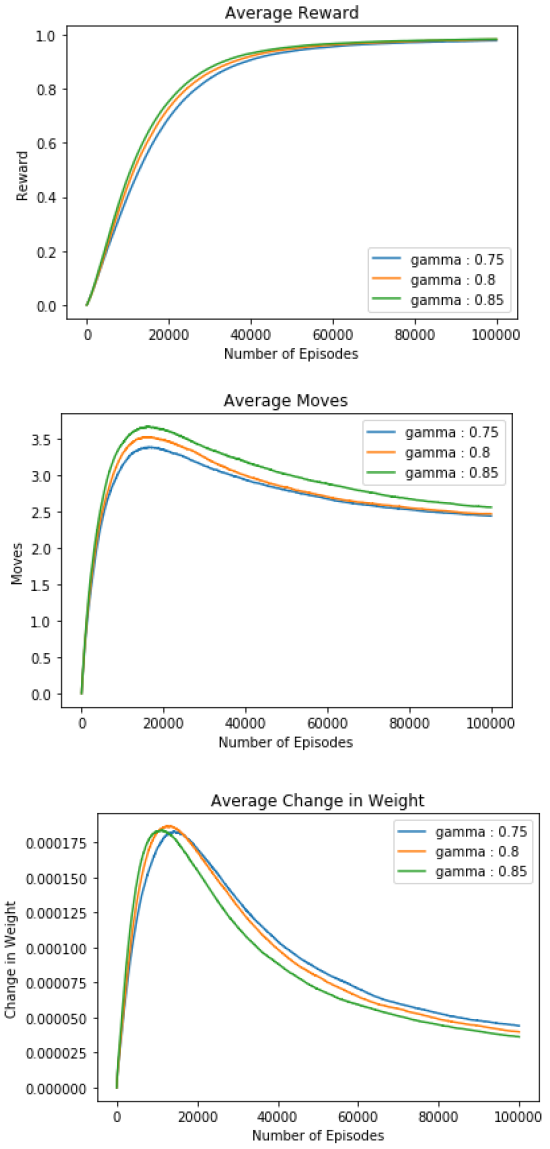

Fig. 7: Testing decreasing gamma. As gamma decreases from default value the reward curve slightly shifts down and at the same time the average move curve shift upward. There is a trade off, but due to very small differences it was decided to stick to default value of gamma.

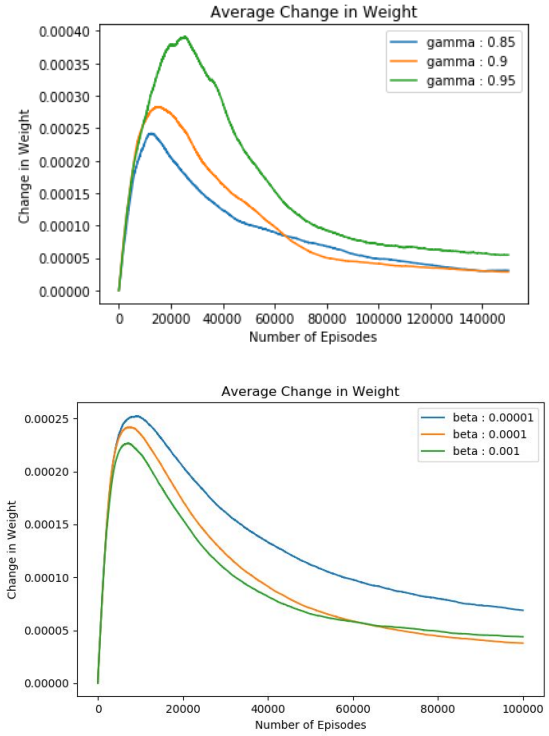

Fig. 8: Average change in weight for increasing gamma on the top and beta on the bottom. Gamma figure further confirm the choice of the optimal parameter. As for the beta the difference between two larger values is minor and results from average moves should play a more significant role in choosing an optimal parameter.

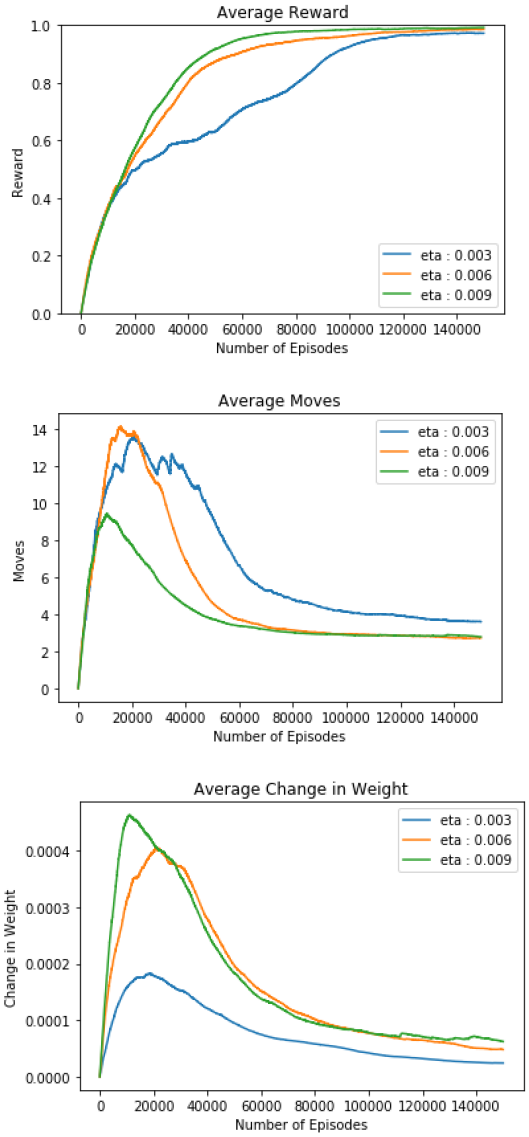

Fig. 9: Testing eta parameter. The eta parameter was only run once and thus results are noisy. However, it is clear that larger eta is more optimal in terms of average reward growth and optimal number of movement throughout training. Higher eta also causes higher increase in weights by definition. Too high eta could result in exploding weights which is further explored in the dedicated section.

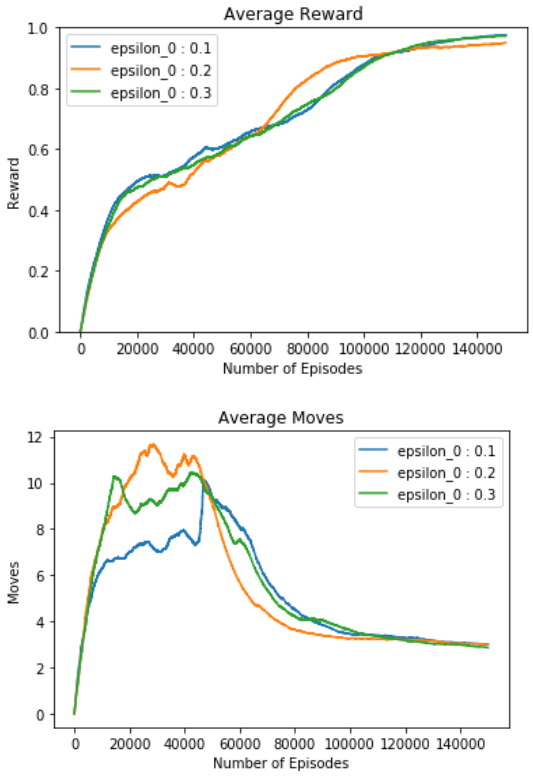

Fig. 10: Testing epsilon parameter. The epsilon parameter was run only once and thus results are noisy. There is no significant difference when the variation is minor thus the parameter was kept at it default value of 0.2. This also allows to keep the current optimal beta value without further hyper-parameter optimization.

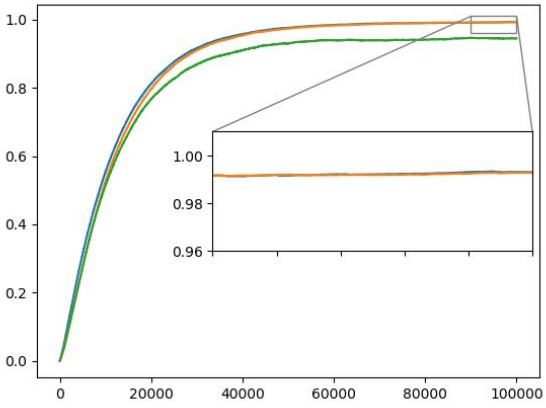

Fig. 11: Average reward per episode graph for SARSA, SARSA0 and Q-learning, zoomed in on final episodes. SARSA0 is integrated with L2 normalisation to ensure weight stability, but it has negative impact on learning rate. This figure shows that without normalisation the maximum performance achieved was over 0.99 and still converging towards 1.

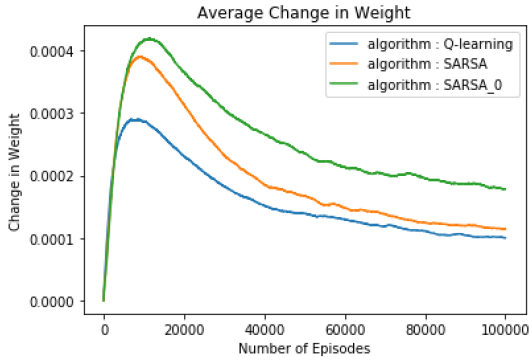

Fig. 12: The average weight change for the main run described in the SARSA vs Q chapter. This shows that Q-learning has the lowest error because it always goes for the most optimal action. SARSA has a smaller change in weight since as epsilon converges the error claculated from Q-values is reduced converging at a similiar value as Q-learning.

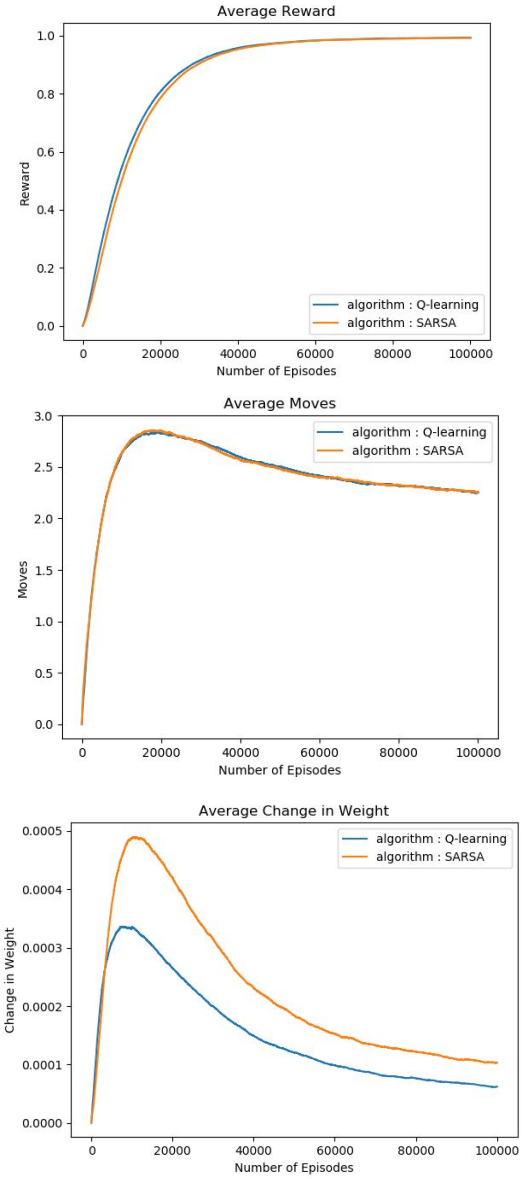

Fig. 13: Further confirming results for comparison of SARSA and Q-learning when the code is run without L2 normalisation. Both reward and average moves are almost identical and there is only a difference in average change in weight.

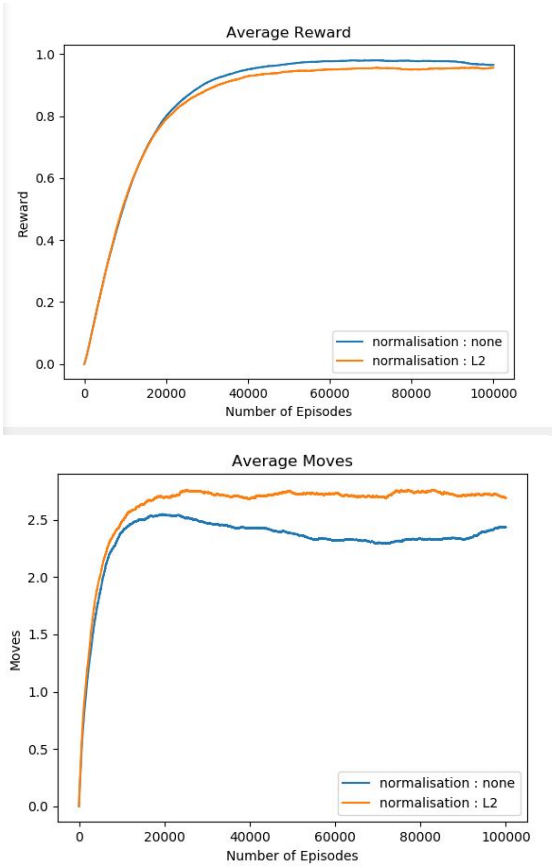

Fig. 14: Average reward and average moves per episode for normalisation experiment. With no normalisation the weights explode and towards the end of the training the reward starts to decrease and number of moves increases. It is expected that this would proceed further if training would continue.

C. Code Snippets

```
def initW(x, N_a, init):

    n_input_layer = x.shape[0]  # Number of neurons of the input layer.
    n_hidden_layer = 200  # Number of neurons of the hidden layer
    n_output_layer = N_a  # Number of neurons of the output layer.

    # Initialise the weights between all layers.
    if init == 'uniform':
        W1 = np.random.uniform(0,1, (n_hidden_layer, n_input_layer))
        W1 /= (n_input_layer * n_hidden_layer)
        W2 = np.random.uniform(0,1, (n_output_layer, n_hidden_layer))
        W2 /= (n_hidden_layer * n_output_layer)
    elif init == 'henormal':
        W1 = np.random.randn(n_hidden_layer, n_input_layer)
        W1 *= np.sqrt(2/(n_input_layer+n_hidden_layer))
        W2 = np.random.randn(n_output_layer, n_hidden_layer)
        W2 *= np.sqrt(2/(n_hidden_layer+n_output_layer))
    elif init == 'xavier':
        W1 = np.random.randn(n_hidden_layer, n_input_layer)
        W1 *= np.sqrt(1/(n_input_layer))
        W2 = np.random.randn(n_output_layer, n_hidden_layer)
        W2 *= np.sqrt(1/(n_hidden_layer))

    # Initialise the biases for both layers
    bias_W1 = np.ones(n_hidden_layer) * 0.1
    bias_W2 = np.ones(n_output_layer) * 0.1

    # Store in the dictionary
    W = {}
    W['W1'] = W1
    W['W2'] = W2
    W['bias_W1'] = bias_W1
    W['bias_W2'] = bias_W2
    W['dW'] = 0

    # YOUR CODES ENDS HERE

    return W
```

Fig. 15: Code for weight initialisation. The size of layers are dependent on the shape of input features x and total number of available action. Tree different initialisation techniques are implemented where optimal one found to be *henormal*. It initialises the weight randomly from standard normal distribution and normalises them based on number of connection from previous layers. The biases are initialised to 0.1. The weights are then stored in the dictionary which allows to pass all the values between methods in a single values and makes code more clear and readable.

```
def Q_values(x, W):

    # Neural activation: input layer -> hidden layer
    act1 = np.dot(W['W1'], x) + W['bias_W1']
    out1 = act1 * (act1 > 0)

    # Neural activation: hidden layer -> output layer
    act2 = np.dot(W['W2'], out1) + W['bias_W2']
    out2 = act2 * (act2 > 0)

    Q = out2

    # YOUR CODE ENDS HERE
    return Q, (act1, out1, act2)
```

Fig. 16: The Q-values for current state are calculated using a two layer network. The output at each layer is calculated using a ReLU activation function.

```
def eGreedy(epsilon, allowed_a, Q):

    eGreedy = int(np.random.rand() < epsilon)
    if eGreedy:
        # Select a random action
        a_agent = np.random.randint(0, allowed_a.shape[0])
        a_agent = allowed_a[a_agent]
    else:
        # Select the most optimal action
        a_agent = allowed_a[np.argmax(Q[allowed_a])]

    #THE CODE ENDS HERE.

    return a_agent
```

Fig. 17: E-greedy policy implementation. If a randomly selected number is below epsilon then a random action from allowed is selected. Else, the most optimal action is selected.

```
    # TD error with future reward = 0
    error = R - Q[a_agent]

    # Apply back propagation to change weights.
    W = backProp(x, error, network, a_agent, eta, W)
```

Fig. 18: Error in final states is calculated using TD algorithm with $Q(s_{t+1}, a_{t+1})$ set to 0. It is then passed into a back-propagation method which updates the weights.

```
    # Calculate the allowed actions for the next move.
    a = np.concatenate([np.array(a_q1), np.array(a_k1)])
    allowed_a = np.where(a > 0)[0]

    if alg == 'Q-learning':
        # Select the next action according to greedy policy.
        a_agent_next = allowed_a[np.argmax(Q_next[allowed_a])]
    elif alg == 'SARSA':
        # Select the next action according to epsilon greedy policy.
        a_agent_next = eGreedy(epsilon_f, allowed_a, Q_next)

    # Calculate TD error
    error = R + (gamma * Q_next[a_agent_next]) - Q[a_agent]

    # Apply back propagation to change weights.
    W = backProp(x, error, network, a_agent, eta, W)
```

Fig. 19: Error in the intermediate states is calculated according to the algorithm used. First the allowed actions from future state are calculated and of Q-learning is used then most optimal action is selected as future action. If SARSA is used, eGreedy action selection schema is applied.

```
def backProp(x, error, network, a_agent, eta, W, L2):

    act1 = 0
    out1 = 1
    act2 = 2

    # Propogate the error from output to hidden layer
    out2delta = error * np.heaviside(network[act2][a_agent], 0)
    dW2 = eta * (out2delta * network[out1])
    # Adjust the weights and biases
    W['W2'][a_agent] += dW2
    W['bias_W2'] += (eta * out2delta)

    # Propogate the error from hidden to input layer
    out1delta = (W['W2'][a_agent] * out2delta) * np.heaviside(network[act1], 0)
    dW1 = eta * np.outer(out1delta, x)
    # Adjust the weights and biases
    W['W1'] += dW1
    W['bias_W1'] += (eta * out1delta)

    W['dW'] += (np.mean(np.abs(dW2)) + np.mean(np.abs(dW1)))

    if L2:
        for key in W.keys():
            W[key] += (-L2 * W[key])

    return W
```

Fig. 20: The error is propagated through the network according to algorithm described in assignment instructions using Heaviside activation. From hidden to output layers only weight for the chosen action are updated. All of the weights are updated from input layer to hidden layer.