INM426 SOFTWARE AGENTS

CITY UNIVERSITY
LONDON

Arshad Ahmed & Konstantinos Stathoulopoulos
MSc Data Science, 2016

TABLE OF CONTENTS

INTRODUCTION

Reinforcement Learning are a class of problems and its solutions are classed as Reinforcement Learning algorithms. These problems are primarily concerned with a machine or software agent learning how to map situations to actions to maximise a certain numerical reward. This reward signal is known as the reinforcement signal. The aim is for the agent to decide on the best action based on its current state. When this step is repeated this type of problems are called Markov Decision Process (MDP). One of these solutions is the Q Learning algorithm[1][2]. The Q Learning algorithm is one such method of solving reinforcement learning problems. It is our aim to evaluate the effect of parameterisation on the Q Learning algorithm.

PROBLEM DEFINITION

We formulate our problem as that of finding the shortest path in a grid by our agent. To do this we initially define a (6, 6) grid and then increase the scope of the problem to be more challenging by making the grid size larger to (10, 10) and then (100,100). In all cases the task is for the agent to find the shortest route or optimal policy through this. We consider two formulations of the Q Learning algorithm in this study: deterministic and stochastic. Some of the analysis is performed with the deterministic version and some are conducted with the stochastic version. Hence, going forward we will present these two cases separately and assess the effect of different parameterisation on their respective convergence and final results.

TOOLS

We use IPython [3] for this analysis in addition to the NUMPY[4], MATPLOTLIB[5], SEABORN[6] and PANDAS[7] packages. In addition to these the stochastic version of the Q Learning algorithm utilises the MDPTOOLBOX[8] package. It is worth noting that Python uses 0 based indexing.

ANALYSIS ROADMAP

Since we are considering two cases of the Q Learning algorithm we present an overview of the analysis conducted with each implementation.

Q Learning: Deterministic	Q Learning: Stochastic
We consider a small (6, 6) grid from now on referred to as small grid and we conduct the following analysis: **Basic Case:** • For the small grid define a learning rate as a function of the number of episodes • Define a State Transition function and the Reward matrix, R • Define an epsilon greedy policy, epsilon = 0.8 • Define discount factor, gamma = 0.2 • Initialise the Q matrix as zeros and show updates after 1200 iterations • Represent performance by looking at the discrepancy value at each episode, the final V values and comparing number of steps at each episode **Advanced Case 3: Different Learning Rates** • We define a fixed learning rate and compare the results for epsilon greedy policy **Advanced Case 4: Different policy** • Define the learning rate as a function of the number of episodes • Consider the results of an epsilon-greedy and softmax policy with a constant gamma value of 0.2 • Consider the results of an epsilon-greedy and softmax policy with a constant gamma value of 0.8	We consider the larger grid sizes of (10, 10) and (100,100) for this implementation in addition to the (6, 6) case. **Advanced Case 3: Different Learning Rates** • We define learning rate and epsilon as a function of number of episodes. **Advanced Case 2: Different Gamma Values** For the small grid: • We rerun the initial analysis with different discount factor values of 0,0.5 and1 **Advanced Case 5: Different State and Reward functions** A Medium grid world (10, 10) • We change the transition probability matrix and reward matrix by changing the dimensions of the problem from a (6, 6) to a (10, 10) grid. • Here we use the stable parameters from the previous experiments which we deem to be discount factor of 0.5 and vary the maximum number of iterations from 10k to 20k and compare the results.
Extra 1: Expanding the scope of the problem A Big grid world (100,100) • We scale our problem to a 100 by 100 grid and define the P and R matrix to run our experiment. Here we use an agent with discount factors 0 and 0.5 with 50K iterations **Extra 2: Comparison of all the above cases** • We compare the optimal policies, V values, performance values and runtimes for all our stochastic experiments. • We compare performance values, V values and number of steps for all our deterministic experiments.	

THE Q LEARNING ALGORITHM

Q Learning is an off policy temporal difference learning algorithm which in its simplest form can be defined as follows:

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

Equation 1: Q learning algorithm update rule[1]

$$\mathrm{dQ} = \alpha[R_{t+1} + \gamma max_a \mathrm{Q}(S_{t+1}, \mathrm{a}) - \mathrm{Q}(S_t, A_t)]$$

Equation 2: The update term for the Q learning algorithm

$$\text{discrepancy} = \text{abs}(dQ)$$

Equation 3: This is the performance measure that we use for the Q Learning algorithm.

We note that the discount factor and learning rate are incorporated in the delta and they both act as a damping factor on the values that the Q matrix is updated with. Therefore the relevant inputs to the Q Learning algorithm then become:

1. The Reward Matrix, R
2. The discount parameter, γ
3. The learning rate, α
4. The Q matrix, initialised as zeros

Q LEARNING: THE LEARNING RATE, α

The learning rate α determines how much the new information that the agent is acquiring will change what he already knows. Instead of using a fixed value for the learning rate, we defined α as a function of the number of training episodes. The alpha parameter is defined as follows:

$$\alpha = \frac{1}{(\sqrt{n+2})}; \quad n = \textit{episode number}.$$

Equation 4: The learning rate, alpha is defined as a function of the number of episodes. It is defined as the inverse square root of the number of episodes with a constant added to prevent division by zeros. .

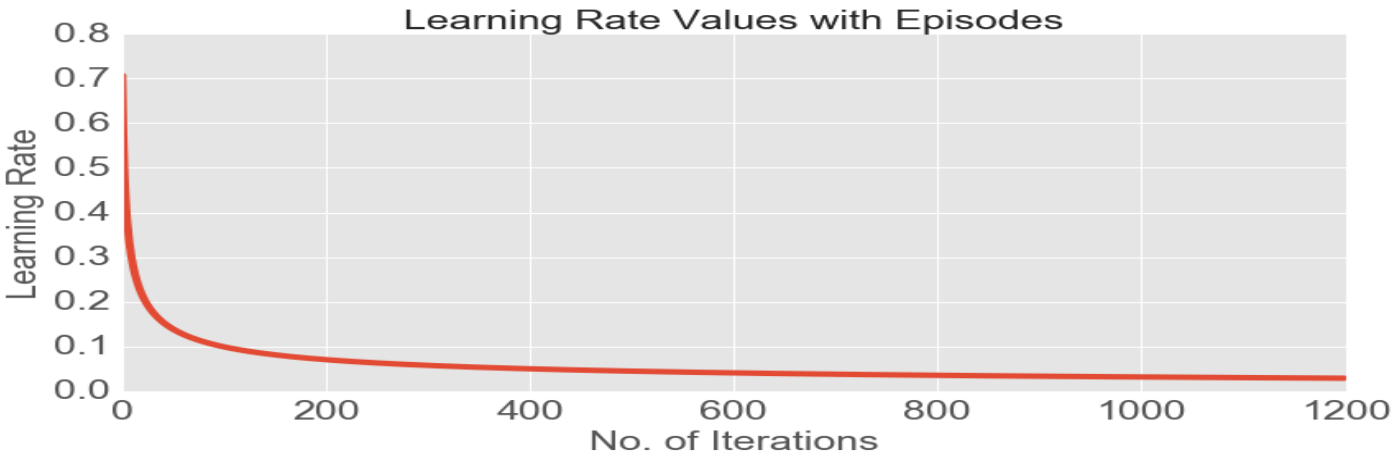


Figure 1: Plot of learning rate, alpha with increasing number of episodes. This is a more dynamic approach to having an adaptive learning rate than a static fixed value.

The impulse response of the learning rate is shown in Fig 1 as defined by Equation 4.This has the effect of applying an exponential smoothing to the Q learning update values. Also the magnitude of its impact is much smaller over time than the gamma value which we keep constant. Since our alpha values change with each episode we say this represents our **Advanced Case 3** and this underlies all our experiments that we present from this point unless explicitly stated otherwise.

Q LEARNING: DETERMINISTIC CASE

In this section we present our analysis with the deterministic implementation of the Q Learning algorithm. Here we define our problem space as a (6, 6) grid. Which gives our agents 6 states and actions to consider. The agent starts at a random position and our goal for him is to reach the state zero. The agent can move only one state at a time. Since we implement an epsilon greedy policy, it compares the epsilon value 0.8 to a number generated from uniform random distribution. When this number is lower than the epsilon the agent is motivated to exploit and in the converse case the agent is motivated to explore. For faster convergence we preferred to bias our agent more towards exploitation than exploration.

Q LEARNING: BASIC CASE

We represent the small grid case and its associated state and reward functions in Fig 1 and Table 1. Fig 1 shows the areas of the grid with high numerical reward with the yellow squares representing the terminal areas. These are the areas that we want the agent to find its way to and the negative reward areas are the ones which we want the agent to avoid. These represent obstacles and in the real world would slow the agent down. But it is more realistic as for example when finding the shortest route one may encounter road works or traffic jams which these could represent. The rewards are given in terms of the states that the agent passed through. The only high positive rewards are given for the terminal state 0, since we want him to stay there as well as for state 2 which is the only point from which the agent is permitted to pass without a penalty.

To do so we initialised a (6, 6) matrix with zeros and assigned to it the values in Table 1. The agent is encouraged to move to the terminal state which has a reward of 100 and avoid the areas marked by negative rewards indicated in yellow.

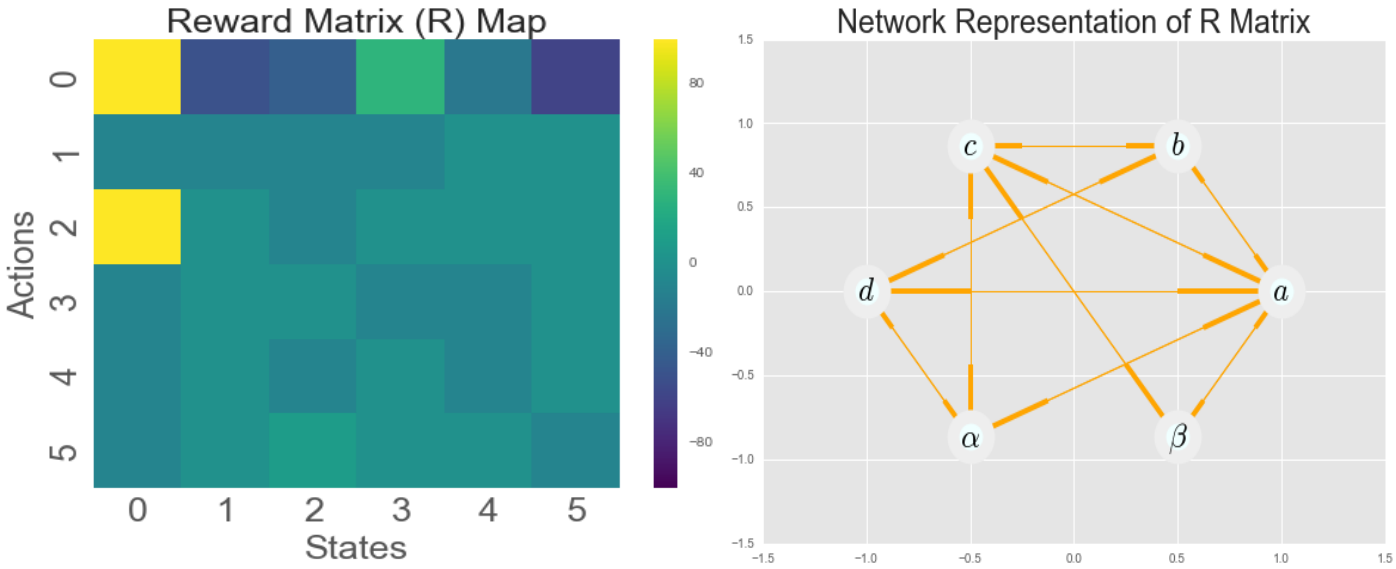


Figure 2: (Left) Map of Reward values showing positive and negative value areas. (Right) Network representation of our R matrix with the terminal state being represented by node 'a'. Also the directed edges shows that the matrix produces a fully connected graph with all states being reachable. The thickened arrows at the incidence of the nodes represent directions.

Table 1: shows our R matrix which contains all the instant rewards that are available to the agent. As it was graphically represented above, our agent can go to any state from any starting position since there are not any states without a link. The rows of Table 2 represent the current state S_t and its columns the next available state S_{t+1}.

	0	1	2	3	4	5
0	100	-50	-40	-30	-20	-60
1	-10	-10	-10	-10	0	0
2	100	0	-10	0	0	0
3	-10	0	0	-10	-10	0
4	-10	0	-10	0	-10	0
5	-10	0	10	0	0	-10

Table 2: State transition function

State, S_t		State, S_{t+1}
0	->	0 or 1 or 2 or 3 or 4 or 5
1	->	0 or 1 or 2 or 3 or 4 or 5
2	->	0 or 1 or 2 or 3 or 4 or 5
3	->	0 or 1 or 2 or 3 or 4 or 5
4	->	0 or 1 or 2 or 3 or 4 or 5
5	->	0 or 1 or 2 or 3 or 4 or 5

For the Q learning algorithm, we use the following parameterisation:

- Learning rate, alpha defined in Fig 1
- Reward Matrix R, defined in Fig 2
- Discount factors, gamma = 0.2
- Maximum number of iterations, n=1200

Q LEARNING: BASIC CASE: epsilon-greedy + γ = 0.2

The parameter γ determines the value of future rewards. It affects the learning of the agent and can be static or dynamic. For the small grid, γ was set equal to 0.2 for the first trial and then it was increased to 0.8, while the rest of the variables remained fixed. This was done in order to compare the magnitude of gamma to the learning process. It should be mentioned that gamma can take values between 0 and 1 and has two extreme cases. For γ = 0, the agent will be myopic with the γmax Q(s, a) becoming 0, so it will not consider future rewards but only the immediate rewards. This approach can only be useful when the reward function is described in great detail and at a later section we will present the findings of this trial. A value of γ=1 will make the agent value the future rewards same as the current ones. This means that there is practically no difference in selecting an action now and in 5 moves. This practice was tested and it is presented later on, but as it will be shown, the learning process doesn't work properly with extremely high gamma values.

Q Matrix initialised as zeros

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

Q learning matrix after an episode

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	-7.011	0	0
2	70.71	0	0	0	0	0
3	0	0	0	0	-9.142	0
4	0	0	-7.071	0	0	0
5	0	0	7.071	0	0	-9.142

Table 3: Q Matrices for the small grid before and after an episode

The Q matrix represents the "brain" of the agent. Initially was full of zeroes because the agent knew nothing about the environment. Its size was the same with the R matrix, (6, 6). Table 3 (left) shows the original Q matrix and (right) its values after the first episode.

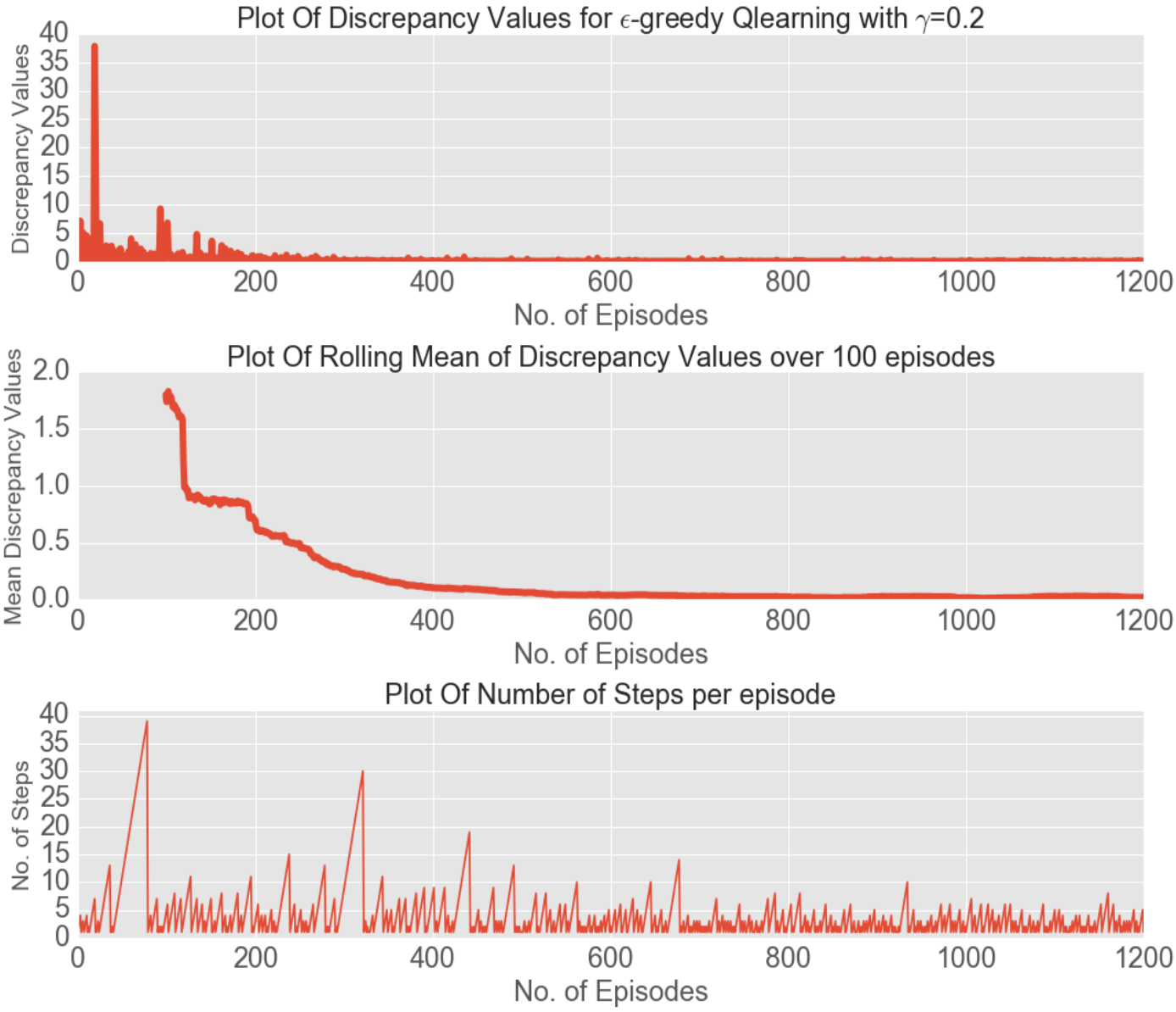


Figure 3: (Top) Plot of discrepancy values for each of the 1200 episodes learnt by the agent calculated according to Equation 3. (Middle) Plot of a rolling with a period of 100, showing the smoothed trend over 100 episodes of the discrepancy values. We observe that the agent converges around 400 episodes without any significant large changes to the Q matrix after this. (Bottom) Plot of the number of steps taken by the agent during each episode. We note that initially the agent explores more so there are a greater number of steps but this drops around 400 episodes and the number of steps reduces due comparatively due its greedy nature.

Final Q Matrix after 1200 episodes for small grid					
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
-5.6366	-6.1608	9.9991	-6.8325	0.9279	5.9923
100.0000	1.6172	9.7332	3.8612	1.1246	5.9384
-9.4915	1.1630	20.0000	-6.5812	-6.1124	5.0640
-7.9945	1.0296	4.3090	2.6536	-9.0312	6.0000
-9.7212	1.3623	30.0000	3.7607	1.1401	-4.4128

Table 4: Final Q matrix learnt by agent after 1200 episodes

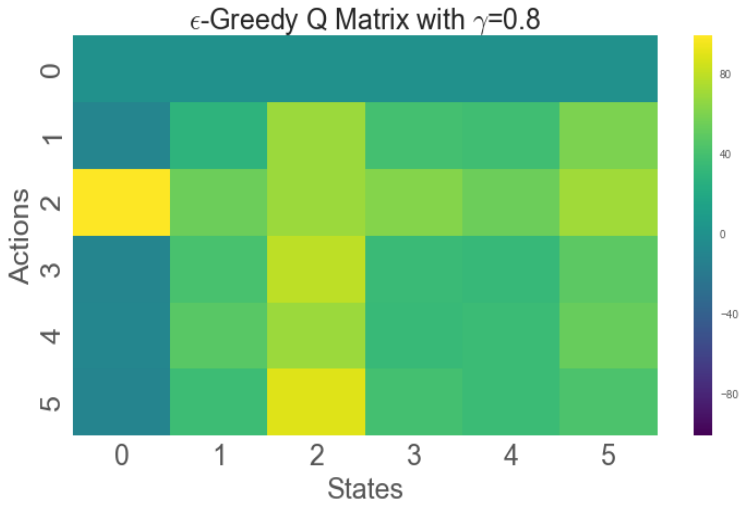


Figure 4: (Right) Final Q Matrix for the small grid case with epsilon greedy policy and discount factor = 0.2. The yellow square represents our terminal state and this corresponds to our R, matrix in Fig 1. The agent has managed to successfully find its way to its destination and also notable in its absence is the lack of large negative values for the Q Matrix. This shows that the reinforcement signal indeed modified the agent's behaviour in avoiding these areas.

Q LEARNING: epsilon-greedy + γ = 0.8

The above results are with an agent that is very short term in nature as defined by its discount factor being so low. So in order to make the agent a more of a long term thinker we increase the gamma value to 0.8 now and consider the results. Since we have shown that the agent behaves as expected in the following sections we present final results only and skip intermediate results.

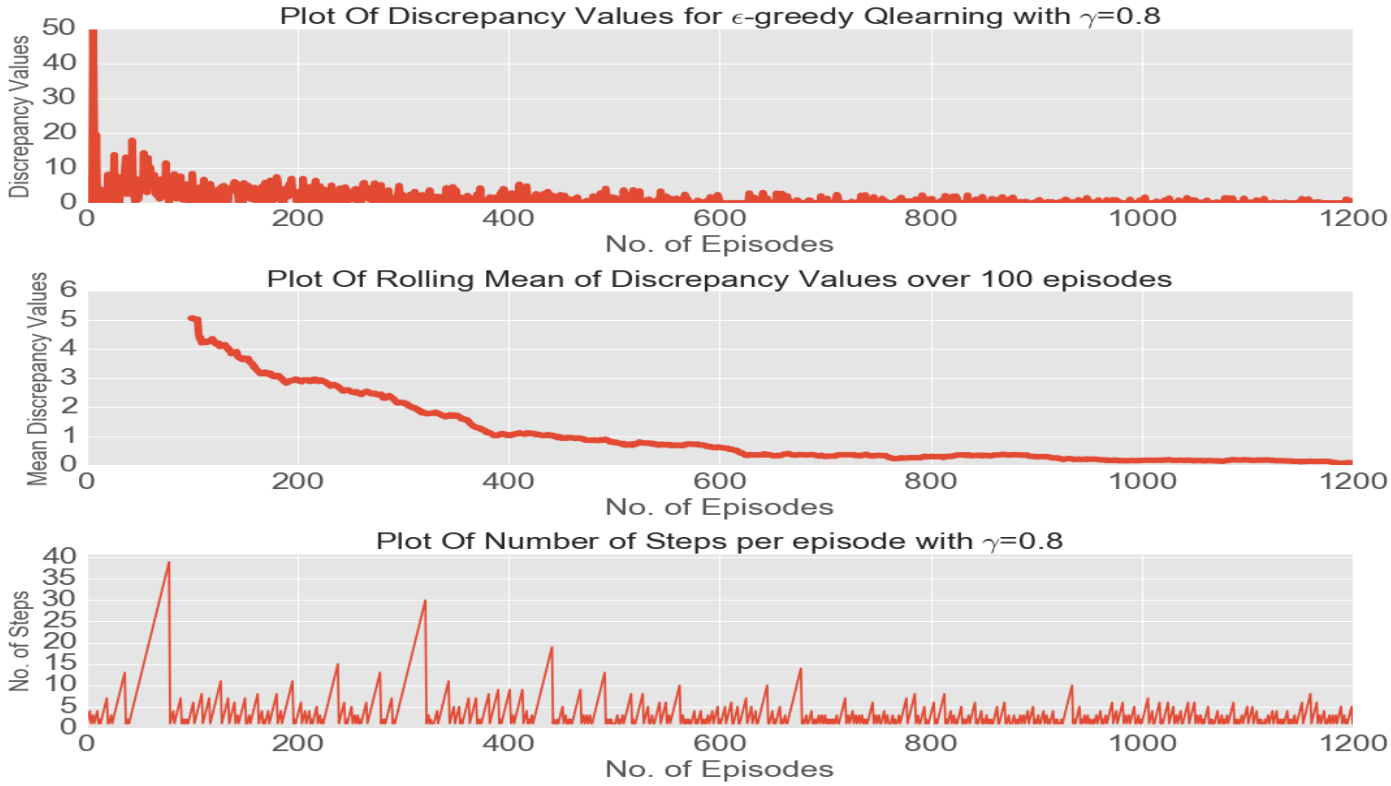


Figure 5: Performance measures for our agents but this time with an increased learning rate. The first point to note is that the discrepancy values are not as smooth as in Fig 3. In the prior case we noticed convergence around 400 episodes but in this case we see that there are significant amount of spikes well into the 600 episode mark. It appears to reach convergence at around 1000 episodes compared to 400 before. The mean plot here shows a much gentler decline over a much greater number of episodes than before. Also the bottom plot of our steps we see that there is a greater amount of steps even in the later episodes in the higher gamma case than the lower gamma case. This indicates that higher gamma values lead to slower convergence due to the long term nature of the agent.

Figure 6: (left) Final Q Matrix for agent after 1200 episodes with gamma = 0.8. The agent eventually finds its way to the terminal state but spends a lot of time in the rest of the grid as part of its long term approach. Hence it accrues higher Q values in some of the other areas defined by the R matrix than the agent in the previous case.

Advanced Case 3: Epsilon greedy + different α

In the previous section we have shown that high gamma value causes the epsilon greedy agent to take longer to converge while the lower gamma values causes faster convergence. So for this set of testing we kept gamma values constant at 0.2 and used three different settings for the learning rate. Firstly, we set alpha as in Equation 4 and then we set it to a fixed value of 0.2 and 0.8 to assess its impact on the agent.

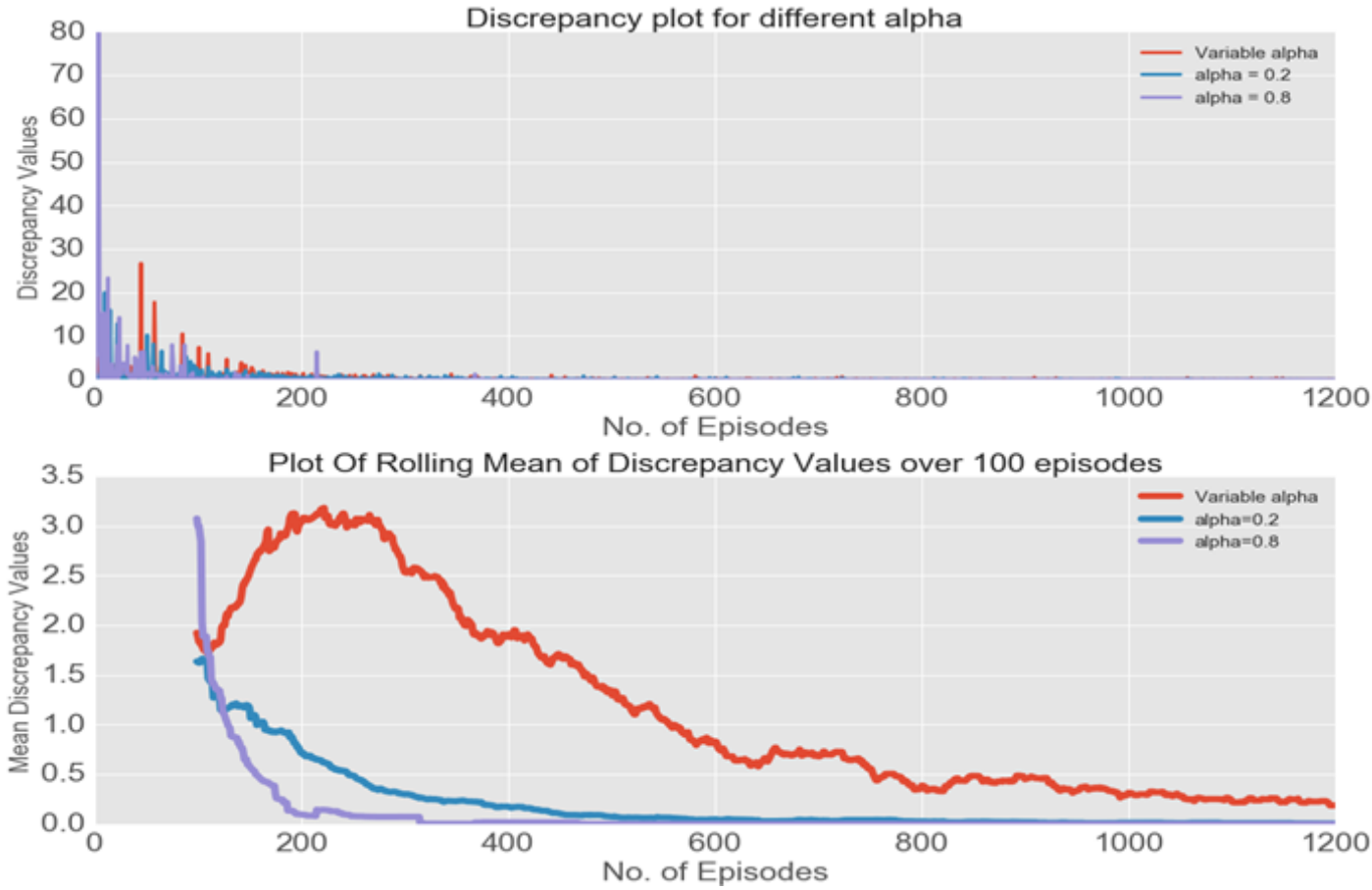
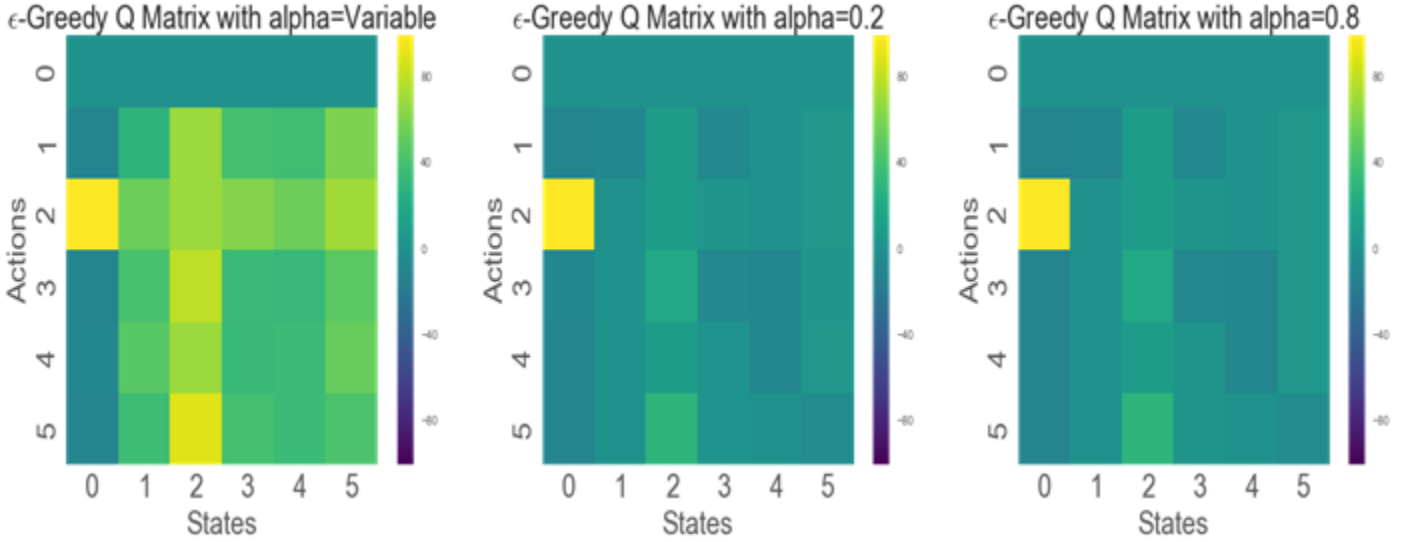


Figure 7: (top) Performance measures for the different learning parameters for the agent and (bottom) associated final Q matrices.



We see that for the different values of alpha the epsilon greedy agent produces fairly consistent behaviour as we can see from the final Q matrix plots. They are nearly identical suggesting that the learning does not have as big of an impact as the gamma parameter which we saw produced much greater variation in the final Q matrix. This is not surprising given how close the trend in the discrepancy plots are. However, it is interesting to see the trends in the rolling mean plot of the discrepancy. Here we see the fixed alpha values converging faster than the variable alpha and higher alpha seems to converge faster than the lower and variable alpha. This could be explained by the update rule for Q learning where the alpha term is used to damp the update terms bigger terms mean bigger damping and while lower terms would cause slower damping of the terms. This is essentially what the plots show so gives us confidence in our implementation of the Q learning algorithm and suggests that despite the variation in the update values the end result for different alphas look similar so there is not much difference to the final solution as to which version of alpha chosen. The only difference would be the convergence time. Even in this case we find that our upper bound of 1200 iterations is sufficient for this to converge.

Advanced Case 4: Softmax Policy + $\gamma = 0.2$

Softmax Policy: The softmax function is used in order to pass from a set of values to a list of probabilities to take an action. These probabilities are calculated with the following formula:

$$P_t(a) = \frac{\exp(q_t(a)/\tau)}{\sum_{i=1}^{n} \exp(q_t(i)/\tau)}$$

Equation 5: The calculation of softmax probabilities given an input. The $q_t(a)$ is the expected value of the reward that will be earned if the agent follows the action a. As it can be observed, the action with the highest value will have a probability closer to 1. The second parameter that we have to mention is the temperature, τ, and parameter. For low temperatures, the expected rewards affect the probability while for high temperature values, all the possible actions have quite similar probabilities.

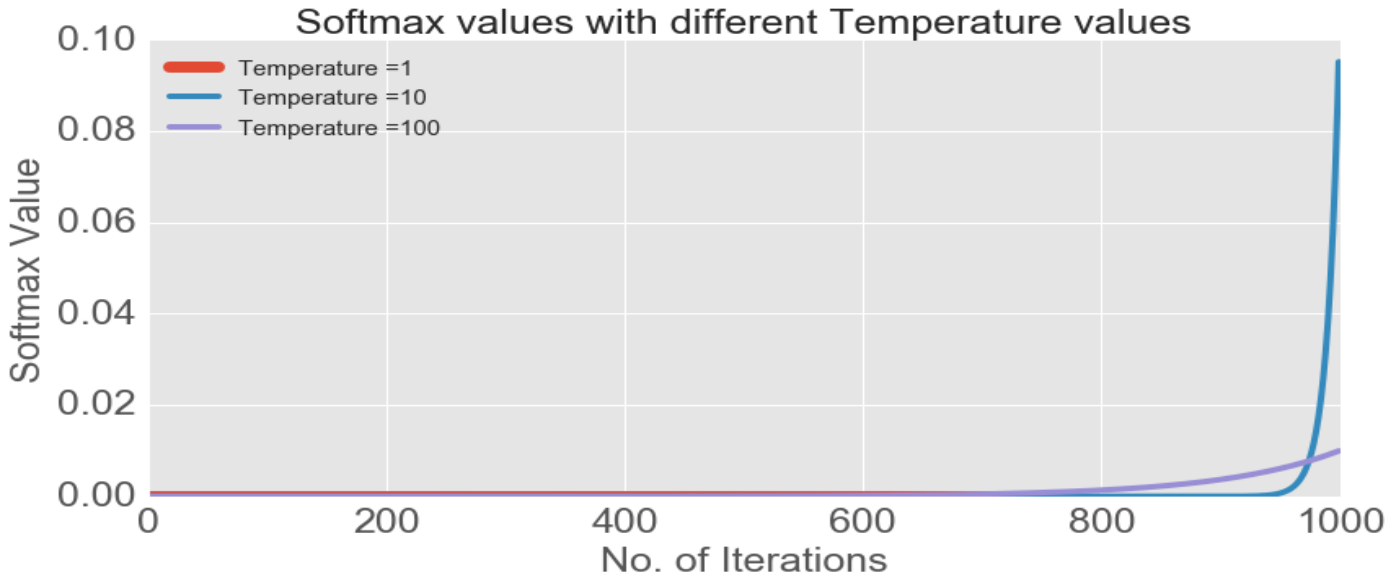


Figure 8: Impulse response for the softmax function with different temperature values. We see that a value of 1 has no effect, while 100 causes a very gentle increase but our chosen value of 10 gives a good balance of increasing probabilities for the maximum number of iterations we have chosen.

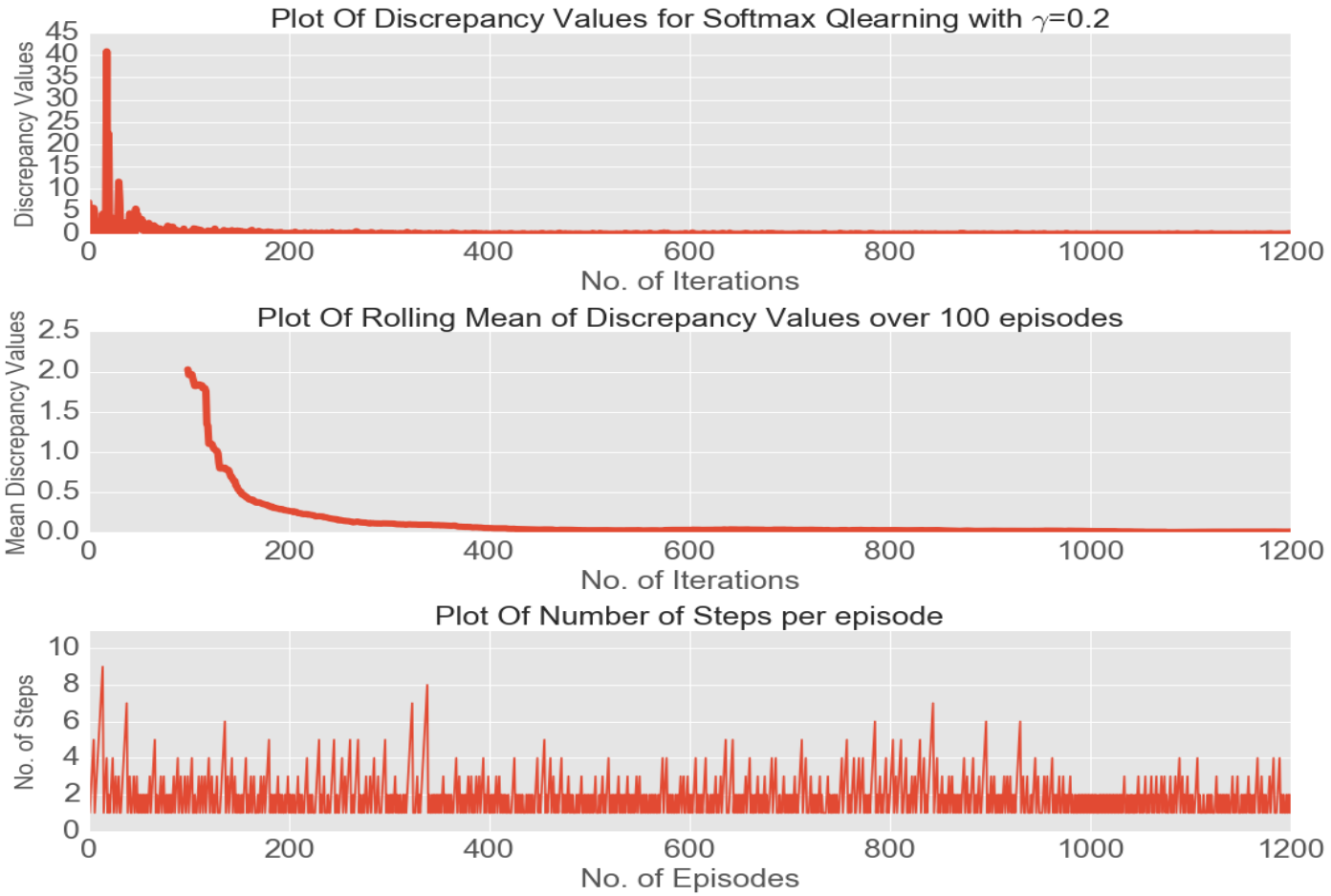
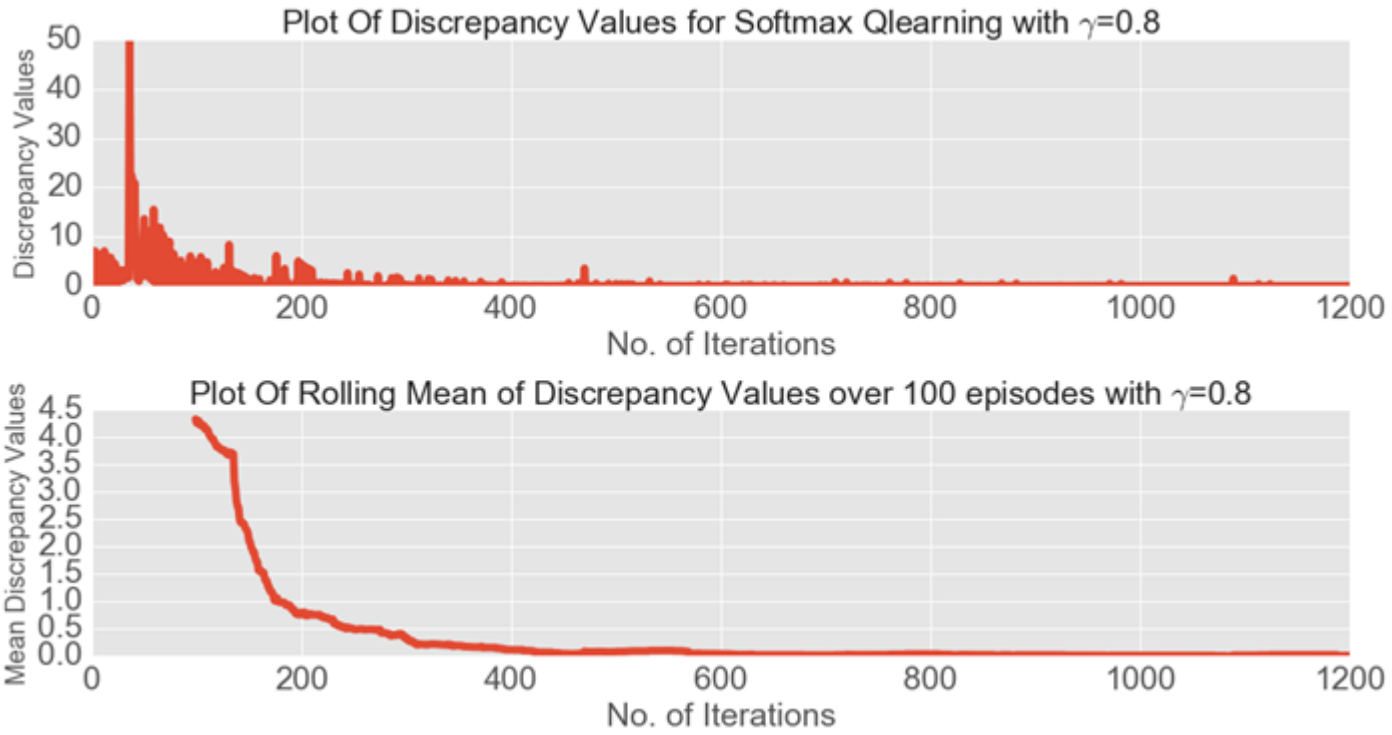


Figure 9: Softmax policy performance with gamma = 0.2. We see that this policy is clearly superior to the epsilon-greedy approach for this problem. While the agent with the epsilon greedy and gamma = 0.2 converged around 400 episodes we see that the agent here converges around 250 episodes in the mean discrepancy plot. Also from the top plot, the agent learns the highest Q values in under 100 episodes.

Advanced Case 4: Softmax Policy + γ = 0.8

To enable us to compare with the epsilon greedy gamma = 0.8 case we repeat the previous experiment with the icnreased gamma value.



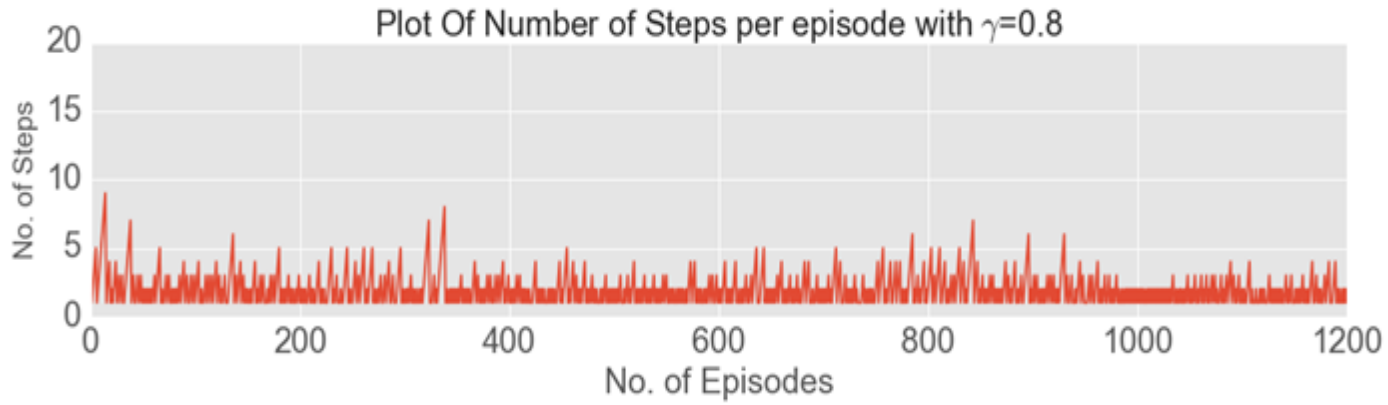


Figure 10: Plot of performance measures for Softmax with increased gamma values. Here the softmax really shows its stability over the epsilon greedy method. With the similar epsilon greedy case we saw that the convergence was pushed back to around 1000 episode mark but even with the increased gamma the softmax converges around 300 episodes. This is much less than the increase observed in the epsilon greedy case. The trends in the discrepancy and mean discrepancy are consistent with Fig 9.

Advanced Case 4: Comparison of Softmax and Epsilon-greedy

In this section we consider the two policies that we have tested so far in our small grid problem: epsilon greedy and softmax. We analyse the gamma value 0.2 and 0.8 cases for each policy together.

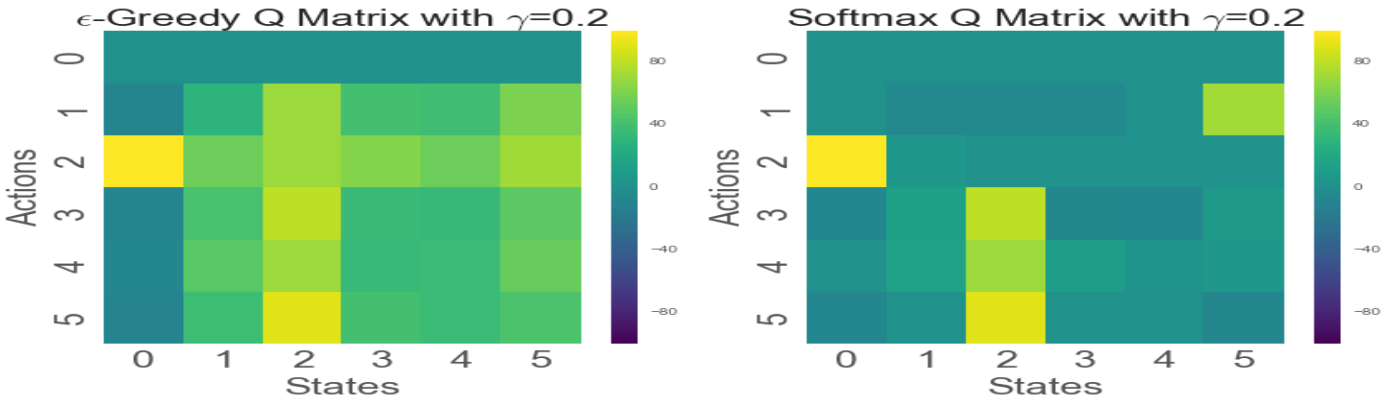


Figure 11: Comparison of the final Q matrices learnt by the agent with the different policies for the same gamma values. We do not observe much difference here and in both cases the agent finds the terminal state although it does so faster with softmax as shown in Fig 9,10.

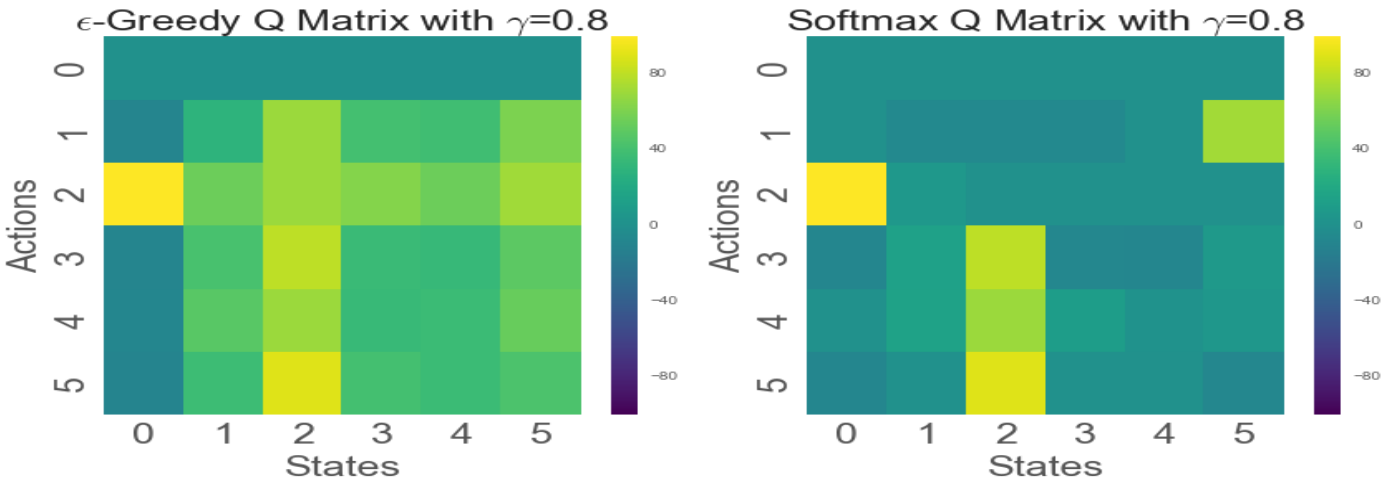


Figure 12: Comparison of final Q matrices for the agent with gamma = 0.8. The divergence between the two policies is much clearer at this value of the discount factor. We see that in both cases there is more exploration by the agent of the rest of the grid and we see more bright spots in the Q matrix as a result of this. This is to be expected as the agent is taking a more long term view of rewards but amount of the bright spots in the map is of interest as we see that the softmax policy is much better. The agent with softmax explores but its exploration is not erratic and all over the map as it is with the epsilon greedy policy.

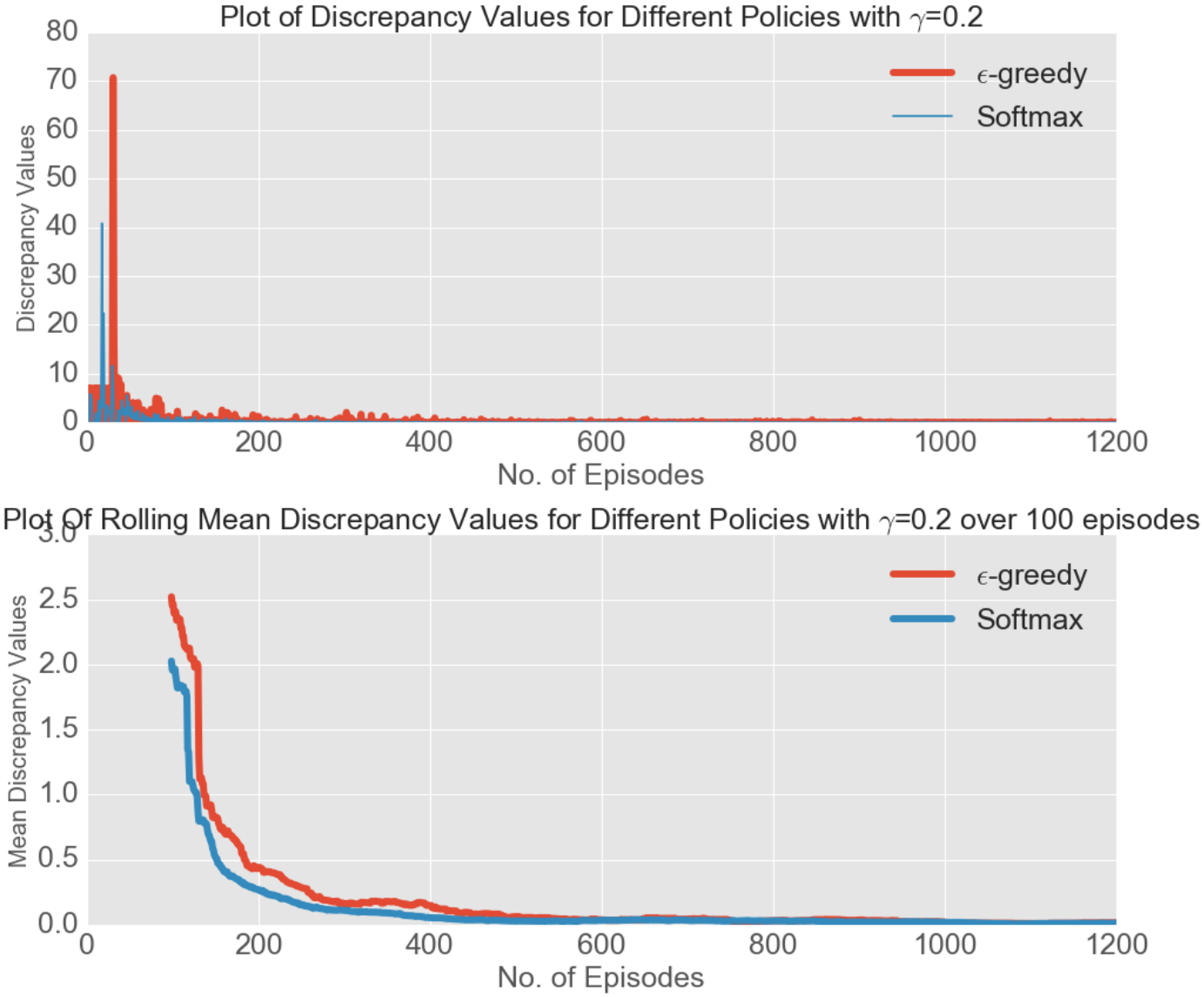


Figure 13 (top): Performance comparison for epsilon greedy and softmax policy for gamma = 0.2. We see that the performance is similar for both policies with this gamma values however, the softmax policy has Q values of smaller magnitude in comparison to the epsilon greedy and converges faster as noted previously.

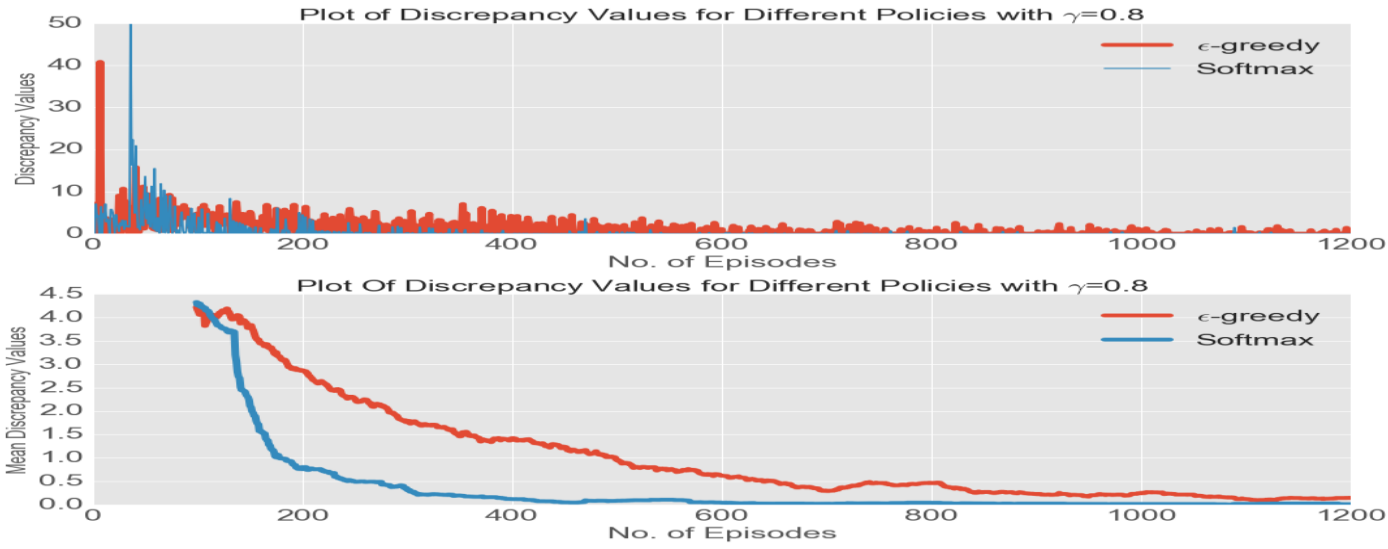


Figure 14 (top): Performance plot of epsilon greedy and softmax policies with gamma = 0.8. As noted we see clearly in the plot of the mean discrepancy that the softmax does not require more episodes to converge as the epsilon greedy does with higher gamma values but still converges on a similar number of episodes. From this we can say that an epsilon greedy policy is more susceptible to gamma parameter changes than the softmax policy.

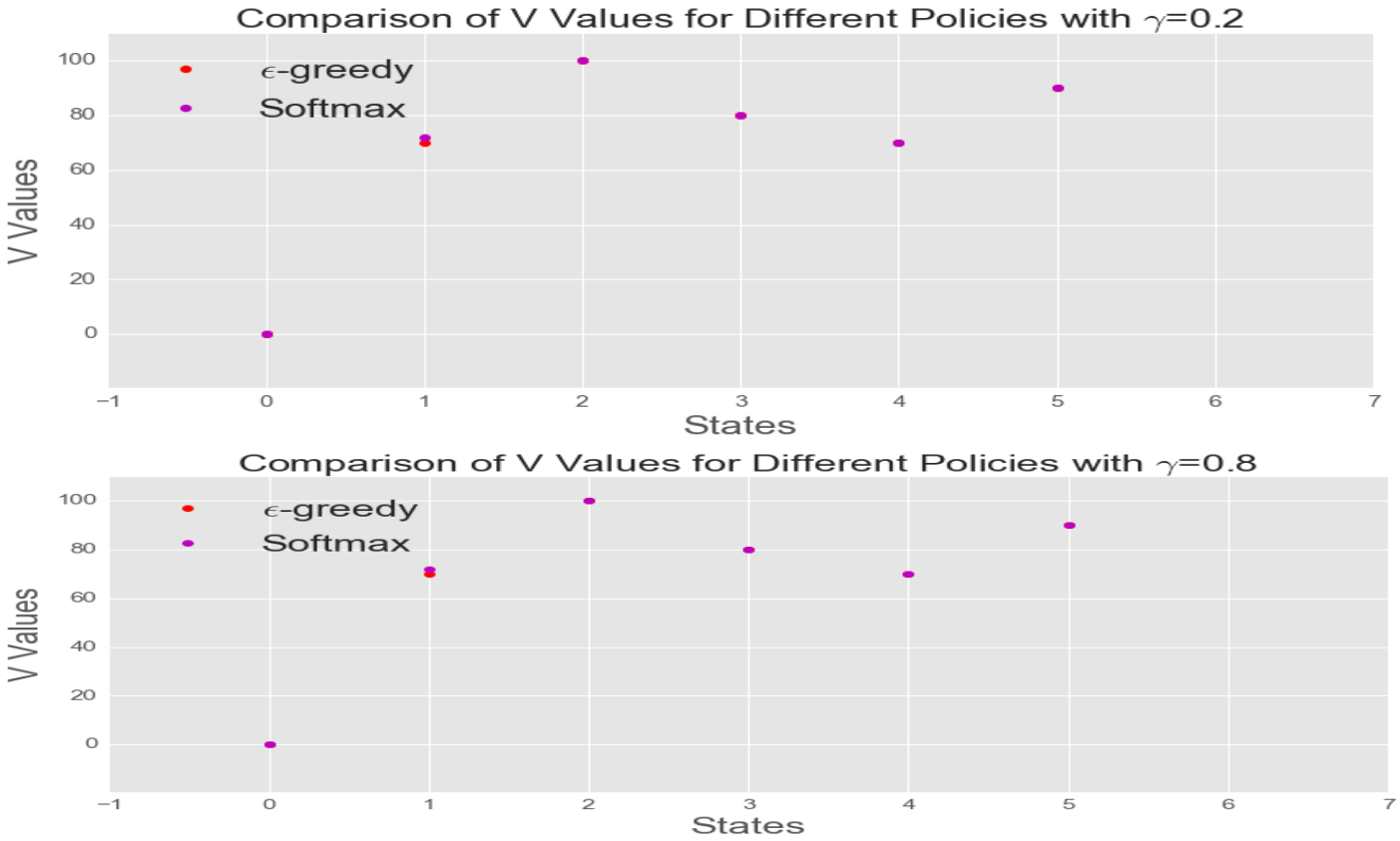


Figure 15: (top) V values for both policies with gamma = 0.2 and 0.8. We define the V values as the maximum Q values along the column direction. The softmax learns mostly similar V Values as the epsilon greedy with only one point being significantly different for small gamma. With the higher gamma value however we more variation in the V values of the epsilon greedy policy. This again gives evidence that gamma affects epsilon greedy policy more than softmax policy as the V values for the softmax are fairly similar in both cases. This could be explained by the fact that the softmax assigns higher probabilities for actions over increasing episodes which favours exploitation rather than exploration. With the epsilon greedy policy the greedy nature of the policy probably does not complement the longer term outlook we are trying to encourage the agent to take with the higher gamma values. Therefore it ends up being greedy in its exploration rather than the exploitation which was the case with low gamma.

Q LEARNING: STOCHASTIC CASE

This set of analysis is conducted with the MDPTOOLBOX. Also, the MDPTOOLBOX represents a state transition probability matrices and the reward matrices as a 3 dimensional matrix of the form (Action, State, State). When these hypothesis spaces are visualised, we stack them in the vertical direction to allow representation onto a 2D plane. Also the default number of iterations is 10K and we use this for all the analysis unless stated otherwise. The alpha value is defined as a function of the number of episodes as given in Equation 4. The policy selection is epsilon greedy with increasing probability also defined as a function of the number of episodes.

$$\epsilon = 1 - \frac{1}{\log(n+2)} : n \ = number\ of\ episodes$$

Equation 5: Epsilon greedy policy selection as function of the episode number.

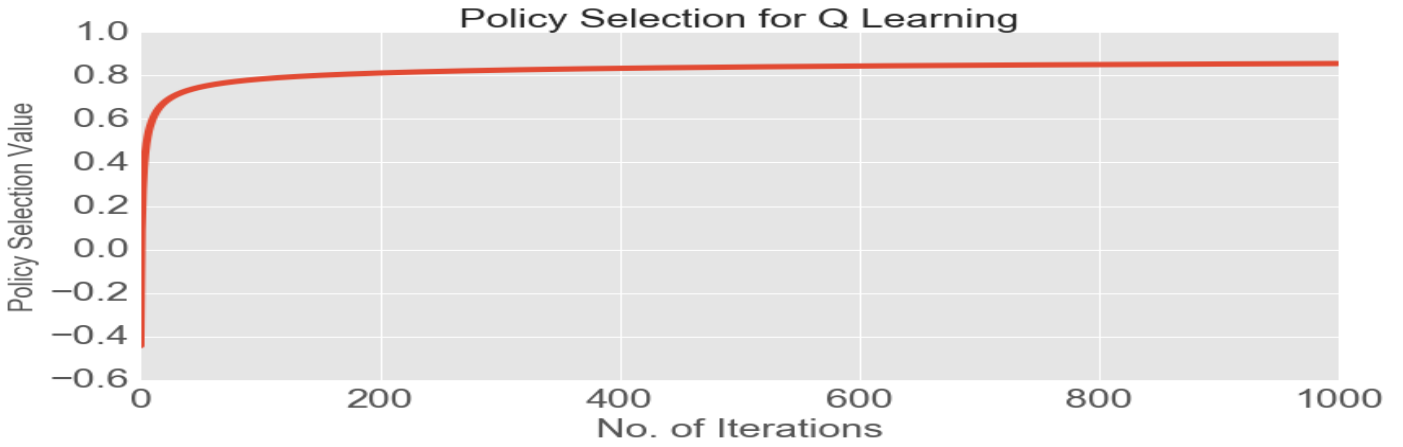


Figure 16: Impulse response for the epsilon greedy policy selection for the stochastic case

We see that the value increases over time meaning our agent gets greedier as the number of training episode increases so we move from more exploration to more exploitation. This is in our opinion better than having a static value of epsilon as it allows for a good balance between exploration and exploitation. This strategy allows the agent to follow exploratory moves at the first episodes and as it approaches the end of the iterations, its behaviour becomes greedier. In this way, the agent learns an optimal solution.

How it works

This package solves for discrete time Markov Decision Processes. It also features a rich set of algorithms such and utilities to solve these problems. We will use the Q Learning implementation from here to analyse the performance our agent in a stochastic environment of increasing dimensionality. The package features an 'examples' method which allows the user to generate a valid MDP consisting of valid state transition probabilities and associated reward functions. We use the random function from the example class to generate an MDP of the desired size. So the size of our stochastic grids increase from (6, 6) to (10, 10) and finally to (100, 100).

This function takes as an argument two integers for the number of states and actions in addition to return a sparse matrix. We choose the dense matrix representation of our problem. Based on the input number of states and actions the state transition probability matrix, P is initialised with zeros in the form (Action, State, State) and the Reward matrix, R is also initialised as zeros in the form (Action, State, State). Then for each value in the range of actions and states the function iterates over the initialised matrices to populate them. The values are draw from a continuous uniform distribution over a stated interval. For the P matrix this is [0, 1] because it represents probabilities. For the R matrix, the range is [-1, 1] because we want to reinforce our agent to avoid some areas while reaching high probability areas. The function also checks the P matrix to ensure that each state has at least 1 transition so the agent does not get stuck. The P matrix is then normalised to ensure that the probabilities are valid probabilities and the R matrix is checked to ensure that its values lies in the [-1, 1] interval. The output is a dense P and R matrix. We use this function to generate valid probabilistic MDP of dimensions (6, 6), (10, 10) and (100, 100).

So far we have presented results of a deterministic Q learning agent and have seen that the discount factor has a strong impact on its performance. We would like to assess whether we see the same pattern in our stochastic case. To do so we consider a (6, 6) grid again and consider gamma values of 0, 0.5 and 1. The rationale for choosing these values are to see how the agent performs at the edge of myopia and long term outlook.

This section is organised as follows. We first present the case where we evaluate the extreme gamma values and its impact on a (6, 6) stochastic grid. We then evaluate a larger stochastic grid, hence a different transition and reward matrix and evaluate the effect of increasing the number of iterations from 10K to 20K on the performance of the agent. Then we expand the scope of our problem by considering a Big Grid of (100, 100) and evaluate the agent with gamma = 0, 0.5 and 50K iterations. Lastly, we conclude by comparing all the different cases and presenting our conclusions.

Q LEARNING: Advanced Case 2: Different γ on stochastic grid

We first present our Transition Probability Matrix, P and the Reward Matrix, R for the stochastic case in Fig 17. We only have positive values in the P matrix as they represent probabilities.

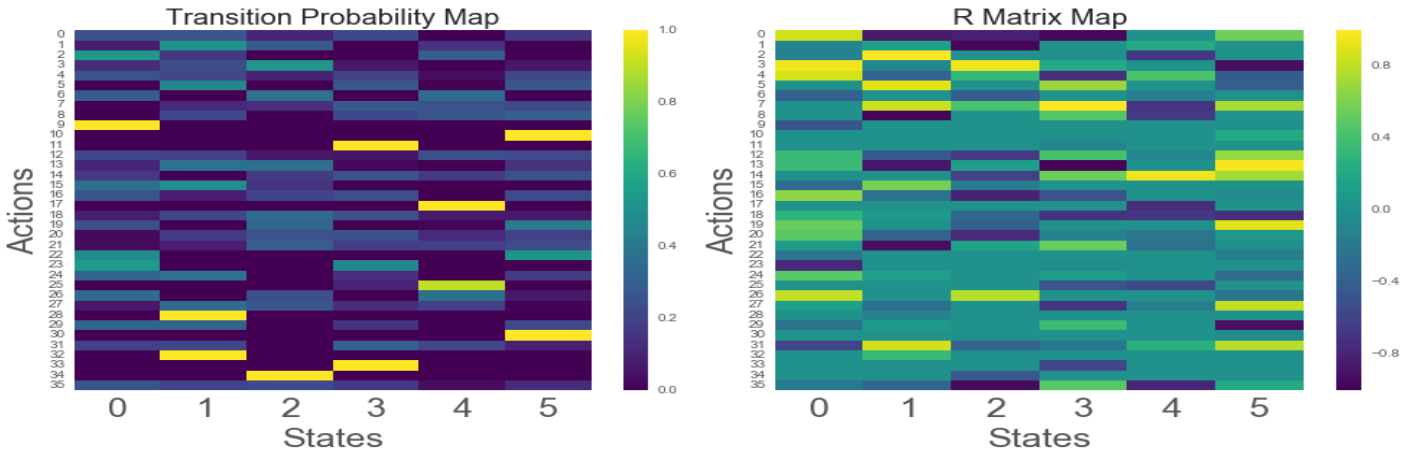


Figure 17: (Left) Probability Transition Matrix, P for our grid problem. The light colours represent areas of high probability and dark colours represent areas of low probability. (Right) Reward Matrix, R through which we tell our agent how to determine the optimal policy. Here the light areas represent where we would like our agent to go and the negative values or dark areas where we would like the agent not to go.

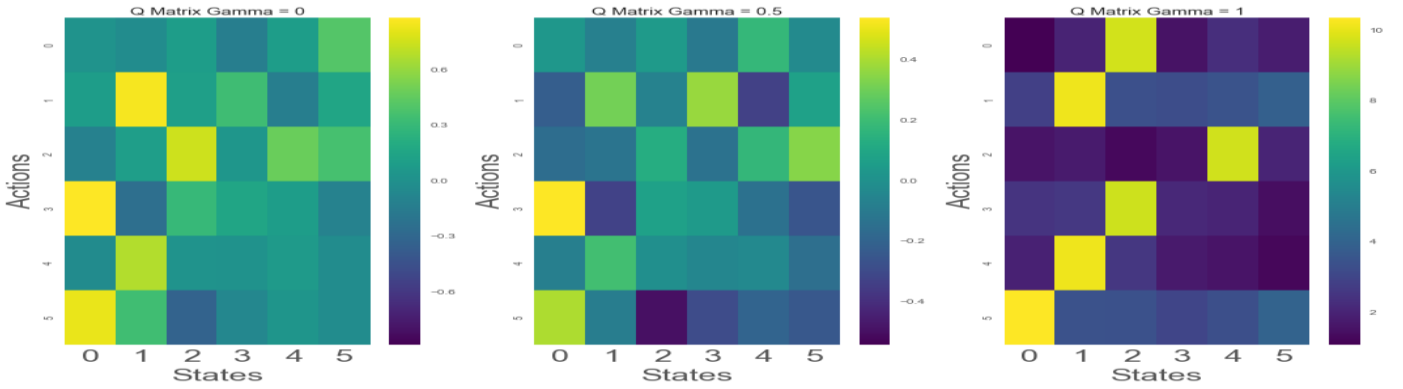


Figure 18: Final Q Matrix comparison plot for Q Learning with different gamma values.

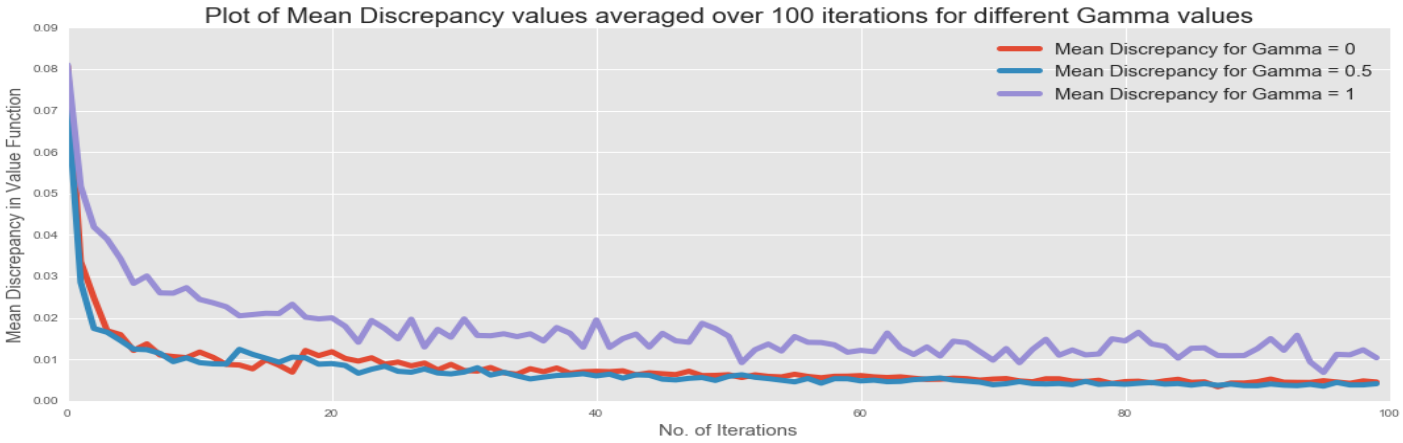


Figure 19: Mean Discrepancy plot for 3 runs of Q Learning with different gamma values. We observe that the 0 and 0.5 cases achieve convergence around 40 while the 1 does not appear to converge. When gamma parameter is equal to one, the agent weights future rewards equal to the current ones. This means that there is no difference between the reward of an action taken now and taken some moves later. In this extreme case, learning does not work and consequently, it fails to converge. **The mean discrepancy is averaged over (n/100) episodes**.

We see that there is quite a lot of variation between the final Q matrices the agent learns with the different gamma values. Of the different Q matrices we see that the 0 and 0.5 case are fairly similar in the sense that they learn almost similar areas of high Q values although the magnitude in the 0.5 case is slightly smaller. But the gamma = 1 produces a completely different map compared to the rest so we can assume that this value failed to converge. Again, we have strong evidence that gamma values have a strong impact on an epsilon greedy learning agent and that higher values lead to longer convergence times and where it approaches 1 it fails to converge. The curious case is that of gamma = 0 where we would have expected to observe similarly extreme behaviour. But the agent is more stable in its myopia than its long term outlook. This could be down to the probabilistic nature of the P and R matrices for this particular analysis.

Q LEARNING: Advanced Case 5: Different state and reward functions

Here we generate a (10, 10) P and R matrix. Since the dimensions are different the state and reward functions are also different compared to the previous (6, 6) case. In this section we run our Q Learning analysis again with the gamma value of 0.5 and conduct our experiment on a larger grid. We choose 0.5 as this gives a stable result as noted in the previous section. Here, we start with a (10, 10) state transition probability matrix and reward matrix. We run our Q learning algorithm with 10K and 20K iterations and compare the results. We see that both the cases converge to a similar solution after around 5000 episodes. Note that **the mean discrepancy is averaged over (n/100) episodes.**

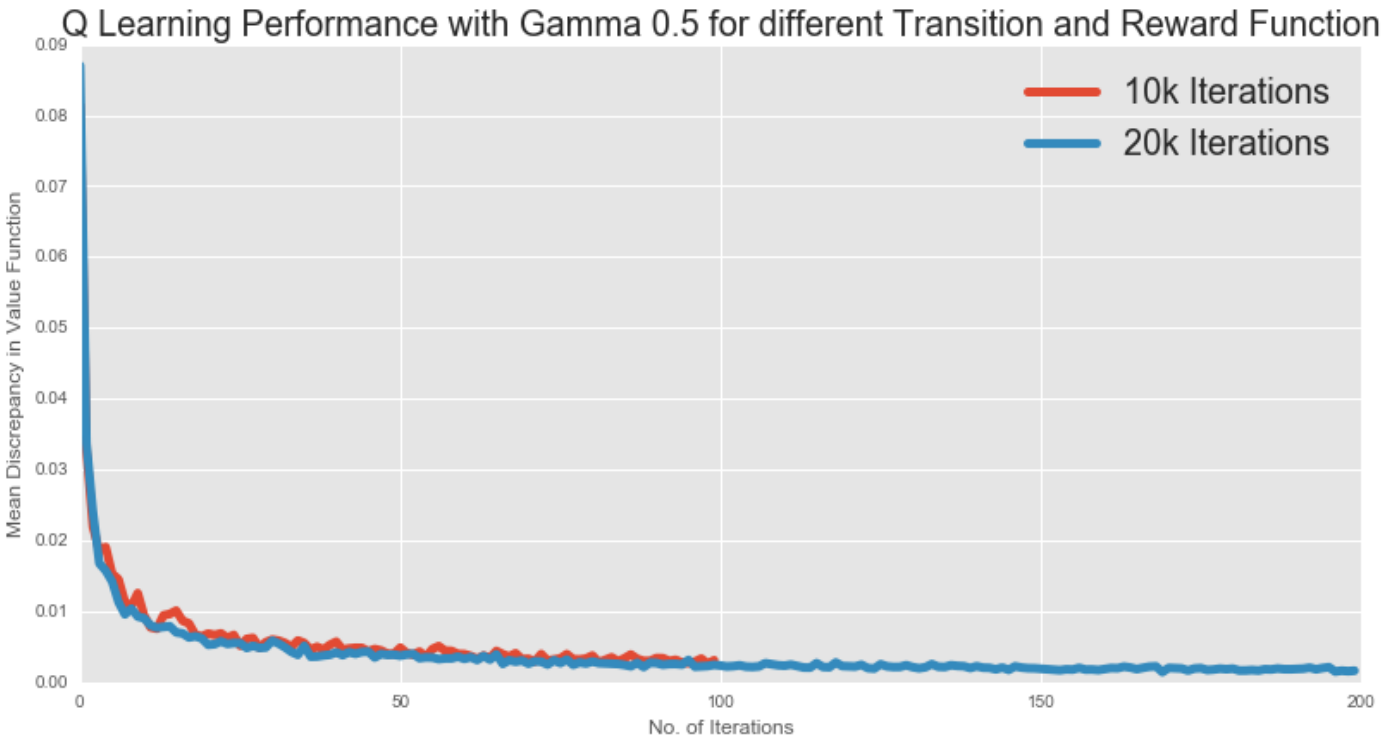


Figure 20: Mean discrepancy plot for Q Learning on a (10, 10) grid with gamma = 0.5. The 20K iterations has more iterations hence the scale for this is longer than the 10k case. Note that these values are smoother over 100 episodes. Both runs seem to achieve a stable solution.

Figure 21 Final Q Matrices for the (10, 10) grid case. We see that some areas have high values and are common to both while the 20K case finds other high value regions.

Q LEARNING: Extra 1: Expanding the scope of the problem

We expand the scope of our problem by considering a Big Grid of (100, 100) and evaluate the agent with gamma = 0, 0.5 and 50K iterations.

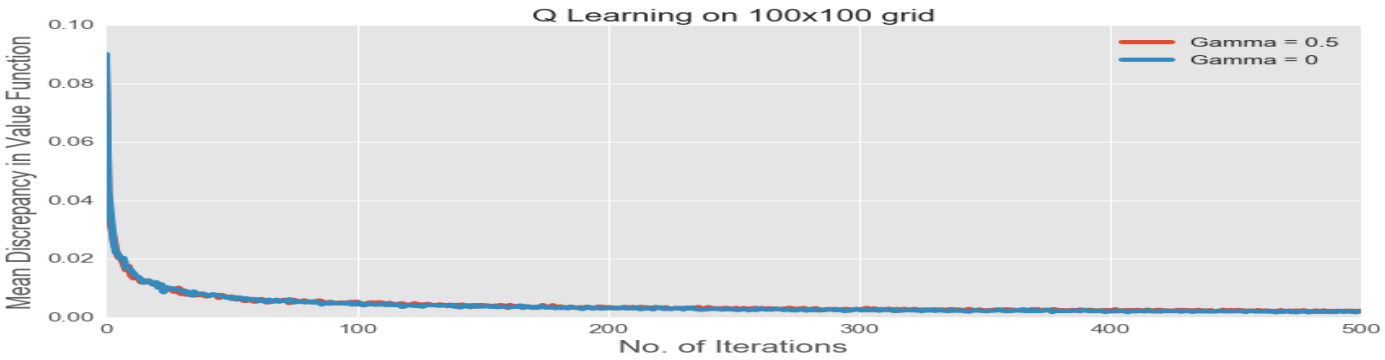


Figure 22: Plot of performance for the agent in a 100,100 grid with discount factor or 0 and 0.5. We see that over 50,000 iterations both the solutions achieve a fairly similar convergence.

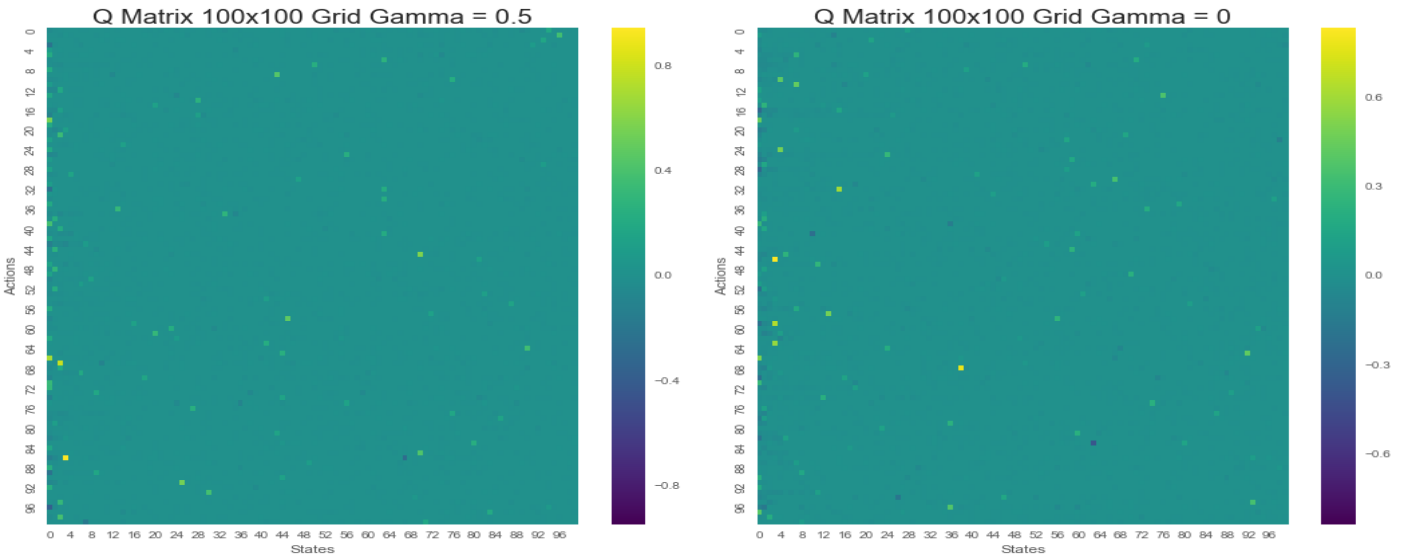


Figure 23: The final Q Matrix for the different cases. The agent with a discount factor of 0.5 in general learns Q values overall of a higher magnitude than the agent with discount factor 0. But we see bright spots on Q matrix for the agent with discount 0 than that of agent with discount 0.5.

Figure 24: Plot of V values for both agents showing that in both cases they learn comparable V values due to the fairly close clustering of points with a few outliers present. These could represent the high probability areas that they find.

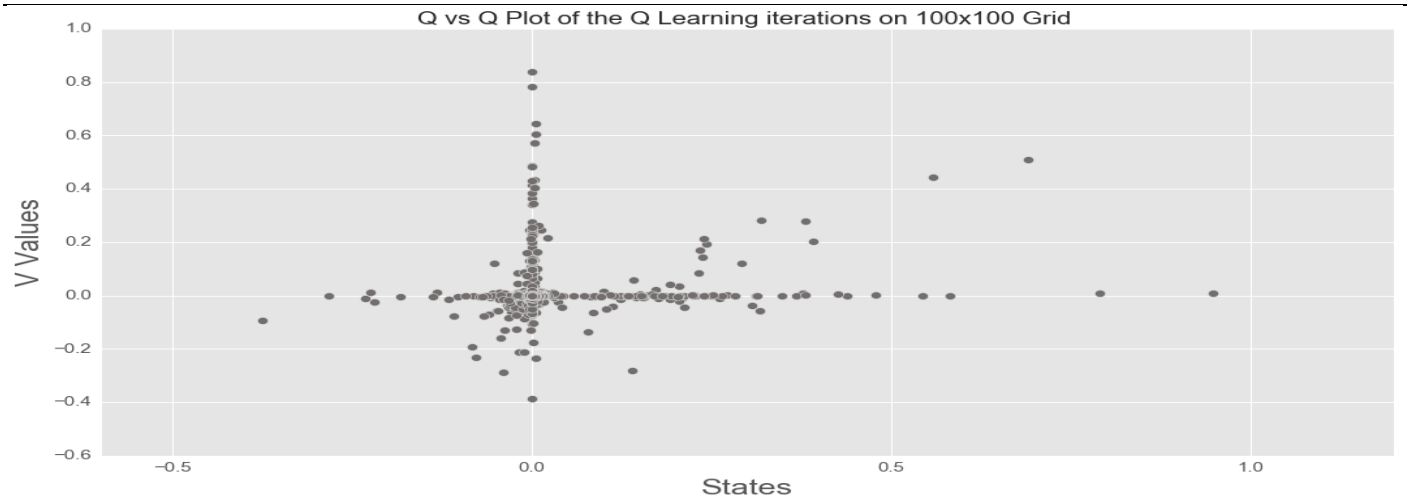


Figure 25: Plot of Q values for both agents showing that most values are clustered close to zero but there is large amount of variation in the Q values learnt. We can also see the direction the agent was going based the trajectory of the Q values.

Q LEARNING: Extra 2: Comparisons Stochastic Cases

Firstly, we consider the different policies that is learnt by the agents under different circumstances in the different hypothesis spaces show in Fig 26. We see when our problem space is fairly small and we vary the discount parameter. The optimal policy learnt by the agents with discount factor 0 and 0.5 are similar while that learnt by agent with discount 1 is highly variable. This suggests that with this value the algorithm probably does not converge within the maximum number of iterations hence could be a reason for the variability.

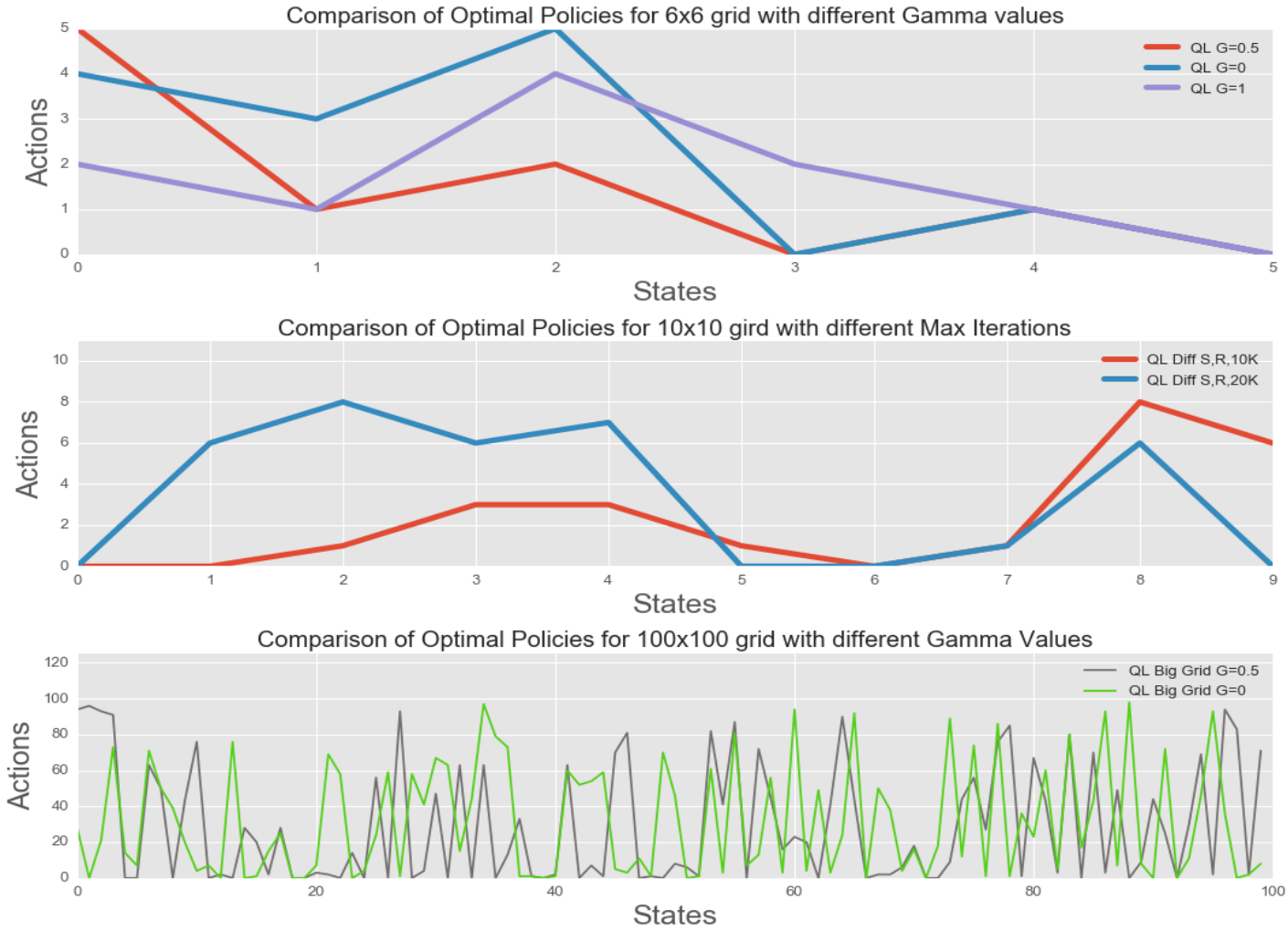


Figure 26: Comparison of optimal policies learnt by the agent for the different cases considered.

Figure 27: V Value comparison across the different runs. The V values learnt by the agent across the different cases we considered. The overall trend is that the agent learns similar V values in all cases where the discount factor is 0 and 0.5 but the case where it is 1 the values diverge sharply. This could be due to the convergence issue we have identified.

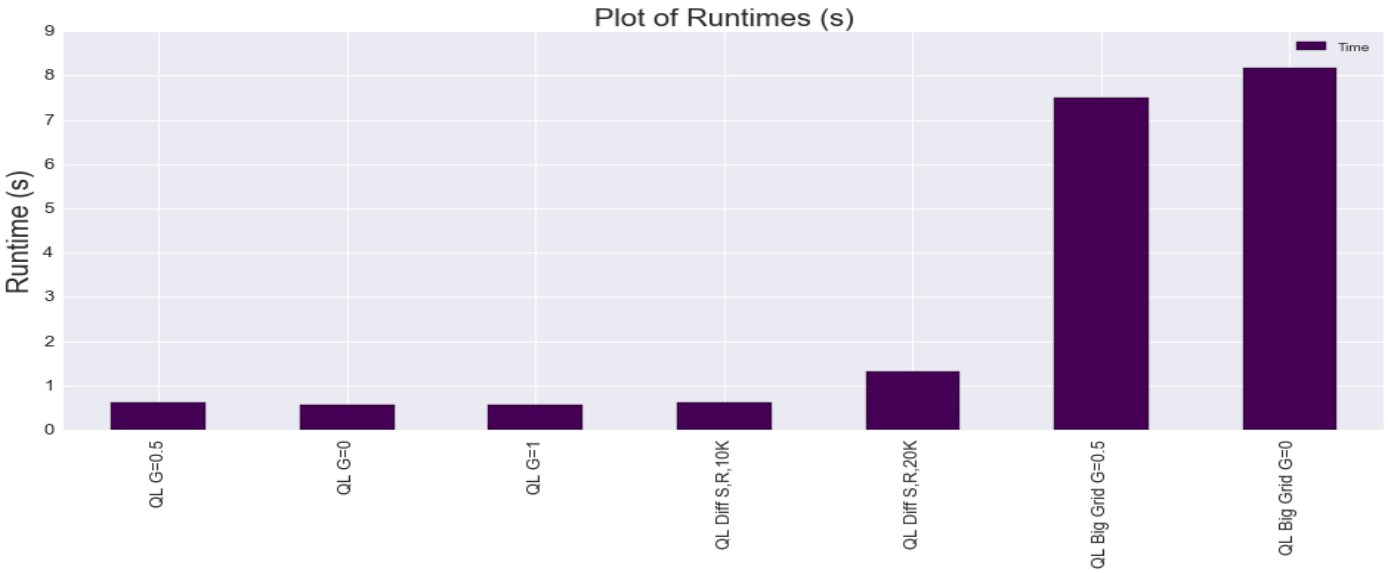


Figure 28: Plot of runtime in seconds for the different experiments. As expected we see that 20K iterations, take longer than 10K iterations and 50K iterations. However we note that discount factor of 0.5 for the 10K iterations was faster than the other values. From a performance perspective it is important to note that a 5 times increase in the number of iterations leads to almost an 8 time increase in the runtime. This suggests nonlinear complexity and could be critical when considering larger hypothesis spaces.

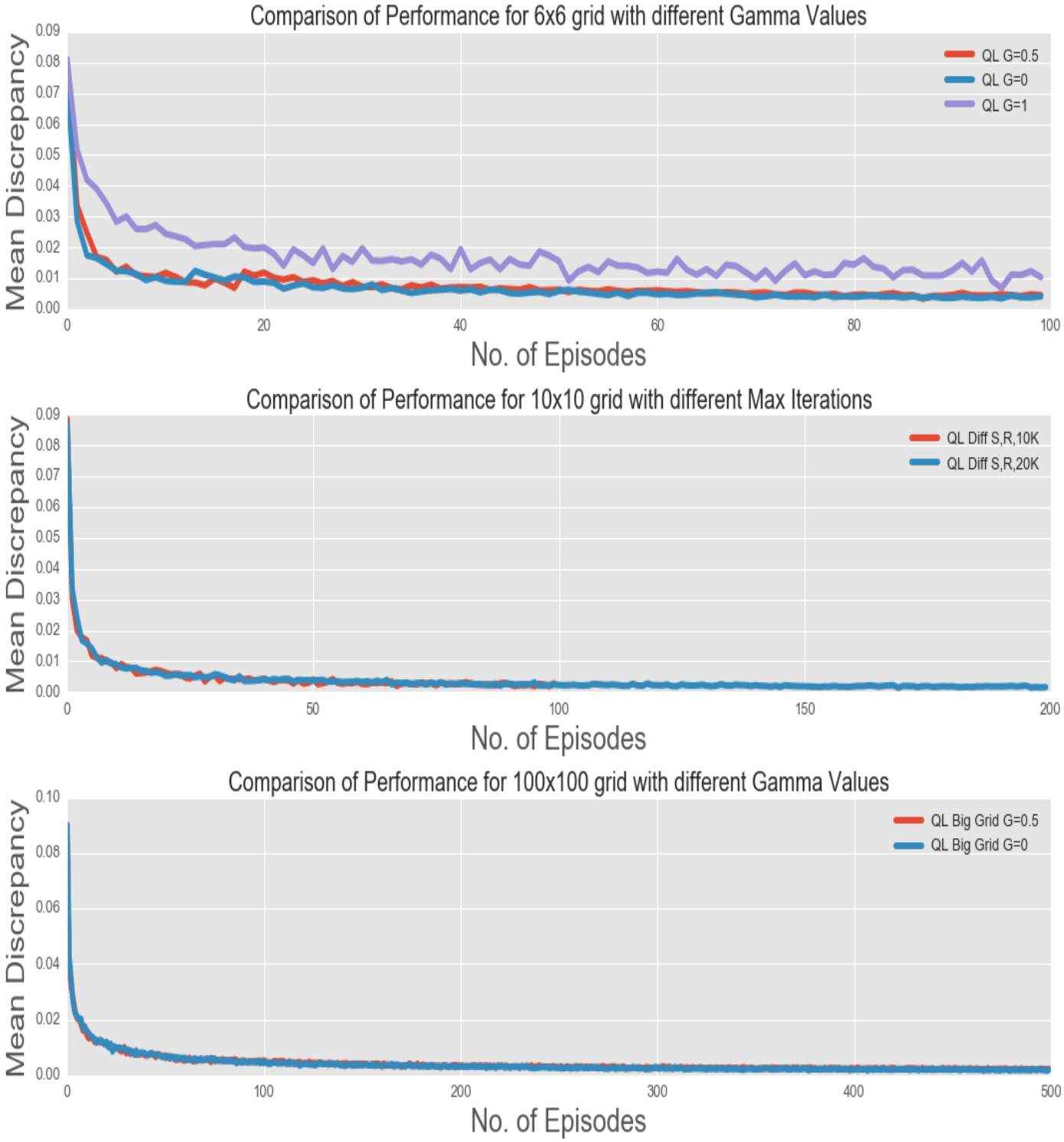


Figure 29: Mean discrepancy over the different runs. The mean discrepancy over the different training runs converge and again we see evidence that discount factor 0 and 0.5 give most stable results even when we vary the number of iterations and enlarge the hypothesis space. The discount factor 1 shows a lack of convergence. This is one the reasons we opted to perform the remaining experiments with discount factor other than 1.

DISCUSSION

Prior to our experiments, we believed that the Q Learning algorithm would be equally affected by changes in parameterisation and that some parameters were not more significant than others. So we expected for a given policy that the agent would behave differently for similar values of gamma and alpha. This intuition is based on Equation 1, because a higher gamma value makes the agent take a longer view but a higher alpha value cause the agent to become greedy as a larger term implies a larger decay of the update value. We also this to be the case in our experiment as Fig 7, shows that for a fixed value of 0.8 we achieve faster convergence then when the value is low or decaying over the number of episodes. Depending on the nature of the problem we would recommend starting with a moderate value of alpha and then tune upwards or downwards as necessary. For our application a high learning rate seems to be beneficial in terms of number of episodes required before convergence. We would recommend a starting value between 0.5 and 0.8 for future applications for similar problems.

With respect to the discount factor we observe some deviations from our expectations. We expect that the agent will take a longer term view when this value is high and act greedily when this value is low. But in our stochastic case with variable gamma values we see that the gamma = 1 case fails to achieve convergence. However, it should be mentioned that what we have deemed to be a lack of convergence over 10K episodes might converge over a larger range. This could be investigated in further work. But the other surprise is the similarity in performance between the gamma =0.5 and 0 cases. For discount factor 0 we expect the agent to be myopic and only consider the immediate rewards. However, in practice for our experiment this is not the case. One possible explanation for this deviation is potentially due to the probabilities associated with the P and R matrix as generated by the package. We find that in general for the discount factor, $0 < \gamma < 1$ are stable. We would recommend a gamma = 0.5 as a good starting value to avoid convergence issues or making the agent far too myopic.

We did not expect that the effects of the learning rate and discount factor to be so policy specific. We have shown that there is almost a coupling between epsilon greedy performance and these parameters. But the same is not true for the softmax policy which appears much more invariant to parameter changes and produces consistent results despite parameter variations. So we recommend using softmax as a policy because it achieves faster convergence and shows greater invariance to parametrisation than epsilon greedy.

We have presented analysis of the Q learning algorithm in both its stochastic and deterministic case. We have compared the parameter variations between them and have gained some interesting insights. With regards to policy, it appears that a softmax policy is much more stable than an epsilon greedy policy at both high low values of the discount factor. We have shown that for an agent with an epsilon greedy policy the convergence breaks down as the discount factor approaches 1 but the converse is not found to be the case in our analysis. We find that low to 0 discount factor values produce stable results which high values and 1 gives very different results. This leads us to conclude that the Q learning agent performs better in our experiments being myopic rather than a long term thinker. We find that low discount factor values are scalable and stable even when the hypothesis space is large as shown by our big grid case.

The learning rate is found to have a smaller impact as we have shown in the deterministic case that varying the alpha or leaving it fixed leads to almost identical final Q matrices and the performance measures show a similar pattern. This is only the case when a low discount factor is used. Given the impact we have observed of high gamma values we do not expect the alpha variability will produce the same impact because the gamma value would lead to convergence issues for the agent. These are only applicable for the epsilon greedy case. For the softmax case we find that varying the temperature parameter by small amounts hardly leads to much difference and that by increasing the order of the magnitude of the temperature parameter we can produce steeper or gentler profiles for the softmax probabilities. A fixed value of 10 is found to be good value for our softmax policy.

For both types of policies considered the agent is able to arrive at the optimal policy in the deterministic case and learns very similar policies with stable gamma values even when the scope of the problem is greatly increased. Another insight is the fact that the Q Learning algorithm has nonlinear complexity. As we see that increasing the number of iterations by 5 leads to an 8 time increase in the run time. This is a very important consideration when considering very large problems.

Also, we propose an original way to visualise the performance measures by plotting a series of transforms of the performance measure shown in Fig 30. We plot a Hilbert transform which applies a linear operator and keeps the function in the same domain. It derives an analytical representation of the signal by extending it into the complex plane. This will help us identify potential periodicity in the data. This is useful because it is indicative of the agent potentially being stuck in a local minimum. The double Hilbert transform reverses the original signal and is perhaps less useful in this case. The Fourier transform gives the frequency spectrum of a given signal and shows dominant frequencies in the data. From this we can easily see if some values are more common and repeatedly learnt by the agent. This could highlight potential convergence of being stuck in a local minima. The Double Fourier Transform gives a spectrum of a spectrum and picks out values not immediately obvious from the first spectrum. The purpose of these are to add to our analysis toolkit and we include them here as potential extensions. The example presented is of the epsilon greedy agent with gamma = 0.2 the FFT and Hilbert plot both show that this agent achieves convergence fairly quickly and is stable from episode to episode without any spikes. We concluded this based on the Q matrix plots and rolling means previously but highlights the utility of these plots.

FURTHER WORK

We suggest trying more creative approaches to defining policies and learning rate functions to assess their impact on the learning agent. In addition we suggest alternative visualisation of the performance measure such as by applying a Fourier Transform, Hilbert Transform, Double Hilbert and Fourier Transforms shown in Fig 30 and Fig 31. In addition, future work could use cosine similarity or other distance metrics between the performance measures to compare the performance of various Q learning parameter setups easily and consistently. Finally, we could add more obstacles to the agent, even moving threats.

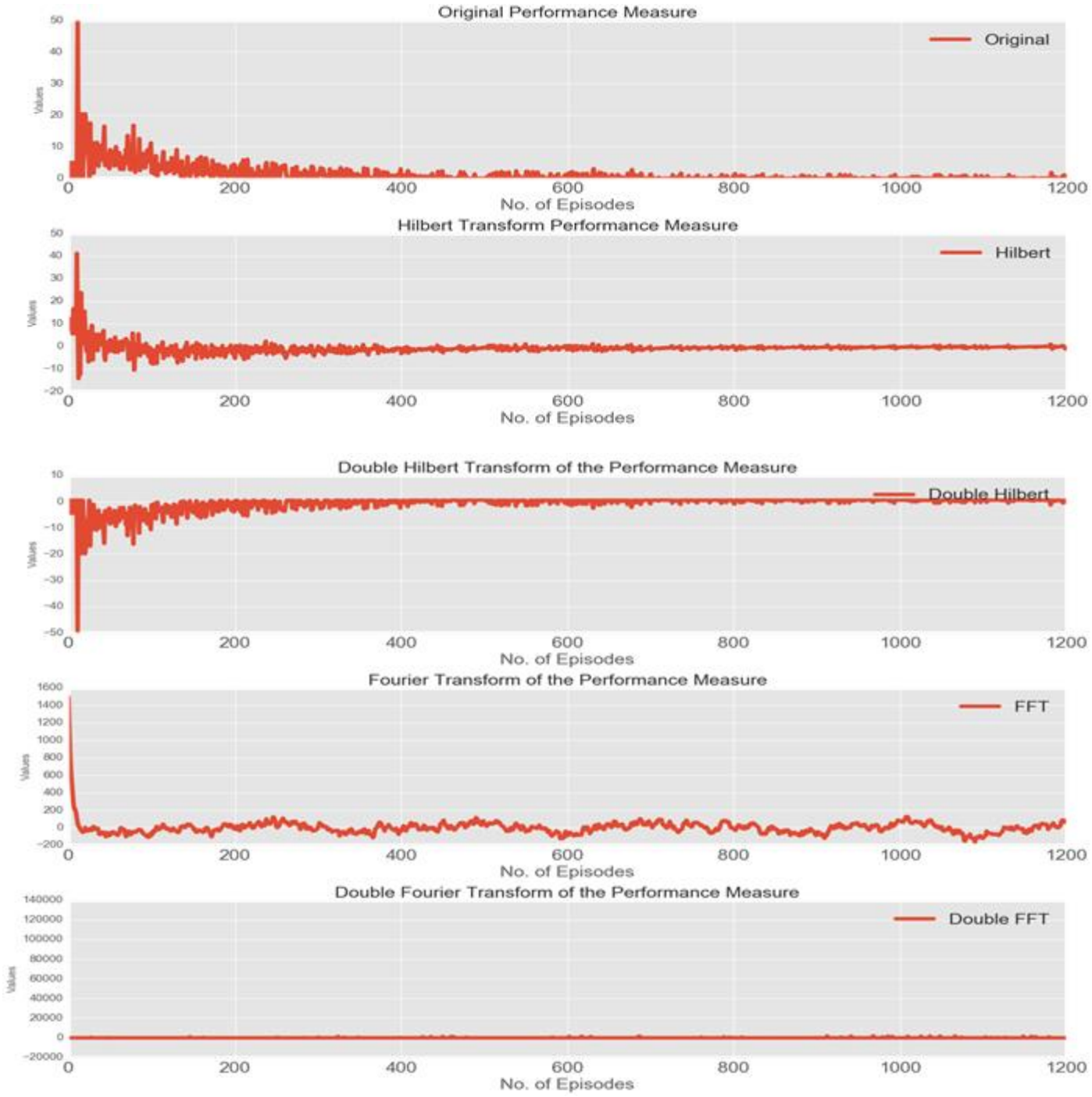


Figure 31: Some suggested alternative visualisations and analysis of performance measures.

CONCLUSIONS

We have found that the Q learning algorithm is good choice for solving MDP problems and that it has a strong dependence on parameterisation for its final performance. The choice of policy is crucial as the epsilon greedy policies and discount factors have a strong coupling this is less the case with the softmax policy. This method is scalable to larger MDP's but its nonlinear complexity could inhibit its use for very large problems without suitable computation capacity. Overall, the Q Learning algorithm is a very good approach for solving MDP problems and lends itself to easy understanding and extension as we have demonstrated. However, care must be taken to choose a policy wisely to avoid unnecessary parameters dependence.

REFERENCES

[1] Richard S. Sutton and Andrew G. Barto, R. S. Sutton, and A. G. Barto, "Introduction to Reinforcement Learning," *Learning*, vol. 4, no. 1996, pp. 1—5, 2005.

[2] "Reinforcement Learning Introduction." [Online]. Available: http://reinforcementlearning.ai-depot.com/. [Accessed: 20-Mar-2016].

[3] F. Pérez and B. E. Granger, "IPython: a System for Interactive Scientific Computing," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 21—29, May 2007.

[4] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 22—30, Mar. 2011.

[5] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90—95, 2007.

[6] "Seaborn: statistical data visualization – seaborn 0.7.0 documentation." [Online]. Available: https://stanford.edu/~mwaskom/software/seaborn/index.html. [Accessed: 20-Mar-2016].

[7] W. McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, 2010, pp. 51—56.

[8] I. Chadès, G. Chapron, M.-J. Cros, F. Garcia, and R. Sabbadin, "MDPtoolbox: a multi-platform toolbox to solve stochastic dynamic programming problems," *Ecography (Cop.).*, vol. 37, no. 9, pp. 916—920, Sep. 2014.

Deterministic_Qlearning_AA_KS

March 23, 2016

1 Table of Contents

2 Introduction

Here we are testing a deterministic Q Learning algorithm with the ϵ-greedy and softmax policy

3 Import Modules

```
In [2]: #Imports
        import numpy as np
        import random
        from random import randrange
        import seaborn as sns
        import matplotlib.pyplot as plt
        np.random.seed(0)
        c= sns.plotting_context("poster", font_scale=3, rc={"lines.linewidth": 6})
        plt.style.use('ggplot')
        from IPython.display import display, Math, Latex
        %matplotlib inline
```

4 Set R Matrix

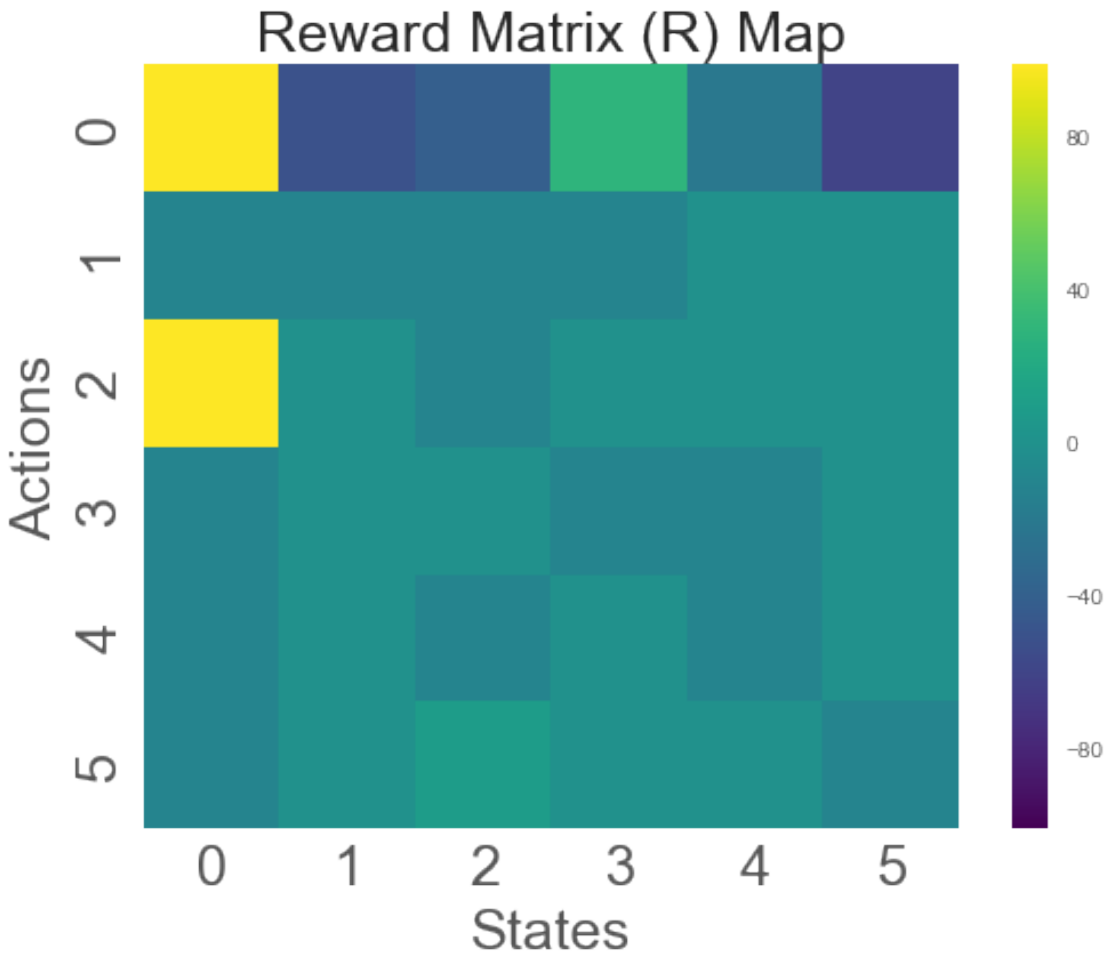
```
In [3]: R = np.zeros((6,6))
        R

Out[3]: array([[ 0.,  0.,  0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [4]: #fill with values
        x0 = [100., -50.,-40.,30.,-20. ,-60.]
        x1 = [-10., -10., -10. , -10., 0. ,0]
        x2 = [100. ,0. , -10., 0., 0. ,0.]
        x3 = [-10. ,0. ,   0. , -10.  ,-10. ,0.]
        x4 = [-10. ,0. , -10. , 0. , -10.,0.]
        x5 = [-10., 0., 10., 0. ,0.,-10.]

In [5]: R[0] =x0;
        R[1] =x1;
        R[2] =x2;
        R[3] =x3;
        R[4] =x4;
        R[5] =x5;
        R  =  R.astype(float)
        print('R Matrix \n',R)

R Matrix
 [[ 100.  -50.  -40.   30.  -20.  -60.]
 [ -10.  -10.  -10.  -10.    0.    0.]
 [ 100.    0.  -10.    0.    0.    0.]
 [ -10.    0.    0.  -10.  -10.    0.]
 [ -10.    0.  -10.    0.  -10.    0.]
 [ -10.    0.   10.    0.    0.  -10.]]
```

4.1 R Matrix Map

```
In [6]: plt.figure(figsize=(8,6))
        sns.heatmap(R, cmap=plt.cm.viridis)
```

```
plt.title('Reward Matrix (R) Map', fontsize=25)
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
```

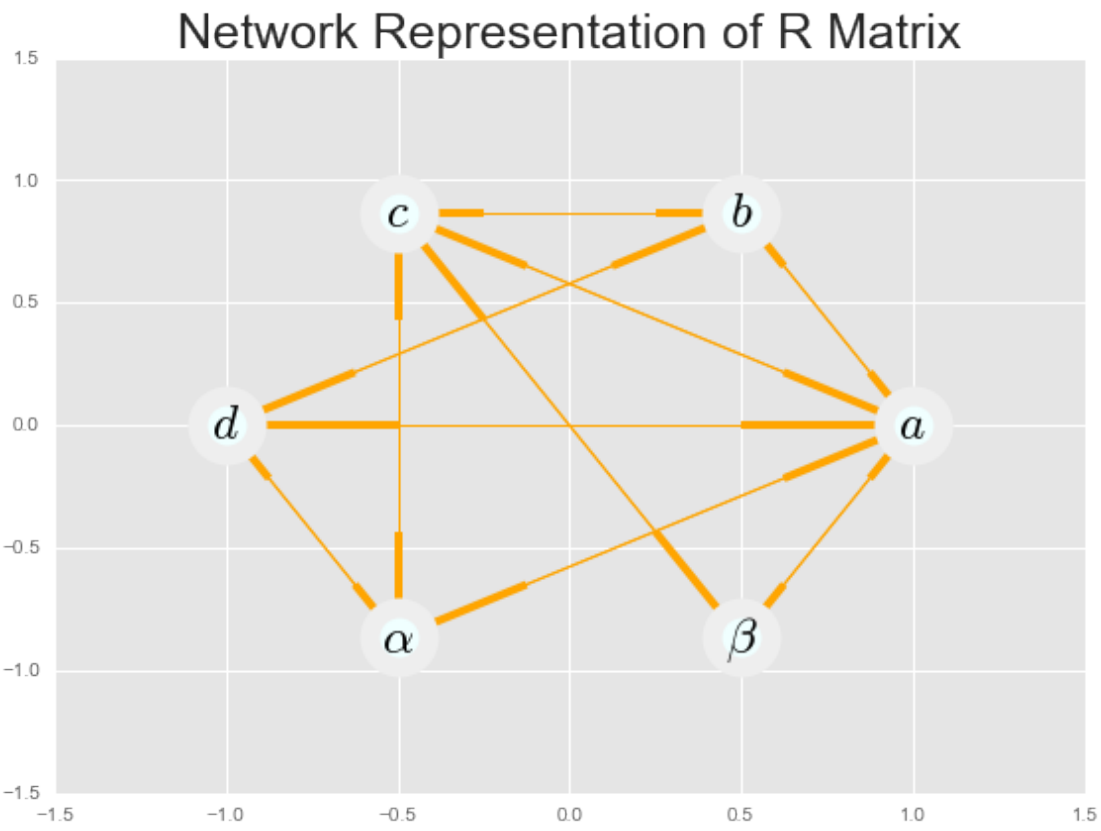
```
Out[6]: (array([ 0.5,  1.5,  2.5,  3.5,  4.5,  5.5]),
         <a list of 6 Text yticklabel objects>)
```



4.2 R Matrix Network Diagram

```
In [7]: import networkx as nx
        G = nx.from_numpy_matrix(R)
        Gd = nx.DiGraph(G)
        plt.figure(figsize=(8,6))
        labels={}
        labels[0]=r'$a$'
        labels[1]=r'$b$'
        labels[2]=r'$c$'
        labels[3]=r'$d$'
```

```
labels[4]=r'$\alpha$'
labels[5]=r'$\beta$'
pos = nx.shell_layout(Gd)
nx.draw_networkx(Gd, pos, labels=labels, font_size =25, node_size=900, edge_color='#FFA500', no
                 linewidths=10)
plt.title('Network Representation of R Matrix', fontsize=25)
plt.tight_layout()
```



5 Set Q Matrix

In [8]:
```
Q = np.zeros_like(R)
Q
```

Out[8]:
```
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

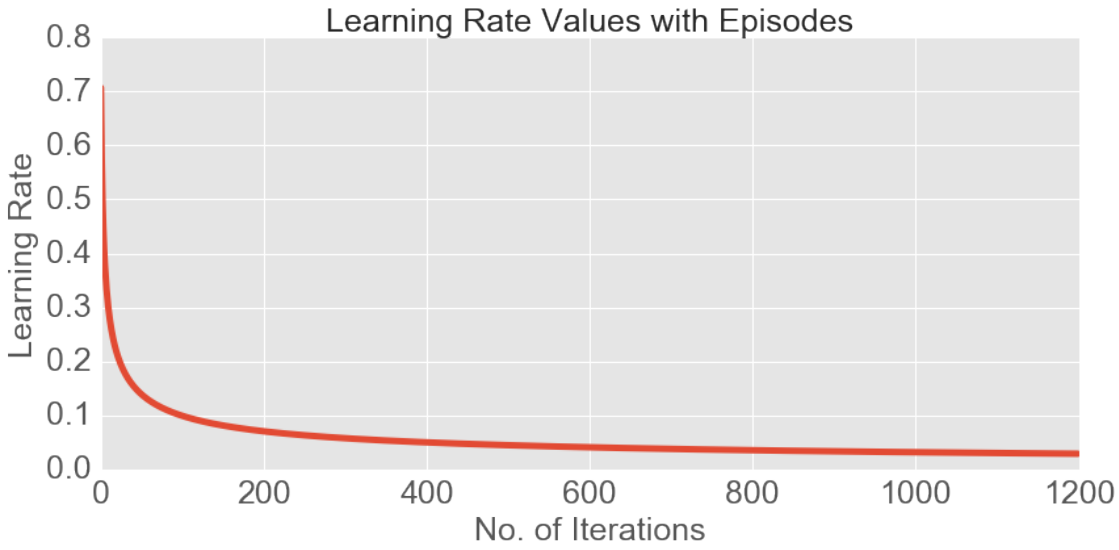
In [9]:
```
max_iter = 1200
```

6 Learning Rate, α

We define our learning rate as a function of our number of episodes, n. This ensures a more dynamic learning rate as opposed a static value. We define our alpha s follows:

$$\alpha = \frac{1}{\sqrt{(n+2)}}$$

```
In [10]: plt.figure(figsize=(12,6))
         plt.plot(1/np.sqrt((np.arange(1200)+2)), linewidth =5)
         plt.xlabel("No. of Iterations", fontsize=25)
         plt.ylabel('Learning Rate', fontsize=25)
         plt.title('Learning Rate Values with Episodes', fontsize=25)
         plt.xticks(fontsize=25)
         plt.yticks(fontsize=25)
         plt.tight_layout()
```



7 Q Learning: Basic Case 1: ϵ-greedy policy, with γ=0.2

```
In [11]: #random initial positions
         pos = [0,1,2,3,4,5]

In [12]: move_greedy = []

         def QLearning_epsilon(Q,R,explore,gamma,max_iter=max_iter):

             for episode in range(0, max_iter):
                 #print('Start of episode: ', episode)
                 #Random initial position
                 initial_pos = random.choice(pos)
                 #print(initial_pos)
                 #learning rate
```

```
learning_rate = 1 / (np.sqrt(episode + 2))

#
#two_terminal_in_a_row = -2
step = 1
#While we haven't reached our terminal state, continue searching
#
#while two_terminal_in_a_row < 0:
while initial_pos != 0:
    #It will be used later for the e-greedy policy
    greedy_threshold = random.uniform(0.0,1.0)


    #empty list which will be filled with all the available positions
    NEW_pos = []
    #the available position have to be one of the pos variable
    for next_pos in range(0, len(pos)):
        #search only for positions with a value
        if np.isnan(R[initial_pos, next_pos]) == False:
        #Append to NEW_pos list the Q value of all the available positions
            NEW_pos.append(Q[initial_pos, next_pos])

    #the maximum value of the above list. It will be used for the movement part
    greedy_Q = max(NEW_pos)

    #MOVEMENT PART

    #We follow the e-greedy policy
    if greedy_Q > 0.0 and greedy_threshold >= explore:
        #Find the index of the maximum Q value
         next_state = NEW_pos.index(greedy_Q)
    else:
        #If the greedy_threshold is smaller than the exploration rate, then we select
         next_state = randrange(0,len(NEW_pos))



    #Decrease explore value very slowly in every iteration. This ensures that at the b
    #when most of the Q values are zeroes, that the algorithm has a higher chance of e
    #that are many iterations the chances of exploit are higher
    explore = explore * 0.999


    #calculate max[Q(next s, all a)]
    list_Q = []
    for future_pos in range(0, len(pos)):
        #check that the next position isn't empty
        if np.isnan(R[next_state, future_pos]) == False:
            list_Q.append(Q[next_state, future_pos])
    #maximum Q
    maximum_Q = max(list_Q)

    #define dQ - update rule
    dQ = learning_rate*(R[initial_pos, next_state] + gamma* maximum_Q -  Q[initial_pos
```

```
                    #Q learning update
                    Q[initial_pos, next_state] = Q[initial_pos, next_state] + dQ

                    #calculate performance measure
                    discrepancy.append(np.absolute(dQ))

                    move_greedy.append(step)
                    step = step + 1
                    initial_pos = next_state
                    #print('-----------------------------------')
                    #print('Initial Position:', initial_pos)
                    #print('-----------------------------------')
```

```
In [13]: #Initialise performance measure, policy each time
         discrepancy = []
         QLearning_epsilon(Q,R,0.8,0.2)
```

7.1 Q Learning with ϵ-greedy policy results

```
In [14]: #calculate value
         policy = np.argmax(Q, axis=1)
         Value = (np.max(Q, axis=1))
         print('Policy\n',policy)
         print('V Values\n', Value)
         print('Final Q Values:\n',Q)
```

```
Policy
 [0 2 0 2 2 2]
V Values
 [  0.          9.99999318 100.         19.99999343   9.99994249
  29.99997949]
Final Q Values:
 [[  0.           0.           0.           0.           0.           0.        ]
 [ -9.58056887  -7.49769422   9.99999318  -5.48764517   1.41725678
    5.06877808]
 [100.          1.98178175   9.92628679   3.86018217   1.91095719
    5.78549293]
 [ -9.48072439   1.58447315  19.99999343  -4.60677112  -5.26071096
    4.1296911 ]
 [ -8.38945079   1.42554249   9.99994249   2.91686545  -8.34715248
    4.73780648]
 [ -7.95945087   1.02614553  29.99997949   3.24042702   1.57937092
   -4.26730281]]
```

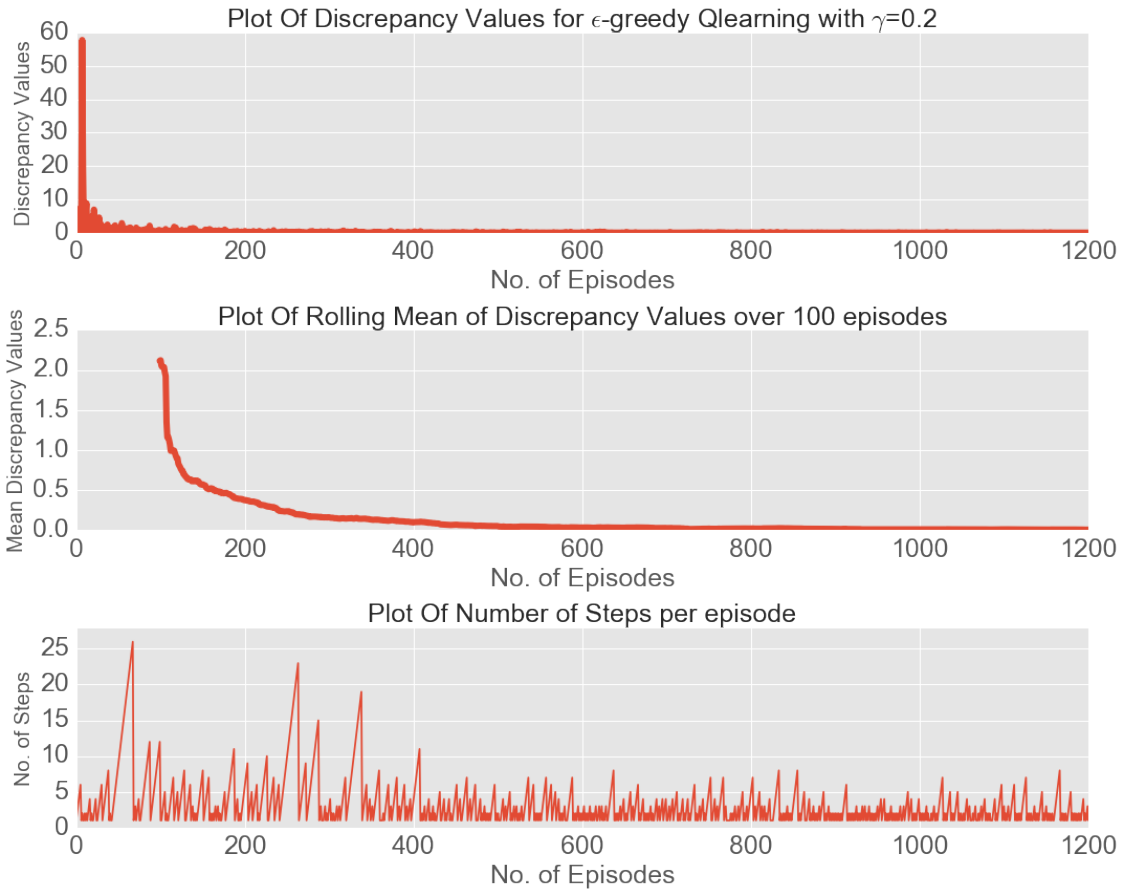
```
In [15]: import tabulate
         from tabulate import tabulate as tb
         print("Q Matrix \n",tb(Q, tablefmt="simple", numalign="left", floatfmt=".4f"))
```

```
Q Matrix
 --------  -------  -------  -------  -------  -------
0.0000    0.0000   0.0000   0.0000   0.0000   0.0000
-9.5806   -7.4977  10.0000  -5.4876  1.4173   5.0688
100.0000  1.9818   9.9263   3.8602   1.9110   5.7855
-9.4807   1.5845   20.0000  -4.6068  -5.2607  4.1297
```

```
-8.3895   1.4255   9.9999   2.9169   -8.3472  4.7378
-7.9595   1.0261   30.0000  3.2404   1.5794   -4.2673
--------  -------  -------  -------  -------  -------
```

In [16]:
```
import pandas as pd
disc_pd = pd.Series(discrepancy)
mean_dis = disc_pd.rolling(window=100,center=False).mean()
```

In [17]:
```
plt.figure(figsize=(15,12))

plt.subplot(311)
plt.plot(discrepancy, linewidth=6)
plt.title('Plot Of Discrepancy Values for $\epsilon$-greedy Qlearning with $\gamma$=0.2', font
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("Discrepancy Values", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.xlim(0,max_iter)

plt.subplot(312)
plt.title('Plot Of Rolling Mean with 100 epsodes')
plt.plot(mean_dis, linewidth=6)
plt.title('Plot Of Rolling Mean of Discrepancy Values over 100 episodes', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("Mean Discrepancy Values", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.xlim(0,max_iter)

plt.subplot(313)
plt.plot(move_greedy)
plt.title('Plot Of Number of Steps per episode', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("No. of Steps", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.ylim(0, max(move_greedy)+2)
plt.xlim(0,max_iter)

plt.tight_layout()
plt.show()
```



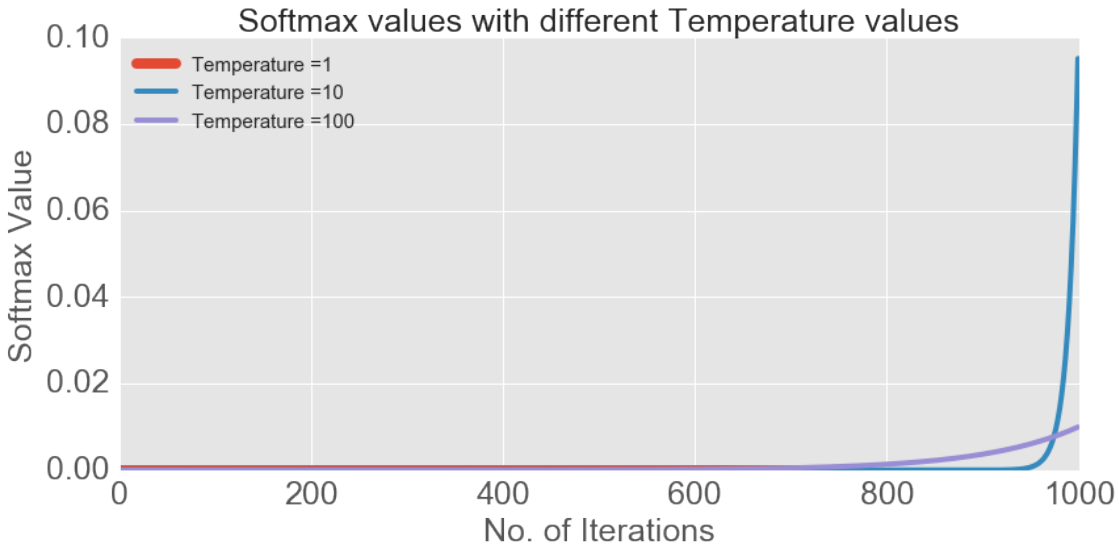
7.2 Q Matrix for ϵ-greedy

```
In [18]: plt.figure(figsize=(12,6))
         sns.heatmap(Q, cmap=plt.cm.viridis)
         plt.title("$\epsilon$-Greedy Q Matrix with $\gamma$=0.2", fontsize=25)
         plt.xlabel("States", fontsize=25)
         plt.ylabel('Actions', fontsize=25)
         plt.xticks(fontsize=25)
         plt.yticks(fontsize=25)

Out[18]: (array([ 0.5,  1.5,  2.5,  3.5,  4.5,  5.5]),
          <a list of 6 Text yticklabel objects>)
```

8 Q Learning: Case 4: Different Policy: Softmax, with γ=0.2

8.1 Softmax impulse Response

In [19]:
```
plt.figure(figsize=(12,6))
num = np.arange(1000)
plt.plot(np.exp(num/1)/sum(np.exp(num/1)),  linewidth=8, label='Temperature =1')
plt.plot(np.exp(num/10)/sum(np.exp(num/10)), linewidth=4, label='Temperature =10')
plt.plot(np.exp(num/100)/sum(np.exp(num/100)), linewidth=4, label='Temperature =100')
plt.xlabel("No. of Iterations", fontsize=25)
plt.ylabel('Softmax Value', fontsize=25)
plt.title('Softmax values with different Temperature values', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.legend(fontsize=15, loc=2)
plt.tight_layout()
```




```
In [20]: Q2 = np.zeros_like(R)
         Q2

Out[20]: array([[ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [21]: move_softmax = []
         def QLearning_softmax(Q,R, gamma, temperature=10, max_iter=max_iter):

         #Initiate episodes
             for episode in range(0, max_iter):
                 #print('Start of episode: ', episode)
                 #Random initial position
                 initial_pos = np.random.choice(pos)
                 #learning rate, alpha
                 alpha = 1 / (np.sqrt(episode + 2))

                 step = 1
                 #While we haven't reached our terminal state, continue searching

                 while initial_pos != 0:

                    #It will be used later for the softmax policy
                     threshold = random.uniform(0.0,1.0)
                     prob_t = [0,0,0,0,0,0]       #initialise

                     #temperature = 10.0
                     #empty list which will be filled with all the available positions
                     NEW_pos = []
                     #the available position have to be one of the pos variable
```

```
for next_pos in range(len(pos)):
    #search only for positions with a value
    #if np.isnan(R[initial_pos, next_pos]) == False:
    #Append to NEW_pos list the Q value of all the available positions
        NEW_pos.append(Q[initial_pos, next_pos])

#MOVEMENT PART
for a in range(len(NEW_pos)):
    prob_t[a] = np.exp(NEW_pos[a]/temperature)  #calculate numerators

#numpy matrix element-wise division for denominator (sum of numerators)
prob_t = prob_t / sum(prob_t)
prob = list(prob_t)

counter = 0
#We follow the softmax policy
for i in range(len(NEW_pos)):
    if threshold <= prob[i] + counter:
        break

    counter = counter + prob_t[i]
next_state = i

    #Decrease explore value very slowly in every iteration. This ensures that at t
    #when most of the Q values are zeroes, that the algorithm has a higher chance
    #that are many iterations the chances of exploit are higher
    #temperature = temperature * 0.99999


#calculate max[Q(next s, all a)]
list_Q = []
for future_pos in range(len(pos)):
    #check that the next position isn't empty
    #if np.isnan(R[next_state, future_pos]) == False:
        list_Q.append(Q[next_state, future_pos])
#maximum Q
maximum_Q = max(list_Q)

#define dQ - update rule
dQ = alpha*(R[initial_pos, next_state] + gamma* maximum_Q -  Q2[initial_pos, next_s

#Q learning update
Q2[initial_pos, next_state] = Q2[initial_pos, next_state] + dQ

#calculate performance measure
discrepancy2.append(np.absolute(dQ))
move_softmax.append(step)
step = step + 1

#Set the next state as initial position
#If it is zero, then it will exit the loop and the next episode will begin

#if initial_pos == 0 and initial_pos == next_state + NEW_pos.index(greedy_Q):
#   two_terminal_in_a_row = two_terminal_in_a_row + 1
```

```
                    #print([initial_pos, next_state])
                    #print(next_state)
                    initial_pos = next_state
```

In [22]:
```
discrepancy2 = []
QLearning_softmax(Q2,R,0.2)
```

8.2 Q Learning with Softmax Results

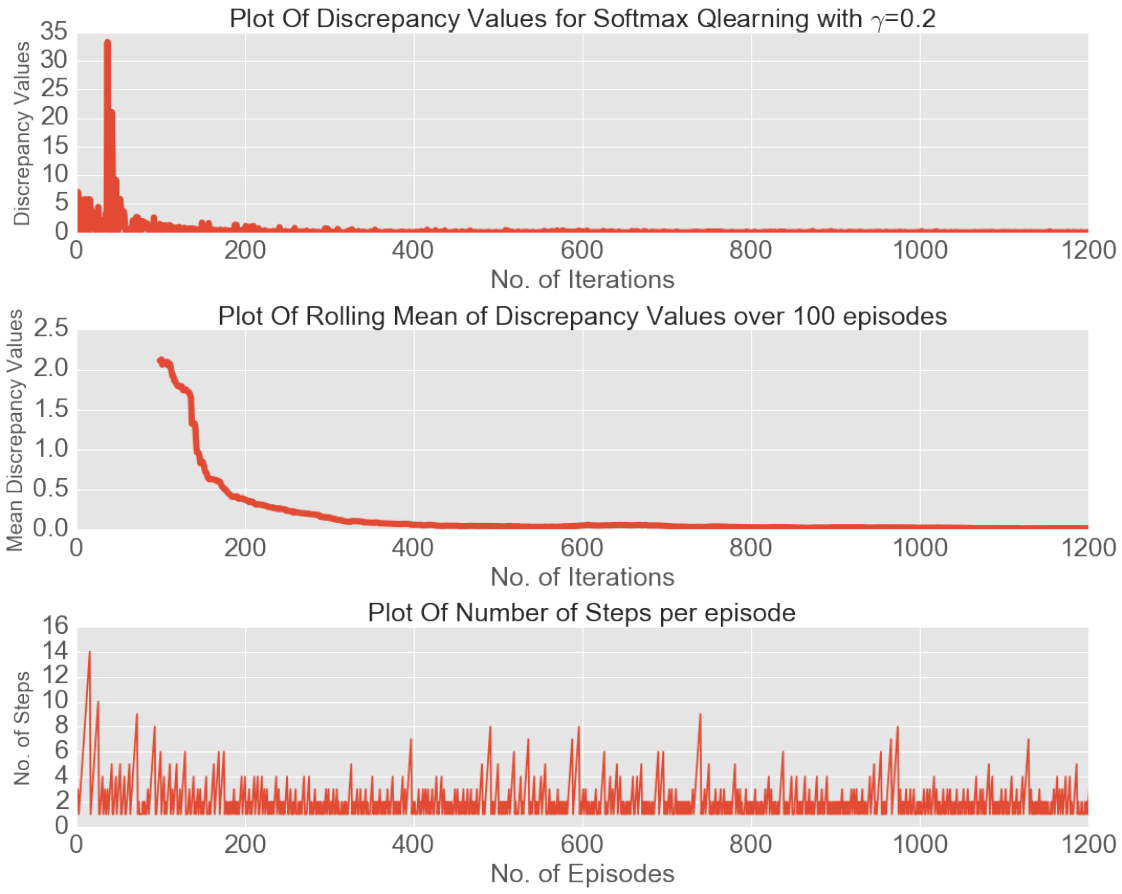
In [23]:
```
#calculate policy
policy2 = (np.argmax(Q2, axis=1))
#calculate value
Value2 = (np.max(Q2, axis=1))
print('Policy\n',policy)
print('V Values\n', Value)
print('Final Q Values:\n',Q2)
```

```
Policy
 [0 2 0 2 2 2]
V Values
 [  0.          9.99999318 100.         19.99999343   9.99994249
  29.99997949]
Final Q Values:
 [[  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00]
 [ -9.23081396e+00  -8.44565594e+00   9.88793815e+00  -5.63479383e+00
    1.64019052e+00   5.83282248e+00]
 [  1.00000000e+02   0.00000000e+00  -8.70849735e+00   4.61880215e-02
    0.00000000e+00   6.85457672e-01]
 [ -9.40323482e+00   1.20257538e+00   1.99999663e+01  -4.78255524e+00
   -7.18251609e+00   5.40816307e+00]
 [ -7.39545518e+00   1.51421367e+00   9.81230515e+00   3.87084648e+00
   -7.57782670e+00   5.81412760e+00]
 [ -8.80470922e+00   1.09759835e+00   2.99999970e+01   3.01364774e+00
    9.85598410e-01  -5.38598642e+00]]
```

In [24]:
```
disc_pd2 = pd.Series(discrepancy2)
mean_dis2 = disc_pd2.rolling(window=100,center=False).mean()
```

In [25]:
```
#plot discrepancy
plt.figure(figsize=(15,12))

plt.subplot(311)
plt.plot(discrepancy2, linewidth=6)
plt.title('Plot Of Discrepancy Values for Softmax Qlearning with $\gamma$=0.2', fontsize=25)
plt.xlabel("No. of Iterations", fontsize=25)
plt.ylabel("Discrepancy Values", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.xlim(0,max_iter)

plt.subplot(312)
plt.title('Pot Of Rolling Mean with 100 epsodes')
plt.plot(mean_dis2, linewidth=6)
plt.title('Plot Of Rolling Mean of Discrepancy Values over 100 episodes', fontsize=25)
```

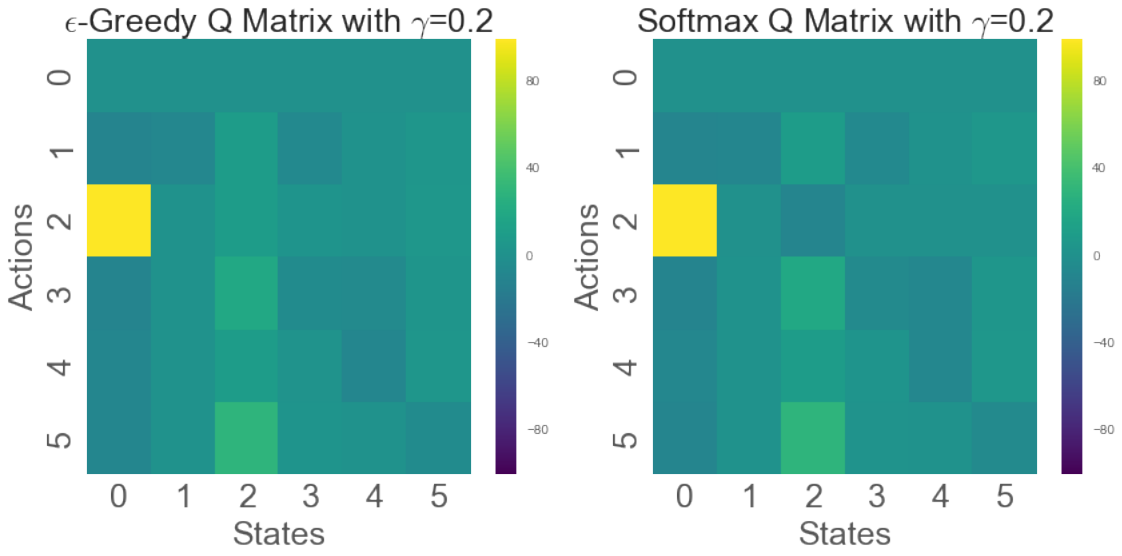
```
plt.xlabel("No. of Iterations", fontsize=25)
plt.ylabel("Mean Discrepancy Values", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.xlim(0,max_iter)

plt.subplot(313)
plt.plot(move_softmax)
plt.title('Plot Of Number of Steps per episode', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("No. of Steps", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.ylim(0, max(move_softmax)+2)
plt.xlim(0,max_iter)

plt.tight_layout()
```



9 Comparisons

9.1 FInal Q Matrices of ϵ-greedy and Softmax

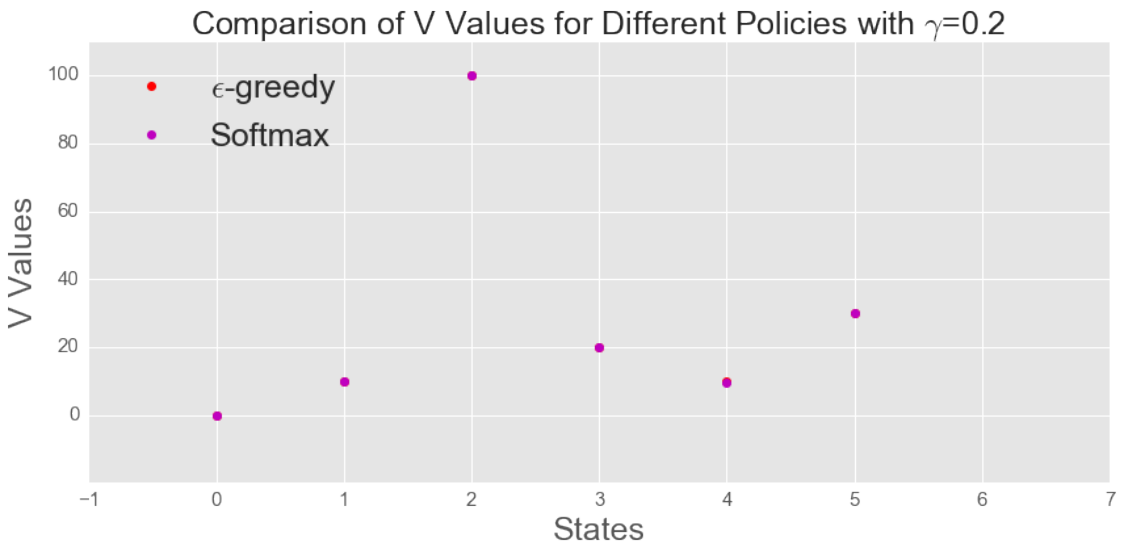
In [26]:
```
plt.figure(figsize=(12,6))

plt.subplot(121)
sns.heatmap(Q, cmap=plt.cm.viridis)
plt.title("$\epsilon$-Greedy Q Matrix with $\gamma$=0.2", fontsize=25)
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)

plt.subplot(122)
sns.heatmap(Q2, cmap=plt.cm.viridis)
plt.title("Softmax Q Matrix with $\gamma$=0.2", fontsize=25)
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.tight_layout()
```



9.2 Comparison of V Values

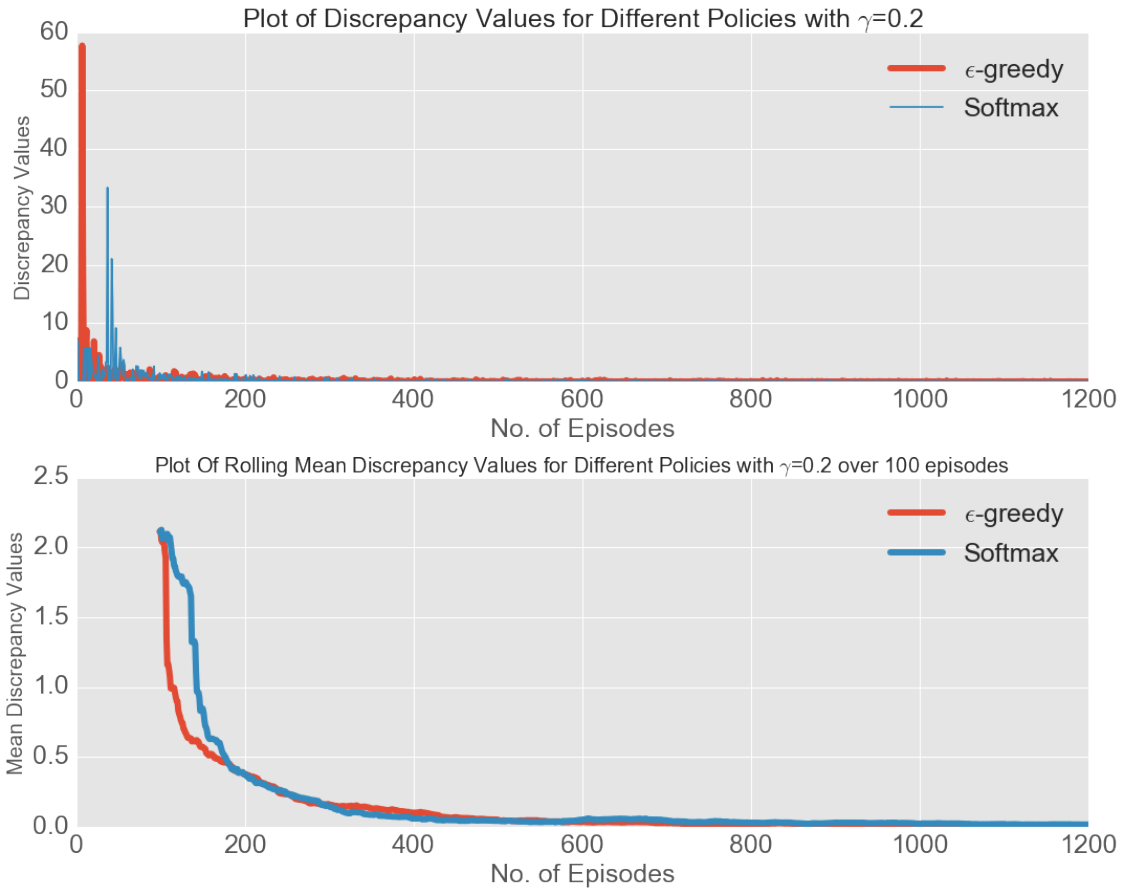
In [27]:
```
plt.figure(figsize=(12,6))
p1= plt.plot(Value,'ro', label='$\epsilon$-greedy')
p2= plt.plot(Value2,'mo', label='Softmax')

plt.title("Comparison of V Values for Different Policies with $\gamma$=0.2",  fontsize=25)
plt.ylim(Q.min()-10, Q.max()+10)
plt.xlim(-1,len(Q)+1)
plt.ylabel("V Values", fontsize=25)
```

```
plt.xlabel('States', fontsize=25)
plt.xticks( fontsize=15)
plt.yticks( fontsize=15)
plt.legend(fontsize=25, loc=2)
plt.tight_layout()
```



In [28]:
```
plt.figure(figsize=(15,12))

plt.subplot(211)

plt.plot(discrepancy, linewidth=6, label='$\epsilon$-greedy')
plt.plot(discrepancy2, label='Softmax')

plt.title('Plot of Discrepancy Values for Different Policies with $\gamma$=0.2', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("Discrepancy Values", fontsize=20)
plt.legend(fontsize=25, loc=1)
plt.xlim(0,max_iter)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)

plt.subplot(212)

plt.plot(mean_dis, linewidth=6, label='$\epsilon$-greedy')
plt.plot(mean_dis2, linewidth=6,label='Softmax')

plt.title('Plot Of Rolling Mean Discrepancy Values for Different Policies with $\gamma$=0.2 ov
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("Mean Discrepancy Values", fontsize=20)
plt.legend(fontsize=25, loc=1)
plt.xlim(0,max_iter)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
```

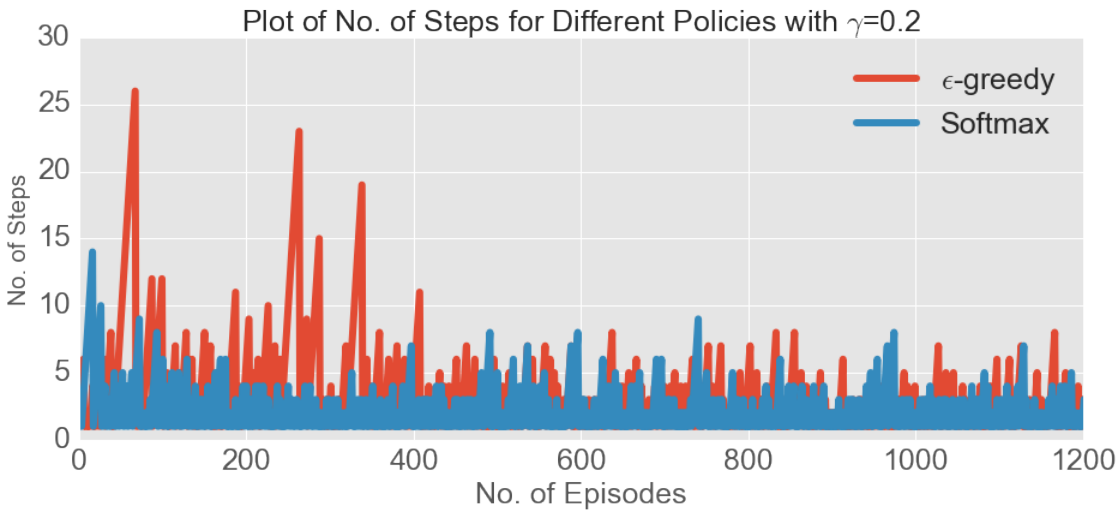
```
plt.tight_layout()
```



9.3 Comparison of number of Steps

In [29]:
```
plt.figure(figsize=(15,6))
plt.plot(move_greedy, linewidth=6, label='$\epsilon$-greedy')
plt.plot(move_softmax, linewidth=6,label='Softmax')

plt.title('Plot of No. of Steps for Different Policies with $\gamma$=0.2', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("No. of Steps", fontsize=20)
plt.legend(fontsize=25, loc=1)
plt.xlim(0,max_iter)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
```

Out[29]:
```
(array([  0.,   5.,  10.,  15.,  20.,  25.,  30.]),
 <a list of 7 Text yticklabel objects>)
```



10 ϵ-greedy with with γ=0.8

```
In [30]: Q = np.zeros_like(R)
         Q

Out[30]: array([[ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [31]: #Initialise performance measure, policy each time
         discrepancy = []
         QLearning_epsilon(Q,R,0.8,0.8)
```

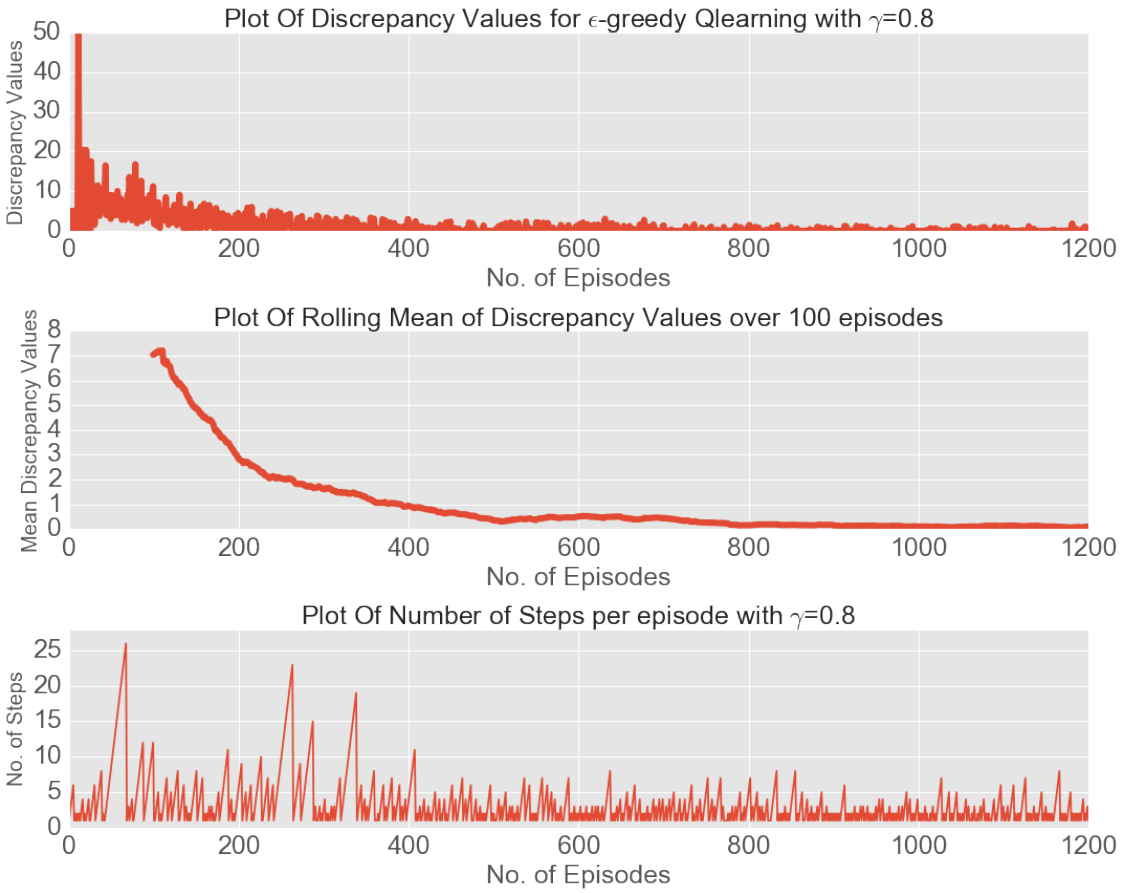
10.1 Q Learning with ϵ-greedy policy results with γ=0.8

```
In [32]: #calculate value
         policy = np.argmax(Q, axis=1)
         Value = (np.max(Q, axis=1))
         print('Policy\n',policy)
         print('V Values\n', Value)
         print('Final Q Values:\n',Q)

Policy
 [0 2 0 2 5 2]
V Values
 [  0.          69.99971241 100.          79.99996629  71.99954943
  89.99999999]
Final Q Values:
 [[  0.           0.           0.           0.           0.           0.        ]
 [ -8.47516088  28.85501922  69.99971241  38.88327287  42.33194567
   49.16925676]
```

```
 [100.          49.2112256   66.77996037  62.72875357  56.13899718
   70.7270934 ]
 [ -8.41404132  42.44448673  79.99996629  43.04145526  29.95537957
   34.96718119]
 [ -7.14267796  43.65962658  63.78008714  53.72862296  37.81542323
   71.99954943]
 [ -8.71677981  42.32426539  89.99999999  56.41084873  41.65347906
   55.01128242]]
```

In [33]:
```
disc_pd = pd.Series(discrepancy)
mean_dis = disc_pd.rolling(window=100,center=False).mean()
```

In [34]:
```
#plot discrepancy
plt.figure(figsize=(15,12))

plt.subplot(311)
plt.plot(discrepancy, linewidth=6)
plt.title('Plot Of Discrepancy Values for $\epsilon$-greedy Qlearning with $\gamma$=0.8', font
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("Discrepancy Values", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.xlim(0,max_iter)

plt.subplot(312)
plt.title('Pot Of Rolling Mean with 100 epsodes with $\gamma$=0.8')
plt.plot(mean_dis, linewidth=6)
plt.title('Plot Of Rolling Mean of Discrepancy Values over 100 episodes', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("Mean Discrepancy Values", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.xlim(0,max_iter)

plt.subplot(313)
plt.plot(move_greedy)
plt.title('Plot Of Number of Steps per episode with $\gamma$=0.8', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("No. of Steps", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.ylim(0, max(move_greedy)+2)
plt.xlim(0,max_iter)

plt.tight_layout()
plt.show()
```



10.2 Q Matrix for ϵ-greedy with γ=0.8

In [35]:
```
plt.figure(figsize=(12,6))

        sns.heatmap(Q, cmap=plt.cm.viridis)
        plt.title("$\epsilon$-Greedy Q Matrix with $\gamma$=0.8", fontsize=25)
        plt.xlabel("States", fontsize=25)
        plt.ylabel('Actions', fontsize=25)
        plt.xticks(fontsize=25)
        plt.yticks(fontsize=25)
```

Out[35]:
```
(array([ 0.5,  1.5,  2.5,  3.5,  4.5,  5.5]),
 <a list of 6 Text yticklabel objects>)
```

11 Softmax with with γ=0.8

```
In [36]: Q2 = np.zeros_like(R)
         Q2

Out[36]: array([[ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [37]: discrepancy2 = []
         QLearning_softmax(Q2,R,0.8)
```

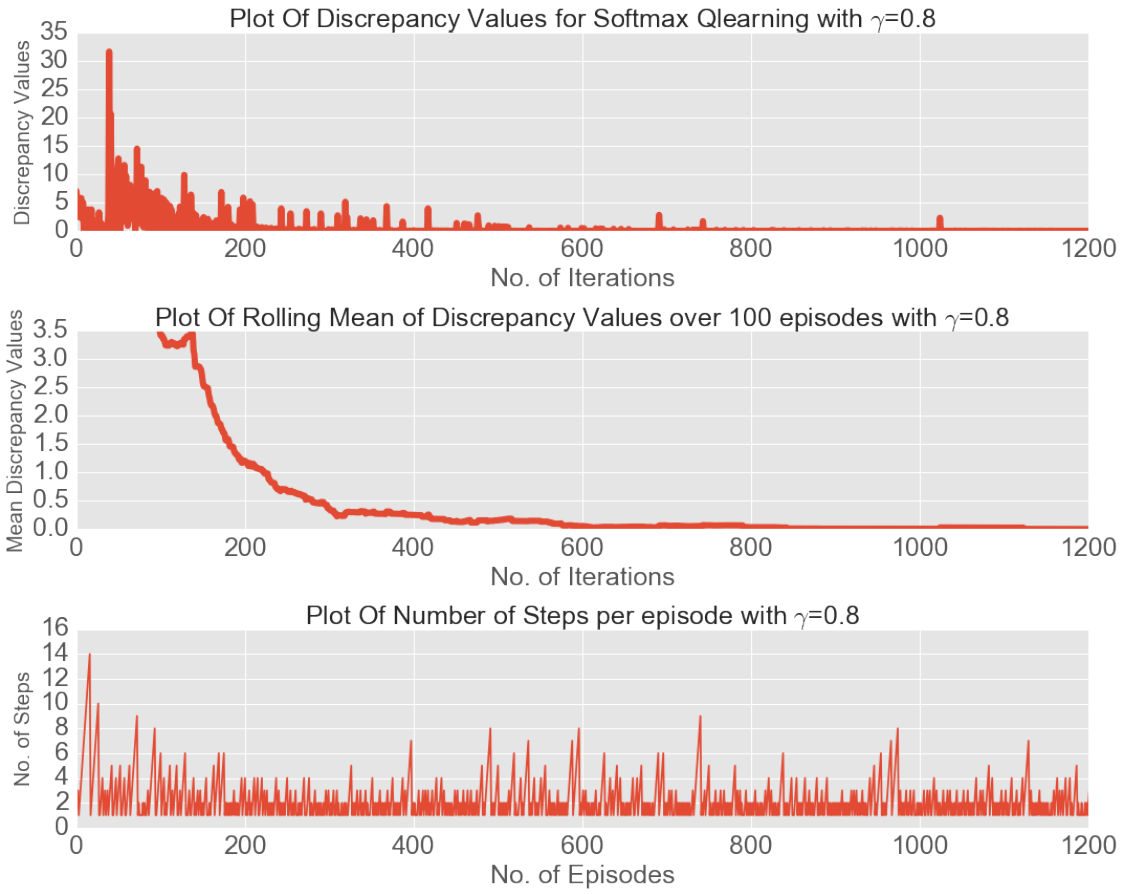
11.1 Q Learning with Softmax Results

```
In [38]: #calculate policy
         policy2 = (np.argmax(Q2, axis=1))
         #calculate value
         Value2 = (np.max(Q2, axis=1))
         print('Policy\n',policy)
         print('V Values\n', Value)
         print('Final Q Values:\n',Q2)

Policy
 [0 2 0 2 5 2]
V Values
```

```
 [  0.          69.99971241 100.          79.99996629  71.99954943
  89.99999999]
Final Q Values:
 [[  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00]
 [ -8.82286442e+00  -5.33290905e+00   1.60866279e+01  -8.00210131e+00
    1.90579534e+01   7.19801580e+01]
 [  1.00000000e+02   5.67413332e-01  -3.01873218e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00]
 [ -7.98943621e+00   4.23951903e-01   8.00000000e+01  -7.88675135e+00
   -7.37096852e+00   4.86697100e+00]
 [ -7.66364975e+00   0.00000000e+00   0.00000000e+00   6.39975031e+01
   -1.14045299e+00   2.35303210e+01]
 [ -4.58012574e+00   2.80690267e+00   8.99999998e+01   1.19578487e+01
    1.24141733e-02  -6.41135884e-01]]
```

In [39]:
```
disc_pd2 = pd.Series(discrepancy2)
mean_dis2 = disc_pd2.rolling(window=100,center=False).mean()
```

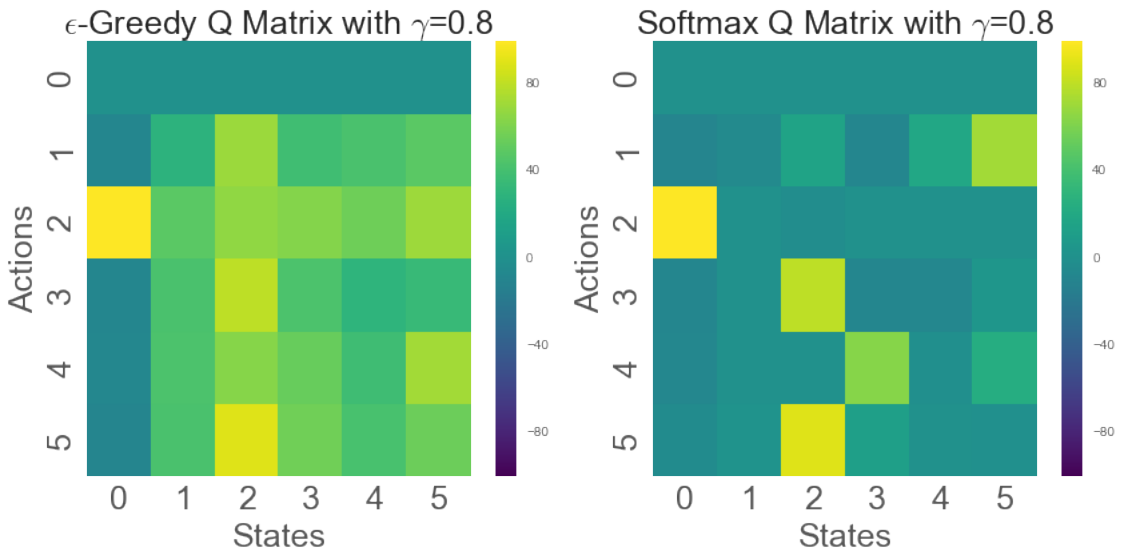
In [40]:
```
#plot discrepancy
plt.figure(figsize=(15,12))

plt.subplot(311)
plt.plot(discrepancy2, linewidth=6)
plt.title('Plot Of Discrepancy Values for Softmax Qlearning with $\gamma$=0.8', fontsize=25)
plt.xlabel("No. of Iterations", fontsize=25)
plt.ylabel("Discrepancy Values", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.xlim(0,max_iter)

plt.subplot(312)
plt.title('Pot Of Rolling Mean with 100 epsodes')
plt.plot(mean_dis2, linewidth=6)
plt.title('Plot Of Rolling Mean of Discrepancy Values over 100 episodes with $\gamma$=0.8', fo
plt.xlabel("No. of Iterations", fontsize=25)
plt.ylabel("Mean Discrepancy Values", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.xlim(0,max_iter)

plt.subplot(313)
plt.plot(move_softmax)
plt.title('Plot Of Number of Steps per episode with $\gamma$=0.8', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("No. of Steps", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.ylim(0, max(move_softmax)+2)
plt.xlim(0,max_iter)

plt.tight_layout()
```



12 Comparisons

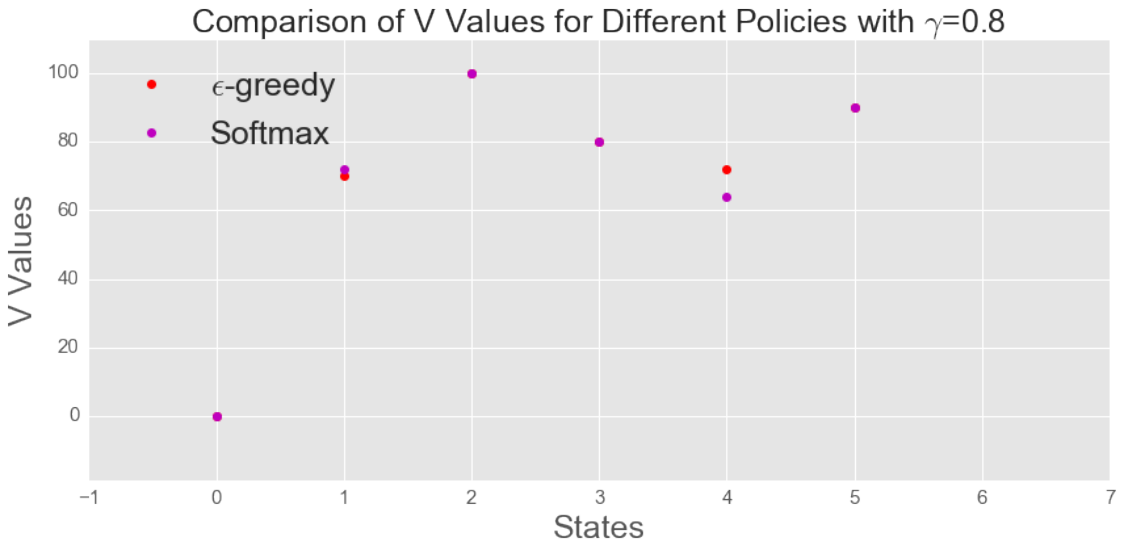
12.1 FInal Q Matrices of ϵ-greedy and Softmax

In [41]:
```
plt.figure(figsize=(12,6))

plt.subplot(121)
sns.heatmap(Q, cmap=plt.cm.viridis)
plt.title("$\epsilon$-Greedy Q Matrix with $\gamma$=0.8", fontsize=25)
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)

plt.subplot(122)
sns.heatmap(Q2, cmap=plt.cm.viridis)
plt.title("Softmax Q Matrix with $\gamma$=0.8", fontsize=25)
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.tight_layout()
```

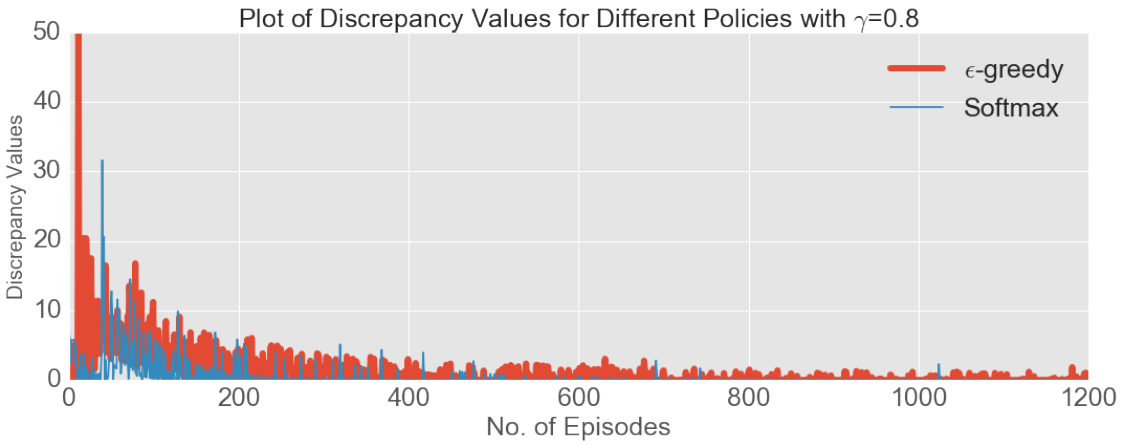


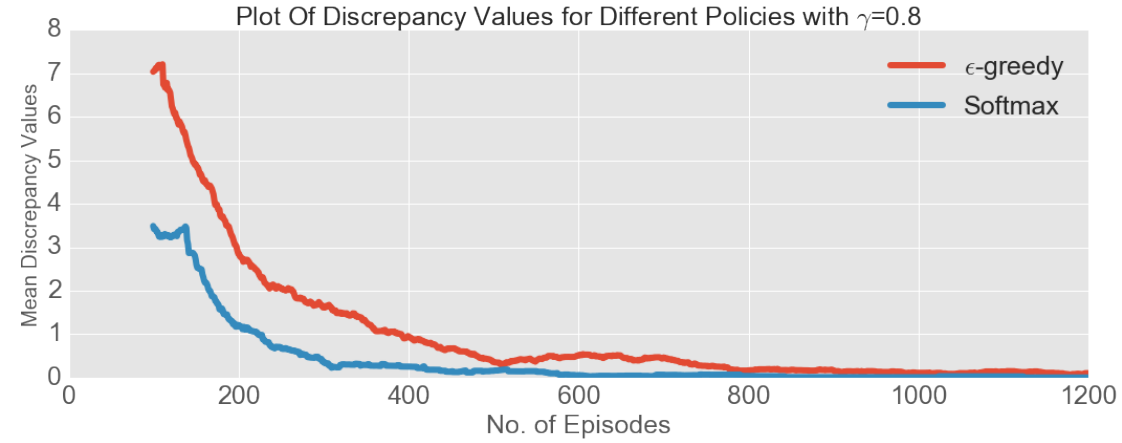
12.2 Comparison of V Values

In [42]:
```
plt.figure(figsize=(12,6))
p1= plt.plot(Value,'ro', label='$\epsilon$-greedy')
p2= plt.plot(Value2,'mo', label='Softmax')

plt.title("Comparison of V Values for Different Policies with $\gamma$=0.8",  fontsize=25)
plt.ylim(Q.min()-10, Q.max()+10)
plt.xlim(-1,len(Q)+1)
plt.ylabel("V Values", fontsize=25)
plt.xlabel('States', fontsize=25)
plt.xticks( fontsize=15)
plt.yticks( fontsize=15)
plt.legend(fontsize=25, loc=2)
plt.tight_layout()
```



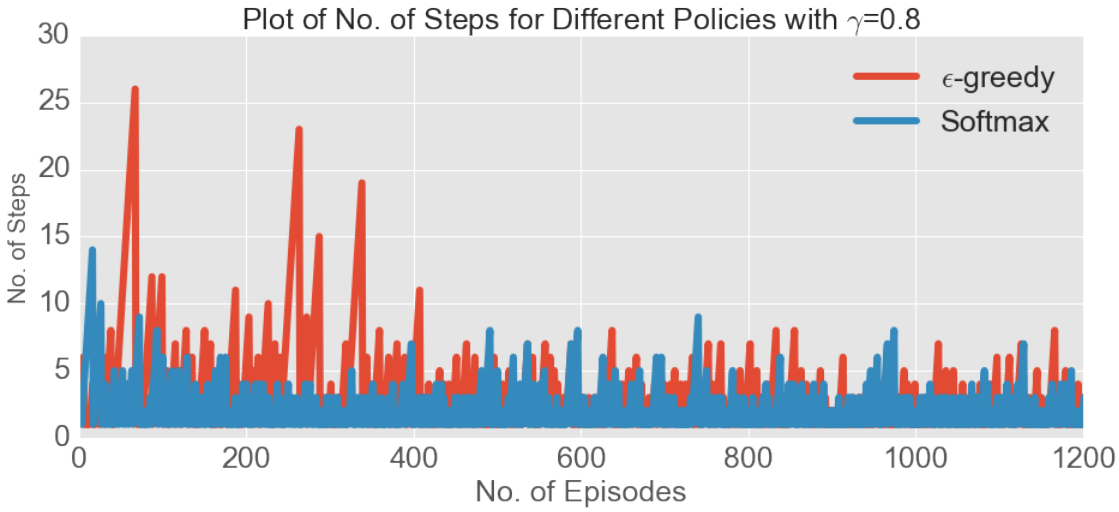
In [43]:
```
plt.figure(figsize=(15,12))

plt.subplot(211)

plt.plot(discrepancy, linewidth=6, label='$\epsilon$-greedy')
plt.plot(discrepancy2, label='Softmax')

plt.title('Plot of Discrepancy Values for Different Policies with $\gamma$=0.8', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("Discrepancy Values", fontsize=20)
plt.legend(fontsize=25, loc=1)
plt.xlim(0,max_iter)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)

plt.subplot(212)

plt.plot(mean_dis, linewidth=6, label='$\epsilon$-greedy')
plt.plot(mean_dis2, linewidth=6,label='Softmax')

plt.title('Plot Of Discrepancy Values for Different Policies with $\gamma$=0.8', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("Mean Discrepancy Values", fontsize=20)
plt.legend(fontsize=25, loc=1)
plt.xlim(0,max_iter)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)


plt.tight_layout()
```





12.3 Comparison of number of Steps

```
In [44]: plt.figure(figsize=(15,6))
         plt.plot(move_greedy, linewidth=6, label='$\epsilon$-greedy')
         plt.plot(move_softmax, linewidth=6,label='Softmax')

         plt.title('Plot of No. of Steps for Different Policies with $\gamma$=0.8', fontsize=25)
         plt.xlabel("No. of Episodes", fontsize=25)
         plt.ylabel("No. of Steps", fontsize=20)
         plt.legend(fontsize=25, loc=1)
         plt.xlim(0,max_iter)
         plt.xticks(fontsize=25)
         plt.yticks(fontsize=25)

Out[44]: (array([  0.,   5.,  10.,  15.,  20.,  25.,  30.]),
          <a list of 7 Text yticklabel objects>)
```



13 Fixed Learning Rate + ϵ-greedy policy

13.1 ϵ-greedy with α=0.2

In [45]:
```
Q4 = np.zeros_like(R)
Q4
```

Out[45]:
```
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

In [46]:
```
move_greedy = []

def QLearning_epsilon(Q,R,explore,gamma,learning_rate,max_iter=max_iter):

    for episode in range(0, max_iter):
        #print('Start of episode: ', episode)
        #Random initial position
        initial_pos = random.choice(pos)
        #print(initial_pos)
        #learning rate
        #learning_rate = 1 / (np.sqrt(episode + 2))

        #
        #two_terminal_in_a_row = -2
        step = 1
        #While we haven't reached our terminal state, continue searching
        #
        #while two_terminal_in_a_row < 0:
        while initial_pos != 0:
            #It will be used later for the e-greedy policy
```

```
greedy_threshold = random.uniform(0.0,1.0)


#empty list which will be filled with all the available positions
NEW_pos = []
#the available position have to be one of the pos variable
for next_pos in range(0, len(pos)):
    #search only for positions with a value
    if np.isnan(R[initial_pos, next_pos]) == False:
    #Append to NEW_pos list the Q value of all the available positions
        NEW_pos.append(Q4[initial_pos, next_pos])

#the maximum value of the above list. It will be used for the movement part
greedy_Q = max(NEW_pos)

#MOVEMENT PART

#We follow the e-greedy policy
if greedy_Q > 0.0 and greedy_threshold >= explore:
    #Find the index of the maximum Q value
     next_state = NEW_pos.index(greedy_Q)
else:
    #If the greedy_threshold is smaller than the exploration rate, then we select
     next_state = randrange(0,len(NEW_pos))



#Decrease explore value very slowly in every iteration. This ensures that at the b
#when most of the Q values are zeroes, that the algorithm has a higher chance of e
#that are many iterations the chances of exploit are higher
explore = explore * 0.999


#calculate max[Q(next s, all a)]
list_Q = []
for future_pos in range(0, len(pos)):
    #check that the next position isn't empty
    if np.isnan(R[next_state, future_pos]) == False:
        list_Q.append(Q4[next_state, future_pos])
#maximum Q
maximum_Q = max(list_Q)

#define dQ - update rule
dQ = learning_rate*(R[initial_pos, next_state] + gamma* maximum_Q -  Q4[initial_pos

#Q learning update
Q4[initial_pos, next_state] = Q4[initial_pos, next_state] + dQ

#calculate performance measure
discrepancy4.append(np.absolute(dQ))

move_greedy.append(step)
step = step + 1
initial_pos = next_state
```

```
In [47]: #Initialise performance measure, policy each time
         discrepancy4 = []
         QLearning_epsilon(Q4,R,0.8,0.2,0.2)

In [48]: #calculate value
         policy4 = np.argmax(Q4, axis=1)
         Value4 = (np.max(Q4, axis=1))
         print('Policy\n',policy)
         print('V Values\n', Value)
         print('Final Q Values:\n',Q)

Policy
 [0 2 0 2 5 2]
V Values
 [  0.          69.99971241 100.          79.99996629  71.99954943
  89.99999999]
Final Q Values:
 [[  0.           0.           0.           0.           0.           0.        ]
 [ -8.47516088  28.85501922  69.99971241  38.88327287  42.33194567
   49.16925676]
 [100.          49.2112256   66.77996037  62.72875357  56.13899718
   70.7270934 ]
 [ -8.41404132  42.44448673  79.99996629  43.04145526  29.95537957
   34.96718119]
 [ -7.14267796  43.65962658  63.78008714  53.72862296  37.81542323
   71.99954943]
 [ -8.71677981  42.32426539  89.99999999  56.41084873  41.65347906
   55.01128242]]
```

13.2 ϵ-greedy with α=0.8

```
In [49]: Q5 = np.zeros_like(R)
         Q5

Out[49]: array([[ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [50]: move_greedy = []

         def QLearning_epsilon(Q,R,explore,gamma,learning_rate,max_iter=max_iter):

             for episode in range(0, max_iter):
                 #print('Start of episode: ', episode)
                 #Random initial position
                 initial_pos = random.choice(pos)
                 #print(initial_pos)
                 #learning rate
                 #learning_rate = 1 / (np.sqrt(episode + 2))
```

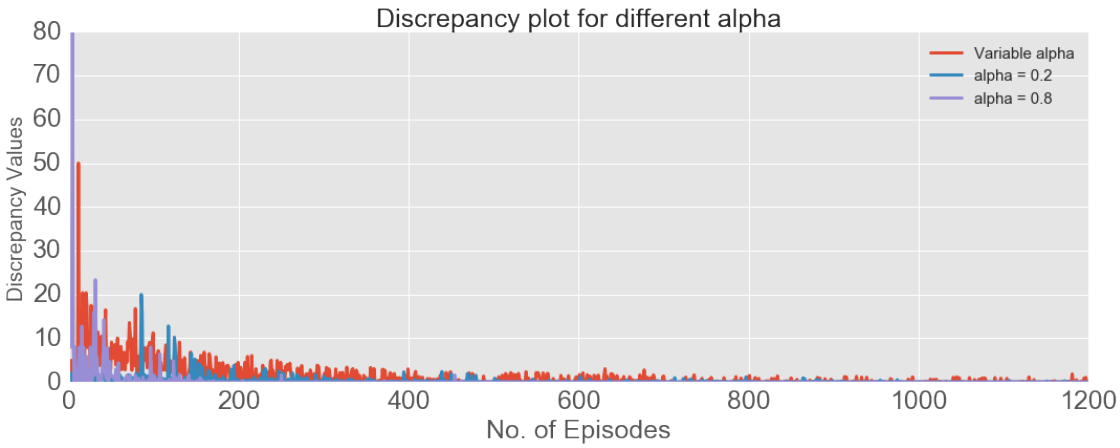
```
#
#two_terminal_in_a_row = -2
step = 1
#While we haven't reached our terminal state, continue searching
#
#while two_terminal_in_a_row < 0:
while initial_pos != 0:
    #It will be used later for the e-greedy policy
    greedy_threshold = random.uniform(0.0,1.0)


    #empty list which will be filled with all the available positions
    NEW_pos = []
    #the available position have to be one of the pos variable
    for next_pos in range(0, len(pos)):
        #search only for positions with a value
        if np.isnan(R[initial_pos, next_pos]) == False:
        #Append to NEW_pos list the Q value of all the available positions
            NEW_pos.append(Q5[initial_pos, next_pos])

    #the maximum value of the above list. It will be used for the movement part
    greedy_Q = max(NEW_pos)

    #MOVEMENT PART

    #We follow the e-greedy policy
    if greedy_Q > 0.0 and greedy_threshold >= explore:
        #Find the index of the maximum Q value
         next_state = NEW_pos.index(greedy_Q)
    else:
        #If the greedy_threshold is smaller than the exploration rate, then we select
         next_state = randrange(0,len(NEW_pos))



    #Decrease explore value very slowly in every iteration. This ensures that at the b
    #when most of the Q values are zeroes, that the algorithm has a higher chance of e
    #that are many iterations the chances of exploit are higher
    explore = explore * 0.999


    #calculate max[Q(next s, all a)]
    list_Q = []
    for future_pos in range(0, len(pos)):
        #check that the next position isn't empty
        if np.isnan(R[next_state, future_pos]) == False:
            list_Q.append(Q5[next_state, future_pos])
    #maximum Q
    maximum_Q = max(list_Q)

    #define dQ - update rule
    dQ = learning_rate*(R[initial_pos, next_state] + gamma* maximum_Q -  Q5[initial_pos

    #Q learning update
```

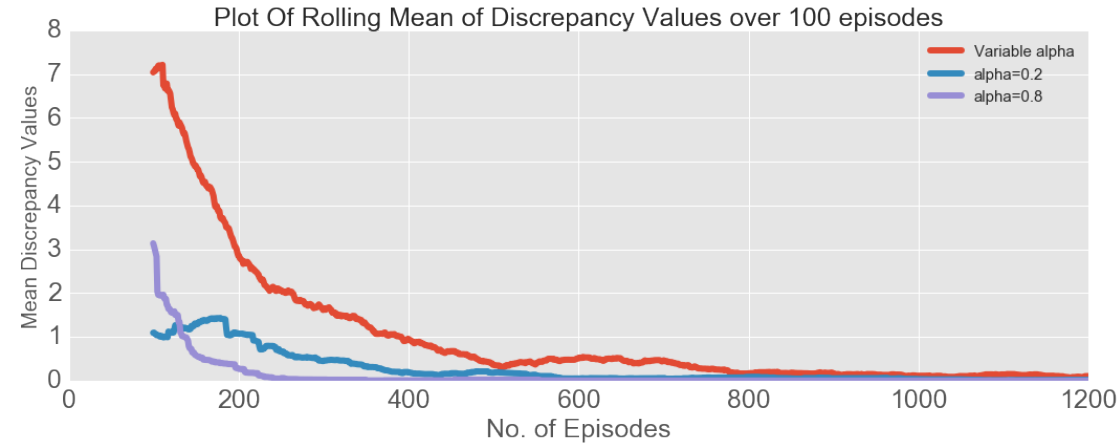
```
                    Q5[initial_pos, next_state] = Q5[initial_pos, next_state] + dQ

                    #calculate performance measure
                    discrepancy5.append(np.absolute(dQ))

                    move_greedy.append(step)
                    step = step + 1
                    initial_pos = next_state
```

In [51]:
```
#Initialise performance measure, policy each time
discrepancy5 = []
QLearning_epsilon(Q5,R,0.8,0.2,0.8)
```

In [52]:
```
disc_pd4 = pd.Series(discrepancy4)
mean_dis4 = disc_pd4.rolling(window=100,center=False).mean()

disc_pd5 = pd.Series(discrepancy5)
mean_dis5 = disc_pd5.rolling(window=100,center=False).mean()
```

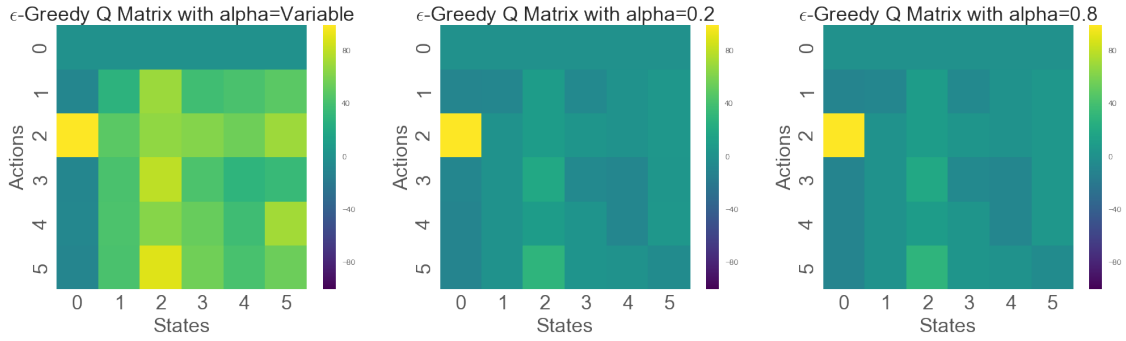
In [53]:
```
plt.figure(figsize=(15,12))
plt.subplot(211)
plt.plot(discrepancy, linewidth=3, label="Variable alpha")
plt.plot(discrepancy4, linewidth=3, label="alpha = 0.2")
plt.plot(discrepancy5, linewidth=3, label="alpha = 0.8")
plt.title('Discrepancy plot for different alpha', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("Discrepancy Values", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.xlim(0,max_iter)
plt.legend(fontsize=15, loc=1)

plt.subplot(212)
plt.plot(mean_dis, linewidth=6, label='Variable alpha')
plt.plot(mean_dis4, linewidth=6, label='alpha=0.2')
plt.plot(mean_dis5, linewidth=6, label='alpha=0.8')
plt.title('Plot Of Rolling Mean of Discrepancy Values over 100 episodes', fontsize=25)
plt.xlabel("No. of Episodes", fontsize=25)
plt.ylabel("Mean Discrepancy Values", fontsize=20)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.xlim(0,max_iter)
plt.legend(fontsize=15, loc=1)

plt.tight_layout()
```





In [54]:
```
plt.figure(figsize=(20,6))

plt.subplot(131)
sns.heatmap(Q, cmap=plt.cm.viridis)
plt.title("$\epsilon$-Greedy Q Matrix with alpha=Variable", fontsize=25)
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)

plt.subplot(132)
sns.heatmap(Q4, cmap=plt.cm.viridis)
plt.title("$\epsilon$-Greedy Q Matrix with alpha=0.2", fontsize=25)
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)

plt.subplot(133)
sns.heatmap(Q5, cmap=plt.cm.viridis)
plt.title("$\epsilon$-Greedy Q Matrix with alpha=0.8", fontsize=25)
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
```

```
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
plt.tight_layout()
```

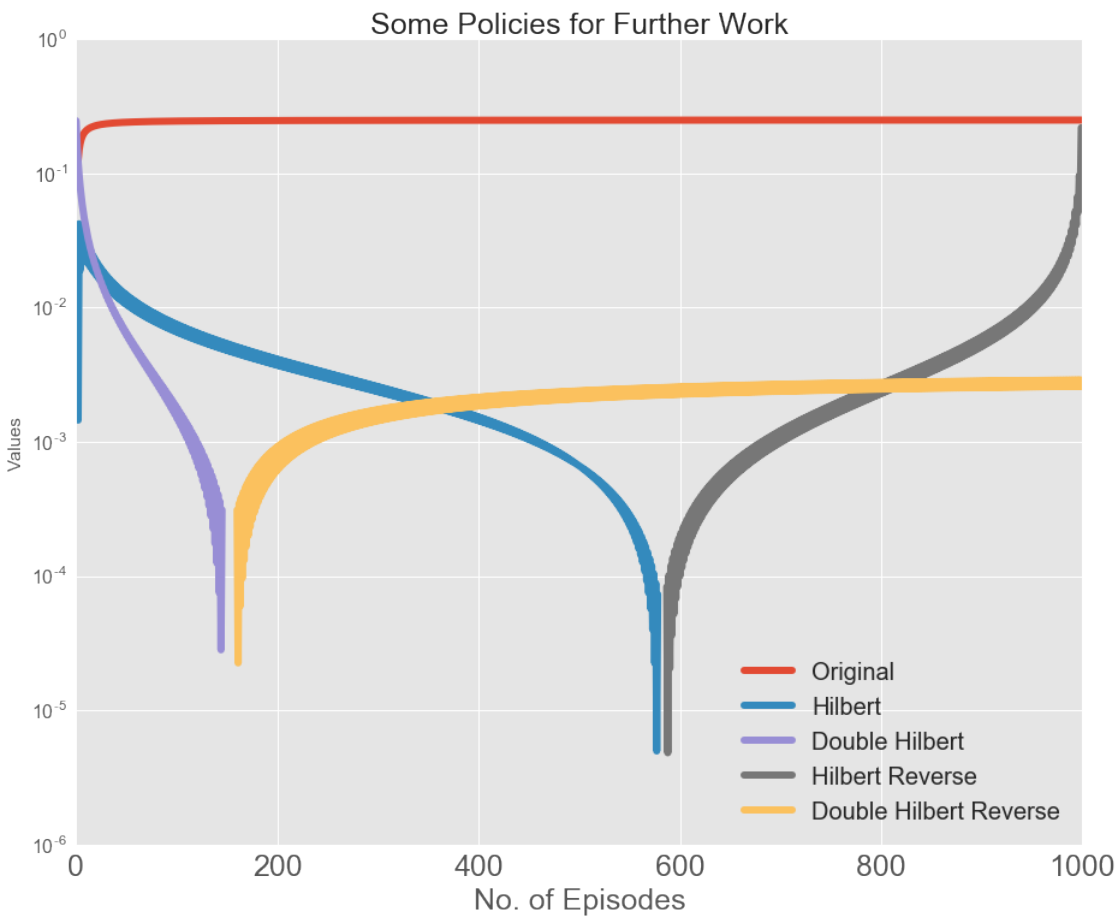


14 Extensions

```
In [55]: import scipy
         from scipy.fftpack import *
         f = lambda x: (x / (x + 2)**1/4)
         val =  f(np.arange(1000))
```

```
In [56]: plt.figure(figsize=(15,12))
         plt.semilogy(val, label='Original', linewidth=6)
         plt.semilogy(hilbert(val), label='Hilbert', linewidth=6)
         plt.semilogy(hilbert(hilbert(val)), label='Double Hilbert', linewidth=6)
         plt.semilogy(hilbert(val)*-1, label='Hilbert Reverse', linewidth=6)
         plt.semilogy(hilbert(hilbert(val))*-1, label='Double Hilbert Reverse', linewidth=6)
         plt.legend(fontsize=20, loc=4)

         plt.title('Some Policies for Further Work', fontsize=25)
         plt.xlabel("No. of Episodes", fontsize=25)
         plt.ylabel("Values", fontsize=15)
         plt.xticks(fontsize=25)
         plt.yticks(fontsize=16)
```

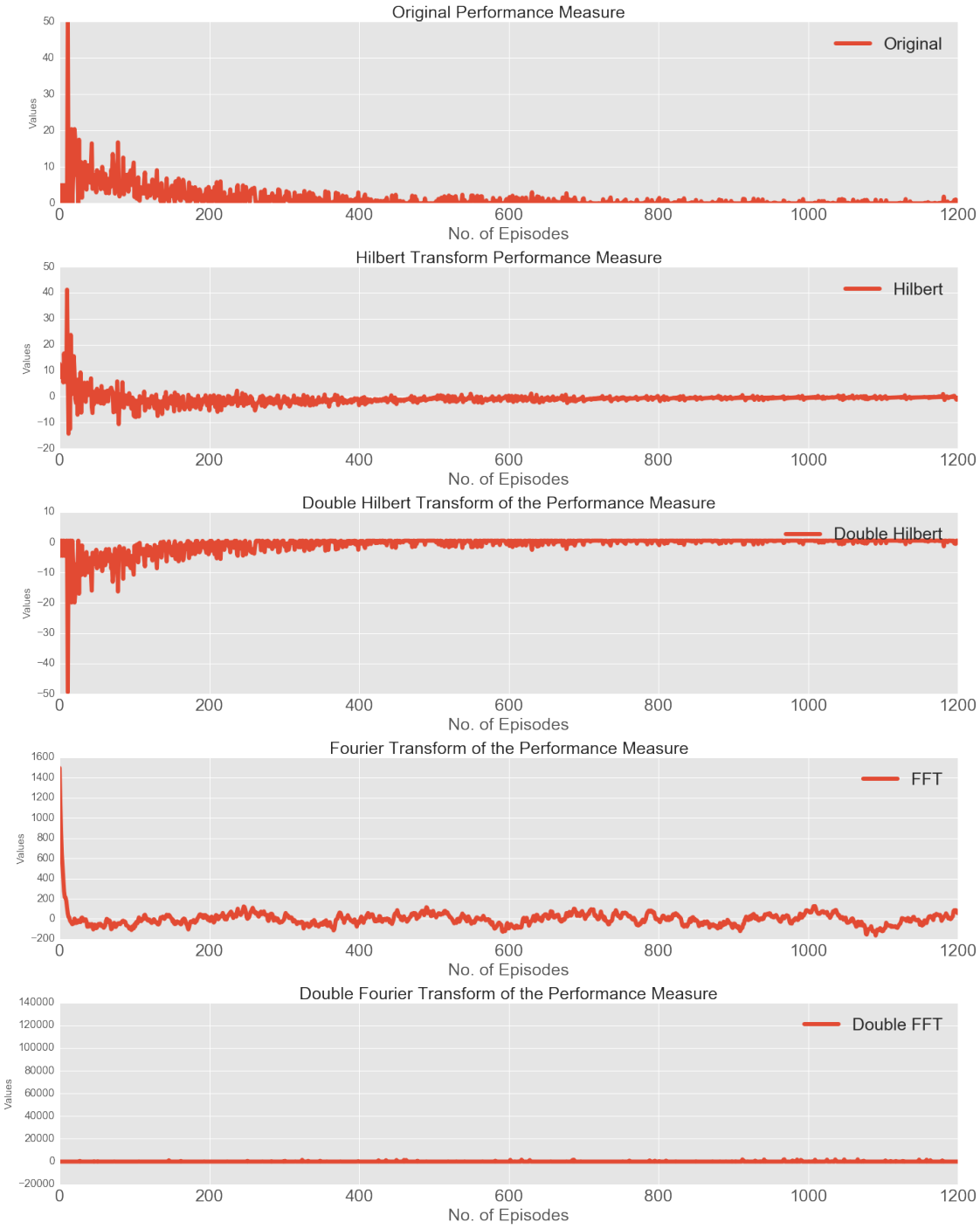
```
Out[56]: (array([  1.00000000e-07,   1.00000000e-06,   1.00000000e-05,
                   1.00000000e-04,   1.00000000e-03,   1.00000000e-02,
                   1.00000000e-01,   1.00000000e+00,   1.00000000e+01]),
          <a list of 9 Text yticklabel objects>)
```




```
In [70]: plt.figure(figsize=(20,25))
         plt.subplot(511)
         plt.plot(discrepancy, label='Original', linewidth=6)
         plt.title('Original Performance Measure', fontsize=25)
         plt.xlabel("No. of Episodes", fontsize=25)
         plt.ylabel("Values", fontsize=15)
         plt.xlim(0,max_iter)
         plt.legend(fontsize=25, loc=1)
         plt.xticks(fontsize=25)
         plt.yticks(fontsize=16)

         plt.subplot(512)
         plt.plot(hilbert(discrepancy), label='Hilbert', linewidth=6)
         plt.title('Hilbert Transform Performance Measure', fontsize=25)
         plt.xlabel("No. of Episodes", fontsize=25)
         plt.ylabel("Values", fontsize=15)
         plt.xlim(0,max_iter)
         plt.legend(fontsize=25, loc=1)
         plt.xticks(fontsize=25)
         plt.yticks(fontsize=16)

         plt.subplot(513)
```

```
        plt.plot(hilbert(hilbert(discrepancy)), label='Double Hilbert', linewidth=6)
        plt.title('Double Hilbert Transform of the Performance Measure', fontsize=25)
        plt.xlabel("No. of Episodes", fontsize=25)
        plt.ylabel("Values", fontsize=15)
        plt.xlim(0,max_iter)
        plt.legend(fontsize=25, loc=1)
        plt.xticks(fontsize=25)
        plt.yticks(fontsize=16)

        plt.subplot(514)
        plt.plot(fft(discrepancy), label='FFT', linewidth=6)
        plt.title('Fourier Transform of the Performance Measure', fontsize=25)
        plt.xlabel("No. of Episodes", fontsize=25)
        plt.ylabel("Values", fontsize=15)
        plt.xlim(0,max_iter)
        plt.legend(fontsize=25, loc=1)
        plt.xticks(fontsize=25)
        plt.yticks(fontsize=16)

        plt.subplot(515)
        plt.plot(fft(fft(discrepancy)), label='Double FFT', linewidth=6)
        plt.title('Double Fourier Transform of the Performance Measure', fontsize=25)
        plt.xlabel("No. of Episodes", fontsize=25)
        plt.ylabel("Values", fontsize=15)
        plt.xlim(0,max_iter)
        plt.legend(fontsize=25, loc=1)
        plt.xticks(fontsize=25)
        plt.yticks(fontsize=16)

        plt.tight_layout()
```

```
C:\Users\arsha_000\Anaconda3\lib\site-packages\numpy\core\numeric.py:474: ComplexWarning: Casting comple
  return array(a, dtype, copy=False, order=order)
```


Original Performance Measure
Original
Values
No. of Episodes
Hilbert Transform Performance Measure
Hilbert
Values
No. of Episodes
Double Hilbert Transform of the Performance Measure
Double Hilbert
Values
No. of Episodes
Fourier Transform of the Performance Measure
FFT
Values
No. of Episodes
Double Fourier Transform of the Performance Measure
Double FFT
Values
No. of Episodes

Stochastic_QLearning_AA

March 23, 2016

1 Table of Contents

2 Introduction

2.1 Domain

We consider a case where an agent moves in a NxN grid and tries to find the optimal way around this grid while navigating various obstacles. We represent all possible states that the agent can move into with either

0 or positive numbers. We represent obstacles with negative numbers. In order to do so we will need to define a Markov Decision Process (MDP). This will be defined in a matrix form. The analysis will ber performed in Python using the MDP Toolbox, Numpy and Matplotlib packages.

The package at the moment works in a 3D matrix sense only so the Q Learning algorithm implementation here expects a (A,S,S) transition probability matrix and a (A,S) reward matrix. So we use the utility to generate a (N,N,N) problem space to be compliant with the requirements.

Note that Python uses 0 based indexing

```
In [1]: #Import some libraries
        import numpy as np
        import random
        import matplotlib.pyplot as plt
        import mdptoolbox as mdp
        import mdptoolbox.example
        import seaborn as sns
        np.random.seed(0)
        %matplotlib inline
        plt.style.use('ggplot')
        from IPython.display import display, Math, Latex
```

```
In [86]: #Set plotting styles
         c= sns.plotting_context("poster", font_scale=3, rc={"lines.linewidth": 3.5})
```

2.2 Number of States, S

```
In [3]: S=6
```

2.3 Number of Actions, A

```
In [4]: A=6
```

2.4 Transition Probability Matrix, P

```
In [5]: P, R = mdptoolbox.example.rand(S,A, is_sparse=False)
```

```
In [6]: P[1:2,]
```

```
Out[6]: array([[[ 0.28619179,  0.        ,  0.36763563,  0.        ,  0.34617258,
                  0.        ],
                [ 0.        ,  0.11466841,  0.13597694,  0.27825365,  0.25150863,
                  0.21959237],
                [ 0.        ,  0.21384415,  0.        ,  0.22139517,  0.26564833,
                  0.29911235],
                [ 1.        ,  0.        ,  0.        ,  0.        ,  0.        ,
                  0.        ],
                [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
                  1.        ],
                [ 0.        ,  0.        ,  0.        ,  1.        ,  0.        ,
                  0.        ]]])
```

```
In [7]: print(P.shape[1])
        print(P.shape[0])
```

```
6
6
```

2.5 R Matrix

```
In [8]: R[1:2,]
```

```
Out[8]: array([[[-0.39685037,  0.        , -0.41984479,  0.        , -0.1424626 ,
                 -0.        ],
                [-0.        ,  0.83896523,  0.4284826 ,  0.99769401, -0.70110339,
                  0.73625211],
                [ 0.        , -0.97657183, -0.        ,  0.45998112, -0.65674065,
                  0.04207321],
                [-0.46547492,  0.        ,  0.        ,  0.        ,  0.        ,
                  0.        ],
                [ 0.        ,  0.        ,  0.        , -0.        ,  0.        ,
                  0.21409012],
                [-0.        ,  0.        ,  0.        , -0.08774046, -0.        ,
                  0.        ]]])
```

```
In [9]: print(R.shape[1])
        print(R.shape[0])
```

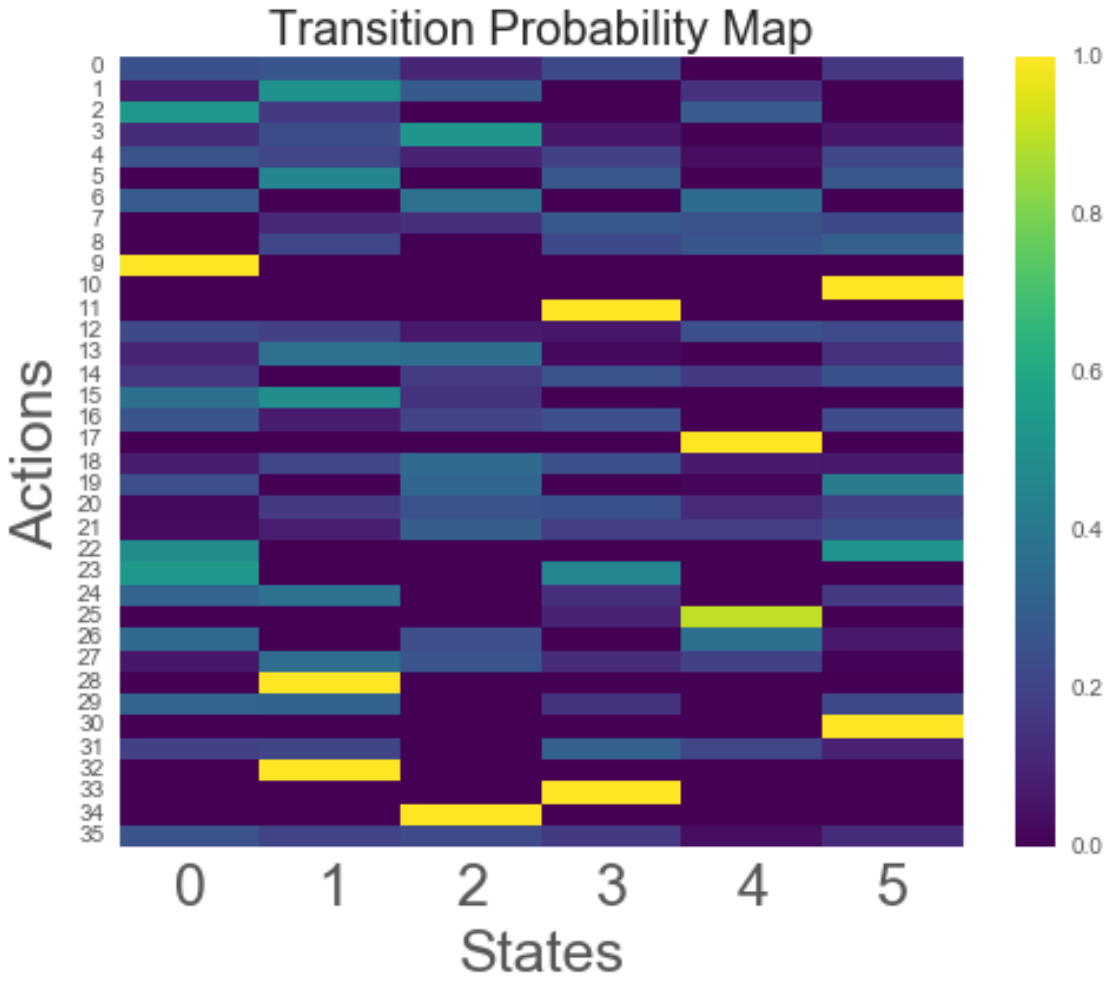
```
6
6
```

2.6 Plot of Transition Probabilities

Here we plot the transition probability matrix and show in light colors the low probability areas and in dark colors the high probability areas. Due to the 3D nature of these matrices we stack in the vertical direction for plotting

```
In [57]: plt.figure(figsize=(8,6))
         plt.title("Transition Probability Map", fontsize=20)
         sns.heatmap(np.vstack(P), cmap=plt.cm.viridis)
         plt.xlabel("States", fontsize=25)
         plt.ylabel('Actions', fontsize=25)
         plt.xticks(fontsize=25)
         plt.yticks(fontsize=10)
```

```
Out[57]: (array([ 0.5,  1.5,  2.5,  3.5,  4.5,  5.5,  6.5,  7.5,  8.5,
                  9.5, 10.5, 11.5, 12.5, 13.5, 14.5, 15.5, 16.5, 17.5,
                 18.5, 19.5, 20.5, 21.5, 22.5, 23.5, 24.5, 25.5, 26.5,
                 27.5, 28.5, 29.5, 30.5, 31.5, 32.5, 33.5, 34.5, 35.5]),
          <a list of 36 Text yticklabel objects>)
```

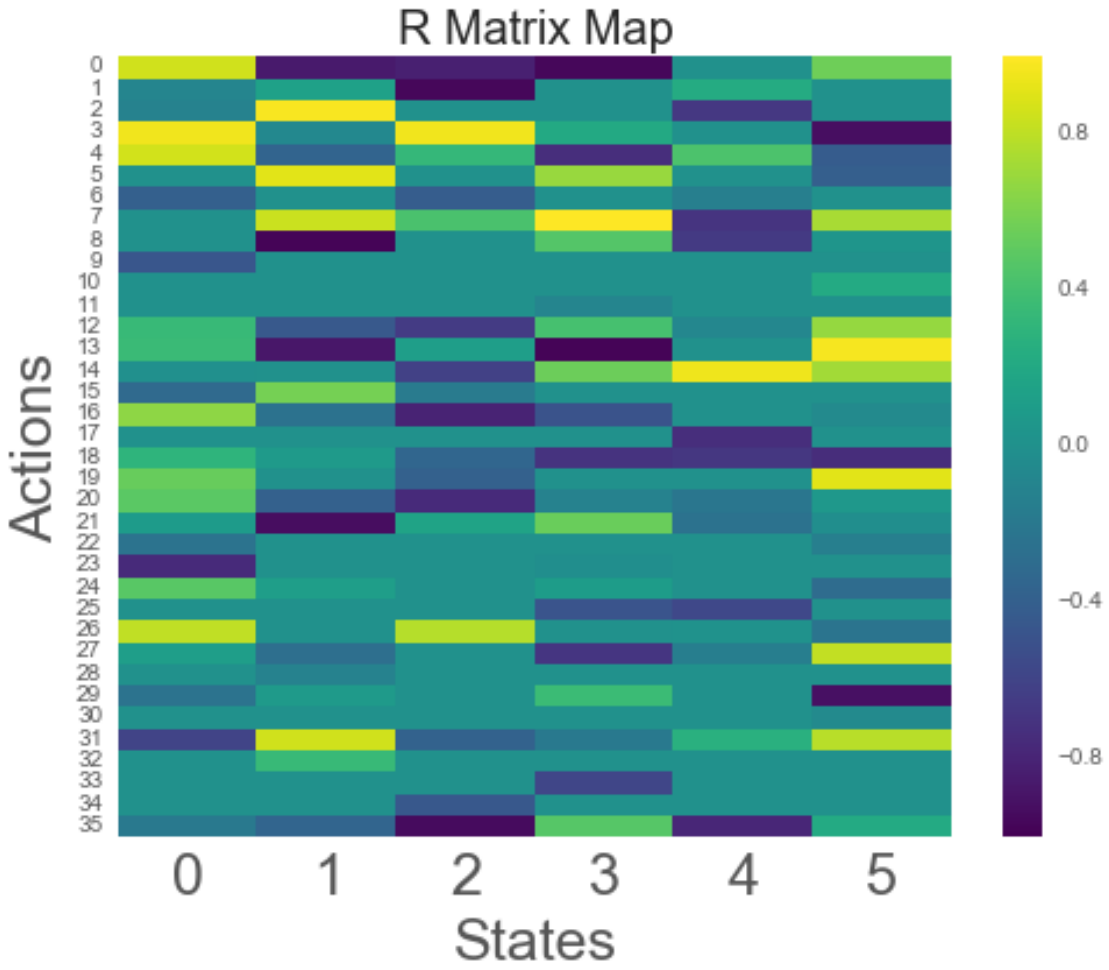


2.7 Plot of R Matrix

Here we plot the transition probability matrix and show in light colors the low probability areas and in dark colors the high probability areas.

```
In [58]: plt.figure(figsize=(8,6))
         sns.heatmap(np.vstack(R), cmap=plt.cm.viridis)
         plt.title("R Matrix Map", fontsize=20)
         plt.xlabel("States", fontsize=25)
         plt.ylabel('Actions', fontsize=25)
         plt.xticks(fontsize=25)
         plt.yticks(fontsize=10)
```

```
Out[58]: (array([  0.5,   1.5,   2.5,   3.5,   4.5,   5.5,   6.5,   7.5,   8.5,
                    9.5,  10.5,  11.5,  12.5,  13.5,  14.5,  15.5,  16.5,  17.5,
                   18.5,  19.5,  20.5,  21.5,  22.5,  23.5,  24.5,  25.5,  26.5,
                   27.5,  28.5,  29.5,  30.5,  31.5,  32.5,  33.5,  34.5,  35.5]),
          <a list of 36 Text yticklabel objects>)
```

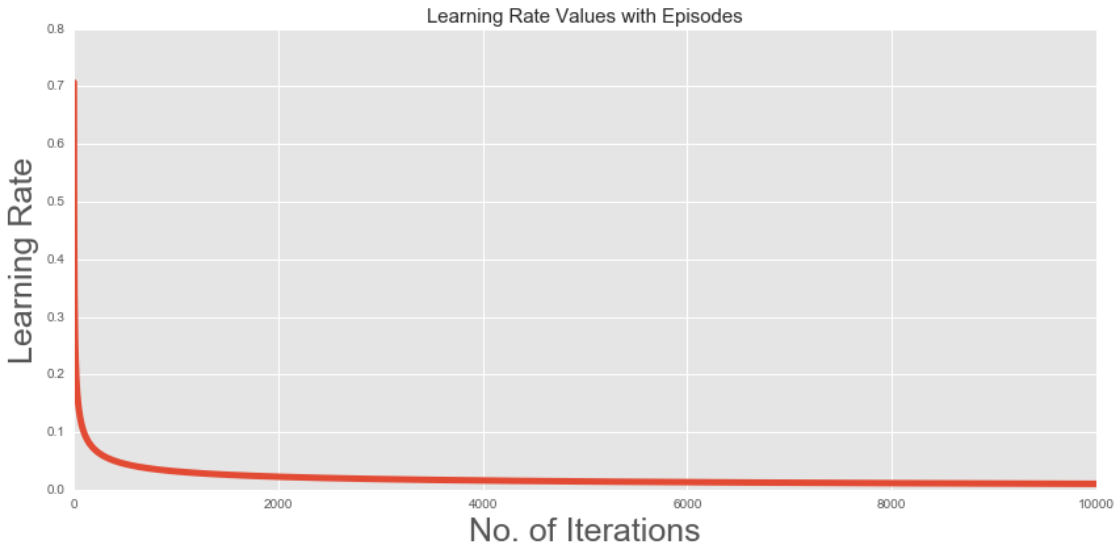


2.8 Learning Rate, α

We define our learning rate as a function of our number of episodes, n. This ensures a more dynamic learning rate as opposed a static value. We define our alpha s follows:

$$\alpha = \frac{1}{\sqrt{(n+2)}}$$

```
In [13]: plt.figure(figsize=(12,6))
         plt.plot(1/np.sqrt((np.arange(10000)+2)), linewidth =5)
         plt.xlabel("No. of Iterations", fontsize=25)
         plt.ylabel("Learning Rate", fontsize=25)
         plt.title('Learning Rate Values with Episodes', fontsize=15)
         plt.tight_layout()
```



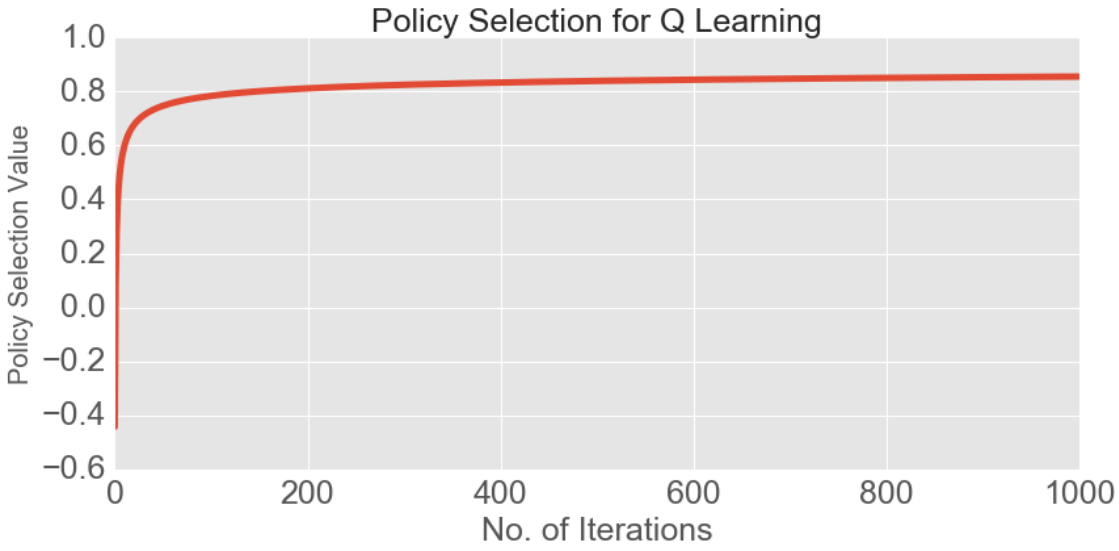
2.9 Policy Selection, $\epsilon - greedy$

We use a ϵ-greedy policy for all the testing in this section. We define our ϵ as:

$$\epsilon = 1 - \frac{1}{\log(n+2)}$$

This is compared against a random number sampled from a continuous uniform distribution between the interval of [0,1]. When $\epsilon >$ than the random number the agent exploits and when it is less the agent explores.

```
In [14]: plt.figure(figsize=(12,6))
         plt.plot((1- (1/(np.log(np.arange(1000)+2)))) , linewidth =5)
         plt.xlabel("No. of Iterations", fontsize=25)
         plt.ylabel("Policy Selection Value", fontsize=20)
         plt.title('Policy Selection for Q Learning', fontsize=25)
         plt.xticks(fontsize=25)
         plt.yticks(fontsize=25)
         plt.tight_layout()
```



3 Q Learning with different gamma values

3.1 Q Learning with Gamma = 0.5

```
In [15]: ql_1 = mdp.mdp.QLearning(P,R,0.5)
         print("Initial Values")
         print("Actions \n", ql_1.A)
         print("Q Matrix \n",ql_1.Q )
         print("Rewards \n", ql_1.R[1:2,])
         ql_1.run()
```

```
Initial Values
Actions
 6
Q Matrix
 [[ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]]
Rewards
 [[[-0.39685037  0.         -0.41984479  0.         -0.1424626  -0.        ]
  [-0.          0.83896523  0.4284826   0.99769401 -0.70110339  0.73625211]
  [ 0.         -0.97657183 -0.          0.45998112 -0.65674065  0.04207321]
  [-0.46547492  0.          0.          0.          0.          0.        ]
  [ 0.          0.          0.         -0.          0.          0.21409012]
  [-0.          0.          0.         -0.08774046 -0.          0.        ]]]
```

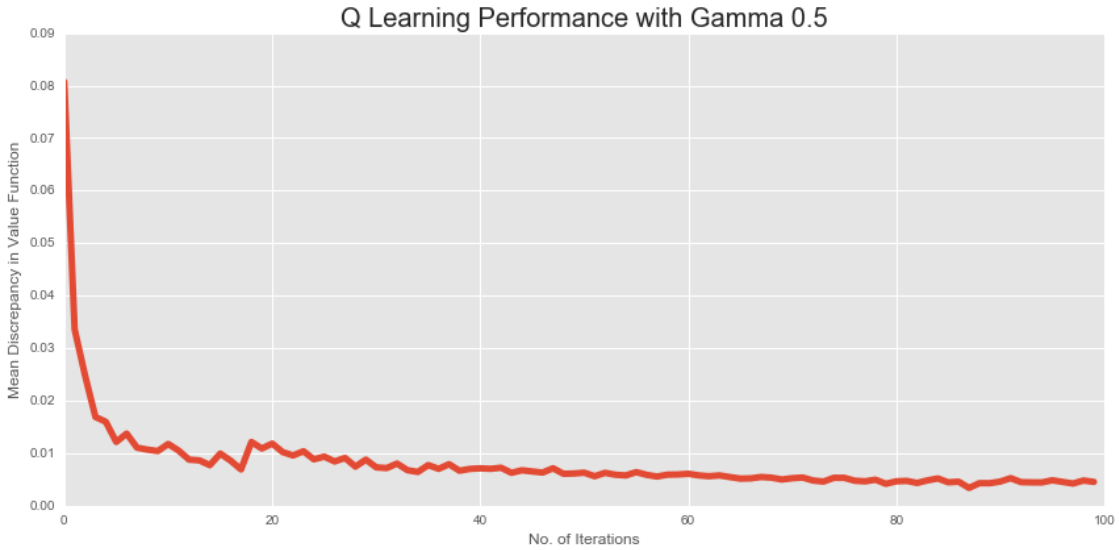
3.1.1 Results Q Learning Gamma = 0.5

```
In [16]: print('After Q Learing with Gamma %s' %ql_1.discount)
         print("Actions :", ql_1.A)
```

```
        print("Q Matrix \n",ql_1.Q )
        print("Rewards \n", ql_1.R[1:2,])
        print("Optimal Policy :", ql_1.policy)
        print("Time taken :%g"%ql_1.time)
        print("Total Iterations :", ql_1.max_iter)
```

```
After Q Learing with Gamma 0.5
Actions : 6
Q Matrix
 [[ 0.02449109 -0.0344293   0.09234019 -0.13636232  0.0930445   0.41183435]
 [ 0.09017421  0.85850942  0.1058576   0.33269218 -0.13129809  0.15465009]
 [-0.1129266   0.09831385  0.75103282  0.0426225   0.4801443   0.36179264]
 [ 0.8827493  -0.24674176  0.29824492  0.15420645  0.09012653 -0.10706585]
 [-0.03450163  0.67621459  0.02383737  0.00795256  0.0724088  -0.03121429]
 [ 0.82631928  0.33833197 -0.32547418 -0.06665251  0.03771055 -0.03344386]]
Rewards
 [[[-0.39685037  0.         -0.41984479  0.         -0.1424626  -0.        ]
  [-0.          0.83896523  0.4284826   0.99769401 -0.70110339  0.73625211]
  [ 0.         -0.97657183 -0.          0.45998112 -0.65674065  0.04207321]
  [-0.46547492  0.          0.          0.          0.          0.        ]
  [ 0.          0.          0.         -0.          0.          0.21409012]
  [-0.          0.          0.         -0.08774046 -0.          0.        ]]]
Optimal Policy : (5, 1, 2, 0, 1, 0)
Time taken :0.605089
Total Iterations : 10000
```

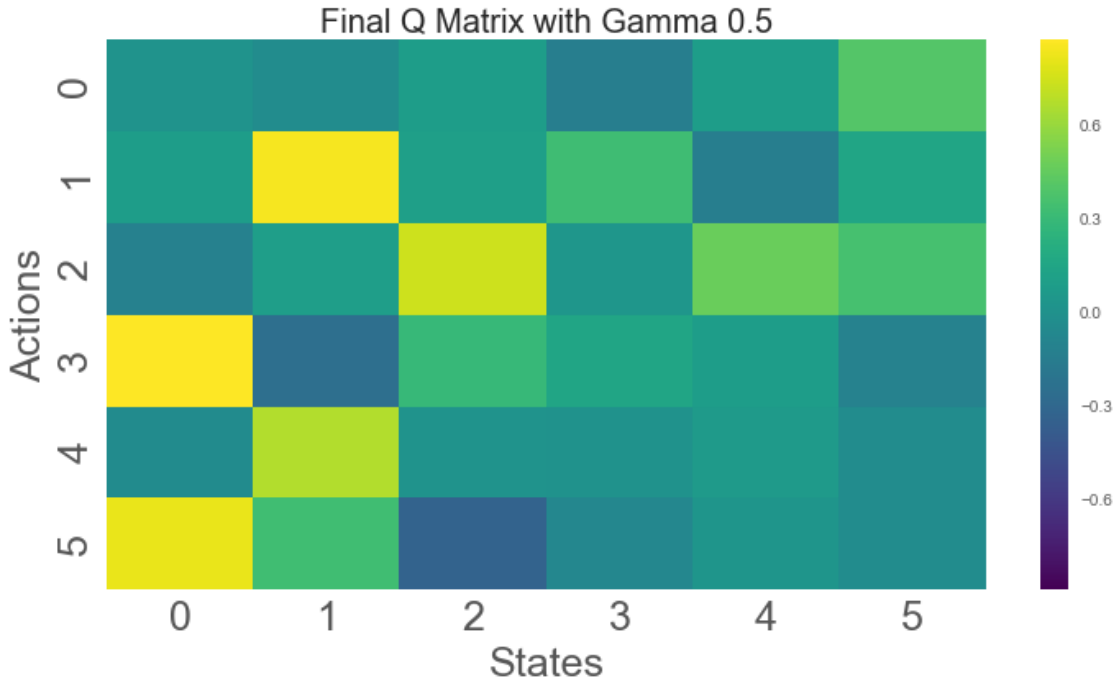
In [17]:
```
        plt.figure(figsize=(12,6))
        plt.plot(ql_1.mean_discrepancy, linewidth =5)
        plt.xlabel("No. of Iterations")
        plt.ylabel("Mean Discrepancy in Value Function")
        plt.title('Q Learning Performance with Gamma %s' %ql_1.discount, fontsize=20)
        plt.tight_layout()
```



In [60]:
```
        plt.figure(figsize=(12,6))
        sns.heatmap(ql_1.Q, cmap=plt.cm.viridis)
```

```
plt.title('Final Q Matrix with Gamma %s' %ql_1.discount, fontsize=20)
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=25)
```

```
Out[60]: (array([ 0.5,  1.5,  2.5,  3.5,  4.5,  5.5]),
          <a list of 6 Text yticklabel objects>)
```



3.2 Q Learning with Gamma = 0

```
In [19]: ql_2 = mdp.mdp.QLearning(P,R,0)
         print("Initial Values")
         print("Actions \n", ql_2.A)
         print("Q Matrix \n",ql_2.Q )
         print("Rewards \n", ql_2.R[1:2,])
         ql_2.run()
```

```
Initial Values
Actions
 6
Q Matrix
 [[ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]]
Rewards
```

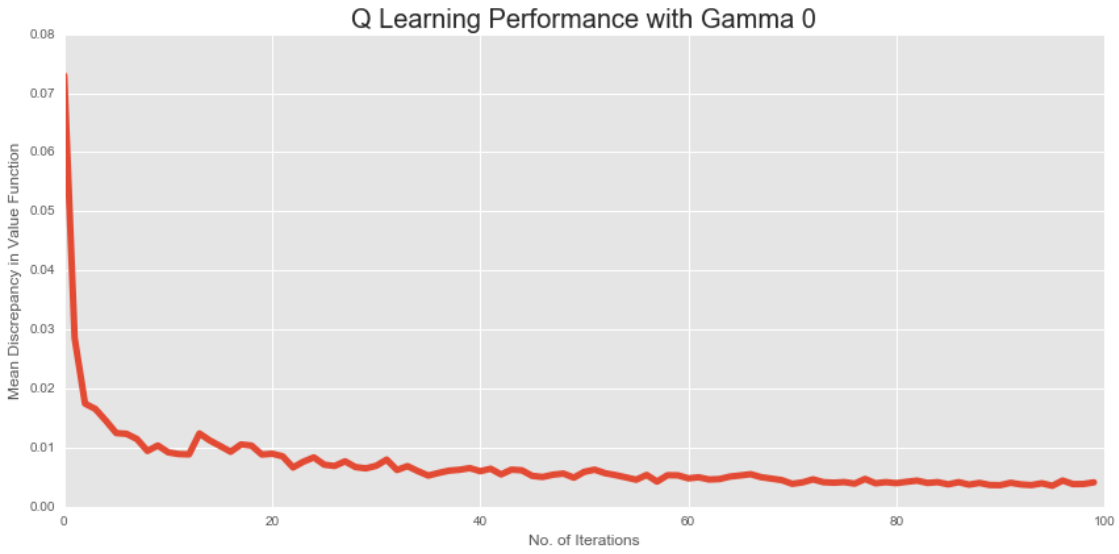
```
 [[[-0.39685037  0.         -0.41984479  0.         -0.1424626  -0.        ]
  [-0.          0.83896523  0.4284826   0.99769401 -0.70110339  0.73625211]
  [ 0.         -0.97657183 -0.          0.45998112 -0.65674065  0.04207321]
  [-0.46547492  0.          0.          0.          0.          0.        ]
  [ 0.          0.          0.         -0.          0.          0.21409012]
  [-0.          0.          0.         -0.08774046 -0.          0.        ]]]
```

3.2.1 Results Q Learning Gamma = 0

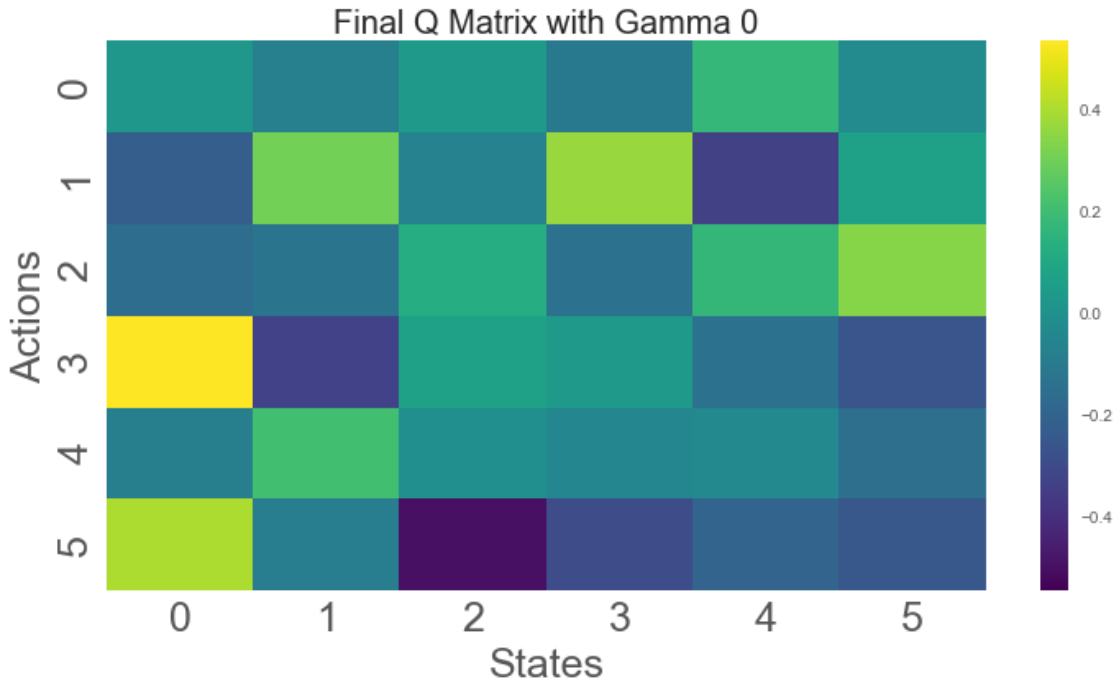
```
In [20]: print('After Q Learing with Gamma %s' %ql_2.discount)
         print("Actions :", ql_2.A)
         print("Q Matrix \n",ql_2.Q )
         print("Rewards \n", ql_2.R[1:2,])
         print("Optimal Policy :", ql_2.policy)
         print("Time taken :%g"%ql_2.time)
         print("Total Iterations :", ql_2.max_iter)
```

```
After Q Learing with Gamma 0
Actions : 6
Q Matrix
 [[ 0.02970631 -0.07597226  0.04629893 -0.10176937  0.18142385 -0.02801513]
 [-0.22144939  0.31211941 -0.05953707  0.37223549 -0.32992478  0.0744957 ]
 [-0.15842262 -0.12579506  0.12906559 -0.13319831  0.17429078  0.34408211]
 [ 0.54065493 -0.3262809   0.07551968  0.04284658 -0.13616605 -0.25762093]
 [-0.07853041  0.21408881 -0.00562782 -0.04469116 -0.03208163 -0.14823256]
 [ 0.40909225 -0.08771431 -0.49506372 -0.29472908 -0.19450835 -0.2481426 ]]
Rewards
 [[[-0.39685037  0.         -0.41984479  0.         -0.1424626  -0.        ]
  [-0.          0.83896523  0.4284826   0.99769401 -0.70110339  0.73625211]
  [ 0.         -0.97657183 -0.          0.45998112 -0.65674065  0.04207321]
  [-0.46547492  0.          0.          0.          0.          0.        ]
  [ 0.          0.          0.         -0.          0.          0.21409012]
  [-0.          0.          0.         -0.08774046 -0.          0.        ]]]
Optimal Policy : (4, 3, 5, 0, 1, 0)
Time taken :0.609527
Total Iterations : 10000
```

```
In [21]: plt.figure(figsize=(12,6))
         plt.plot(ql_2.mean_discrepancy, linewidth =5)
         plt.xlabel("No. of Iterations")
         plt.ylabel("Mean Discrepancy in Value Function")
         plt.title('Q Learning Performance with Gamma %s' %ql_2.discount, fontsize=20)
         plt.tight_layout()
```




```
In [61]: plt.figure(figsize=(12,6))
         sns.heatmap(ql_2.Q, cmap=plt.cm.viridis)
         plt.title('Final Q Matrix with Gamma %s' %ql_2.discount,  fontsize=20)
         plt.xlabel("States", fontsize=25)
         plt.ylabel('Actions', fontsize=25)
         plt.xticks(fontsize=25)
         plt.yticks(fontsize=25)

Out[61]: (array([ 0.5,  1.5,  2.5,  3.5,  4.5,  5.5]),
          <a list of 6 Text yticklabel objects>)
```



3.3 Q Learning with Gamma = 1

```
In [23]: ql_3 = mdp.mdp.QLearning(P,R,1)
         print("Initial Values")
         print("Actions \n", ql_3.A)
         print("Q Matrix \n",ql_3.Q )
         print("Rewards \n", ql_3.R[1:2,])
         ql_3.run()
```

```
Initial Values
Actions
 6
Q Matrix
 [[ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]]
Rewards
 [[[-0.39685037  0.         -0.41984479  0.         -0.1424626  -0.        ]
  [-0.          0.83896523  0.4284826   0.99769401 -0.70110339  0.73625211]
  [ 0.         -0.97657183 -0.          0.45998112 -0.65674065  0.04207321]
  [-0.46547492  0.          0.          0.          0.          0.        ]
  [ 0.          0.          0.         -0.          0.          0.21409012]
  [-0.          0.          0.         -0.08774046 -0.          0.        ]]]
```

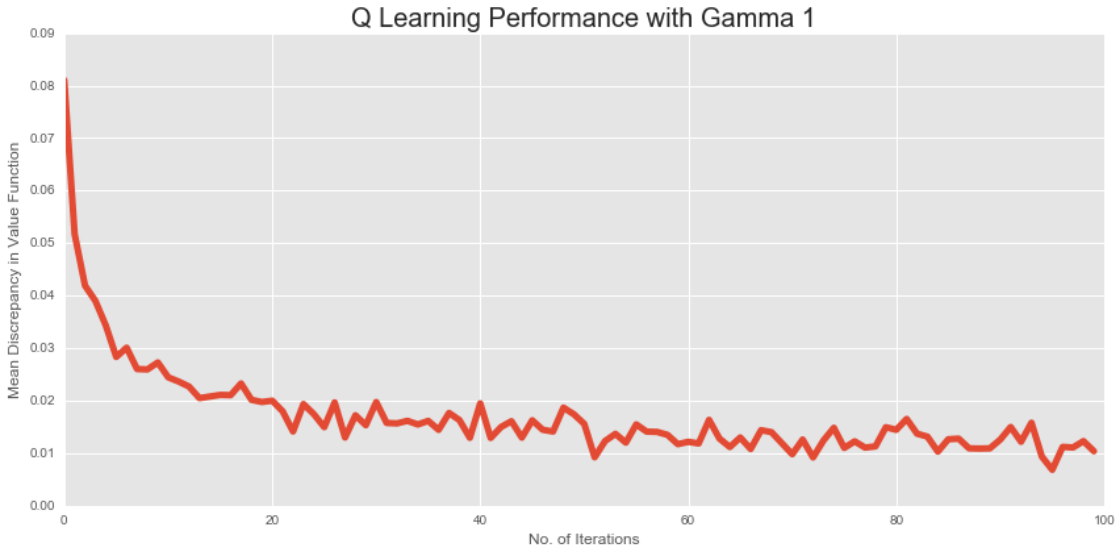
3.3.1 Results Q Learning Gamma = 1

```
In [24]: print('After Q Learing with Gamma %s' %ql_3.discount)
         print("Actions :", ql_3.A)
         print("Q Matrix \n %s"%ql_3.Q )
         print("Rewards \n", ql_3.R[1:2,])
         print("Optimal Policy :", ql_3.policy)
         print("Time taken :%g"%ql_3.time)
         print("Total Iterations :", ql_3.max_iter)
```

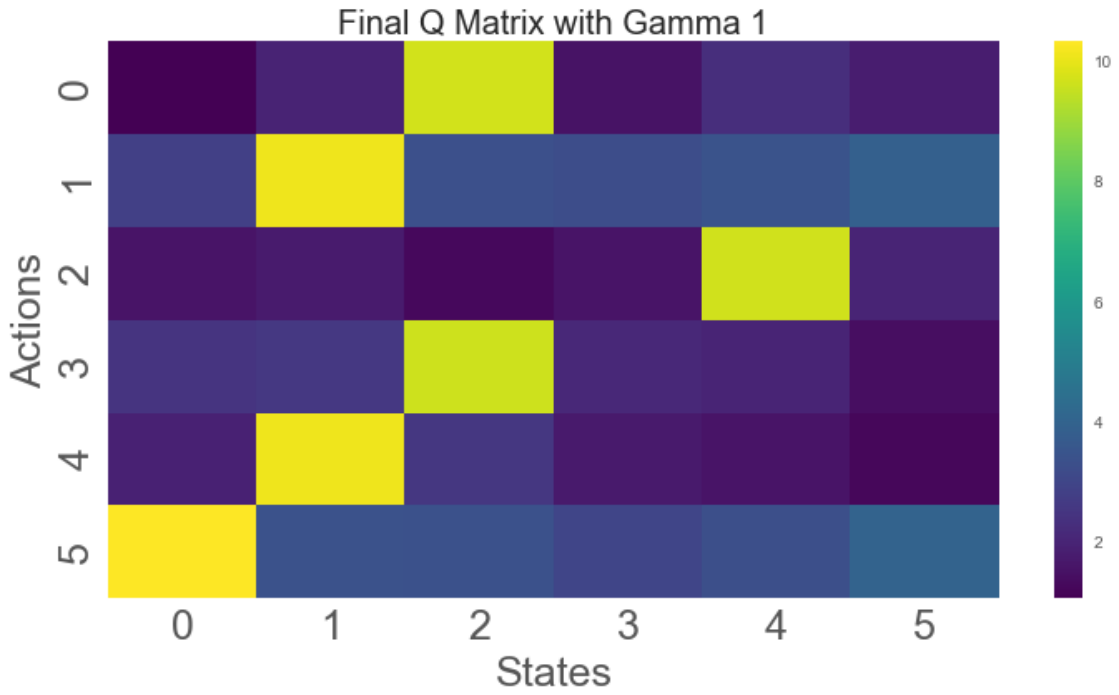
```
After Q Learing with Gamma 1
Actions : 6
Q Matrix
 [[  1.0953202    1.97898059   9.73705125   1.55170817   2.31570529
    1.85440223]
 [  2.85830871  10.13743485   3.37362486   3.2434592    3.48206867
    3.9140436 ]
 [  1.57610248   1.76368886   1.30402078   1.58793037   9.70286436
    2.0146696 ]
 [  2.5027308    2.59995538   9.66113082   2.14001023   2.02851541
    1.45330065]
 [  1.94716061  10.14066281   2.57428113   1.72111458   1.57468812
    1.25902517]
 [ 10.37301913   3.42348474   3.38902207   2.99763139   3.33823125
    3.99578894]]
```

```
Rewards
 [[[-0.39685037  0.         -0.41984479  0.         -0.1424626  -0.        ]
  [-0.          0.83896523  0.4284826   0.99769401 -0.70110339  0.73625211]
  [ 0.         -0.97657183 -0.          0.45998112 -0.65674065  0.04207321]
  [-0.46547492  0.          0.          0.          0.          0.        ]
  [ 0.          0.          0.         -0.          0.          0.21409012]
  [-0.          0.          0.         -0.08774046 -0.          0.        ]]]
Optimal Policy : (2, 1, 4, 2, 1, 0)
Time taken :0.609409
Total Iterations : 10000
```

```
In [25]: plt.figure(figsize=(12,6))
         plt.plot(ql_3.mean_discrepancy, linewidth =5)
         plt.xlabel("No. of Iterations")
         plt.ylabel("Mean Discrepancy in Value Function")
         plt.title('Q Learning Performance with Gamma %s' %ql_3.discount, fontsize=20)
         plt.tight_layout()
```

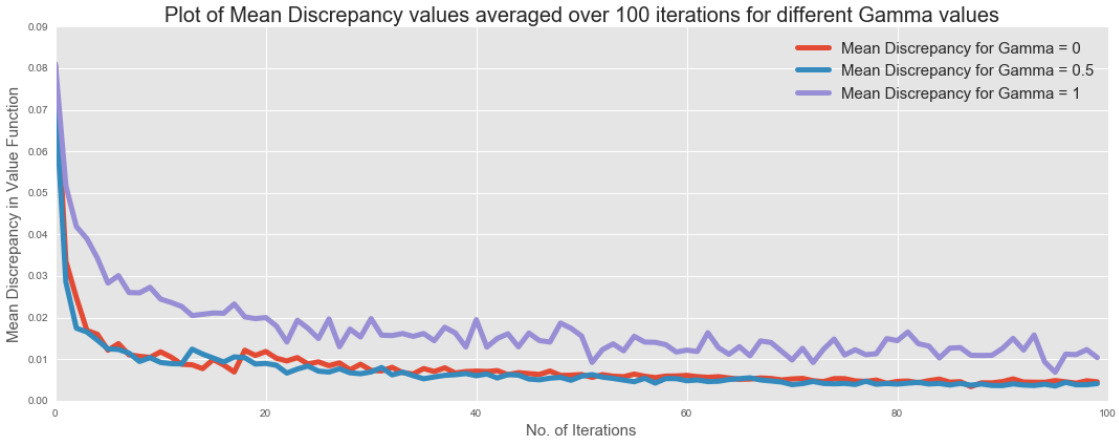



```
In [62]: plt.figure(figsize=(12,6))
         sns.heatmap(ql_3.Q, cmap=plt.cm.viridis)
         plt.title('Final Q Matrix with Gamma %s' %ql_3.discount, fontsize=20)
         plt.xlabel("States", fontsize=25)
         plt.ylabel('Actions', fontsize=25)
         plt.xticks(fontsize=25)
         plt.yticks(fontsize=25)

Out[62]: (array([ 0.5,  1.5,  2.5,  3.5,  4.5,  5.5]),
          <a list of 6 Text yticklabel objects>)
```

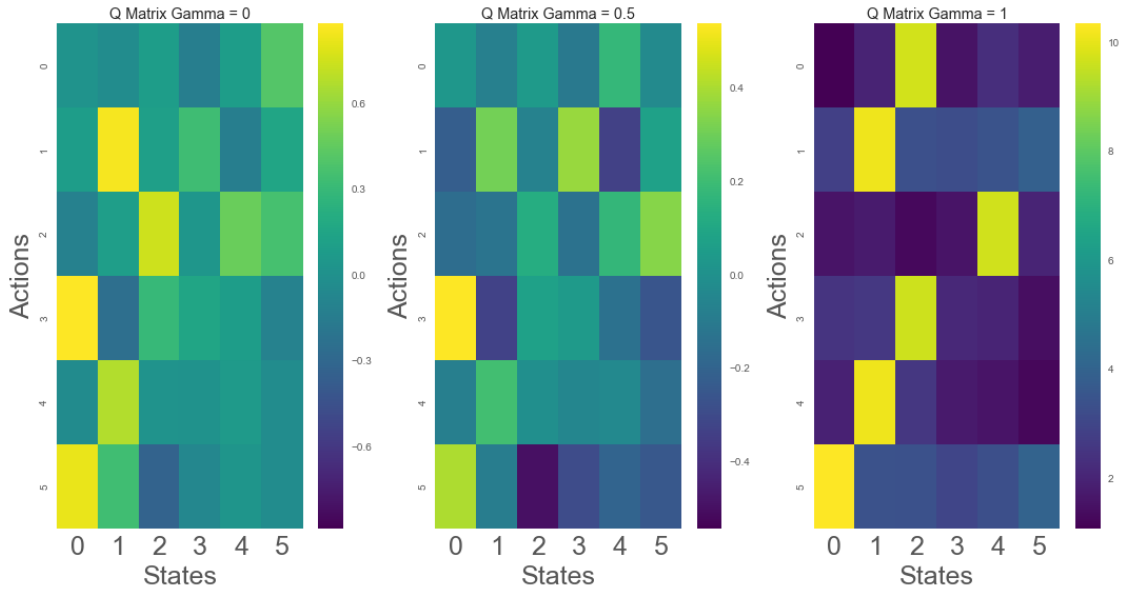


3.4 Comparison of Q Learning with Gamma values [0, 0.5 ,1]

```
In [27]: plt.figure(figsize=(15,6))
         plt.plot(ql_1.mean_discrepancy, label='Mean Discrepancy for Gamma = 0', linewidth =5)
         plt.plot(ql_2.mean_discrepancy, label='Mean Discrepancy for Gamma = 0.5', linewidth =5)
         plt.plot(ql_3.mean_discrepancy, label='Mean Discrepancy for Gamma = 1', linewidth =5)
         plt.title('Plot of Mean Discrepancy values averaged over 100 iterations for different Gamma va
                   fontsize=20)
         plt.xlabel("No. of Iterations", fontsize=15)
         plt.ylabel("Mean Discrepancy in Value Function", fontsize=15)
         plt.legend(fontsize=15)
         plt.tight_layout()
```



In [63]:
```
plt.figure(figsize=(15,8))
plt.subplot(1,3,1)
sns.heatmap(ql_1.Q, cmap=plt.cm.viridis)
plt.title("Q Matrix Gamma = 0")
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=10)

plt.subplot(1,3,2)
sns.heatmap(ql_2.Q, cmap=plt.cm.viridis)
plt.title("Q Matrix Gamma = 0.5")
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=10)

plt.subplot(1,3,3)
sns.heatmap(ql_3.Q, cmap=plt.cm.viridis)
plt.title("Q Matrix Gamma = 1")
plt.xlabel("States", fontsize=25)
plt.ylabel('Actions', fontsize=25)
plt.xticks(fontsize=25)
plt.yticks(fontsize=10)

plt.tight_layout()
```



It appears Q with Gamma = 0.5 gives the best results

4 Q Learning with different state and reward functions

We expand the scope of the problem where the agent now tries to find its way around a (10,10,10) grid. The agent has graduated to the real world since it is still a probabilistic situation we assume to have fullfilled the differnt state and reward function criterion from the previous phase

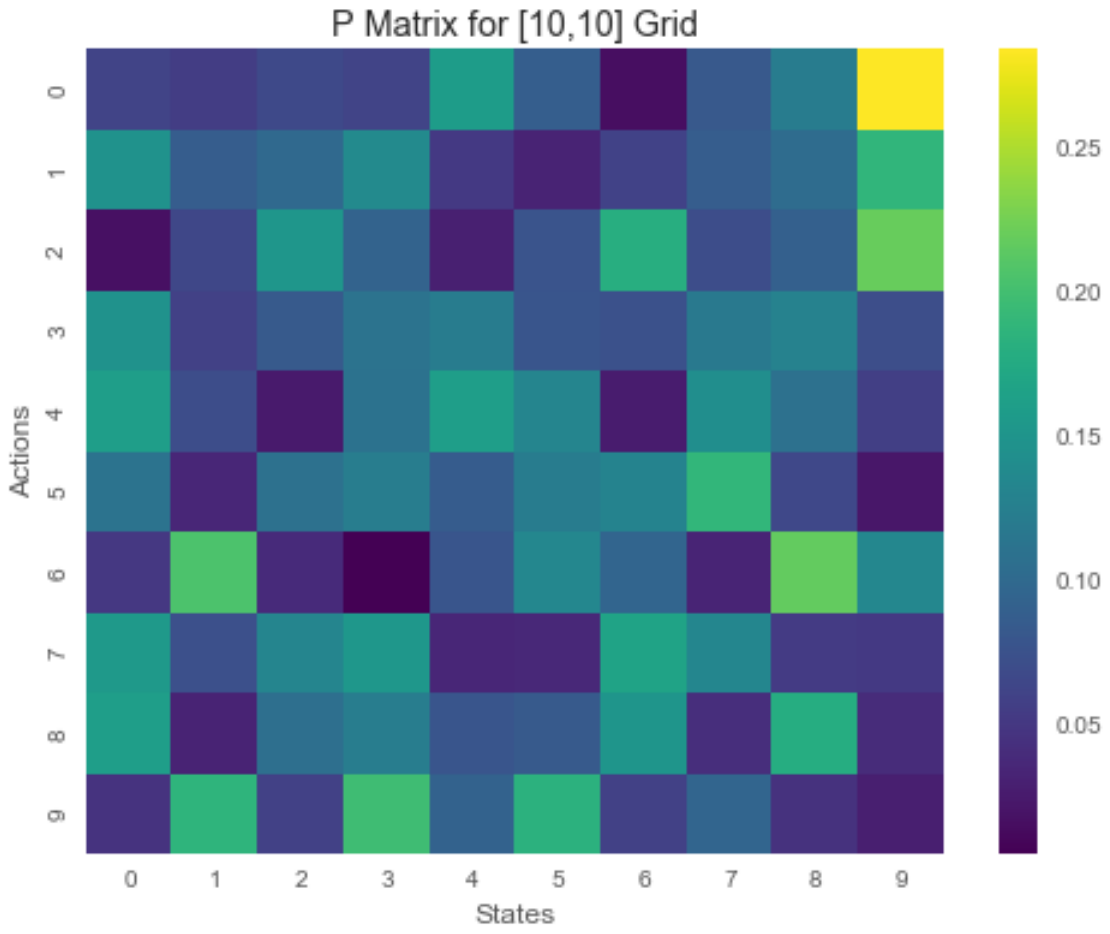
```
In [29]: P2, R2 = mdptoolbox.example.rand(10,10, is_sparse=False)
```

```
In [30]: P2[1:2,]
```

```
Out[30]: array([[[ 0.47638633,  0.        ,  0.        ,  0.        ,  0.        ,
                   0.4786749 ,  0.        ,  0.        ,  0.        ,  0.04493877],
                 [ 0.02128449,  0.08057477,  0.11273276,  0.06205292,  0.        ,
                   0.09440775,  0.23739067,  0.00110388,  0.25034947,  0.14010328],
                 [ 0.02480776,  0.02786922,  0.09032595,  0.24558928,  0.01430236,
                   0.15815549,  0.24781808,  0.19113185,  0.        ,  0.        ],
                 [ 0.09007138,  0.02072792,  0.07628214,  0.16840256,  0.02956166,
                   0.23542671,  0.        ,  0.        ,  0.15210942,  0.2274182 ],
                 [ 0.        ,  0.        ,  0.        ,  0.        ,  1.        ,
                   0.        ,  0.        ,  0.        ,  0.        ,  0.        ],
                 [ 0.        ,  0.        ,  0.05882986,  0.12044782,  0.15320672,
                   0.02422786,  0.15458296,  0.30484383,  0.        ,  0.18386096],
                 [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
                   0.        ,  0.26435534,  0.        ,  0.73564466,  0.        ],
                 [ 0.        ,  0.        ,  0.        ,  1.        ,  0.        ,
                   0.        ,  0.        ,  0.        ,  0.        ,  0.        ],
                 [ 0.        ,  0.        ,  0.        ,  0.32607184,  0.        ,
                   0.29580749,  0.37812068,  0.        ,  0.        ,  0.        ],
                 [ 0.11985248,  0.18251823,  0.        ,  0.03556512,  0.        ,
                   0.36339275,  0.2512804 ,  0.04739102,  0.        ,  0.        ]]])
```

```
In [87]: P2_x = np.sum(P2, axis=0)/P2.shape[0]
         plt.figure(figsize=(8,6))
         ax = sns.heatmap(P2_x, cmap=plt.cm.viridis)
         ax.set(xlabel ="States", ylabel="Actions", title="P Matrix for [10,10] Grid")
```

```
Out[87]: [<matplotlib.text.Text at 0x183d6c0d68>,
          <matplotlib.text.Text at 0x183d6ba9e8>,
          <matplotlib.text.Text at 0x183d560860>]
```

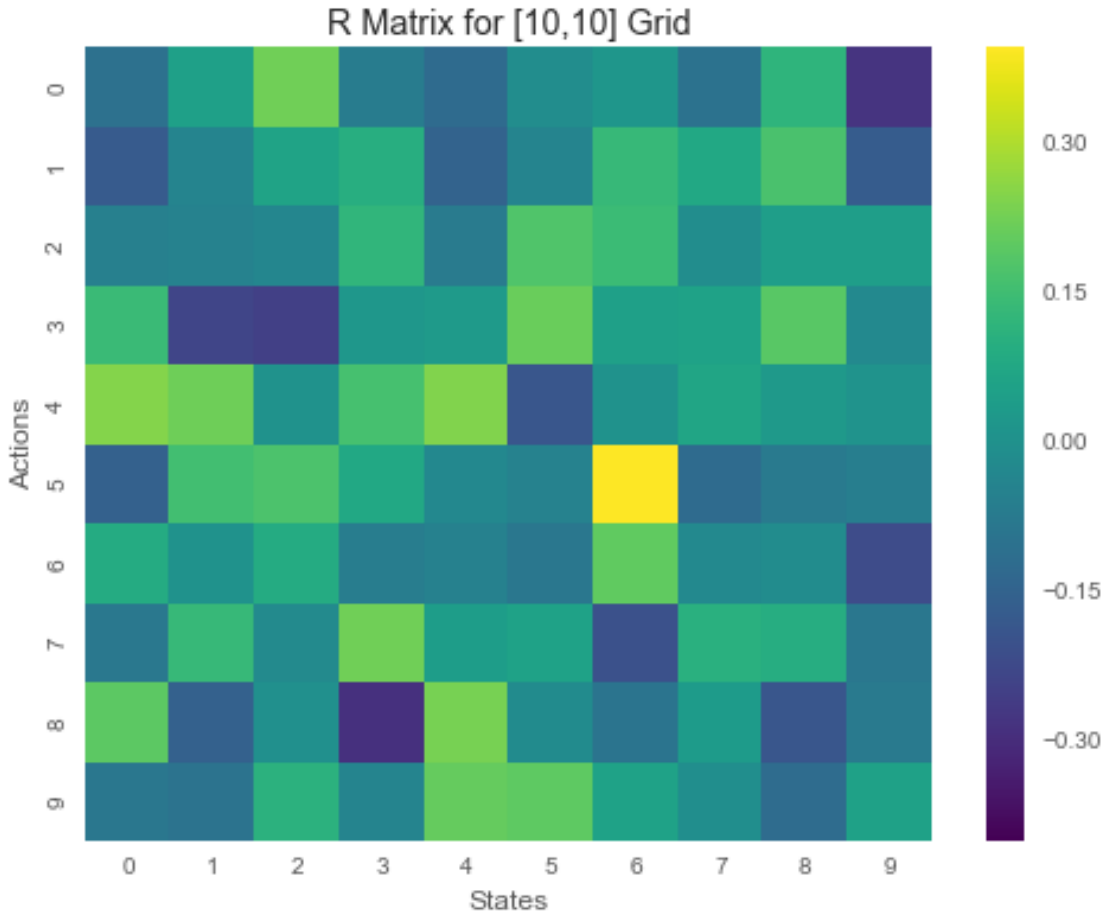



```
In [31]: R2[1:2,]
```

```
Out[31]: array([[[ 0.08852718,  0.        ,  0.        ,  0.        ,  0.        ,
                   -0.6724465 ,  0.        , -0.        , -0.        , -0.4449276 ],
                  [-0.97499635, -0.45640552,  0.65612984,  0.22572188,  0.        ,
                    0.13983976,  0.15654991, -0.75820366, -0.35675485, -0.5483869 ],
                  [ 0.08771878, -0.09092874, -0.2381622 , -0.0960959 ,  0.92957956,
                    0.79738878,  0.08843746, -0.09917354, -0.        , -0.        ],
                  [ 0.46929671, -0.83019184, -0.76611705, -0.95488333, -0.67524102,
                    0.70972695, -0.        , -0.        ,  0.41396572,  0.53484852],
                  [ 0.        ,  0.        , -0.        , -0.        , -0.01678224,
                   -0.        , -0.        ,  0.        , -0.        , -0.        ],
                  [-0.        , -0.        ,  0.76214798, -0.40819893, -0.06120668,
                   -0.31257548,  0.71658568,  0.05378245,  0.        ,  0.10429216],
                  [-0.        , -0.        , -0.        , -0.        , -0.        ,
                   -0.        ,  0.55195389,  0.        , -0.38649619, -0.        ],
                  [-0.        , -0.        ,  0.        ,  0.88401461,  0.        ,
                   -0.        ,  0.        , -0.        , -0.        ,  0.        ],
                  [ 0.        , -0.        ,  0.        , -0.4699761 ,  0.        ,
                   -0.8978813 , -0.98575318, -0.        ,  0.        ,  0.        ],
                  [-0.59791281, -0.28301558,  0.        , -0.93090613, -0.        ,
                    0.69167486,  0.5956771 , -0.46268113, -0.        , -0.        ]]])
```

```
In [89]: R2_x = np.sum(R2, axis=0)/R2.shape[0]
         plt.figure(figsize=(8,6))
         ax = sns.heatmap(R2_x, cmap=plt.cm.viridis)
         ax.set(xlabel ="States", ylabel="Actions", title="R Matrix for [10,10] Grid")
```

```
Out[89]: [<matplotlib.text.Text at 0x183d839be0>,
          <matplotlib.text.Text at 0x183d5eab70>,
          <matplotlib.text.Text at 0x183d6d25f8>]
```




```
In [91]: ql_4 = mdp.mdp.QLearning(P2,R2,0.5)
         print("Initial Values")
         print("States \n", ql_4.S)
         print("Actions \n", ql_4.A)
         print("Q Matrix \n",ql_4.Q )
         print("Rewards \n", ql_4.R[1:2,])
         ql_4.run()
```

```
Initial Values
States
 10
Actions
 10
```

```
Q Matrix
 [[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
Rewards
 [[[ 0.08852718  0.          0.          0.          0.         -0.6724465
    0.         -0.         -0.         -0.4449276 ]
  [-0.97499635 -0.45640552  0.65612984  0.22572188  0.          0.13983976
    0.15654991 -0.75820366 -0.35675485 -0.5483869 ]
  [ 0.08771878 -0.09092874 -0.2381622  -0.0960959   0.92957956  0.79738878
    0.08843746 -0.09917354 -0.         -0.        ]
  [ 0.46929671 -0.83019184 -0.76611705 -0.95488333 -0.67524102  0.70972695
   -0.         -0.          0.41396572  0.53484852]
  [ 0.          0.         -0.         -0.         -0.01678224 -0.         -0.
    0.         -0.         -0.        ]
  [-0.         -0.          0.76214798 -0.40819893 -0.06120668 -0.31257548
    0.71658568  0.05378245  0.          0.10429216]
  [-0.         -0.         -0.         -0.         -0.         -0.
    0.55195389  0.         -0.38649619 -0.        ]
  [-0.         -0.          0.          0.88401461  0.         -0.          0.
   -0.         -0.          0.        ]
  [ 0.         -0.          0.         -0.4699761   0.         -0.8978813
   -0.98575318 -0.          0.          0.        ]
  [-0.59791281 -0.28301558  0.         -0.93090613 -0.          0.69167486
    0.5956771  -0.46268113 -0.         -0.        ]]]
```

In [33]:
```python
import tabulate
from tabulate import tabulate as tb
print("Q Matrix \n",tb(ql_4.Q, tablefmt="simple", numalign="left", floatfmt=".4f"))
```

Q Matrix

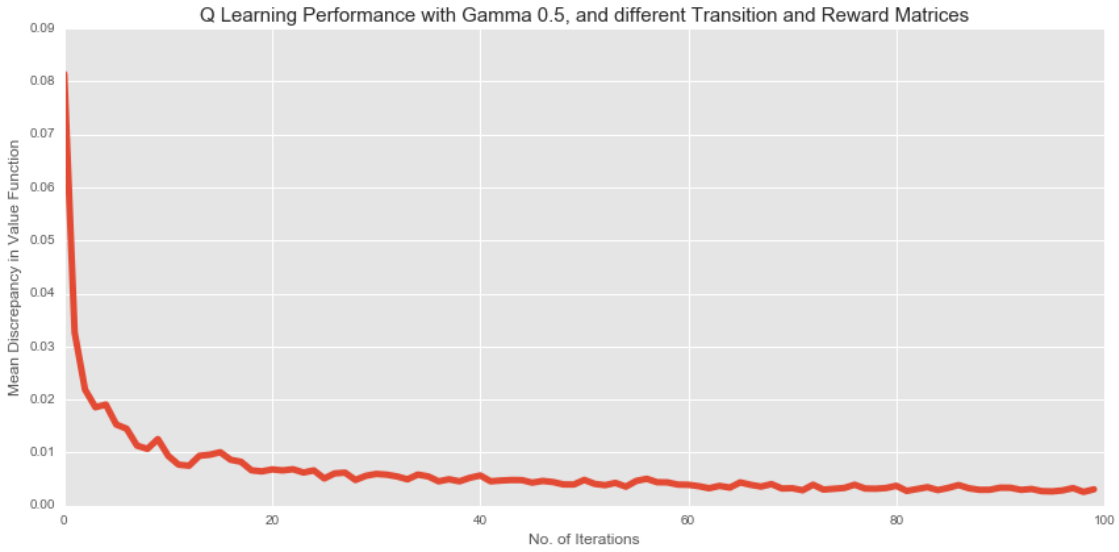
1.1294	-0.0168	0.1122	-0.0370	0.1651	-0.0782	-0.0635	0.2046	-0.2307	0.1357
0.1544	-0.0062	0.0214	0.0805	0.0160	0.0343	0.8512	0.1388	0.0208	0.0338
-0.1391	0.6187	0.0739	0.0556	-0.0631	0.1972	0.0295	-0.0196	0.1702	0.0034
0.7268	0.2559	0.2226	0.6180	0.1109	0.0615	0.2680	0.2609	0.0892	0.4497
0.0000	0.0095	0.0237	0.0493	0.0000	0.6340	0.0018	0.1571	0.0035	0.0627
0.8445	0.1200	0.1790	0.0984	0.0716	0.1542	0.0335	0.0424	0.1680	0.0794
1.3083	-0.0111	0.0108	0.1064	0.0089	0.0522	0.1320	0.0442	-0.1249	0.0511
0.2548	1.2668	-0.3058	0.4330	0.2494	-0.0903	0.1909	-0.0812	0.0333	0.0783
-0.0065	-0.0601	0.0998	0.1136	0.0010	0.5977	0.1142	0.2091	0.0660	-0.0739
0.1972	0.1023	-0.0133	0.0052	0.0009	0.0420	1.0688	0.0000	0.0000	-0.0181

In [34]:
```python
print('After Q Learing with Gamma %s' %ql_1.discount)
print("Actions :", ql_4.A)
print("Rewards \n",  ql_4.R[1:2,])
print("Optimal Policy :", ql_4.policy)
```

```
        print("Time taken :%g"%ql_4.time)
        print("Total Iterations :", ql_4.max_iter)
```

```
After Q Learing with Gamma 0.5
Actions : 10
Rewards
 [[[ 0.08852718  0.          0.          0.          0.         -0.6724465
    0.         -0.         -0.         -0.4449276 ]
  [-0.97499635 -0.45640552  0.65612984  0.22572188  0.          0.13983976
    0.15654991 -0.75820366 -0.35675485 -0.5483869 ]
  [ 0.08771878 -0.09092874 -0.2381622  -0.0960959   0.92957956  0.79738878
    0.08843746 -0.09917354 -0.         -0.        ]
  [ 0.46929671 -0.83019184 -0.76611705 -0.95488333 -0.67524102  0.70972695
   -0.         -0.          0.41396572  0.53484852]
  [ 0.          0.         -0.         -0.         -0.01678224 -0.         -0.
    0.         -0.         -0.        ]
  [-0.         -0.          0.76214798 -0.40819893 -0.06120668 -0.31257548
    0.71658568  0.05378245  0.          0.10429216]
  [-0.         -0.         -0.         -0.         -0.         -0.
    0.55195389  0.         -0.38649619 -0.        ]
  [-0.         -0.          0.          0.88401461  0.         -0.          0.
   -0.         -0.          0.        ]
  [ 0.         -0.          0.         -0.4699761   0.         -0.8978813
   -0.98575318 -0.          0.          0.        ]
  [-0.59791281 -0.28301558  0.         -0.93090613 -0.          0.69167486
    0.5956771  -0.46268113 -0.         -0.        ]]]
Optimal Policy : (0, 6, 1, 0, 5, 0, 0, 1, 5, 6)
Time taken :0.64066
Total Iterations : 10000
```

In [35]:
```
        plt.figure(figsize=(12,6))
        plt.plot(ql_4.mean_discrepancy, linewidth = 5)
        plt.xlabel("No. of Iterations")
        plt.ylabel("Mean Discrepancy in Value Function")
        plt.title('Q Learning Performance with Gamma %s, and different Transition and Reward Matrices'
                  fontsize = 15)
        plt.tight_layout()
```



4.1 With 20,000 Iterations

```
In [36]: ql_5 = mdp.mdp.QLearning(P2,R2,0.5,  n_iter=20000)
         print("Initial Values")
         print("States \n", ql_5.S)
         print("Actions \n", ql_5.A)
         print("Q Matrix \n",ql_5.Q )
         print("Rewards \n", ql_5.R[1:2,])
         ql_5.run()
```

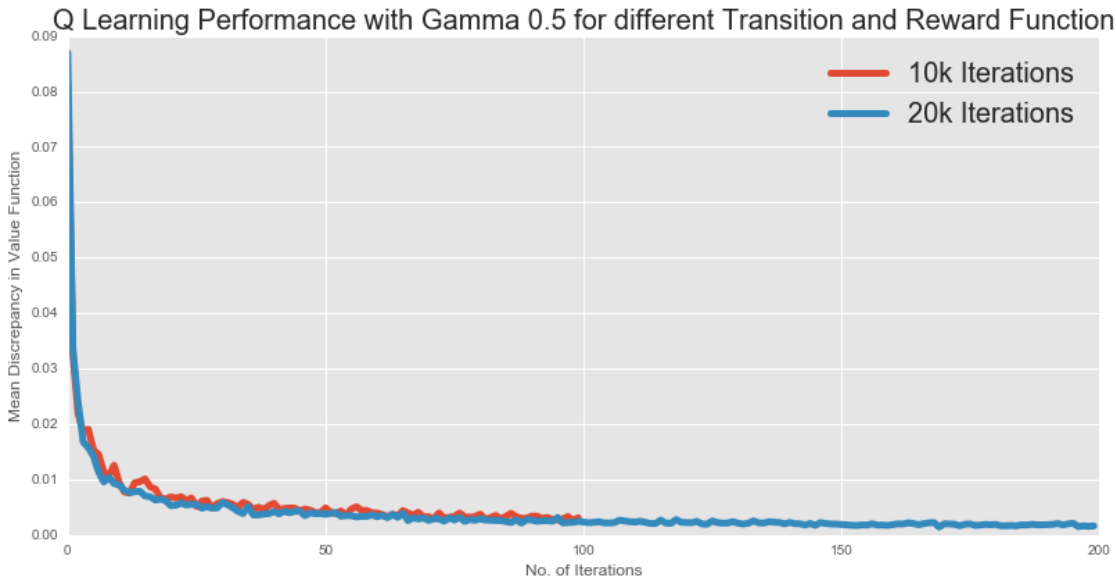
```
Initial Values
States
 10
Actions
 10
Q Matrix
 [[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
Rewards
 [[[ 0.08852718  0.          0.          0.          0.         -0.6724465
    0.         -0.         -0.         -0.4449276 ]
  [-0.97499635 -0.45640552  0.65612984  0.22572188  0.          0.13983976
    0.15654991 -0.75820366 -0.35675485 -0.5483869 ]
  [ 0.08771878 -0.09092874 -0.2381622  -0.0960959   0.92957956  0.79738878
    0.08843746 -0.09917354 -0.         -0.        ]
  [ 0.46929671 -0.83019184 -0.76611705 -0.95488333 -0.67524102  0.70972695
   -0.         -0.          0.41396572  0.53484852]
  [ 0.          0.         -0.         -0.         -0.01678224 -0.         -0.
    0.         -0.         -0.        ]
  [-0.         -0.          0.76214798 -0.40819893 -0.06120668 -0.31257548
    0.71658568  0.05378245  0.          0.10429216]
  [-0.         -0.         -0.         -0.         -0.         -0.
    0.55195389  0.         -0.38649619 -0.        ]
  [-0.         -0.          0.          0.88401461  0.         -0.          0.
   -0.         -0.          0.        ]
  [ 0.         -0.          0.         -0.4699761   0.         -0.8978813
   -0.98575318 -0.          0.          0.        ]
  [-0.59791281 -0.28301558  0.         -0.93090613 -0.          0.69167486
    0.5956771  -0.46268113 -0.         -0.        ]]]
```

```
In [37]: print('After Q Learing with Gamma %s' %ql_5.discount)
         print("Actions :", ql_5.A)
         print("Q Matrix \n",tb(ql_5.Q, tablefmt="plain", numalign="left", floatfmt=".4f"))
         print("Rewards \n", ql_5.R[1:2,])
```

```
        print("Optimal Policy :", ql_5.policy)
        print("Time taken :%g"%ql_5.time)
        print("Total Iterations :", ql_5.max_iter)
```

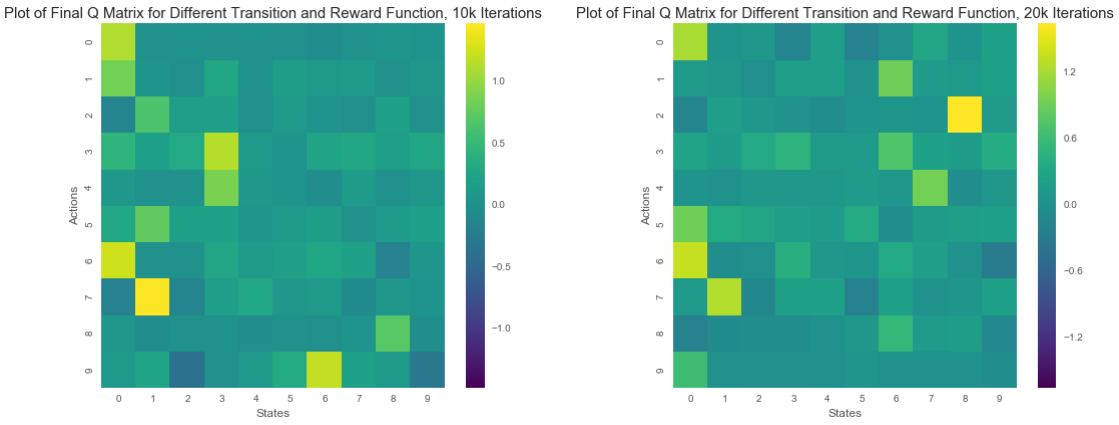
```
After Q Learing with Gamma 0.5
Actions : 10
Q Matrix
 1.2166   0.0536   0.1081   -0.1444  0.1894   -0.2022  -0.0080  0.2881  0.0549   0.2234
0.1254   0.0990   -0.0152  0.1908   0.1881   0.0383   0.9297   0.1471  0.1358   0.2353
-0.1419  0.1934   0.1153   0.0426   -0.0438  0.0589   0.0883   0.0593  1.6533   0.1233
0.2609   0.1672   0.3415   0.4863   0.1452   0.1460   0.7403   0.2447  0.1658   0.3759
0.0582   0.0352   0.1037   0.1024   0.1059   0.1533   0.1085   0.9512  -0.0456  0.1158
0.9252   0.3491   0.2879   0.1870   0.1448   0.3571   -0.0402  0.1559  0.1940   0.2369
1.3657   -0.0378  -0.0110  0.4322   0.0944   0.0824   0.3390   0.1907  0.0006   -0.2900
0.1302   1.2734   -0.1080  0.2335   0.2132   -0.1927  0.2157   0.0253  0.0843   0.2186
-0.1810  -0.0684  -0.0271  -0.0391  0.0111   0.1079   0.5546   0.1479  0.1706   -0.0933
0.6302   0.0271   0.0015   0.0111   0.0200   0.0759   0.0233   0.0193  0.0297   -0.0306
Rewards
 [[[ 0.08852718  0.          0.          0.          0.         -0.6724465
    0.         -0.         -0.         -0.4449276 ]
  [-0.97499635 -0.45640552  0.65612984  0.22572188  0.          0.13983976
    0.15654991 -0.75820366 -0.35675485 -0.5483869 ]
  [ 0.08771878 -0.09092874 -0.2381622  -0.0960959   0.92957956  0.79738878
    0.08843746 -0.09917354 -0.         -0.        ]
  [ 0.46929671 -0.83019184 -0.76611705 -0.95488333 -0.67524102  0.70972695
   -0.         -0.          0.41396572  0.53484852]
  [ 0.          0.         -0.         -0.         -0.01678224 -0.         -0.
    0.         -0.         -0.        ]
  [-0.         -0.          0.76214798 -0.40819893 -0.06120668 -0.31257548
    0.71658568  0.05378245  0.          0.10429216]
  [-0.         -0.         -0.         -0.         -0.         -0.
    0.55195389  0.         -0.38649619 -0.        ]
  [-0.         -0.          0.          0.88401461  0.         -0.          0.
   -0.         -0.          0.        ]
  [ 0.         -0.          0.         -0.4699761   0.         -0.8978813
   -0.98575318 -0.          0.          0.        ]
  [-0.59791281 -0.28301558  0.         -0.93090613 -0.          0.69167486
    0.5956771  -0.46268113 -0.         -0.        ]]]
Optimal Policy : (0, 6, 8, 6, 7, 0, 0, 1, 6, 0)
Time taken :1.28131
Total Iterations : 20000
```

In [38]:
```
        plt.figure(figsize=(12,6))
        plt.plot(ql_4.mean_discrepancy, label='10k Iterations', linewidth = 5)
        plt.plot(ql_5.mean_discrepancy, label='20k Iterations', linewidth = 5)
        plt.xlabel("No. of Iterations")
        plt.ylabel("Mean Discrepancy in Value Function")
        plt.title('Q Learning Performance with Gamma %s for different Transition and Reward Function' %
                  fontsize=20)
        plt.legend(fontsize=20)
        plt.tight_layout()
```



The Mean Discrepancy is calculated as Max_Iterations/100 as noted previously so the second case has extra 100 points

```
In [93]: plt.figure(figsize=(16,6))

         plt.subplot(1,2,1)
         plt.title("Plot of Final Q Matrix for Different Transition and Reward Function, 10k Iterations
         ax =sns.heatmap(ql_4.Q, cmap=plt.cm.viridis)
         ax.set(xlabel ="States", ylabel="Actions")

         plt.subplot(1,2,2)
         plt.title("Plot of Final Q Matrix for Different Transition and Reward Function, 20k Iterations
         ax1 = sns.heatmap(ql_5.Q, cmap=plt.cm.viridis)
         ax1.set(xlabel ="States", ylabel="Actions")
         plt.tight_layout()
```



5 100 x100 Grid - navigating a more 'realistic' world

```
In [40]: P4, R4 = mdp.example.rand(100,100)
```

```
In [41]: print(P4.shape[0])
         print(P4.shape[1])
```

```
100
100
```

```
In [42]: print(R4.shape[0])
         print(R4.shape[1])
```

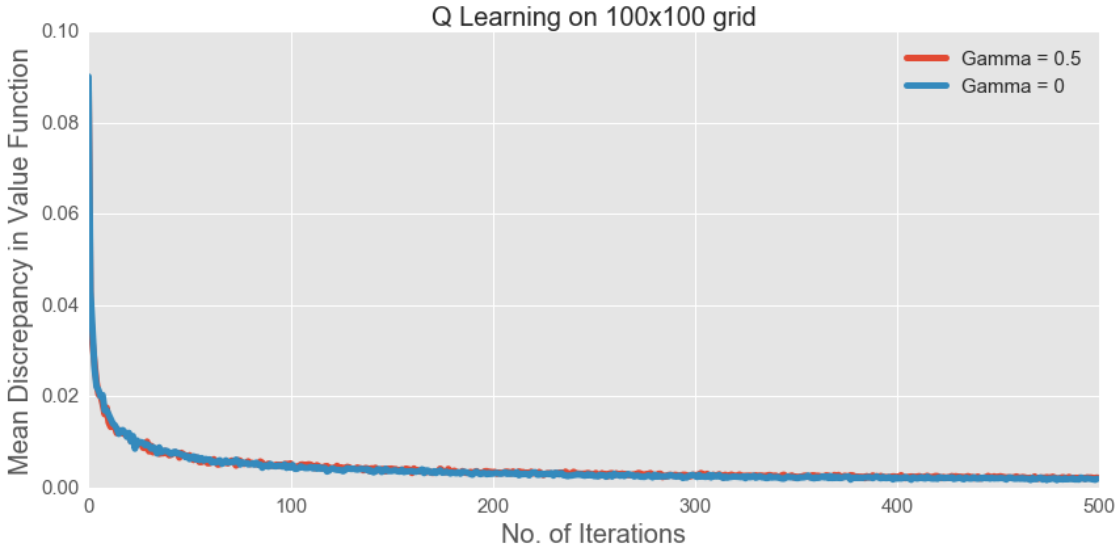
```
100
100
```

We noted that the Gamma values of 0 and 0.5 gave a stable performance so we use these two values to test the Q Learning on the larger grid

```
In [43]: ql_6 = mdp.mdp.QLearning(P4,R4,0.5, n_iter=50000)
         ql_6.run()
         ql_7 = mdp.mdp.QLearning(P4,R4,0, n_iter=50000)
         ql_7.run()
```

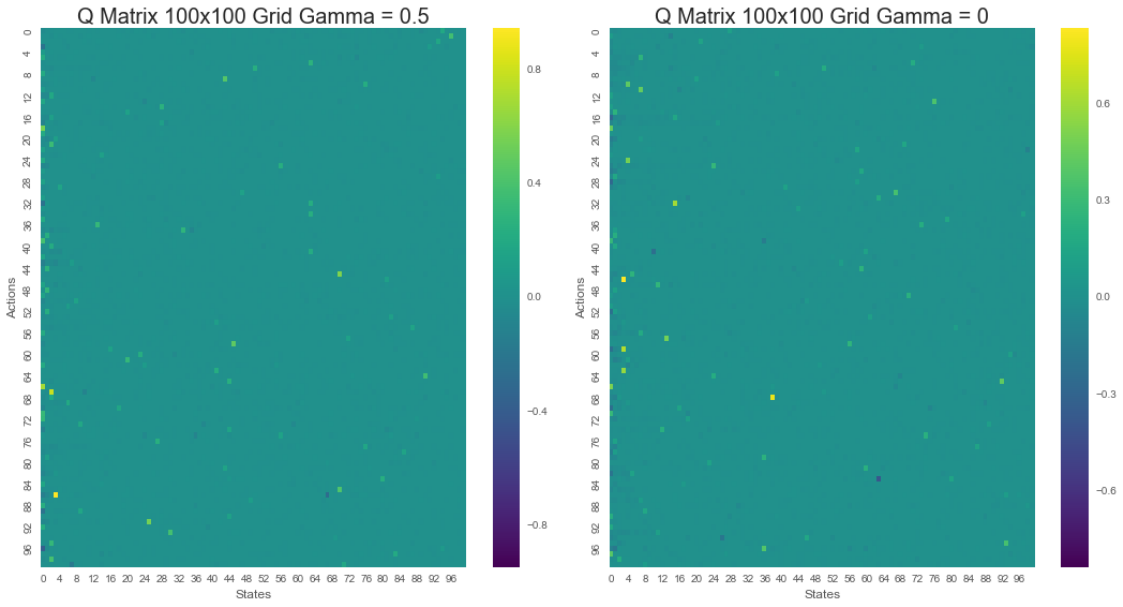
5.1 Performance Comparison

```
In [44]: plt.figure(figsize=(12,6))
         plt.plot(ql_6.mean_discrepancy, linewidth =5, label='Gamma = 0.5')
         plt.plot(ql_7.mean_discrepancy, linewidth =5, label='Gamma = 0')
         plt.xlabel("No. of Iterations", fontsize=20)
         plt.ylabel("Mean Discrepancy in Value Function", fontsize=20)
         plt.title('Q Learning on 100x100 grid', fontsize=20)
         plt.xticks(fontsize=15)
         plt.yticks(fontsize=15)
         plt.legend(fontsize=15)
         plt.tight_layout()
```

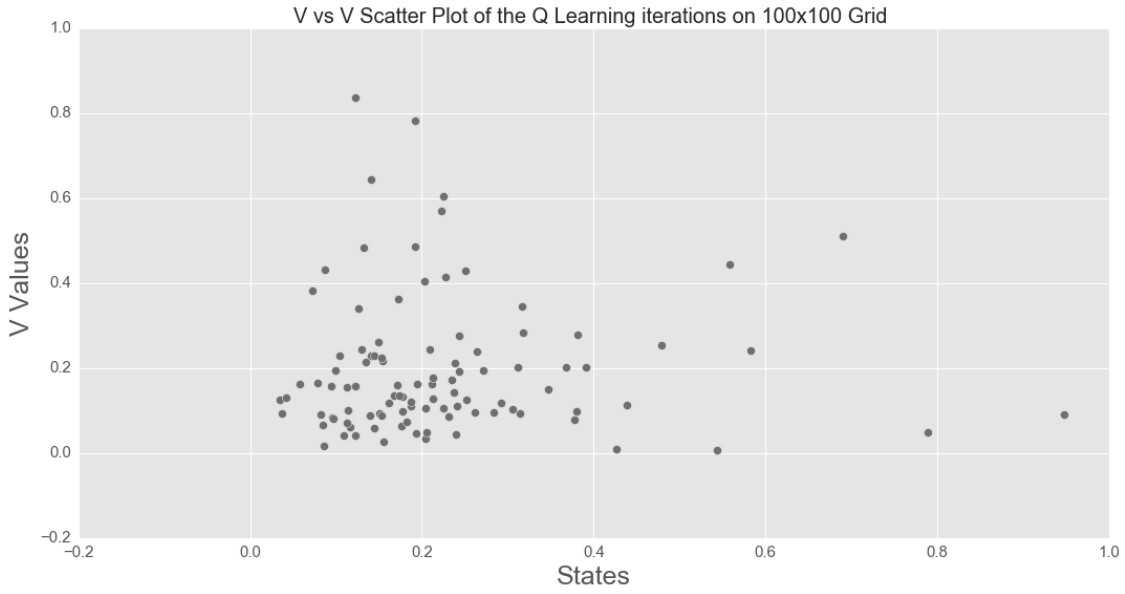


5.2 Q Matrix Comparison

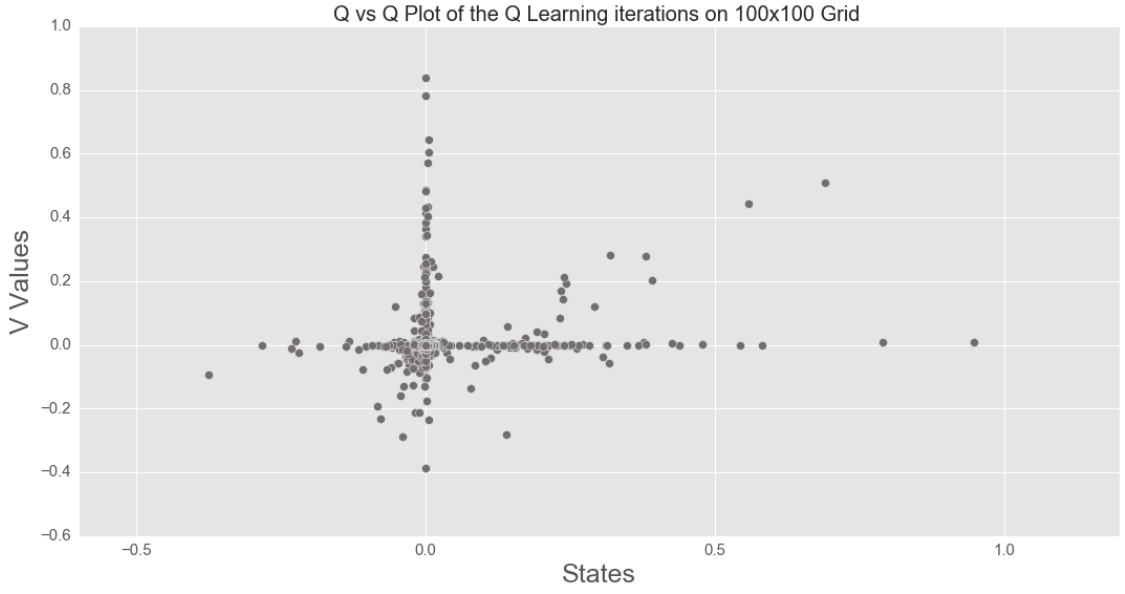
In [95]:
```
plt.figure(figsize=(15,8))

plt.subplot(1,2,1)
plt.title("Q Matrix 100x100 Grid Gamma = 0.5", fontsize = 20)
ax =sns.heatmap(ql_6.Q, cmap=plt.cm.viridis, xticklabels=4, yticklabels=4)
ax.set(xlabel ="States", ylabel="Actions")

plt.subplot(1,2,2)
plt.title("Q Matrix 100x100 Grid Gamma = 0", fontsize=20)
ax1= sns.heatmap(ql_7.Q, cmap=plt.cm.viridis, xticklabels=4, yticklabels=4)
ax1.set(xlabel ="States", ylabel="Actions")
plt.tight_layout()
```



In [96]:
```
plt.figure(figsize=(15,8))
plt.title("V vs V Scatter Plot of the Q Learning iterations on 100x100 Grid ", fontsize=20)
plt.scatter(ql_6.V, ql_7.V,c='#726E6D', s=65)
plt.ylabel("V Values", fontsize=25)
plt.xlabel('States', fontsize=25)
plt.xticks( fontsize=15)
plt.yticks( fontsize=15)
plt.tight_layout()
```



In [97]:
```
plt.figure(figsize=(15,8))
plt.title("Q vs Q Plot of the Q Learning iterations on 100x100 Grid ", fontsize=20)
plt.scatter(ql_6.Q, ql_7.Q, c='#726E6D',s=65)
plt.ylabel("V Values", fontsize=25)
plt.xlabel('States', fontsize=25)
plt.xticks( fontsize=15)
plt.yticks( fontsize=15)
plt.tight_layout()
```



6 Comparisons

In this section we look at some statistics from the different Q Learning iterations that we have performed.

6.1 How do the Optimal Policies compare between the different runs of Q Learning?

```
In [48]: import pandas as pd
```

```
In [49]: ql_ix = ["QL G=0.5",
                  "QL G=0",
                  "QL G=1",
                  "QL Diff S,R,10K",
                  "QL Diff S,R,20K",
                  "QL Big Grid G=0.5",
                  "QL Big Grid G=0"
              ]
```

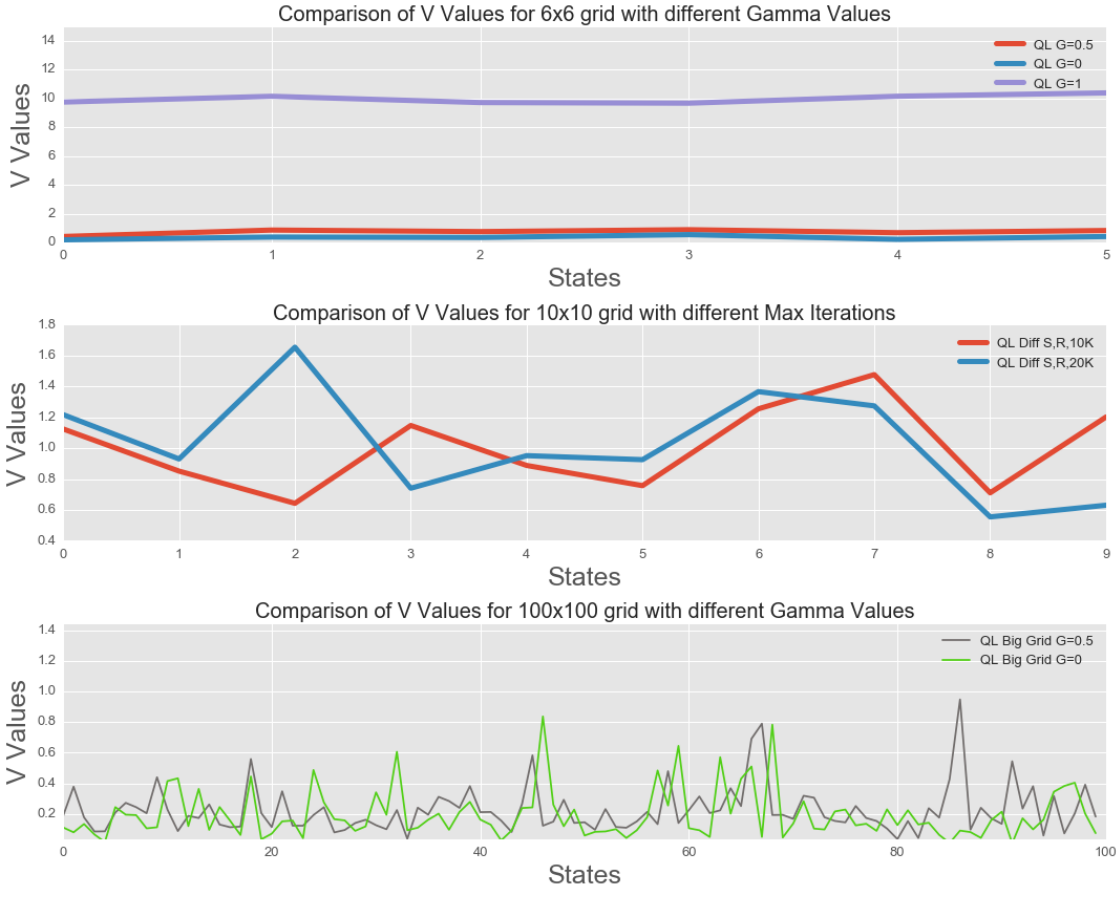
```
In [98]: plt.figure(figsize=(15,12))

         #first grid 6,6
         plt.subplot(311)
         plt.title('Comparison of Optimal Policies for 6x6 grid with different Gamma values',fontsize=2
         plt.plot(ql_1.policy, label=ql_ix[0], linewidth=5)
         plt.plot(ql_2.policy, label=ql_ix[1], linewidth=5)
         plt.plot(ql_3.policy, label=ql_ix[2], linewidth=5)
         plt.legend(fontsize=13)
         plt.xticks(fontsize=13)
         plt.yticks(fontsize=13)
         plt.ylabel("Actions", fontsize=25)
         plt.xlabel('States', fontsize=25)

         #second grid 10,10
         plt.subplot(312)
         plt.title('Comparison of Optimal Policies for 10x10 gird with different Max Iterations',fontsi
         plt.plot(ql_4.policy, label=ql_ix[3], linewidth=5)
         plt.plot(ql_5.policy, label=ql_ix[4], linewidth=5)
         plt.legend(fontsize=13)
         plt.xticks(fontsize=13)
         plt.yticks(fontsize=13)
         plt.ylim(np.min(ql_5.policy),np.max(ql_5.policy)+3)
         plt.ylabel("Actions", fontsize=25)
         plt.xlabel('States', fontsize=25)

         #final big grid 100,100
         plt.subplot(313)
         plt.title("Comparison of Optimal Policies for 100x100 grid with different Gamma Values", fontsi
         plt.plot(ql_6.policy, '#726E6D', label=ql_ix[5])
         plt.plot(ql_7.policy, '#52D017', label=ql_ix[6])
         plt.legend(fontsize=13)
         plt.xticks(fontsize=13)
         plt.yticks(fontsize=13)
         plt.ylim(np.min(ql_6.policy),np.max(ql_6.policy)+30)
         plt.ylabel("Actions", fontsize=25)
```

```
plt.xlabel('States', fontsize=25)
plt.tight_layout()
```

6.2 How do the V Values compare between the different runs?

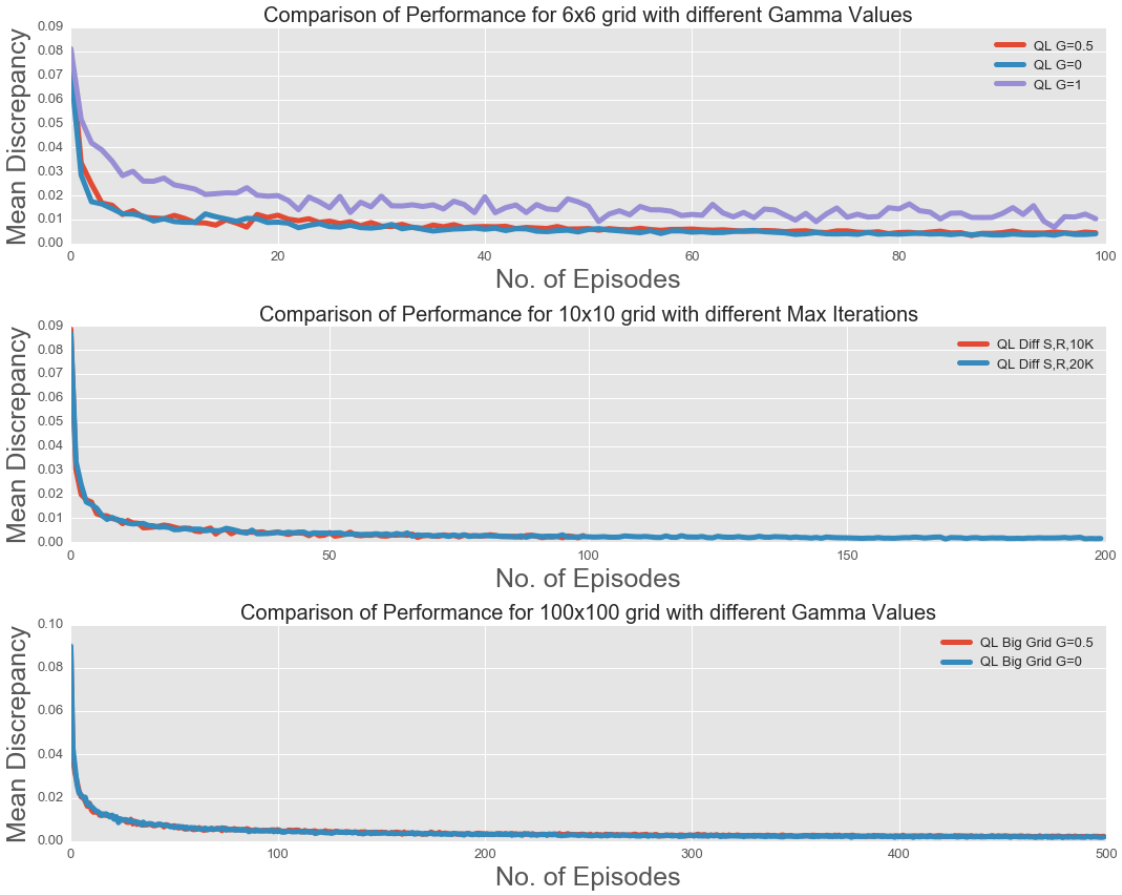
In [99]:
```
plt.figure(figsize=(15,12))

#first grid 6,6
plt.subplot(311)
plt.title('Comparison of V Values for 6x6 grid with different Gamma Values',fontsize=20)
plt.plot(ql_1.V, label=ql_ix[0], linewidth=5)
plt.plot(ql_2.V, label=ql_ix[1], linewidth=5)
plt.plot(ql_3.V, label=ql_ix[2], linewidth=5)
plt.legend(fontsize=13)
plt.xticks(fontsize=13)
plt.ylim(0,15)
plt.yticks(fontsize=13)
plt.ylabel("V Values", fontsize=25)
plt.xlabel('States', fontsize=25)

#second grid 10,10
plt.subplot(312)
```

```
plt.title('Comparison of V Values for 10x10 grid with different Max Iterations',fontsize=20)
plt.plot(ql_4.V, label=ql_ix[3], linewidth=5)
plt.plot(ql_5.V, label=ql_ix[4], linewidth=5)
plt.legend(fontsize=13)
plt.xticks(fontsize=13)
plt.yticks(fontsize=13)
plt.ylabel("V Values", fontsize=25)
plt.xlabel('States', fontsize=25)

#final big grid 100,100
plt.subplot(313)
plt.title("Comparison of V Values for 100x100 grid with different Gamma Values", fontsize=20)
plt.plot(ql_6.V, '#726E6D', label=ql_ix[5])
plt.plot(ql_7.V, '#52D017', label=ql_ix[6])
plt.legend(fontsize=13)
plt.xticks(fontsize=13)
plt.yticks(fontsize=13)
plt.ylim(np.min(ql_6.V),np.max(ql_6.V)+0.5)
plt.ylabel("V Values", fontsize=25)
plt.xlabel('States', fontsize=25)

plt.tight_layout()
```



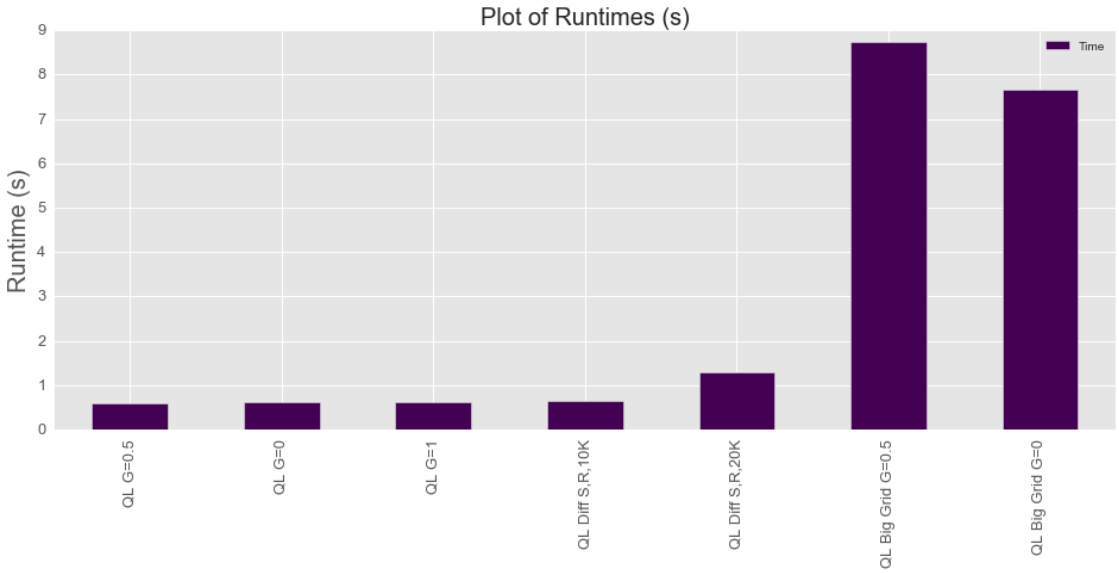
6.3 How does the performance compare for the different iterations?

In [100]:
```
plt.figure(figsize=(15,12))

#first grid 6,6
plt.subplot(311)
plt.title('Comparison of Performance for 6x6 grid with different Gamma Values',fontsize=20)
plt.plot(ql_1.mean_discrepancy, label=ql_ix[0], linewidth=5)
plt.plot(ql_2.mean_discrepancy, label=ql_ix[1], linewidth=5)
plt.plot(ql_3.mean_discrepancy, label=ql_ix[2], linewidth=5)
plt.legend(fontsize=13)
plt.xticks(fontsize=13)
plt.yticks(fontsize=13)
plt.ylabel("Mean Discrepancy", fontsize=25)
plt.xlabel('No. of Episodes', fontsize=25)

#second grid 10,10
plt.subplot(312)
plt.title('Comparison of Performance for 10x10 grid with different Max Iterations',fontsize=2
plt.plot(ql_4.mean_discrepancy, label=ql_ix[3], linewidth=5)
plt.plot(ql_5.mean_discrepancy, label=ql_ix[4], linewidth=5)
plt.legend(fontsize=13)
plt.xticks(fontsize=13)
plt.yticks(fontsize=13)
plt.ylabel("Mean Discrepancy", fontsize=25)
plt.xlabel('No. of Episodes', fontsize=25)

#final big grid 100,100
plt.subplot(313)
plt.title("Comparison of Performance for 100x100 grid with different Gamma Values", fontsize=
plt.plot(ql_6.mean_discrepancy, label=ql_ix[5], linewidth=5)
plt.plot(ql_7.mean_discrepancy, label=ql_ix[6], linewidth=5)
plt.legend(fontsize=13)
plt.xticks(fontsize=13)
plt.yticks(fontsize=13)
plt.ylabel("Mean Discrepancy", fontsize=25)
plt.xlabel('No. of Episodes', fontsize=25)
plt.tight_layout()
```



6.4 How do the runtimes compare between the different iterations?

```
In [53]: time = pd.DataFrame([ql_1.time,ql_2.time,ql_3.time,ql_4.time, ql_5.time,ql_6.time, ql_7.time],
         time.columns = ['Time']

         time.plot(kind='bar', colormap=plt.cm.viridis, figsize=(16,6))
         plt.title('Plot of Runtimes (s)', fontsize=20)
         plt.ylabel('Runtime (s)', fontsize=20)
         plt.xticks(fontsize=13)
         plt.yticks(fontsize=13)
         plt.show()
```




```
In [54]: print(tb(time))
```

QL G=0.5	0.605089
QL G=0	0.609527
QL G=1	0.609409
QL Diff S,R,10K	0.64066
QL Diff S,R,20K	1.28131
QL Big Grid G=0.5	8.72751
QL Big Grid G=0	7.67015

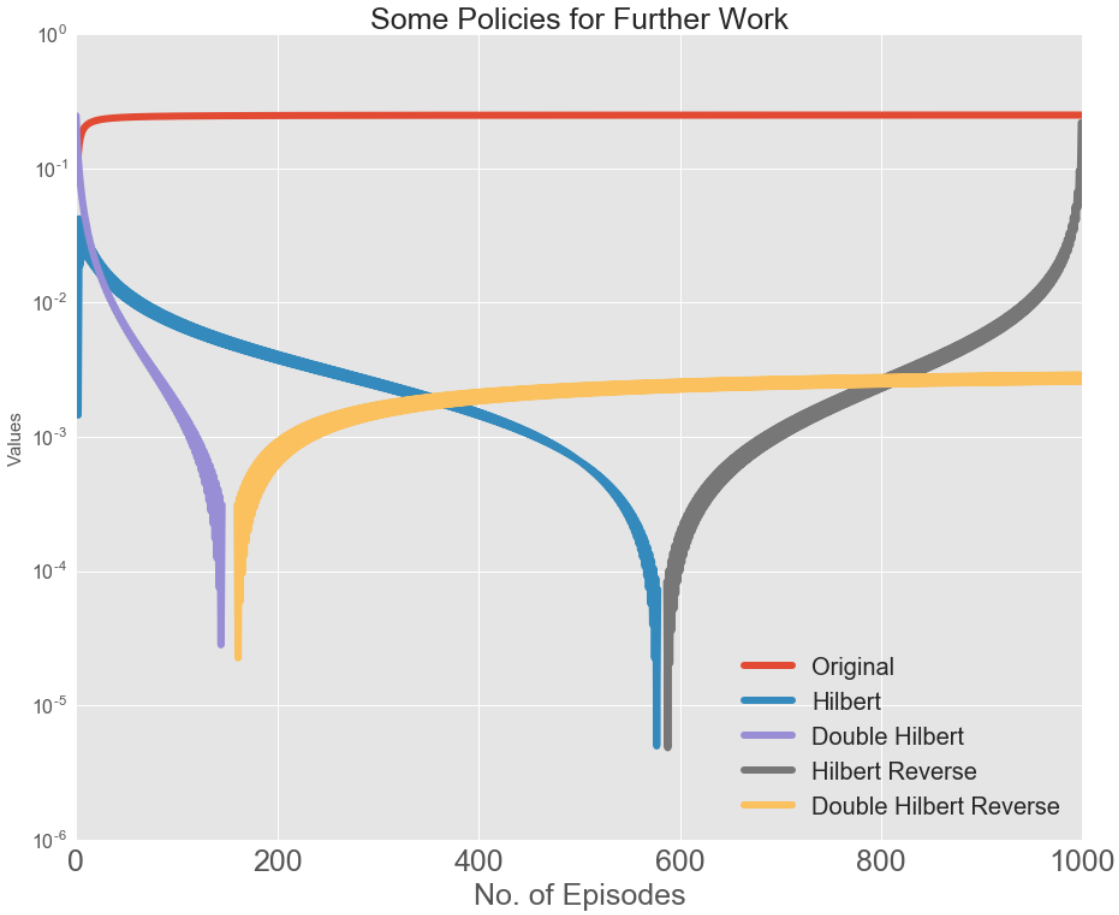
```
In [55]: print("Ratio of fastest to slowest run: %g"%((time.min()/time.max())))

Ratio of fastest to slowest run: 0.0693313
```

7 Extensions

```
In [101]: import scipy
          from scipy.fftpack import *

In [102]: f = lambda x: (x / (x + 2)**1/4)
          val =  f(np.arange(1000))

In [103]: plt.figure(figsize=(15,12))
          plt.semilogy(val, label='Original', linewidth=6)
          plt.semilogy(hilbert(val), label='Hilbert', linewidth=6)
          plt.semilogy(hilbert(hilbert(val)), label='Double Hilbert', linewidth=6)
          plt.semilogy(hilbert(val)*-1, label='Hilbert Reverse', linewidth=6)
          plt.semilogy(hilbert(hilbert(val))*-1, label='Double Hilbert Reverse', linewidth=6)
          plt.legend(fontsize=20, loc=4)

          plt.title('Some Policies for Further Work', fontsize=25)
```

```
        plt.xlabel("No. of Episodes", fontsize=25)
        plt.ylabel("Values", fontsize=15)
        plt.xticks(fontsize=25)
        plt.yticks(fontsize=16)
```

```
Out[103]: (array([  1.00000000e-07,   1.00000000e-06,   1.00000000e-05,
                    1.00000000e-04,   1.00000000e-03,   1.00000000e-02,
                    1.00000000e-01,   1.00000000e+00,   1.00000000e+01]),
           <a list of 9 Text yticklabel objects>)
```




```
In [ ]: plt.figure(figsize=(20,20))
        plt.subplots(411)
        plt.plot(discrepa)
        plt.legend(fontsize=20, loc=1)
        plt.title('Some Policies for Further Work', fontsize=25)
        plt.xlabel("No. of Episodes", fontsize=25)
        plt.ylabel("Values", fontsize=15)
        plt.xticks(fontsize=25)
        plt.yticks(fontsize=16)
```