UDACITY

MACHINE LEARNING – NANODEGREE

TRAIN A SMARTCAB TO DRIVE

BY: TODD FARR

1 Implement a Basic Driving Agent

1.1 Random Actions

Initially the agent was set to operate randomly. Given a list of valid option [None, 'forward', 'right', 'left'] the agent would drive aimlessly around the environment as expected. On occasion the agent would make it to the destination but hardly ever within the allotted time and its success was based more on luck rather than any sort of structured process. Since this is a finite grid-world environment it can be expected the agent will always end up reaching the goal eventually (assuming the deadline isn't enforced). However, in a real world scenario, random actions without a closed-loop learning system would likely result in a never ending taxi ride.

1.2 Identify and Update a State

Since Q-Learning relies on a updating a table of values associated with a given set of states, it's important to identify a state variable that will be informative and contain all valid information required to help the agent identify a set of Q-Values and ultimately learn. For the state variable the following variables were stored in a tuple represented by Equation 1.

$$state = \left(light_{state}, traffic_{oncoming}, traffic_{left}, traffic_{right}, waypoint_{next}\right) \quad \text{(Eq. 1)}$$

It's important to monitor the state of the traffic light as the agent hasn't really 'learned' anything if it hasn't learned how to obey the rules of the road. In addition to this, it was important to store the state of the oncoming, left, and right traffic. If there was another vehicle in any of these positions the agent should keep tabs on their positions and their next actions to avoid traffic collisions. Finally, the last variable supplied to the state variable is the next waypoint. This one's pretty obvious as the agent needs guidance to the final destination. Why the values were stored in a tuple (a python data type) is because tuples are hash-able, meaning they can be used as dictionary keys. This makes it simple to look up past states and update their associated values in the Q-Table (Q-Dictionary actually in this instance). Lastly, it was also considered to include the deadline as a component of this state variable but this may result in the agent violating traffic laws as this value decreases and the promise of that reward diminishes.

2 Implement Q-Learning

2.1 The Q-Table

The Q-Table is the literal backbone of the Q-Learning Algorithm. This is what stores all the Q-Values for any given state/action pair the agent will encounter in the environment. When the agent is initialized the Q-Table is built. A series of nested 'for' loops are used to create a unique tuple for every combination based on each of the state variables above. These tuples are then used as a hash to create a dictionary pointer to each valid action [None, 'left', 'right', 'forward'] that the agent can take at each of these states. This is the completed Q-Table in which each state/action pair now contains the value zero. These values will be updated by the Q-Learning algorithm as the agent begins to make its way around in and learn about the environment.

2.2 Updating the Q-Table

The Q-Table is updated by using the estimation of Q from transitions (Isbell, Littman 2015). This means that the max utility (best action$[a']$) from the next state (s') is used to update the Q for the current state and current action using Equation 2.

$$\hat{Q}(s,a) \xleftarrow{\alpha_t} r + \gamma\, max_{a'} Q(s', a') \qquad \text{(Eq. 2)}$$

In this equation $\boldsymbol{r}$ is the reward obtained at the current state, $\boldsymbol{\gamma}$ (gamma) is future reward multiplier (how much the agent values current reward vs future reward) and $\boldsymbol{max_{a'} Q(s', a')}$ is the maximum utility for the next state/action pair $(s',\ a')$. It's important to note that $\boldsymbol{\gamma}$ is a value between 0 and 1. If $\boldsymbol{\gamma}$ is equal to 0 then the agent would disregard the utility at the next state completely, however if $\boldsymbol{\gamma}$ is equal to 1 then the agent values the future reward as much as the current reward. Lastly, to update $\hat{Q}(s,a)$ a transition using a learning rate $\boldsymbol{\alpha}$ is used where $\boldsymbol{v} \xleftarrow{\boldsymbol{\alpha}} \boldsymbol{x}$ can be represented by Equation 3.

$$v \leftarrow (1 - \alpha)v + \alpha x \qquad \text{(Eq. 3)}$$

2.3 Picking the Best Action

As the agent moves around the environment and collects the associated rewards, the values in the Q-Table are updated as previously stated. However, before the best action can be selected based on Q-Values, there is some exploration that needs to take place. This is because when the agent is initialized the Q-Table is also initialized to all zeros. In order to start populating these values with meaningful numbers the agent needs to randomly select an action at any given state and collect the associated reward. If the action was 'bad' the Q-Value for that state/action pair will be decreased. However, if the action was 'good' the opposite happens and the Q-Value is increased. Eventually at some point the agent needs to stop exploring and start exploiting the values and information in the Q-Table. This is where the ε (epsilon) value comes into play.

One can think of ε as a probability or percentage that a random action vs one selected from the Q-Table will be chosen. Initially, ε was set to 0.5, then a random number is selected between 0-1. If this number happens to be less than ε, then the best action is simply a random choice of all the available actions. This 'best' action is taken and the associated Q-Value is updated for this state/action pair. However, if the random value is greater than ε, then the best action is selected by finding the Q_{max} for the given state. This is done by iterating through every action that can be performed in that state and selecting that action which has the largest Q-Value.

At some point, as the Q-Table becomes populated with useful values a ε value of 0.5 may not make sense anymore. In this case it becomes desired that the agent begins to lean more toward exploiting the Q-Table rather than exploring and randomly collecting rewards. This is where a decay function can be useful. Initially this decay function was simply a multiplication of ε and ε_{decay} values for each new trial (Equation 4).

$$\varepsilon_t = \varepsilon_{t-1} * \varepsilon_{decay} \qquad \text{(Eq. 4)}$$

It was decided empirically that a ε_{decay} value of 0.9-0.99 seemed to give the agent enough time to explore before it began to exploit and utilize the populated Q-Tables values.

3 Enhancing the Agent

3.1 Agent Learns a Feasible Policy within 100 Trials

Once the Q-Learning Algorithm was tuned an automated data collection script was written to run an iterative set of trials using these tuned parameters. Ten sets of 100 trials were exectured and the data from each was collected and parsed into a CSV file. After consistently verifying (during testing) that net reward was always positive with these values, the final metric chosen to validate Q-Learning performance was the number of successes vs the number of trials (fixed to 100 as per the rubric) for each iteration. The final results from the tuned parameters can be seen in the graph below (Figure 1).

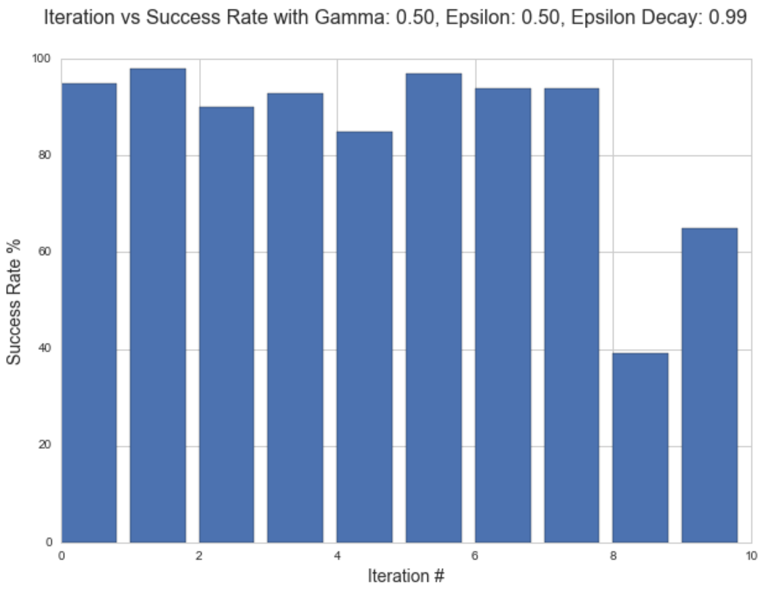

Figure 1: Final Q-Learning Algorithm Performance over 10 Iterations

This policy yields an overall success rate of 85%. There is some randomness introduced due to the initialization of the Environment that may be the cause for the two dips in performance on Iterations 9 & 10. However, overall the agent's ability to consistently learn and implement the policy in a short period of time (often less than 10 trials) is reflected in this data.

3.2 Improvements to Reach this Result

Initially, the Q-Learning algorithm that was used had fixed α, ε and γ values of 0.2, 0.5 and 0.9 respectively. These values were arbitrarily chosen based on some initial research and basic trial and error while observing how the agent interacted with the environment. After confidence was built that the Q-Table was updating correctly and agent was somewhat able to learn a policy based on this data, research began on how to better tune the algorithm.

The first improvement was changing to a decaying α function based on time. This was based off of the "Learning Incrementally" lecture (Isbell, Littman 2015) which yields the following function (Equation 5).

$$V_t \leftarrow^{\alpha_t} X_t \qquad \text{(Eq. 5)}$$

This function draws a series of X_t values to update the V_t values by a series of learning rates represented by α_t. By definition V_t will converge to the expected value of X if α_t satisfies the two properties represented in Equation 6.

$$\sum_{t=1}^{\infty} \alpha_t = \infty \ and \ \sum_{t=1}^{\infty} {\alpha_t}^2 < \infty \qquad \text{(Eq. 6)}$$

One possible learning rate sequence that satisfies these conditions, and that was shared in these lecture materials, is Equation 7.

$$\alpha_t = \frac{1}{t} \qquad \text{(Eq. 7)}$$

However, even after changing the α update to this function the agent had wildly inconsistent results. Often times it would be able to learn the policy quickly, but then on other trials it would fail to reach its destination on almost every single attempt. Based on these results, it was attributed to the hypothesis that other parameters needed to be tuned as well. Utilizing the same iterative, automated data script previously mentioned a series of 72 reinitialized environments and agents with varying γ, ε and ε-deacy values of 100 trials each where collected and their successes plotted (Figure 2).

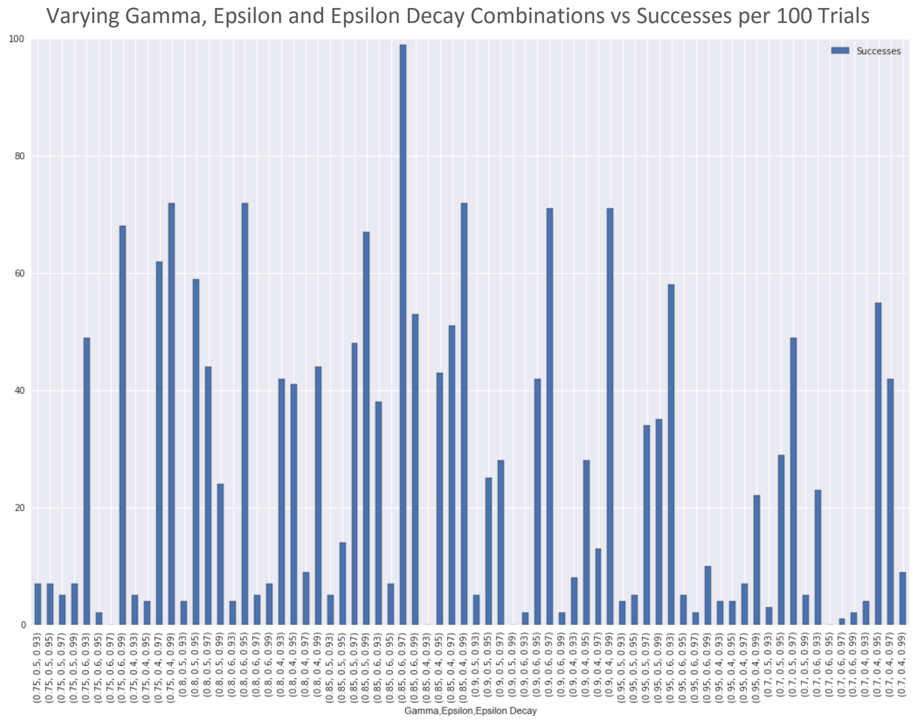

Figure 2: Gamma, Epsilon, and Epsilon Decay vs number of successes per 100 trials.

From this figure it was assumed that there was a clean winner and a correct combination of these values within the tested range. With almost a 100% success rate the obvious winning combination was ($\gamma = 0.85, \varepsilon = 0.6, \varepsilon_{decay} = 0.99$). However, when these values were tested on 10 initializations the overall success rate was only ~37.0%. The automated data script was modified to not only iterate across combinations of γ, ε and ε-deacy values, but also multiple times. It was also decided at this point to narrow the range of values around this 'high-performing' combination. These results can be seen in Figure 3.

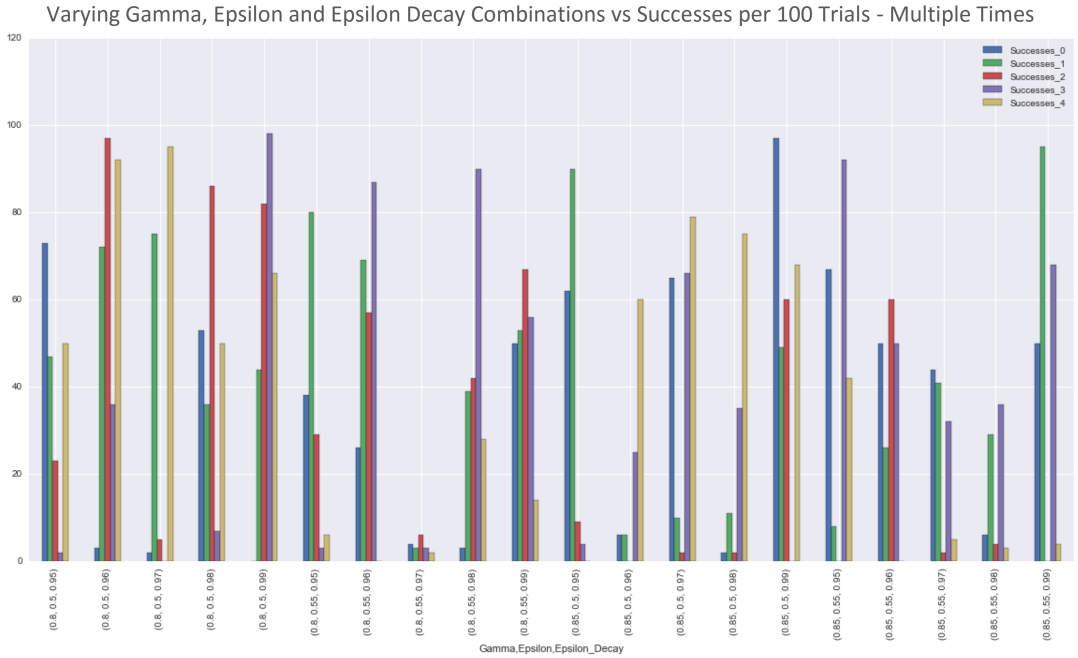

Figure 3: Gamma, Epsilon, and Epsilon Decay vs number of successes per 100 trials completed 5 times each.

Again, there was a lot of variability in the performance of the Q-Learning algorithm even for the same combination of variables. Therefore the approach was to revisit the strategy for the policy as a whole.

This change in direction started with revisiting the alpha decay function. In an article on Q-Learning (Manfredi 2001) an alternate α decay function was proposed and implemented (Equation 8).

$$\alpha_n = \frac{\alpha_0 + (n_0 + 1.0)}{n_0 + current\_trial}, where\ n_0 = \frac{total_{trials}}{10} \qquad \text{(Eq.8)}$$

In addition to adopting this new α decay function, an updated function for ε based solely on ε_{decay} was also implemented (Equation 9).

$$\varepsilon = \frac{1}{current\_trial + \varepsilon_{decay}} \qquad \text{(Eq. 9)}$$

The idea behind this function is not so much relying on ε_{decay} as a true decay value but using it to offset the effects at trial 0. Initially since ε was set to 0.5, there was a 50% chance that a random action wouldn't be chosen on the first trial (0). This was a little counter intuitive as at trial 0, the Q-Table is initialized to all zeros, meaning the best action probably is a random even if it results in a 'bad' one. As the current trial grows the ε value decreases resulting in more exploitation and less exploration. In this environment it seemed to have contributed to the agent learning the policy rather quickly. This is probably due to the fact that it's a relatively small number of inputs, states and actions.

The final improvement was revisiting a wider set of γ values because with the new ε equation this really became the only tunable constant parameter. Again the automated data collection helper script was used to iterate over a wider range of γ values, this time with a larger step to really understand how changing this value affects the performance of the Q-Learning Algorithm. The resulting performance bar graph can be seen below (Figure 4).

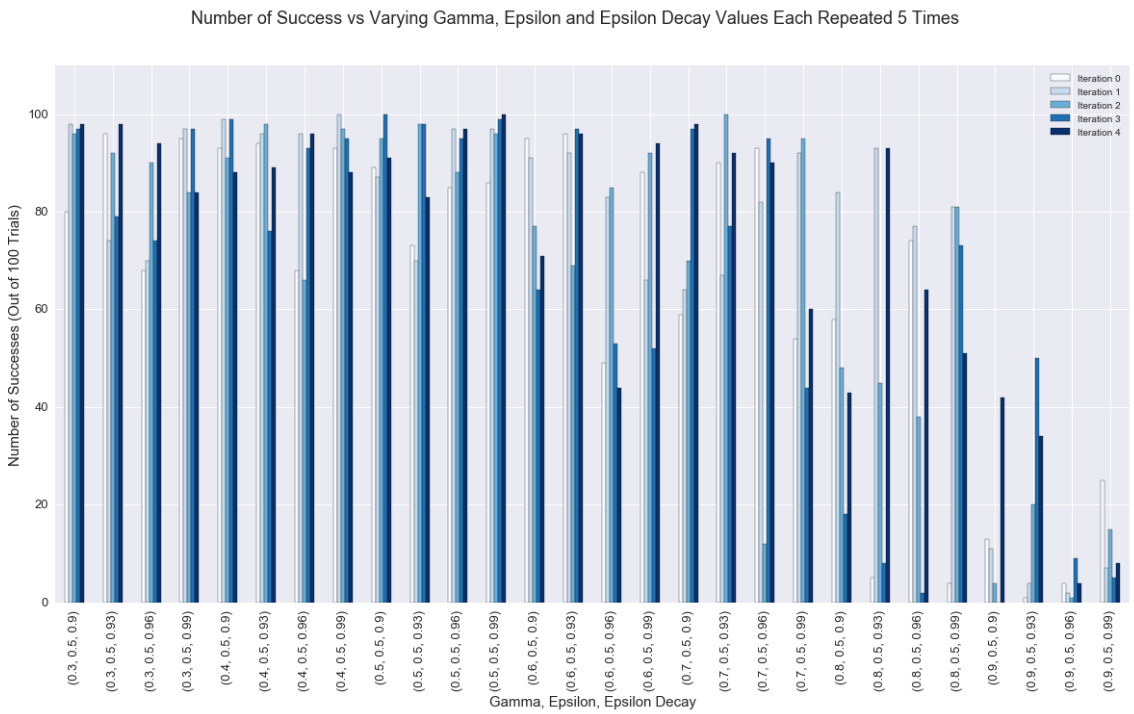

Figure 4: Gamma, Epsilon and Epsilon Decay Combination vs Success. Gamma Range (0.3 – 0.9)

From these results model performance drastically improves and stabilizes at γ values < 0.5. These two attributes (performance and stability) become even more apparent when this same data is plotted via box plots. As γ moves beyond a value of 0.5 the boxplots lengthen as the standard deviation highlights model instability and beyond a value of 0.7 the performance begins to significantly suffer (Figure 5).

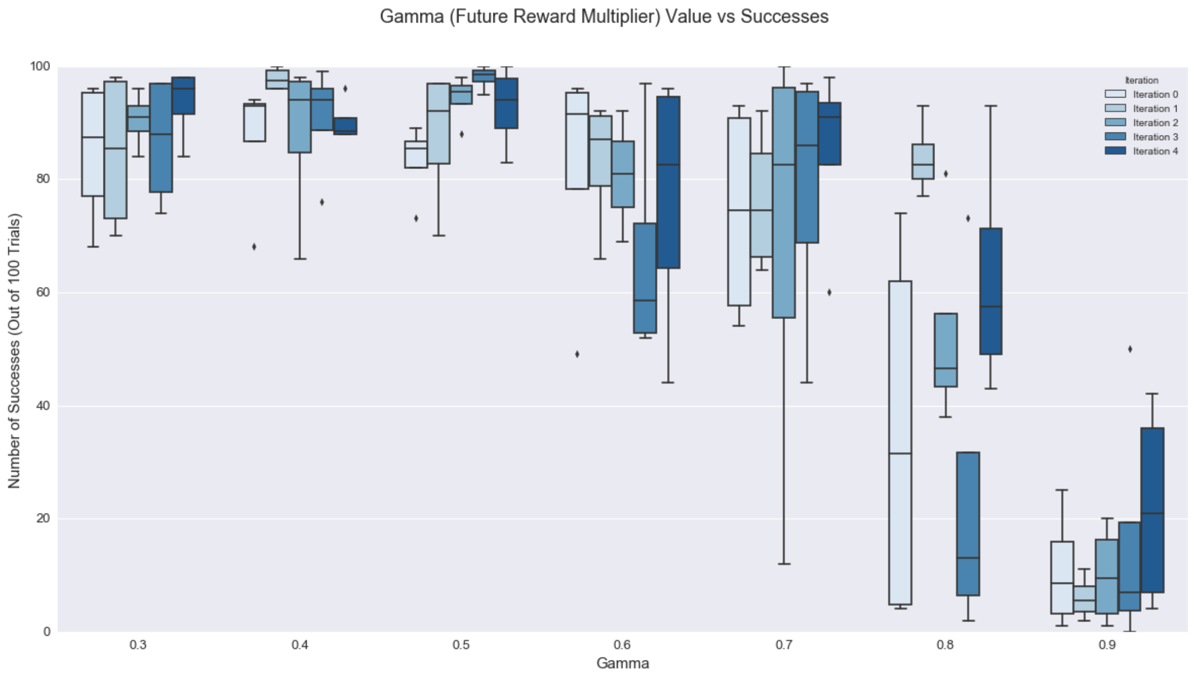

Figure 5: Box plots of the Gamma Value effects on model stability and performance.

Finally, as a simple and quick sanity check and validation of previous comments about γ being the major performance driving variable, the ε_{decay} value was also plotted vs successes over the 5 iterations (Figure 6).

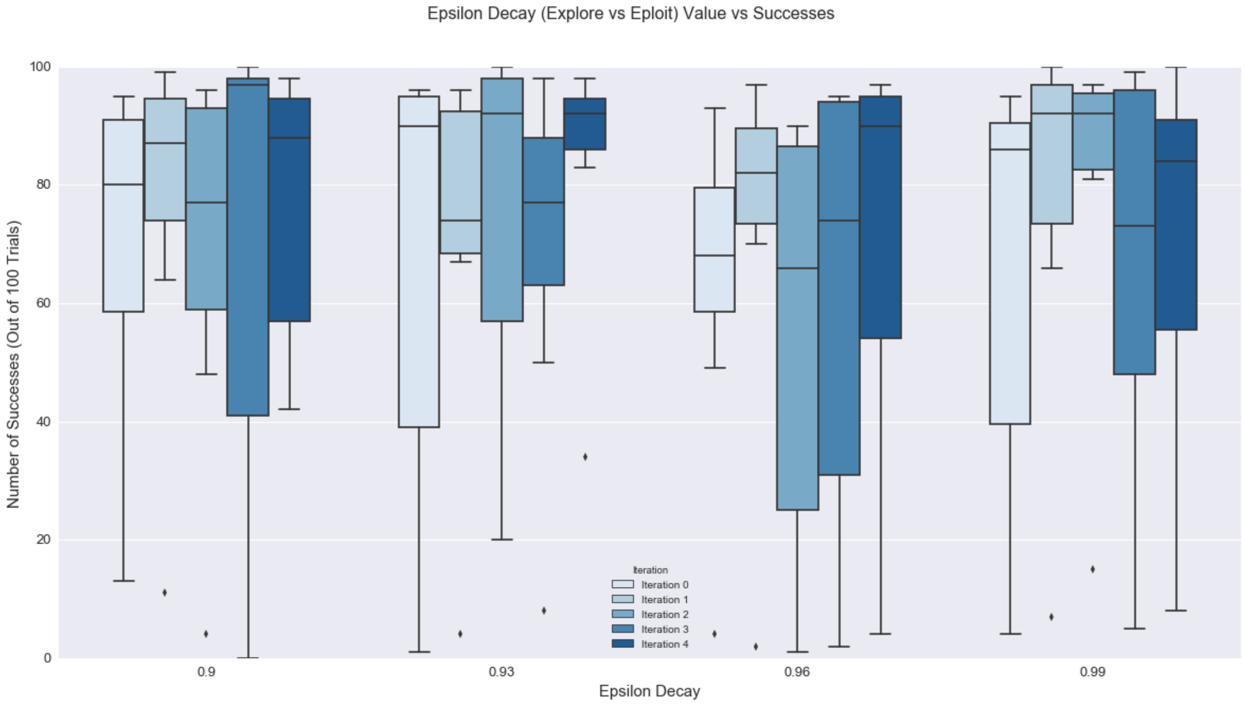

Figure 6: Box plots of the Epsilon Decay Value effects on model stability and performance.

From this plot it can be deducted that, from the various combinations tried, ε_{decay} had no real effects on the model.

3.3 Ideal Policy

By the last 10 trials using an ideal policy it would be expected that the agent had learned the rules of the road and was capable of finding the fastest and safest route to the destination. To determine agent performance in this space each awarded reward value was tallied for the last 10 of 100 trials (Figure 7).

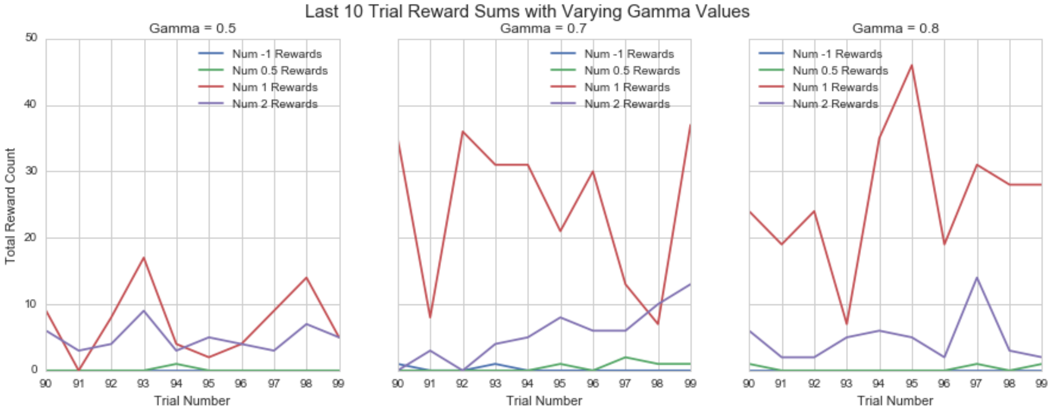

Figure 7: Reward counts for the last 10 trials with varying Gamma values

By analyzing these plots, one can see as γ changes the resulting reward tallies vary drastically. Based on the function of the environment and how rewards are awarded, it can be assumed that the fastest route can be quantified by a higher tally of value '2' rewards (following way points directly) while minimizing the other tally values, especially that of value '0.5' which indicates an action directly contradicts that given by the waypoint (the shortest distance). It's also important to keep tabs on the tally of '1' value rewards in correlation to speed as this reward reflects an agent movement of 'None'. In an ideal policy, if a traffic light is too long or if there's too much oncoming traffic, the agent could determine if a strategy of three consecutive right turns is faster and more efficient than a single left turn. Examining the plots it is clear the policy with a γ value of 0.5 did drastically increase the speed to the destination when compared to the others, however it's still sub-optimal as there is direct deviations from the waypoint on trial 94 and a few questionable trials (93 and 98) where the agent spends a lot of it's time not moving.

In terms of safety this is quantified by the agents ability to minimize the tally of value '-1' rewards which reflects that it has directly committed a traffic violation. For γ values of 0.5 and 0.8 the agents fortunately do not commit any of these violations in the last 10 Trials, however for a γ value of 0.7 the agent committed multiple traffic violations while attempting to reach the destination. Any '-1' rewards after this number trials is unacceptable and yields a sub-optimal policy by highlighting the agents inability to learn the rules of the road.

Finally, one last plot was created to validate policy stability as a function of net reward for the γ values tested (Figure 8).

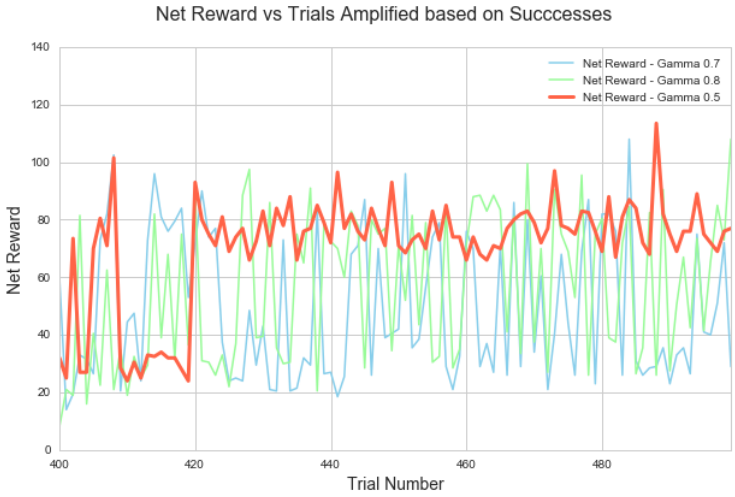

Figure 8: Policy Stability Based on Agent Net Reward.

This plot highlights each policies ability or inability to stabilize and unsuccessfully or successfully reach the destination within the allotted time. In order to magnify inconsistencies based on the policy the net reward was summed over each trial and if the agent successfully reached the destination an inflated value of 40 was added to this sum. This increased reward value amplifies peaks and valleys allowing for quick visualization of model stability. It also reveals how quickly the agent learns based on the policy. From this plot it's clear that the 'tuned' agent using a γ value of 0.5 quickly learns (< 20 trials) and consistently reaches the destination indicated by the associated stabilized red line.

References

Isbell, Charles, and Michael Littman. "Learning Incrementally Quiz - Georgia Tech - Machine Learning." *YouTube*. Udacity, 23 Feb. 2015. Web. 1 May 2016. <https://www.youtube.com/watch?v=FtRJKOvI_fs>.

Isbell, Charles, and Michael Littman. "Greedy Exploration - Georgia Tech - Machine Learning." *YouTube*. 23 Feb. 2015. Web. 5 May 2016. <https://youtu.be/yv8wJiQQ1rc>.

Isbell, Charles, and Michael Littman. "Estimating Q From Transitions - Georgia Tech - Machine Learning." *YouTube*. 23 Feb. 2015. Web. 5 May 2016. <https://youtu.be/Xr2U3BTkifQ>.

Manfredi, Victoria. "Q-learning (QL)." *Q-learning (QL)*. 02 Aug. 2001. Web. 23 May 2016. <http://dreuarchive.cra.org/2001/manfredi/weeklyJournal/pricebot/node10.html>.

McCullock, John. "Q-Learning." *A Painless Q-Learning Tutorial*. 2009. Web. 5 May 2016. <http://mnemstudio.org/path-finding-q-learning-tutorial.htm>.

Poole, David, and Allan Mackworth. "Artificial Intelligence." *Foundations of Computational Agents*. 2010. Web. 26 May 2016.