Information Theory and Reinforcement Learning
Final Project

Fouret, Amaury Vercoustre, Paul Sveen, Andrea Drake

March 29, 2017

Contents

1 Executive Summary

We worked on the LunarLander - V2 on OpenAI, where the goal is to get a lander to land on the moon safely without crashing. We attempted to do this by implementing Q-learning, Kernel Q-learning and a shallow version of Deep Q Network learning (DQN).
We consider Kernel Q-learning as the best algorithm, even though regular Q-learning achieved a higher mean reward overall. This is because Kernel Q-learning obtained the highest mean reward when run for the same number of episodes as Q-learning. As seen in Table 1 the performance improves drastically with the number of episodes, but due to Kernel Q-learning taking much longer to complete an episode we were not able to run as many episodes for this algorithm ($1,000$ episodes took 12 hours to run). We are confident that Kernel Q-learning would have had the best mean reward if we been able to run Kernel Q-learning for more episodes.

Episodes	**Kernel Q-learning 1k**	**Q-learning 1k**	**Q-learning 90k**
Mean Reward	54.54	−8.16	74.75
Median Reward	76.70	−42.99	112.42
Standard Deviation of Rewards	145.26	158.62	134.82
Max Total Reward	272.19	272.51	273.23
Min Total Reward	−548.09	−507.18	−507.18

Table 1: Results of Kernel Q-learning and Q-learning.

2 Project Introduction

The goal is to have a lander land on the moon without crashing. At each step the agent is provided with the current state s, which is an 8 dimensional vector $\mathbb{R}^8$ and the available actions are:

- Do nothing
- Fire left orientation engine
- Fire right orientation engine
- Fire main engine

Once the agent makes an action a, it receives a reward r and the new state s'. The goal is for the agent to learn a policy $\pi(s)$ which decide what action to make based on experience form previous episodes (s, a, r, s').

The landing pad is always at coordinates (0,0), and the lander's goal is to get from the top of the screen to the landing pad. The lander receives between 100 and 140 points for moving from the top of the screen to the pad, with zero speed. Each episode ends when the lander either crashes or lands, receiving and additional -100 or 100 points depending on if it crashes or lands. Each leg contact on ground is +10 points, firing main engine is -0.3 points per frame, and solved is 200 points. Landing outside the pad is possible and fuel is infinite. LunarLander - V2 is considered solved if one gets an average reward of 200 over 100 consecutive tries. The best agent is the agent that is able to solve it in the least amount of steps.

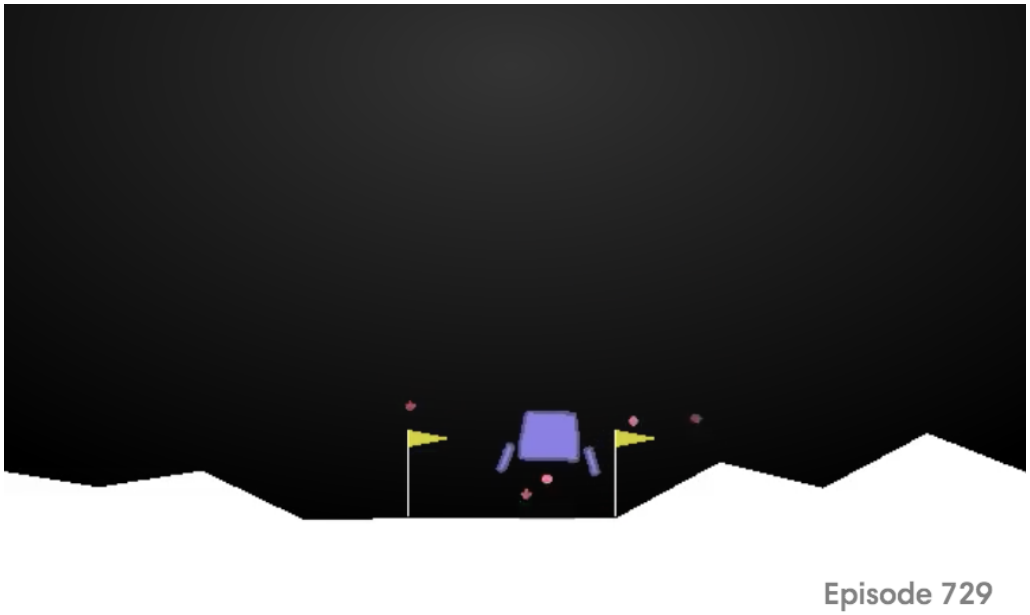


Figure 1: LunarLander - V2

3 Analysis

3.1 Environment Exploration

The state space is $\mathbb{R}^8$. The first and second dimensions are the coordinates of the lunar vehicle. The third and fourth are the x and y coordinates of the speed of the vehicle. The fifth dimension is the angle between the lunar vehicle and the ground. The sixth dimension is the angular speed in respect to the ground, and the seventh and eighth dimension indicates whether the legs have touched the ground or not.

While running our algorithms, we noticed that each dimension is bound between $-\infty$ and $+\infty$. We decided to modify these bounds to avoid too much variance. We did this directly in the environment file 'lunar_lander.py' which is located in the folder '.../gym/envs/box2d'. We put the bounds between -1 and +1, which is advised by the author of the script while this phrase is written as a comment: 'useful range is -1 .. +1, but spikes can be higher'. We also found that the action space was well discretized by default in the script.

3.2 Algorithmic Approach

3.2.1 Random Agent

As a reference we ran a random agent over 200 episodes, and obtained an average reward of -500 as seen in Figure 2. We use this as a benchmark for our other algorithms.

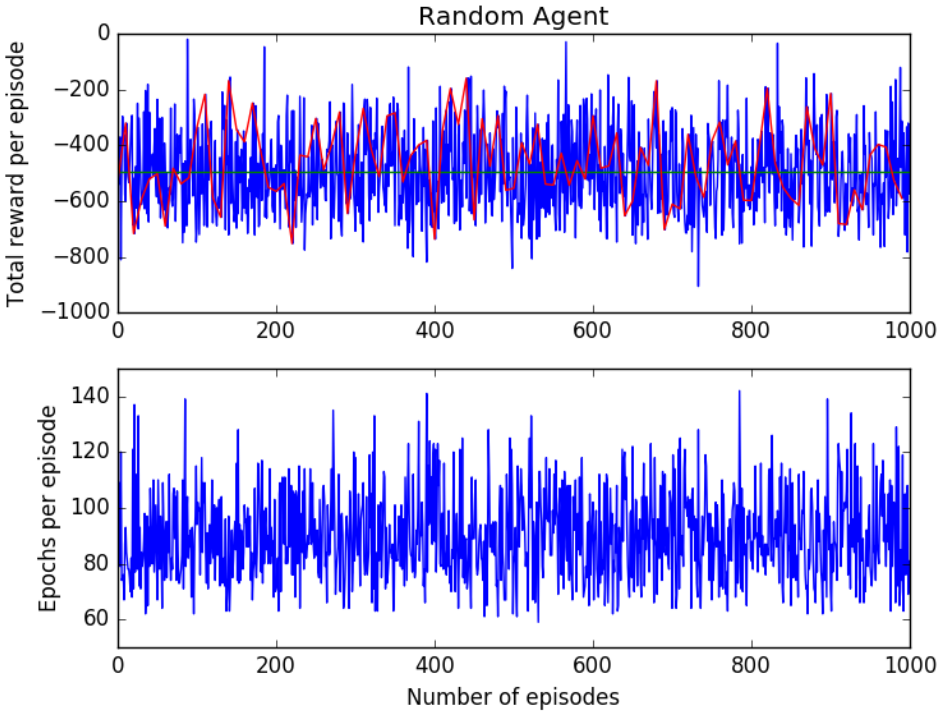


Figure 2: Random Agent

3.2.2 Q Learning

In Q-learning the agent makes decision based on the total reward the agent expects to receive if it takes an action a in state s and continues to follow the policy thereafter. The total reward is $Q^*(s, a)$, can be written as:

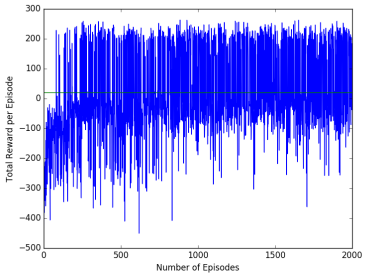
$$Q^*(s, a) = \sum_{i=0}^{\infty} \gamma^i r_i,$$

where γ is the discount factor $(0 < \gamma < 1)$, which ensures a finite value for $Q^*(s, a)$.

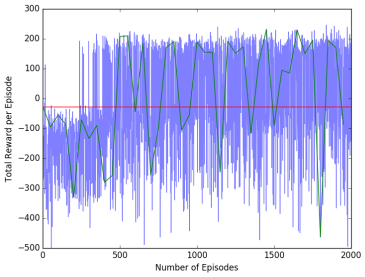
A Q-learning approach does not work on continuous environments because there is an infinite number of states. Therefore, we decided to slice the space into a set of different possibilities. Since the two last dimensions, left leg and right leg, are binaries, we only applied the discretization on the last six dimensions. To do that, we used two functions (build_state and to_bin) written by carlos-aguayo in his solution for the CartPole-v0 environment on gym.ai. These functions created new discrete dimensions in which we could control the number of possible values (*via* the number of bins per dimension). Finally we applied those to the observation space from the gym environment. The result is a discretization of the space in (number of bins per dimension)$^{\text{(number of dimensions discretized)}} *$ 4 (4 possible states relative to the two last dimensions) possible states. For example, if we decide to give 3 bins (3 possible values) per dimension, the total number of possible states in the space is $3^6 * 4 = 2916$.

For all the training tests for this Q-learning approach, we applied an exploration rate ϵ which started at one and decreases by $\epsilon = \epsilon * 0.995$ at each episode until it reach a minimum value of 0.1.

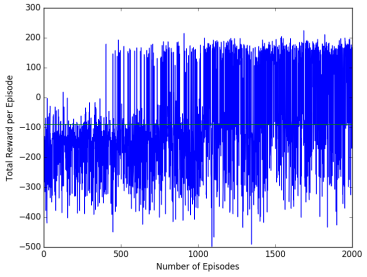
At first we experimented with different bin sizes, and as you can see in Figure 3 we get the highest mean with 3 bins.



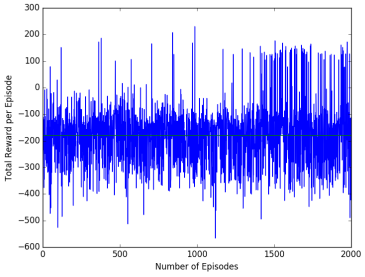
(a) 3 bins, 2,000 episodes.



(b) 5 bins, 2,000 episodes.



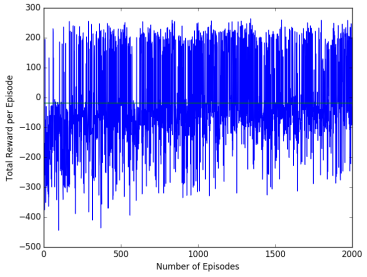
(c) 7 bins, 2,000 episodes.



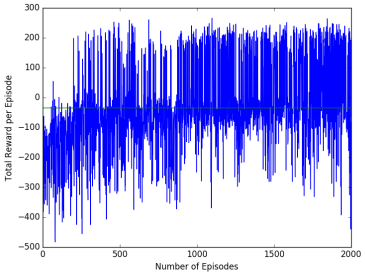
(d) 9 bins, 2,000 episodes.

Figure 3: 3, 5, 7 & 9 bins, $\gamma = 0.99$.

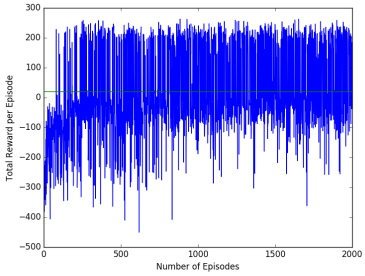
We then proceeded to run 3 bins with γs of 0.90, 0.95 & 0.99 and as seen in Figure 4 $\gamma = 0.99$ clearly has the highest mean reward.



(a) $\gamma = 0.90$.



(b) $\gamma = 0.95$.



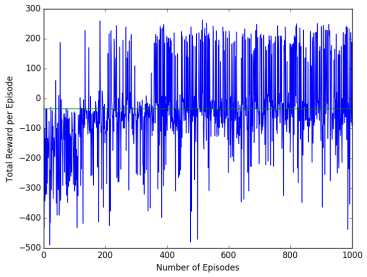
(c) $\gamma = 0.99$.

Figure 4: 3 bins, $\gamma = 0.90$, 0.95 & 0.99.

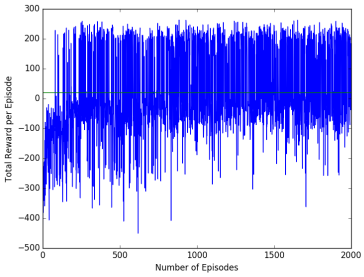
We tried different alphas between 0.001 and 0.5, and found that 0.1 worked best. Below you can see different outputs of the model. After running different variations we saw that we got the best results when using 3 bins so we experimented with 3 different γs, 0.90, 0.95 and 0.99 and saw that 0.99 worked best. In figure 5 we ran the algorithm with $\alpha = 0.1$, bin=3 and $\gamma = 0.99$

Episodes	**2,000**	**20,000**	**90,000**
Mean Reward	24.46	66.70	74.75
Median Reward	−25.39	16.92	112.42
Standard Deviation of Rewards	155.90	138.48	134.82
Max Total Reward	272.51	272.51	273.23
Min Total Reward	−507.18	−507.18	−507.18

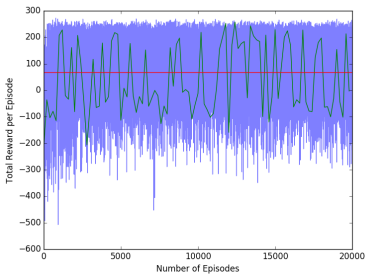
Table 2: Comparison of outputs of Q-learning with 3 bin and $\gamma = 0.99$ for different # of episodes.



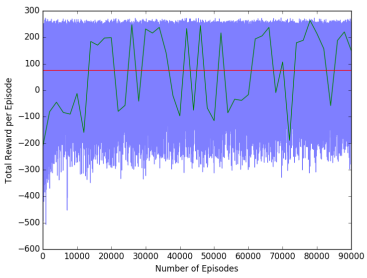
(a) 1,000 episodes.



(b) 2,000 episodes.



(c) 20,000 episodes.



(d) 90,000 episodes.

Figure 5: 3 bins, $\gamma = 0.99$.

On the 20,000 and 90,000 episodes graphs, the red line is mean and the green line is the mean every 2,000 episodes. It has been done to give more clarity to the graphs because of the huge number of episodes.

3.2.3 Deep Q Network learning (DQN)

The DQN uses a neural network to approximate the state value function $Q^*(s, a)$. When we use DQN the policy is redefined to be

$$\pi(s) = \underset{a'}{\operatorname{argmax}}\, \widehat{Q}(s, a'),$$

where $\widehat{Q}$ is the neural network approximation of the true state-action reward Q^*.

From section 3.2.2 we have:

$$Q^*(s, a) = \textstyle\sum_{i=0}^{\infty} \gamma^i r_i,$$

Using the Bellman recursive formulation of $Q^*(s, a)$ we get:

$$Q^*(s, a) = r + \gamma \max_a Q^*(s', a)$$

Based on experience (s, a, r, s') the neural network is trained by taking the following steps:

1. Compute Q value for state s for all a':

$$q_{a'} = Q^*(s, a')\ \forall a$$

2. Compute the Q values for state s' for all a':

$$q_{a'} = Q^*(s', a') \ \forall a'$$

3. Compute the target Q values for s:

$$Set \quad t_{a'} = q_{a'} \forall a' \neq a$$

$$Set \quad t_a = \begin{cases} r + \gamma \max_{a'} p_{a'} & s \text{ is not the end state} \\ r & s \text{ is the end state} \end{cases}$$

We used simple DQN and double DQN, with MSE and huber loss, respectively, as the loss functions. We tried one, two and three hidden layers with $ReLU$ as activation unit and 20, 40, 64 neurons per layer. The results obtained are worse than those obtained with the classical Q learning with standardized space. Figure 6 represents our best results using one hidden layer with 64 neurons and the huber loss function. We can see that the agent is learning and after $5,000$ episodes, we regularly achieve a reward of 200 per episode. However, a weird phenomenon occurs, after $10,000$ episodes, the total reward per episode stabilizes around -100 which is not good. We have not succeed to justify this phenomenon.

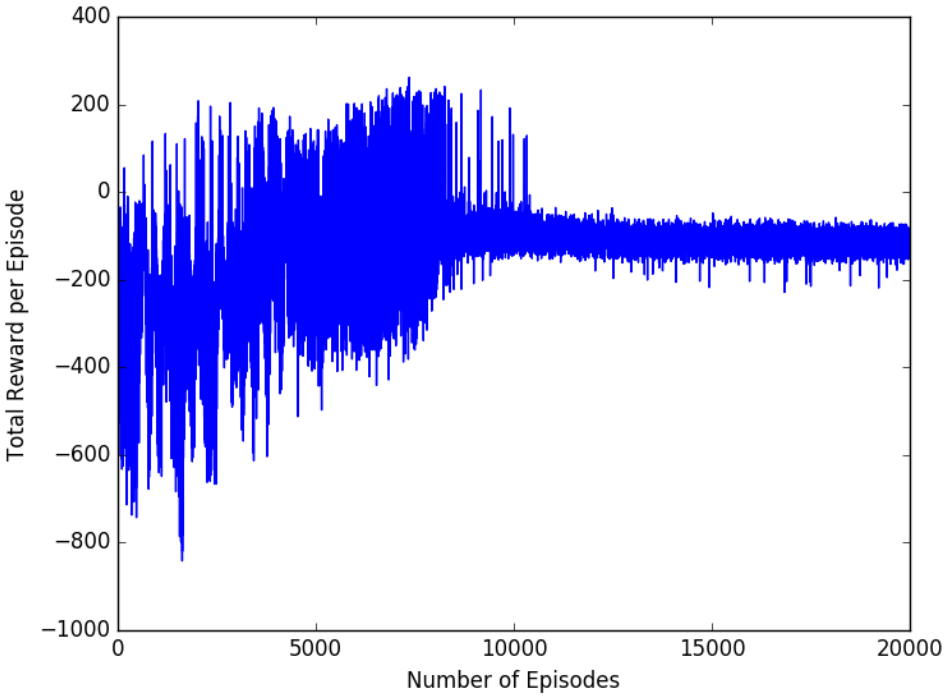


Figure 6: DQN

Since our Deep Q Learning was not giving the expected results and was doing worse than our classical Q learning, we decide to implement a more sophisticated discretization of the state space using a kernel to improve our agent.

3.2.4 Kernel Q-learning

What we refer to as "Kernel Q-learning" is an approximate Q-learning algorithm that maps the original state in $\mathbb{R}^8$ to a higher dimension representation of the state. We then consider linear functions on this new space to approximate the Q-function.

When discretizing the state space we had to be careful regarding the number of nodes (bins) we used since, assuming each dimension is split the same way, the new representation vector would have $(bins+1)^8$ dimensions; meaning the dimensionality would blow up very quickly (with serious consequences on run time). With that in mind we split the last 2 dimensions with 1 bin only (since they are boolean values) and tried different bin values for the other dimensions. A state point $s^{i,j,..,p}$ has new coordinates:

$$s^{i,j,..,p} = (-1 + i\frac{2}{bins}, -1 + j\frac{2}{bins}, ..., -0.5 + p)$$

We then use a kernel to express each coordinate of the $(bins+1)^6 \times 2^2$ dimension representation vector ϕ as function of the closeness between the state vector $(x_1, .., x_8)$ and $s^{i,j,..,p}$. Specifically:

$$\phi_{i,j,..p} = \exp^{-(x_1 - s_1^{i,j,..,p})^2} \times ... \times \exp^{-(x_1 - s_8^{i,j,..,p})^2}$$

The Q-function approximation is :

$$f_W = \begin{cases} R^n \times \{1,2,3,4\} = R \\ \phi, a = W^a \cdot \phi \end{cases} \tag{1}$$

Choosing bin values higher than 3 was prohibited by run time so we tried running our agent with bins = {1,2,3} while tuning the other parameters of the model (i.e. epsilon, epsilon decay rate, gamma, alpha)

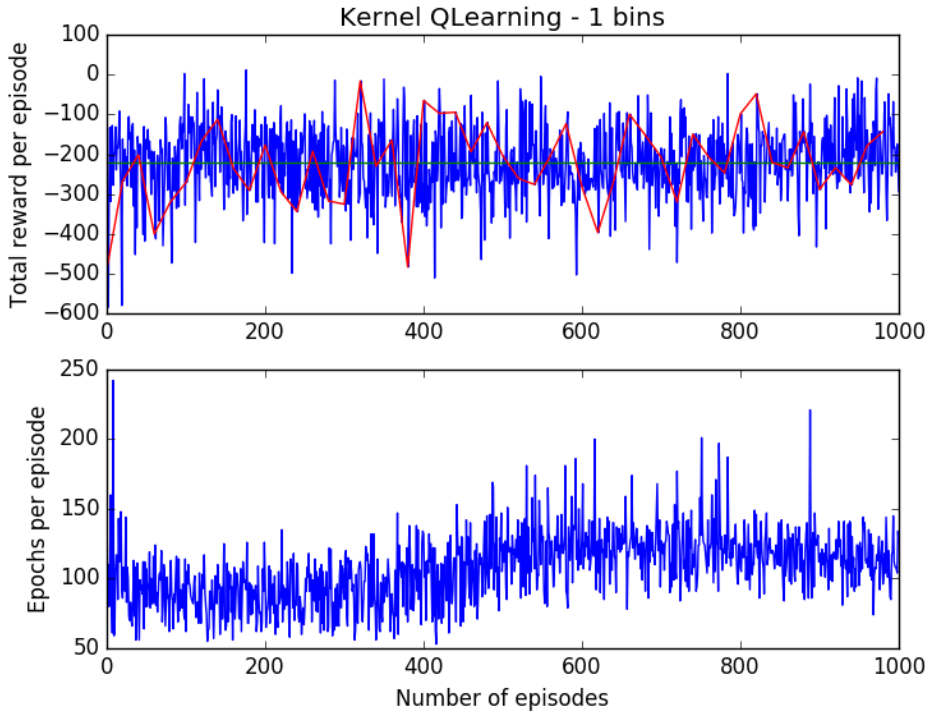


Figure 7: Kernel Q-Learning with 1 bin

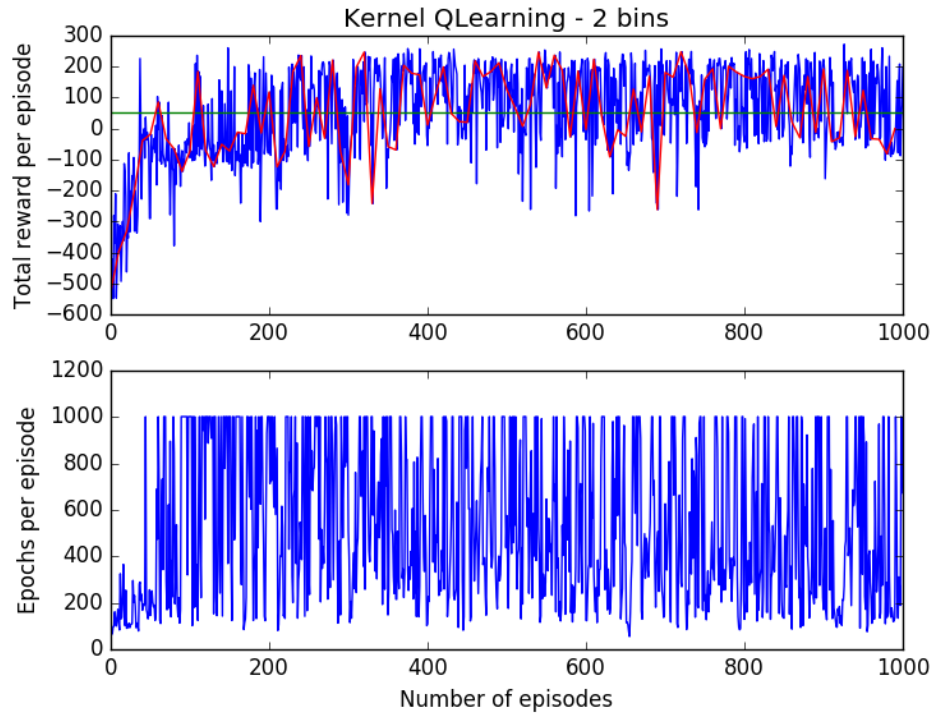


Figure 8: Kernel Q-Learning with 2 bins

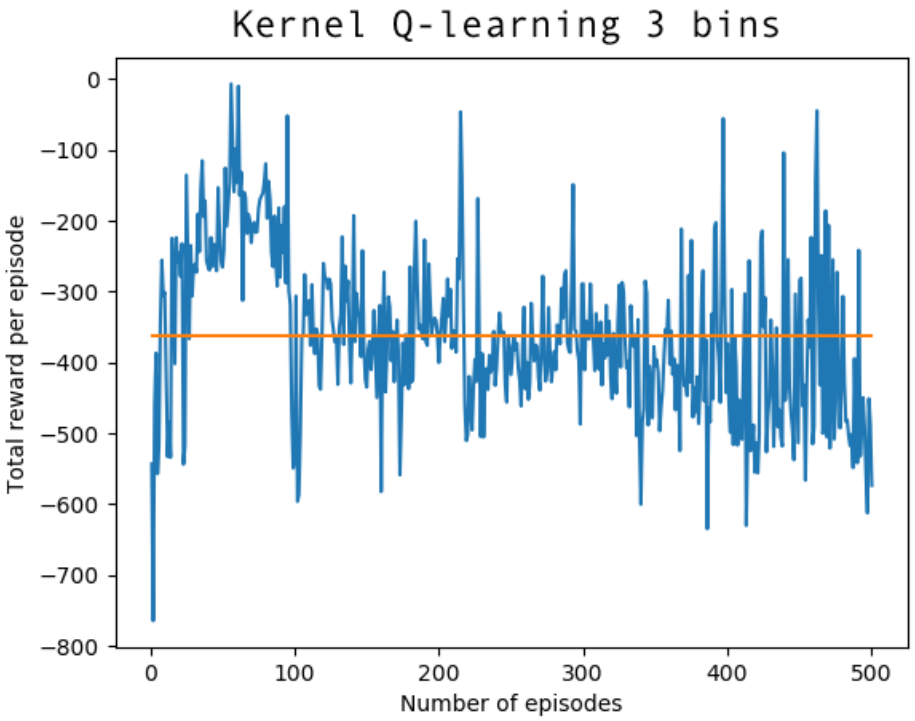


Figure 9: Kernel Q-Learning with 3 bins & $\epsilon = 0.8$

4 Conclusion

We consider Kernel Q-learning as the best algorithm, even though regular Q-learning achieved a higher mean reward overall. This is because Kernel Q-learning obtained the highest mean reward when run for the same number of episodes as Q-learning. As seen in Table 3 the performance improves drastically with the number of episodes, but due to Kernel Q-learning taking much longer to complete an episode we were not able to run as many episodes for this algorithm ($1,000$ episodes took 12 hours to run). We are confident that Kernel Q-learning would have had the best mean reward if we been able to run Kernel Q-learning for more episodes.

Episodes	**Kernel Q-learning 1k**	**Q-learning 1k**	**Q-learning 90k**
Mean Reward	54.54	−8.16	74.75
Median Reward	76.70	−42.99	112.42
Standard Deviation of Rewards	145.26	158.62	134.82
Max Total Reward	272.19	272.51	273.23
Min Total Reward	−548.09	−507.18	−507.18

Table 3: Results of Kernel Q-learning and Q-learning.