Project 2 - CS7642 | Lunar Lander

Andres Miguel Menendez, amenendez6, 902969924

amenendez6@gatech.edu

Commit Hash: 3da7a615c2ee3688549bc192340e6ec449339875

INTRODUCTION

The lunar lander is a small game where a spacecraft has to navigate the moon's surface to land on a landing-pad. The lander is equipped with sensors that provide an input into the navigation agent in the form of an 8-element tuple as follows: $(x, y, x', y', \theta, \theta', leg_L, leg_R)$. This tuple represents a continuous 8 dimensional state space that describes the landers' position, angle and velocity throughout the stochastic environment. The lander has control over 3 jets which it may turn on/off in its attempt to reach its objective. These will toggled discreetly at no throttle or full throttle. The lander, controlled through an agent, will then attempt to successfully land as optimally and reliably as possible. The agent will be guided by a system of rewards that enable to measure the value of each particular state and gauge how successful each successive run is. The agent is penalized with small negative rewards as it moves away from the landing-pad or if its roll angle distances itself from a vertical position, and an inevitable crash is heavily punished with a -100 reward. A proper and successful approach positively rewards a touchdown on both legs and a healthy +200 for landing on the landing-pad. This problem can be rendered as a continuous Markovian Decision Process (Markov Chain), where each state will have policy-prescribed action even when accounting for the stochastic environment.

This Markov Chain can be leveraged to create an agent that can optimally learn through iterative attempts and maximize long term rewards. A Deep Q-Network (DQN) algorithm can be used to solve this MDP by solving for an optimal policy. The DQN network anchors itself on a neural network to efficiently store Q-values of a continuous state-space and thereby develop an optimal policy. This avoids massive tables of Q-values and allows the agent to maximize its learning on a reduced number of experiences. Gradient descent allows the agent to reliably converge on near optimal q-values through the updating of weights within the neural network. This algorithm is driven by reinforcement learning theory where the agent can learn after every action taken. Several hyperparameters play a heavy role in how the agent learns. This is not only with respect to the values of these parameters, but how and when they are updated and controlled. The following discussion looks into the role of this algorithm within the paradigm of reinforcement learning.

DEEP Q-LEARNING

The backbone of DQN is the neural network that maps states to actions. This network enables the agent to efficiently create a policy for the continuous state-space within a stochastic environment. It allows for the use of Bellman equations by effectively bootstrapping through the approximations of a neural network. Backpropagation and gradient descent allows for gradual updates to approach pseudo-true values of each state and therefore converge on near an optimal policy. The goal is not necessarily for the agent to find the global minima of the 8 dimensional state-space, but to reach a local minima that can allow the Lunar Lander to reliably and successfully land.

This algorithm learns through the use of mini batches where it maximizes the utility of an experience by repeatedly using it to evaluate prediction error. In more traditional algorithms, reusing experiences can lead to biased learning and over-fitting as it can over train on specific states. But, because this algorithm randomly selects its batches, each experience is evaluated with a different combination of states and therefore avoids repeated patterns. Because it isn't following a full episode and only sampling a state on its own, it will not dig a channel into the Q table as it will not be relating the state with respect to the entire run, but the state with respect to the entire policy. This means that the memory buffer where the batch size is pulled from has to be big. Both the memory buffer and batch size are hyper parameters that can heavily affect learning but the analysis below assumes a large buffer to avoid overfitting and disregard memory optimization. The buffer also acts as an exploration parameter as its first iterations work with a high epsilon greedy strategy where the intention is to explore the entire state space. As the epsilon value decays, the memory buffer starts to fill up with more repeated states and converges on an optimal path. This is noteworthy as an improper buffer size could lead to too much exploration or not enough exploration. The mini batch size is explored more in depth in the analysis below.

Over a large number of batches and iterations, the algorithm effectively acts as a Bellman update where each state is updated with a new approximation. This bootstrap method relies on prediction error which is solely founded on the neural network. The neural network spits out an action which the agent executes in order to receive a new value that enables the agent to converge on a more correct q-value. The Bellman update equation for q-values, $Q_t(s, a) = R(s, a) + \gamma\, max_a Q_{t-1}(s, a)$, is key to realizing this convergence as each visit to a state creates a new prediction error and thereby allows the neural network to approximate to more true values through gradient descent. The neural network is essential to quick approximations and updates but compels for an appropriately sized model.

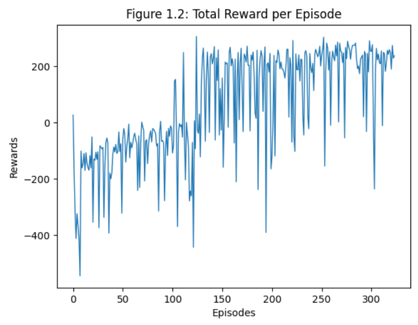
A large neural network model would result in more accurate approximations, but would demand for a lot of data to avoid overfitting. A massive network, both wide and deep, could ideally reach the most ideal and optimal policy for the Lunar Lander agent but would require tons of data to allow it to reach the most perfect weights for each node. Without massive amounts of data, such a large model could over fit to the limited data and fail to reach an optimal policy. Achieving this data is possible but would be contradictory to the primary advantage of DQN: its high learning efficiency. As discussed earlier, this algorithm maximizes the utility of each experience by evaluating each state independently from its origin episode. It would be fundamentally contradictory to reconcile a voluminous network with a learning-efficient algorithm. This dictates also that the size of the network can be treated as a hyperparameter as a small one could result in mediocre granularity of approximations, while a large one would demand more data or threaten the agent with overfitting. The agent analyzed below works with a 3 layer network that inputs an 8-element state tuple through two hidden layers of 516 and 256 nodes respectively. The resulting output is a 4 element tuple that maps to the 4 actions of: 0- do nothing, 1- fire leftmost thruster, 2- fire bottom thruster, 3- fire rightmost thruster. The network uses the *relu* activation for all hidden layers and a *linear* activation function for the output layer. Then, to compile, it uses the common and popular optimizer, Adam.

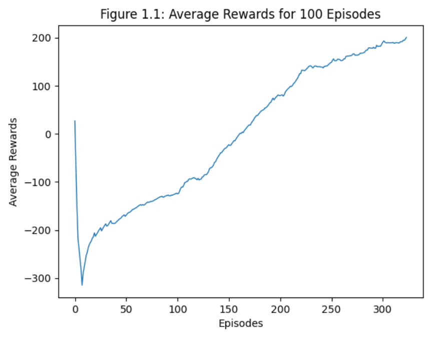
The optimizer is a function used to update weight values across the network. It is the gradient descent function that allows for the convergence of optimal approximations. Each learning step requires the model to fit new data by an evaluated error but is limited to a learning rate. This learning rate is a key hyperparameter.

LEARNING SUCCESSFULLY

An agent was developed to successfully learn and solve the Lunar Lander problem. It's learning performance is shown on the right in figures 1.1 and 1.2. Considering the stochastic nature of the environment, the agent must come to understand the entire state-space in order to navigate from any possible state. Additionally, if the agent does not sufficiently explore the environment, it may fail to find the most optimal path. Yet, after exploration, the craft must be able to precisely understand the right parts of the environment to optimally and reliably land. Thus, it must balance both exploration and expertise. There are various methods of exploration but the primary driver is the ε-factor. The agent takes on a ε-greedy strategy where it will take a random action ε*100% of the time. At first, this agent explores randomly 100% of the time as ε=1. As the agent moves forward it decreases its ε-value by a ε-decay factor. This agent adopts a decay rate of 0.995 and applies it to the ε-value at each learning step. This introduces a new variable: the rate of learning.

As the agent moves across the environment it will learn at a particular rate depending on how often it learns and at a particular learning-rate factor. The tempo of learning indicates how often the agent will learn and apply a gradient descent to the network. This agent skips 5 steps and learns on every 6th step. This not only affects the rate of learning but affects the exploration as the ε-value only decays every 6 steps. Allowing the agent to learn intermittently enables it to fill the buffer with more random states and impacts learning as it has a chance to evaluate and approximate more states. The learning-rate is the other parameter that will directly affect the learning where the neural network will backpropagate and fit to the new prediction error by a small fraction. This is the learning rate and has to be appropriately sized. An excessively large learning rate could prohibit the network from finding the optimal minima yet a small learning rate could lead to overfitting and inefficient learning. This particular agent adapted a learning rate of 0.001.



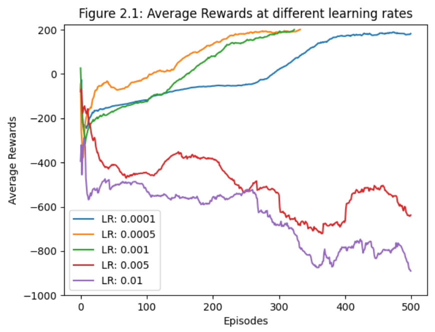


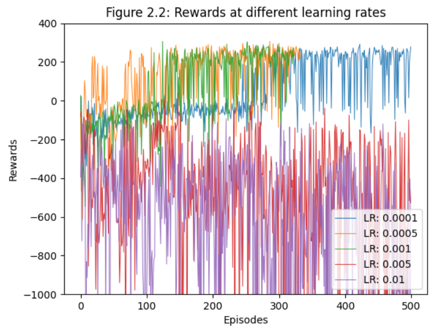
The final primary hyper parameter that was discussed earlier is the buffer and batch size. This agent took on a large memory buffer of 500000 steps and a batch size of 32 steps. The size of these are not only important for the reasons discussed above but also because they may affect the learning efficiency. Larger mini batches slow down as it has to iterate over more values at every learning step. Yet, an excessively small batch size will prohibit the agent from learning enough at each learning step.

There still lie various other hyper parameters that could be considered and optimized, such the neural network size and, convergence criteria, and reward shaping. These would affect how the agent would learn and when it could adapt other hyperparameters. The learning rate could also decay or multiple networks could be used while rewards could be changed to incentivize different strategies. These are not discussed in the discussion but could be considered in new and improved agents. This agent successfully trained the lander to average a score of 200 within 315 episodes.

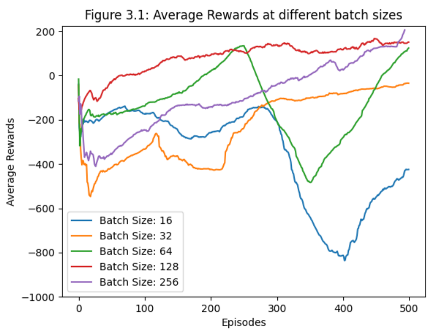
HYPERPARAMETERS

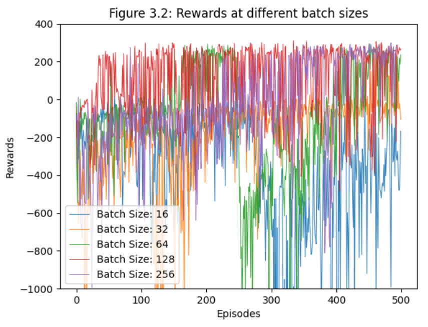
The following section explores the learning rate, batch size and epsilon decay hyper parameters with respect to their effectiveness at different values. The values used are a learning rate of 0.001, batch size at 32 steps, and a ε-decay rate of 0.995, unless used as the target of analysis. Any average discussed is the average rewards over the previous 200 episodes.



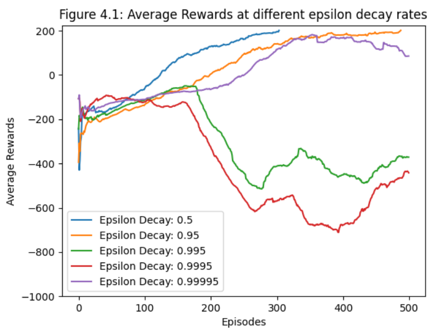


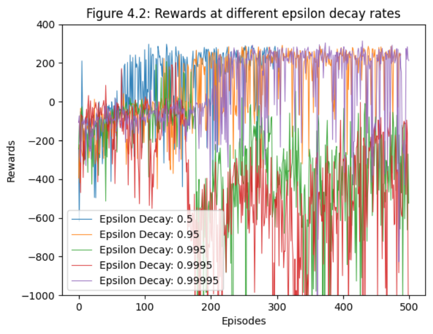
Learning rates, as discussed, affect how much the neural network fits with each progression learning step. The impact of the learning rate is demonstrated in figures 2.1 and 2.2. It can be noted that the two higher learning rates, 0.005 and 0.01, did not allow the gradient descent to successfully converge on the right values. This is likely because at each learning step, the neural network backpropagates too quickly during the exploration phase (high ε value) and is unable to find the local minima. Although 0.0001 allowed the agent to learn correctly, learning rates .0005 and .001 delivered a faster learning process to where the near-optimal policy was achieved earlier, approximately 200 episodes quicker. One noteworthy phenomenon is the behavior that is seen in the learning rate .0001 where the learning starts off slow but suddenly jumps up to 200. This is likely due to the agent's initial intuitive pursuit to reduce negative gains because it does not yet know how to achieve successful landings. After reducing losses it naturally increases its likelihood of landing where it suddenly encounters a episode of a successful +200 reward. At this point, this reward propagates throughout the q-table (neural network) and the agent is able to quickly adapt to a new policy.





Figures 3.1 and 3.2 show the effectiveness of a well sized batch size. All values appeared to eventually reach a correct policy for the lander but at different rates and patterns. The largest batch size, 256 reached an average reward of 200 but took the most amount of time as each learning episode would have to iterate over 256 steps. Figure 3.2 is rather messy but the general behavior is reflected in figure 3.1 that shows the moving average. It can be noted that these averages and the rewards appear to be sporadic as if the agent unlearns information. This is likely due to the seed chosen but may also be attributed to the dependency on the memory buffer. Since the batch size is small it is less likely to make full use of all the stored data. Although, an inappropriately large batch size may overfit due to its repeated use of the limited experiences. After so many steps, the memory buffer will become completely new as it replaces old steps with newly experienced steps. The small batch sizes will learn humbly from the growing buffer, but, as the buffer saturates and becomes replaced, the small batches will begin to learn more and realize the true values of all those states. Thus, it will correctly associate negative rewards with the states it previously considered as reasonable and close in on more correct approximations.





The final hyperparameter that is discussed is the effect of ε decay rates. Figures 4.1 and 4.2 show the behavior effect from varying ε decay rates. These figures show a clear threshold line where epsilon decay rates that are around 0.9995 or larger may not be optimal. Although they may reach the right policies in the long run, they demand a lot of data as an extensive exploration phase will provoke overfitting. Only tons more data may potentially correct the improperly fitted neural network. What was surprising is how a small ε-decay rate of 0.5 allowed the agent to still work excellently. It would be expected that a too small of an ε decay rate would limit the exploration of the state space. This is likely avoided due to the intermittent learning where between every learning step there are 5 exploration steps. This effectively fills the buffer with exploration states and allows each learning step to explore so much more than just the limited states that are confined by an ε-greedy strategy with a small ε-value.

PITFALLS AND PROBLEMS

The DQN algorithm is elegant as the complexity of the algorithm does not change very much across problems. Adapting a DQN agent from a *cartpole* problem1[1] was very useful but unveiled the agent's sensitivity to these hyperparameters. Adapting the right parameters took a long time and knowing where to correctly update dynamic parameters brough many difficulties. Originally, the agent evoked the ε decay after each episode but later, the agent implemented the ε decay after each learning step which compelled the agent to more appropriately balance exploration and learning. But trusting the original algorithm developed for cartpole, inspired patience which was rewarded with a successful agent.

FUTURE EXPLORATION

Each experiment leads to more questions and compels me to wish I had explored a greater domain space for each experiment. The figures demonstrate the issues with using a too large or too small of a hyperparameter but not both. It would have been interesting to see how the agent would have behaved with very large batch sizes and smaller ε decay rates. Additionally, it would have been uniquely useful to implicate agent efficiency with respect to its training time. The number of episodes required to reach a 200 average show the learning efficiency but don't reflect the actual nominal time required for training. Although some hyperparameters may converge quicker with respect to episodes, it may not be reciprocated in its execution time.

There were several more parameters that could have been explained that were not mentioned in the discussion above. This DQN analysis focused on the reinforcement learning side of the algorithm and with exception to the learning rate, it did not consider the parameters of the neural network. The learning rate is not unique to the neural network but rather a strategy inherent to reinforcement learning. The activation functions, beta values, epsilon values for the Adam optimizer, and network depth/breadth sizes are many parameters that could drastically impact the learning and could be incredibly valuable to understand. This discussion could extend so much further in all areas of the paper but the exploration of the learning rate, batch size and epsilon decay rate set up a great understanding to design future agents for all types of environments.

[1]Open AI Gym, Cartpole Environment, https://gym.openai.com/envs/CartPole-v0/

APPENDIX

This appendix is not essential to the paper but provides higher fidelity figures for TAs and graders if needed.

Figure 1.1: Average Rewards for 100 Episodes

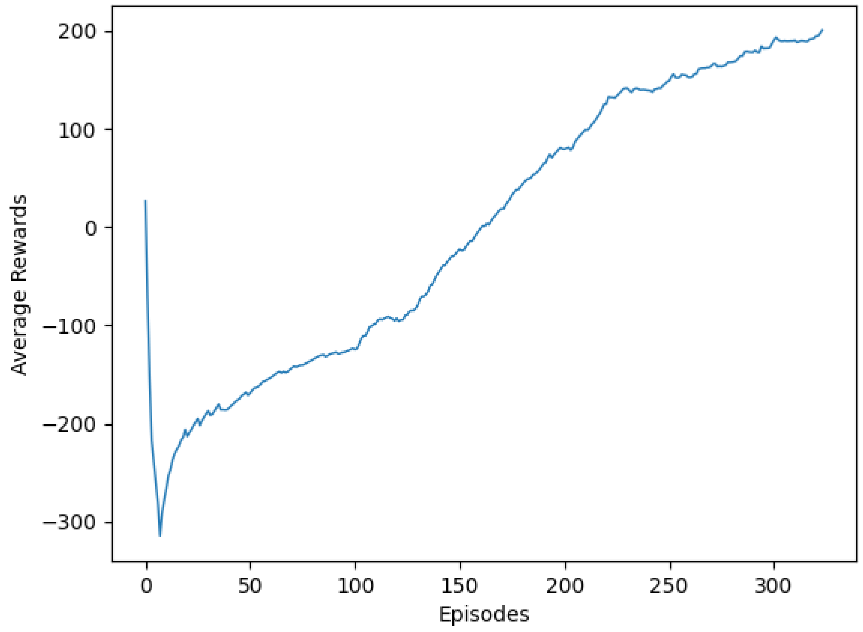
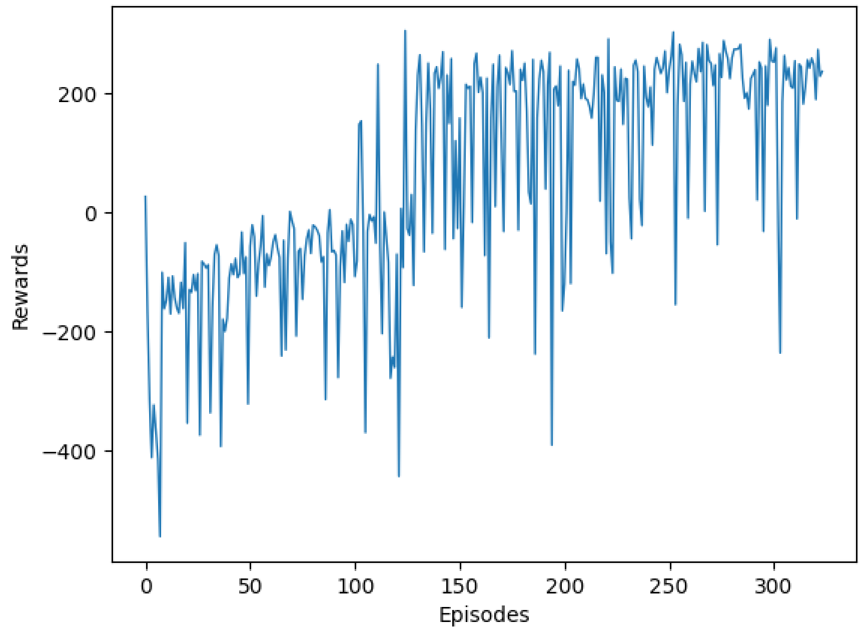


Figure 1.2: Total Reward per Episode



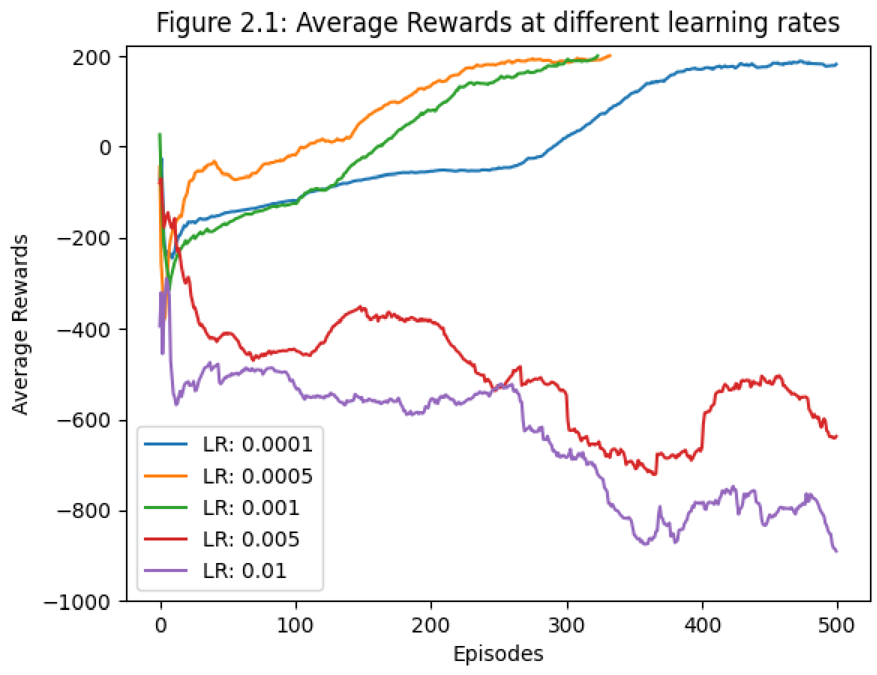


Figure 2.1: Average Rewards at different learning rates
Average Rewards
Episodes
200
0
−200
−400
−600
−800
−1000
0
100
200
300
400
500
LR: 0.0001
LR: 0.0005
LR: 0.001
LR: 0.005
LR: 0.01

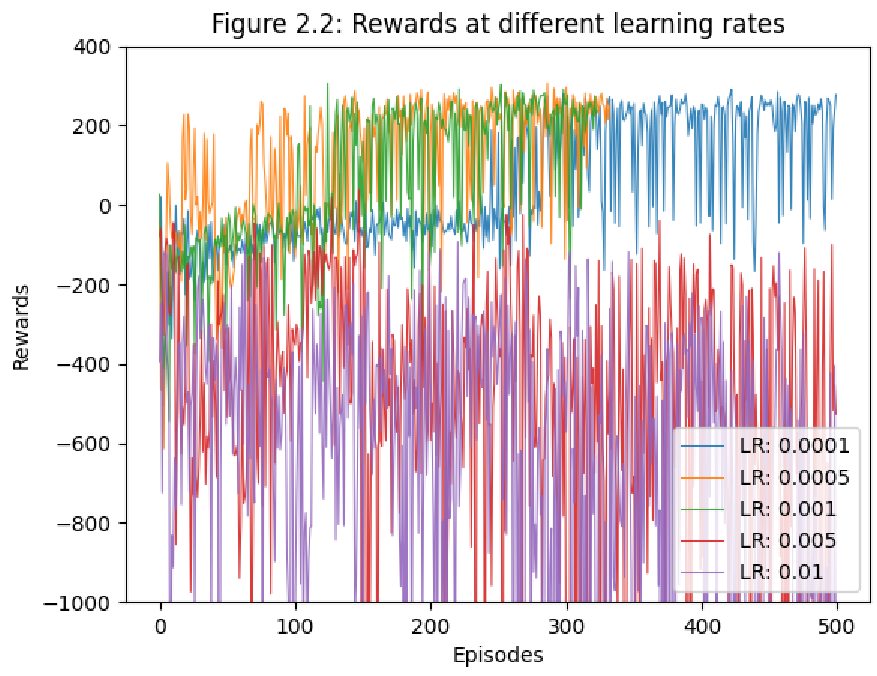


Figure 2.2: Rewards at different learning rates
Rewards
Episodes
400
200
0
−200
−400
−600
−800
−1000
0
100
200
300
400
500
LR: 0.0001
LR: 0.0005
LR: 0.001
LR: 0.005
LR: 0.01

Figure 3.1: Average Rewards at different batch sizes

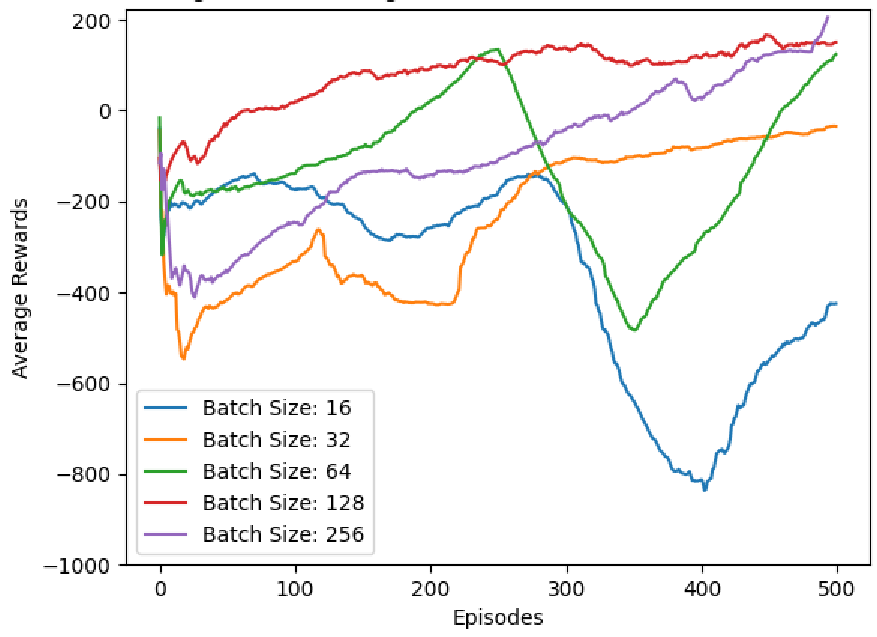


Figure 3.2: Rewards at different batch sizes

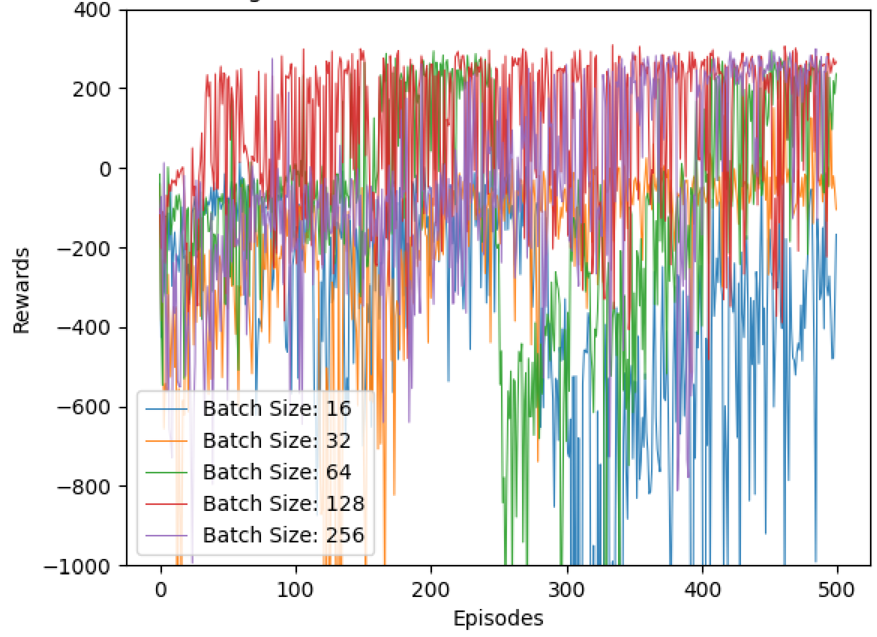


Figure 4.1: Average Rewards at different epsilon decay rates

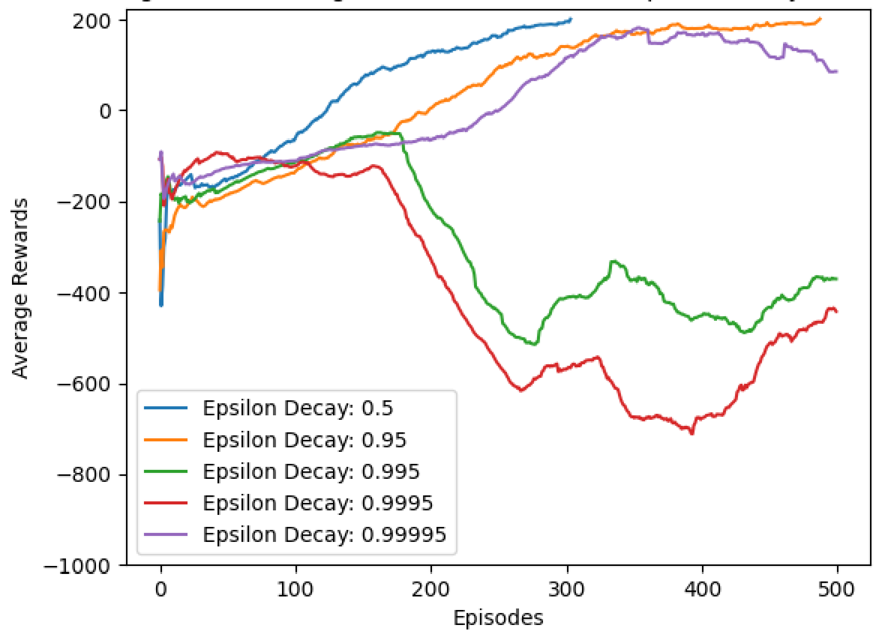
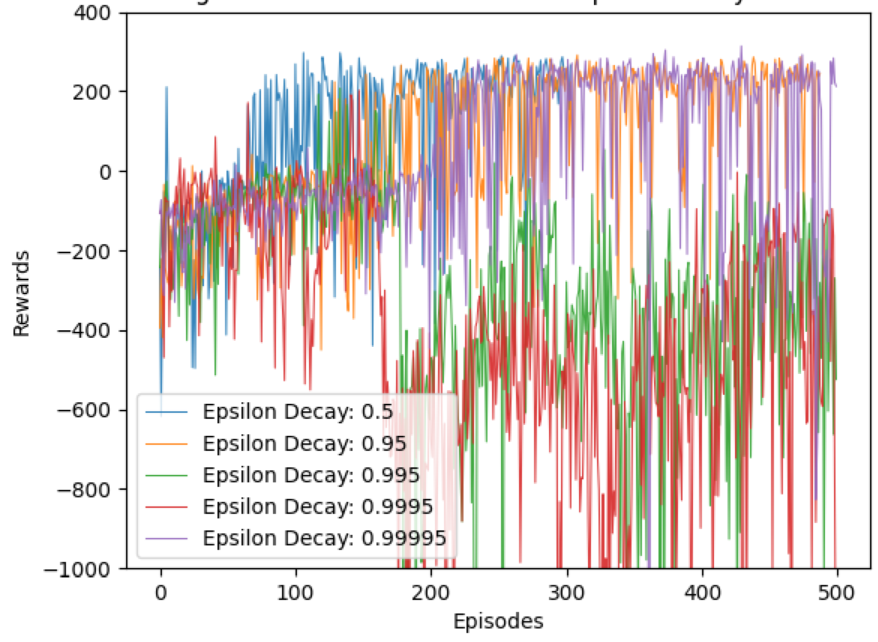


Figure 4.2: Rewards at different epsilon decay rates


Rewards
Episodes
Epsilon Decay: 0.5
Epsilon Decay: 0.95
Epsilon Decay: 0.995
Epsilon Decay: 0.9995
Epsilon Decay: 0.99995