Homework 3 – Reinforcement Learning

Submitted in partial fulfillment of the requirements for the course of

CMSC818B – Decision Making for Robotics

By

Adheesh Chatterjee

Course Faculty
Prof. Pratap Tokekar

A. JAMES CLARK
SCHOOL OF ENGINEERING

INTRODUCTION

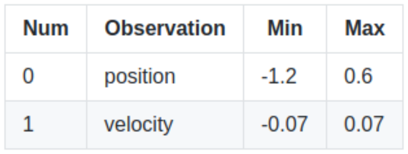
The aim of this project is to use reinforcement learning to solve the MountainCar-v0 problem from OpenAI gym.
Here the state of the agent is given by its position and velocity. The agent can take three discrete actions. Two methods of Q Learning were implemented to solve this problem. Initially, a Q Table was formed by discretizing the state space. Later, a Neural Network was used as a functional approximator. The results of both approaches are discussed below.
Finally, BipedalWalker-v2 problem and MountainCarContinuous-v0 from OpenAI gym were also solved using a Neural Network as a functional approximator. The result is discussed below.

PROBLEM 1
APPROACH 1 – Q table

Q learning was implemented using a Q Table. This was done by discretizing the state space as can be seen in code file (init.py).

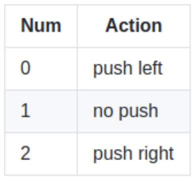
Here first the state space and action space are examined. We observe that the state space is continuous and the action space is discrete.

state space is



Num	Observation	Min	Max
0	position	-1.2	0.6
1	velocity	-0.07	0.07

action space is



Num	Action
0	push left
1	no push
2	push right

So the state space is discretized by multiplying the position values by 10 and the velocity values by 100 and then rounding both numbers so as to get integers.
Thus a Q table is formed with the position values, velocity values and the action.

An epsilon greedy policy is used to choose a random value from the action space epsilon times and use the Q table to predict the next action value 1-epsilon times.

Finally, we check that if the episode is complete and the position of 0.5 is reached by the car, we give reward. Else, we update our Q values using the Q learning formula

$$\underbrace{NewQ(s,a)}_{\text{New Q value for that state and that action}} = \underbrace{Q(s,a)}_{\text{Current Q value}} + \underbrace{\alpha}_{\text{Learning Rate}} [\underbrace{R(s,a)}_{\text{Reward for taking that action at that state}} + \underbrace{\gamma}_{\text{Discount rate}}\, \underbrace{max\, Q'(s',a')}_{\text{Maximum expected future reward given the new s' and all possible actions at that new state}} - Q(s,a)]$$

We use alpha as 0.2, gamma as 0.9 and implement an epsilon decay to improve our exploration-exploitation strategy.

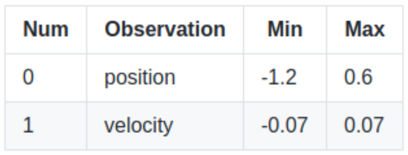
Finally, we log all our rewards and epsilon values as the episodes progress, and train the algorithm. We see that it converges very quickly with minimal computational power.

APPROACH 2 – Neural Network

Q learning was implemented using a NN as a functional approximator. This was done by using Keras as can be seen in code file (final_dqn.py).

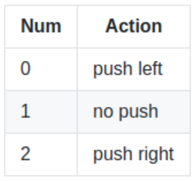
Here first the state space and action space are examined. It is observed that the state space is continuous and the action space is discrete.

state space is



Num	Observation	Min	Max
0	position	-1.2	0.6
1	velocity	-0.07	0.07

action space is



Num	Action
0	push left
1	no push
2	push right

First we get the current state from the environment and start a timer, to record the time of each episode. We then run each episode for a maximum of 200 steps. To speed up the training, the rendering of the environment is only done in the last 2 episodes.

Epsilon Policy
First we implement epsilon greedy strategy to choose our action. Here, essentially, we choose a random value from the action space epsilon times and use our model to predict the next action value 1-epsilon times. We also implement an epsilon decay policy during our training to improve our exploration-exploitation strategy.
Initially we want our agent to explore our environment more and so we let it choose random actions so that it covers more of the state space. Eventually, however, we use our NN model to predict the action values, so that the agent always follows an optimized pattern in selecting the action and thus reaching the goal state and maximizing the reward.

Network Architecture
Now the Keras library is used to create a neural network as it is easier to implement a basic NN model. It uses tensorflow as a backend and is extremely powerful. Now the input to the network is going to be our state space and the output will be the predicted action. Here the input dimension is 2 (position and velocity).

The first hidden layer has 64 neurons. Essentially, there is no way to determine the best number of hidden units without training several networks. If you have too few hidden units, you will get high training error and high generalization error due to underfitting and high statistical bias. If you have too many hidden units, you may get low training error but still have high generalization error due to overfitting and high variance. However, after some research on the various implementations, 64 neurons in the hidden layer seemed like a good assumption.

We then use the ReLU activation function. Most units in neural networks transform their net input by using a scalar-to-scalar function called activation function. Activation functions for the hidden units are needed to introduce nonlinearity into the network. Without nonlinearity, hidden units would not make nets more powerful than just plain perceptrons. Firstly, ReLU is non-linear, so we can easily backpropagate the errors and it doesnt activate all the neurons at the same time. Thus it is more efficient and easy for computation.

It is passed through another hidden layer of 64 neurons which is again activated by the ReLU function. Finally the output layer is the same size as our action space and we use a linear activation function for this, then the model is compiled using the mean squared error loss function and the Adam optimizer with learning rate set to alpha and we have the final model setup to use for our DQN

Reward Setup
We change the reward setup of our model to make it more efficient. So the reward is setup such that if the next state reached is more than the current state and we are on the positive side of the starting point, a reward of +15 is given.
Similarly, if the next state reached is less than the current state and we are on the negative side of the starting point, a reward of +15 is given.
For every time step a negative reward of -10 is given.

This essentially forces our model to always make sure a new state is being discovered so that our model is always forced to explore a new unexplored state, so that it can get a net reward of +5.

Finally a reward of 10000 is given if it reaches the goal position, this forces our model to try to reach the goal position every time.
Eventually our model learns that it needs to build momentum before it can reach the goal position and maximize its reward.
Since it will always get a positive overall reward if it explores a new state, the car always tries to move and thus the running average of the rewards is never negative.

Hyperparameters
We select learning rate as 0.002. Since the learning rate determines to what extent newly acquired information overrides old information, we use a value of 0.002 to as we want our agent to learn from the previous information it gained. Esp as we are using an epsilon greedy policy, our initial exploration shouldn't be explicitly learned by the agent as that will impact our model severely.

We select discount factor of 0.95. Since the discount factor determines the importance of future rewards, we select a high discount factor so that our agent will strive for long-term high reward.

We use the mean squared error as our loss function in the neural network. Mean Squared Error is an estimator which measures the average of error squares i.e. the average squared difference between the estimated values and true value. Since we can use this measure of the discrepancy between the NN output value and the target value, this helps us understand how close our NN approximates the Q value.

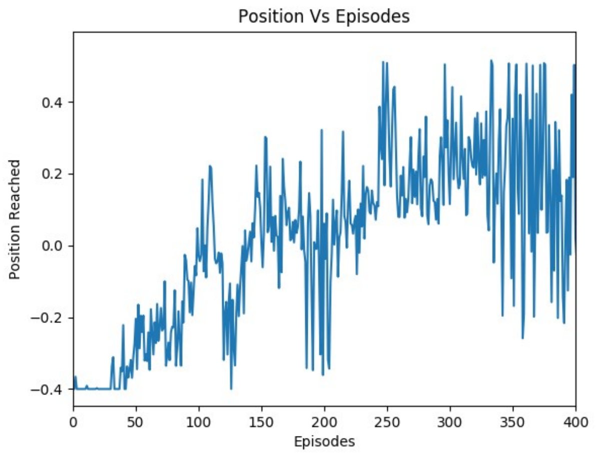
We use a time step as our termination condition. The episode ends either when we reach the goal position or if 200 time steps (iterations) are reached. Finally, if the time step condition is reached, we update our model.

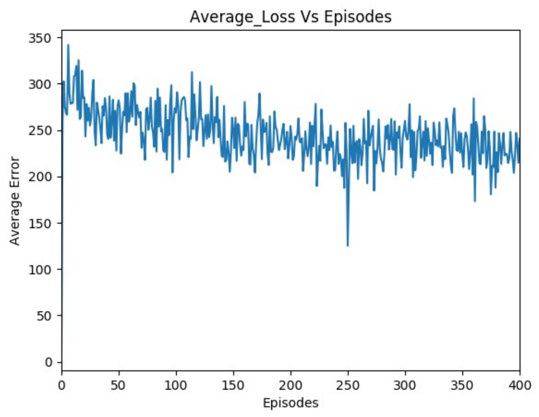
Batch Training
Now we are finally ready to train our model. We first store our current state, action, reward, next state, and done in a list for every time step.
We then create a minibatch which randomly samples our storage with a batch size of 128. We use our model to predict the Q value, as long as our algorithm is not done, we then use the Q learning algorithm to train our model.

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

We get the value of Q(s',a) from our model and use it to update the Q(s,a) value.
Here we also ensure to implement our epsilon decay policy as discussed previously.

RESULTS
Finally we see that our algorithm converges within 400 episodes with a batch size of 128. This is observed by the large sudden increase in our reward plot. For the first 320 episodes, the rewards increase linearly. For the next 80 episodes, we see a sudden increase in the reward, this implies that we hit the goal state. The algorithm then optimizes to make sure it maximizes the overall reward it gets from the entire episode.

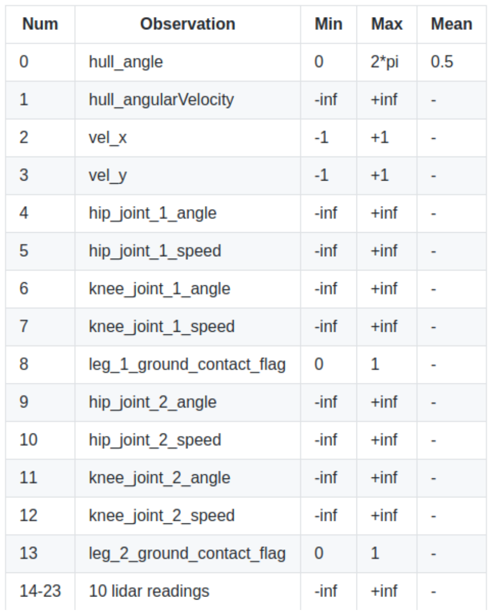




PROBLEM 2
BipedalWalker-v2/ MountainCarContinuous-v0

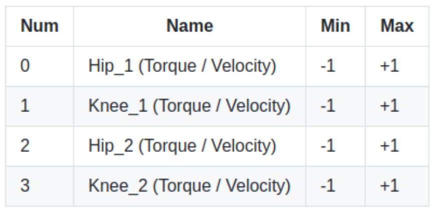
We now setup our model to solve the problems in continuous discrete as well as action space.
We work primarily with the BipedalWalker-v2 and hope our model can also work just as well with the mountainCarContinuous.
Now since we had to change our reward function in mountainCar-v0, we see that we cannot use the same NN model as it will take longer to fit, so **we create a whole new architecture that will basically generalize to any OpenAI gym problem, which has continuous state space and continuous action space.**

state space is



Num	Observation	Min	Max	Mean
0	hull_angle	0	2*pi	0.5
1	hull_angularVelocity	-inf	+inf	-
2	vel_x	-1	+1	-
3	vel_y	-1	+1	-
4	hip_joint_1_angle	-inf	+inf	-
5	hip_joint_1_speed	-inf	+inf	-
6	knee_joint_1_angle	-inf	+inf	-
7	knee_joint_1_speed	-inf	+inf	-
8	leg_1_ground_contact_flag	0	1	-
9	hip_joint_2_angle	-inf	+inf	-
10	hip_joint_2_speed	-inf	+inf	-
11	knee_joint_2_angle	-inf	+inf	-
12	knee_joint_2_speed	-inf	+inf	-
13	leg_2_ground_contact_flag	0	1	-
14-23	10 lidar readings	-inf	+inf	-

action space is



Num	Name	Min	Max
0	Hip_1 (Torque / Velocity)	-1	+1
1	Knee_1 (Torque / Velocity)	-1	+1
2	Hip_2 (Torque / Velocity)	-1	+1
3	Knee_2 (Torque / Velocity)	-1	+1

Epsilon Policy
First we implement epsilon greedy strategy to choose our action. Here, essentially, we choose a random value from the action space epsilon times and use our model to predict the next action value 1-epsilon times. But since the action space is also continuous, we choose a random set of actions (list of 4 actions between -1 and 1) epsilon times and use our model to predict the next action 1-epsilon times.
We also implement an epsilon decay policy during our training to improve our exploration-exploitation strategy.

Initially we want our agent to explore our environment more and so we let it choose random actions so that it covers more of the state space. Eventually, however, we use our NN model to predict the action values, so that the agent always follows an optimized pattern in selecting the action and thus reaching the goal state and maximizing the reward.

Network Architecture
Now the Keras library is used to create a neural network as it is easier to implement a basic NN model. It uses tensorflow as a backend and is extremely powerful. Now the input to the network is going to be our state space and the output will be the predicted action. Here the input dimension is 24, the size of the state space

The first hidden layer has 400 neurons. Essentially, there is no way to determine the best number of hidden units without training several networks. If you have too few hidden units, you will get high training error and high generalization error due to underfitting and high statistical bias. If you have too many hidden units, you may get low training error but still have high generalization error due to overfitting and high variance. However, after some research on the various implementations, 400 neurons in the hidden layer seemed like a good assumption.

We then use the ReLU activation function. Most units in neural networks transform their net input by using a scalar-to-scalar function called activation function. Activation functions for the hidden units are needed to introduce nonlinearity into the network. Without nonlinearity, hidden units would not make nets more powerful than just plain perceptrons. Firstly, ReLU is non-linear, so we can easily backpropagate the errors and it doesnt activate all the neurons at the same time. Thus it is more efficient and easy for computation.

It is passed through another hidden layer of 300 neurons which is again activated by the ReLU function. Finally the output layer is the same size as our action space and we use a linear activation function for this, then the model is compiled using the mean squared error loss function and the Adam optimizer with learning rate set to alpha and we have the final model setup to use for our DQN.

Reward Setup
We do not change the reward setup in this case. Reward is given for moving forward, total 300+ points up to the far end. If the robot falls, it gets -100. Applying motor torque costs a small amount of points.

Hyperparameters
We select learning rate as 0.001. Since the learning rate determines to what extent newly acquired information overrides old information, we use a value of 0.002 to as we want our agent to learn from the previous information it gained. Esp as we are using an epsilon greedy policy, our initial exploration shouldn't be explicitly learned by the agent as that will impact our model severely.

We select discount factor of 0.98. Since the discount factor determines the importance of future rewards, we select a high discount factor so that our agent will strive for long-term high reward.

We use the mean squared error as our loss function in the neural network. Mean Squared Error is an estimator which measures the average of error squares i.e. the average squared difference between the estimated values and true value. Since we can use this measure of the discrepancy between the NN output value and the target value, this helps us understand how close our NN approximates the Q value.

We use a time step as our termination condition. The episode ends either when we reach the goal position or if 200 time steps (iterations) are reached or if the robot body touches the ground.

Batch Training
Now we are finally ready to train our model. We first store our current state, action, reward, next state, and done in a list for every time step.

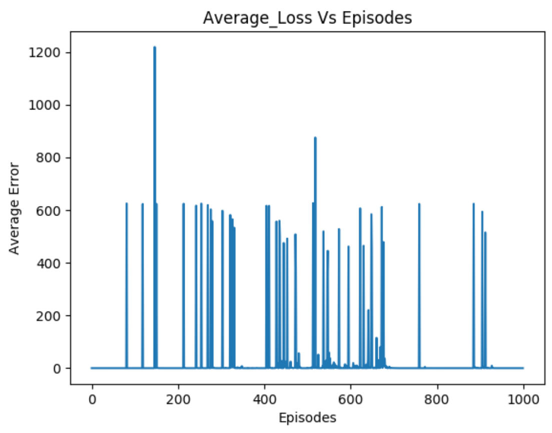
We then create a minibatch which randomly samples our storage with a batch size of 16. We use our model to predict the Q value, as long as our algorithm is not done, we then use the Q learning algorithm to train our model.

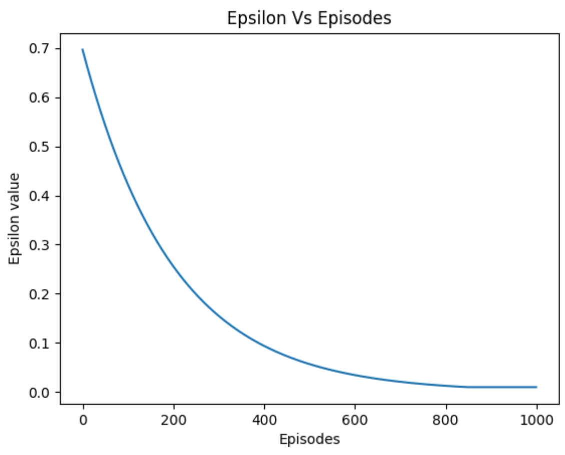
$$Q(s,a) = (1-\alpha)\,Q(s,a) + \alpha\,Q_{obs}(s,a)$$
$$\text{where, } Q_{obs}(s,a) = r(s,a) + \gamma \max_{a'} Q(s',a')$$

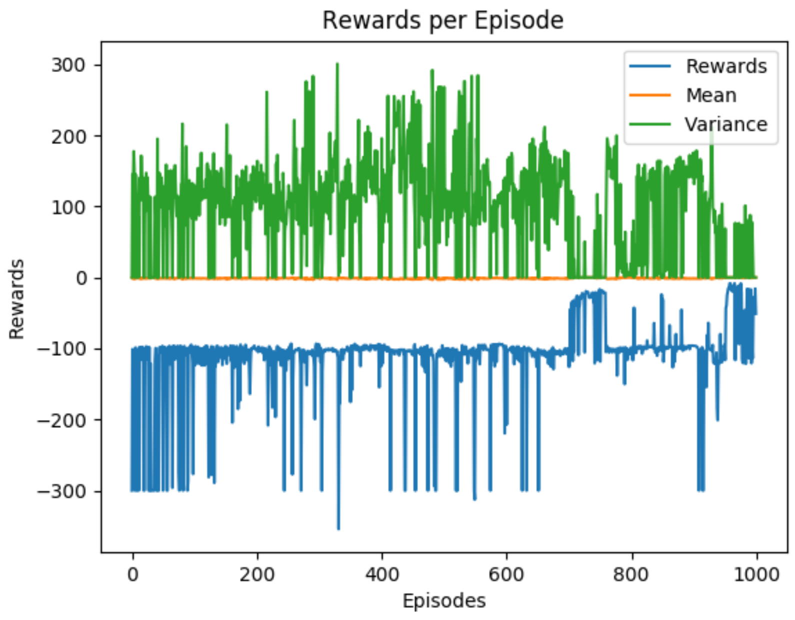
We get the value of Q(s’,a) from our model and use it to update the Q(s,a) value.
Here we also ensure to implement our epsilon decay policy as discussed previously.

RESULTS

Finally we see below that our algorithm doesnt converge within 1000 episodes with a batch size of 16. However, the maximum rewards increases from -300 to a maximum of -8 as seen, thus we can conclude that if trained longer, the algorithm will eventually converge and give us a good output.
We also observe that at around 1000 episodes, the rewards are generally below -100, which is a significant gain. We also see, that the variance between the rewards also decreases significantly, thus our algorithm gets better and with more iterations should give us a very good output. Our loss graph shows us that the average error we get per output is almost 0 for every episode, only sometimes, do we get a high loss value. We also observe that as the iterations increase, the number of times we get a high loss decreases, this implies our agent is learning well and with more iterations should give good output.
Finally, our epsilon graph shows epsilon decay.







FINAL NOTES
The videos for the different approaches are attached and can also be viewed using the 2 tester codes provided. This code although untested for a large number of episodes on MountainCarContinuous-v2, does seem to work for short episode lengths and hence should work for more iterations.

Due to the deadline, the code couldn't be run for more than a 1000 iterations, but the generalized model created for continuous action space and continuous state space, can work well with mountainarContinuous-v2 and should generalize to all problems of this nature.