

# Machine Learning Engineer Nanodegree

---

## Capstone Project

---

Anand Saha <anandsaha@gmail.com>  
August 16th, 2017

## I. Definition

### Project Overview

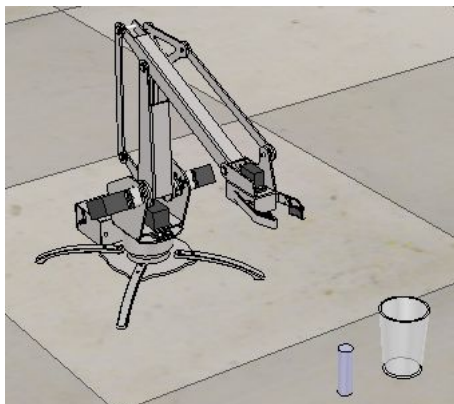
This project explores the process of training a robotic arm to accomplish a given task. The solution has been drawn from the branch of Machine Learning called Reinforcement Learning.

The application of reinforcement learning to robotics is of special significance since it can be very hard otherwise to hand code all the actions a robot should take to execute a task.

“Reinforcement learning offers to robotics a framework and set of tools for the design of sophisticated and hard-to-engineer behaviors.” - J. Kober, J. Andrew (Drew) Bagnell, and J. Peters in Reinforcement Learning in Robotics: A Survey [2]

Our goal is to make the robotic arm learn the shortest way to accomplish the task given the inputs and constraints as mention below.

### Problem Statement



V-REP [1] is a robot simulation environment. Various robotic models are available in V-REP's catalog, ranging from mobile robots like humanoids, hexapods and vehicles, to stationary robots like robotic arms and conveyer belts. These models can be executed in the V-REP environment and interacted with remotely using V-REP's remote client library. Among other applications, V-REP acts as an excellent platform for algorithm development of reinforcement learning approaches.

In this project, a *Uarm with Gripper* model is taken and taught how to hold a given object and place it in a given location. This is a common use case in many industrial and home applications. However, traditionally robots have been programmed assuming a controlled environment, making it impossible for them to adapt to new environments.

In this project, classic *Q-learning* will be used as a learning technique. It is an *online*, *model-free* and *off-policy* reinforcement learning approach. The arm will have to learn to locate the object, grab it, lift it, maneuver it to top of the bin and release it. Appropriate reward strategy needs to be designed to enable the agent to do so.

The task is episodic in nature and is considered successfully finished if the robot arm puts (drops) the object in the bin.

## Metrics

In reinforcement learning, the agent interacts with the environment by taking actions, and in return receive a reward (+ve or -ve) and new state of the environment. This loop continues till a terminal state is reached, which can either signal failure or success. A run of such a loop is called an episode. We run as many episodes as needed for the agent to successfully learn how to accumulate maximum rewards and eventually accomplish the task.

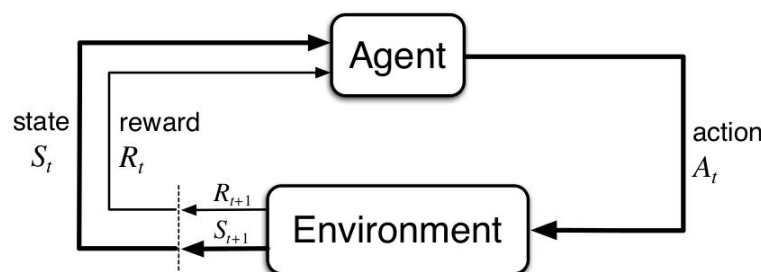


Fig: The reinforcement learning feedback loop (Sutton & Barto 2017)

The following metrics will be tracked for gauging effectiveness of the solution:

- **Success/failure per episode:** When an episode terminates, it will either have failed in its goal or succeeded. Initial episodes will see failures because of novelty of the task and gradually should start succeeding. This metric will let us know how many episodes it took for training before the arm learnt the task.
- **Cumulative reward per episode:** How much reward did the agent collect before the episode ended? We should see a gradual increase in accumulated reward: and it should move from negative to positive. Even when positive, we will see failures initially if the episode ends in one of the fail states. Gradually it should start succeeding.
- **Actions per episode:** The challenge of the arm is to take the least number of actions to accomplish the task. This metric will track that.

## II. Analysis

### State Space, Actions and Rewards Exploration

To solve a reinforcement learning problem, it is critical to segregate the agent from its environment and define the *actions*, *states* and *rewards*.

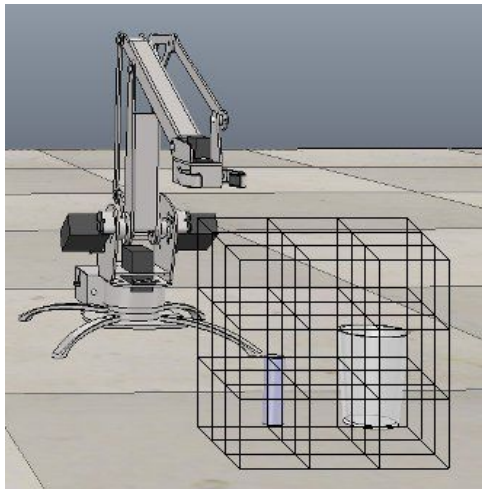
In our solution:

- *The Agent* is the entity which controls the robotic arm. It implements the learning algorithm, takes the actions, perceives the state and receives the rewards.
- *The Environment* consists of the robotic arm, the object and the bin. It reacts to actions taken by the agent.

#### The state space

The state space of our environment consists of:

- a. Position of the gripper of the robotic arm: x, y, z coordinates
- b. Position of the object: x, y, z coordinates
- c. State of the gripper: engaged or disengaged



The active space of the problem was discretized for this project. As can be seen in the suggestive figure, the space around the arm, object and the bin was discretized along the x, y and z axis.

The min and max locations along x, y and z axis as well as the step size has been parameterized and can be set in code.

Therefore, the number of states the environment can be in, is the product of (a) the number of discrete locations where the object can be located, (b) the number of discrete locations where the arm can be positioned and (c) number of states of the gripper

(engaged/disengaged).

#### The actions

There are two types of actions possible in our environment:

- a. Move the arm to a specific location in the active space

b. Engaged/disengage the gripper

Therefore, the total number of actions possible is the product of (a) the number of discrete locations where the arm can be in and (b) number of states of the gripper (engaged/disengaged).

### The rewards strategy

Devising a reward strategy is challenging and was arrived at by observing multiple iterations for this specific problem.

Sr. No.	Condition	Value
1	The environment reached invalid state	-10
2	The arm topples the object	-10
3	The arm displaces the bin	-10
4	The grip was engaged when there was no object in it	-3
5	With the object in grip, the grip was disengaged. But the object did not fall in the bin.	-3
6	The grip could grab the object for the first time in an episode	5
7	The grip could grab the object and drop it in the bin (objective achieved)	10
8	Default reward	-1

Rewards (1), (2) and (3) were most negative because the arm movement would cause the environment to reach a state from where no further actions would be possible.

Rewards (4) and (5) were put in place to suggest the arm what not to do.

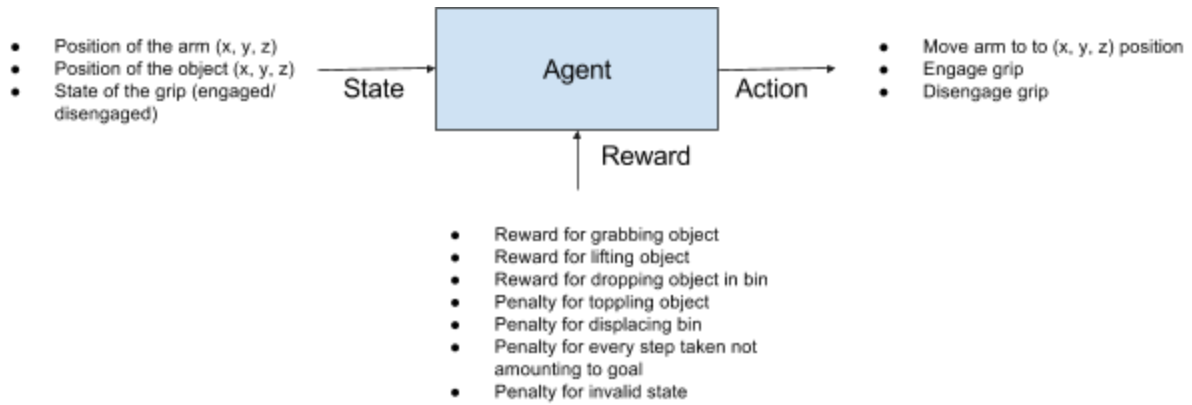
Reward (6) was to encourage the gripping of the object.

Reward (7) was the maximum that any action could get. This would also suggest achieving the goal.

Reward (8) was the default any action would get. A negative value would encourage the arm to reach the goal in as less actions as possible.

## The feedback loop

With the above analysis, we end up with this arrangement for this problem:



## Exploratory Visualization

Here are the states that the environment can get into that needs to be handled.

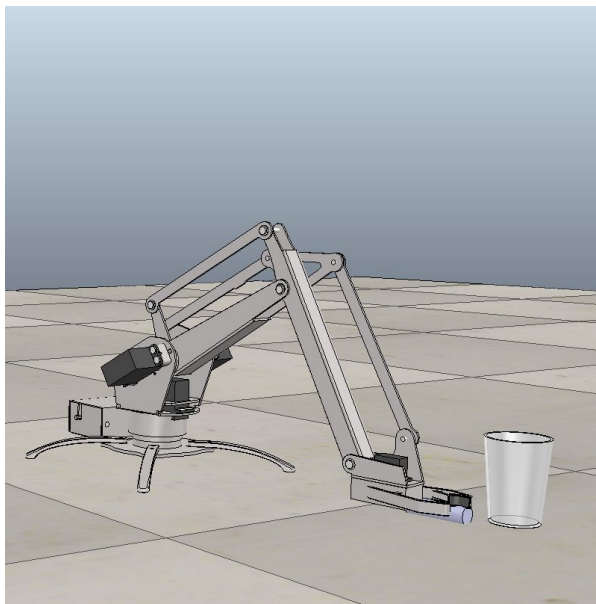


Fig: The arm topples the object

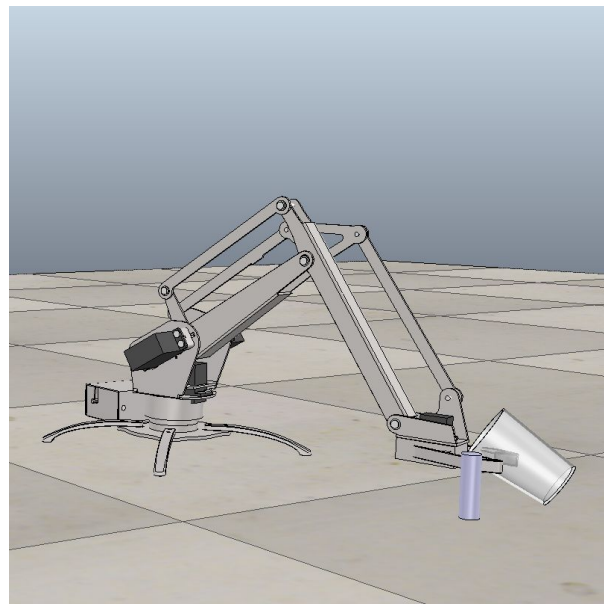


Fig: The arm displaces the bin

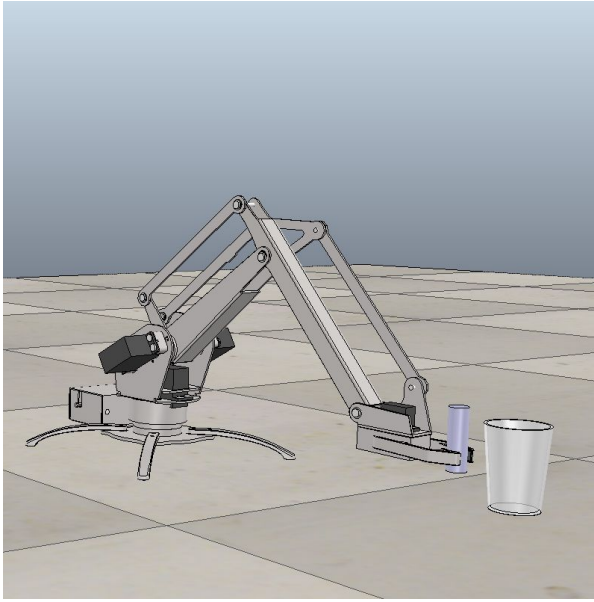


Fig: The grip could grab the object for the first time in an episode

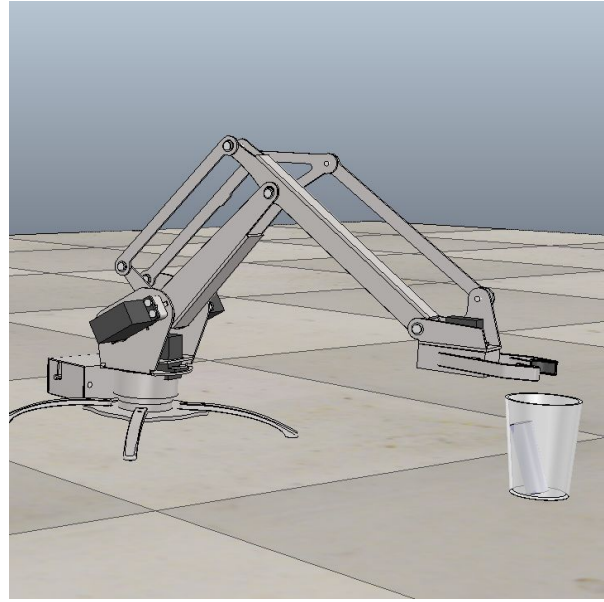


Fig: The robotic arm accomplished the task

## Algorithms and Techniques

Classic Q-Learning technique was used to solve this problem. Several approaches exist to solve reinforcement learning problems, but Q-learning has some useful properties:

- Unlike dynamic programming, Q-learning does not need complete model of the environment (state transition probabilities).
- Unlike dynamic programming, Q-learning learns from experience.
- Unlike monte carlo, Q-learning doesn't need to wait for the episode to end to update the values. It can do so online as the actions are taken and rewards obtained.
- Can also be applied to non-episodic and continuous tasks.

Q-learning however typically needs lots of episodes to be executed for it to learn.

Given (a) a state, (b) an action taken in that state, (c) the reward obtained due to that action, and (d) the resultant new state, q-value of current state-action pair is derived in the following way:

```
def update_q_table(self, state, action, reward, state_new):
    q_current = self.q_table[state, action]
    error = reward + self.discount * np.max(self.q_table[state_new]) - q_current
    self.q_table[state, action] = q_current + self.learn_rate * error
```

Fig: Calculating Q-value of a state-action pair

It is interesting to note here that the q-value depends on that action in the next state which yields the max q-value, even if that action may not be taken. This differentiates it from SARSA.

Two important tunable parameters appear in this formula:

- Discount factor: Also called the gamma factor, this is the weightage given to the max q-value possible in the next state.
- Learning rate: This determines by how much the current q-value needs to be adjusted for the given error.

These parameters are tunable in the code.

### **Exploitation vs Exploration: Epsilon-Greedy with decay**

Q-learning is an *off-policy* algorithm. This means that it learns the values of a state-action pair by choosing and taking actions on its own, and doesn't depend on any policy.

However,

- If the random actions are chosen, it will take enormous time to converge and may not yield optimal solution. Choosing actions randomly is called exploration, since we choose to ignore the current best action in favour of a probable better future action.
- If only the best actions (based on the best q-value available at current state) is always chosen, we may not get the optimal solution. Choosing the best action available is called exploitation, since we choose to exploit our current knowledge base of what is known to be best.

With the epsilon-greedy approach, we allow the agent to sometimes explore rather than exploit. The rate of exploration is determined by a percentage called epsilon.

In this solution, we also decay the epsilon percentage value with each episode to allow more explorations at the start of the learning but reduce the rate as the learning progresses.

### **When to stop learning**

The learning process would keep playing episodes and update the Q table in the process. Four conditions were chosen to decide when to stop the learning process. We would stop learning after an episode if:

- a. Minimum episodes: A minimum number of episodes were executed (parametrised)
- b. Success state: *And* When an episode would end with task accomplished
- c. Least number of actions: *And* when the episode would do so with least number of actions



- d. All steps exploitative: *And* when the episode would do so with all actions taken using exploitation (and not exploration)

The last condition is important. It was observed that episodes could accomplish the goal using exploration early on. However, it would be premature to stop the learning, since the exploration might not yield a positive q-value. Such a Q table may not lead to goal state.

## Benchmark

A *uniformly random agent* was chosen as a benchmark. Though it is highly unlikely that the benchmark agent will be able to accomplish the task, it will be interesting to compare the cumulative reward collected per episode, and how the actual solution improve upon its reward compared to the random agent.

A uniformly random agent was simulated by being *explorative* all the time.

## III. Methodology

### Data Preprocessing

This being a reinforcement learning approach will not need any specific dataset to start with. The solution involves using Q-learning to generate the optimal policy. Hence over time, we will have a Q-table which will capture the optimal values for a state-action pairs.

### Framework setup

V-REP was chosen to simulate a reinforcement learning environment. Towards that,

- V-REP version 3.4 ref 1 was installed.
- A scene with the robotic arm, a cylindrical object and a bin was created.
- Custom child scripts were written to act as hooks into the framework.
- V-REP python remote client scripts were imported into the project



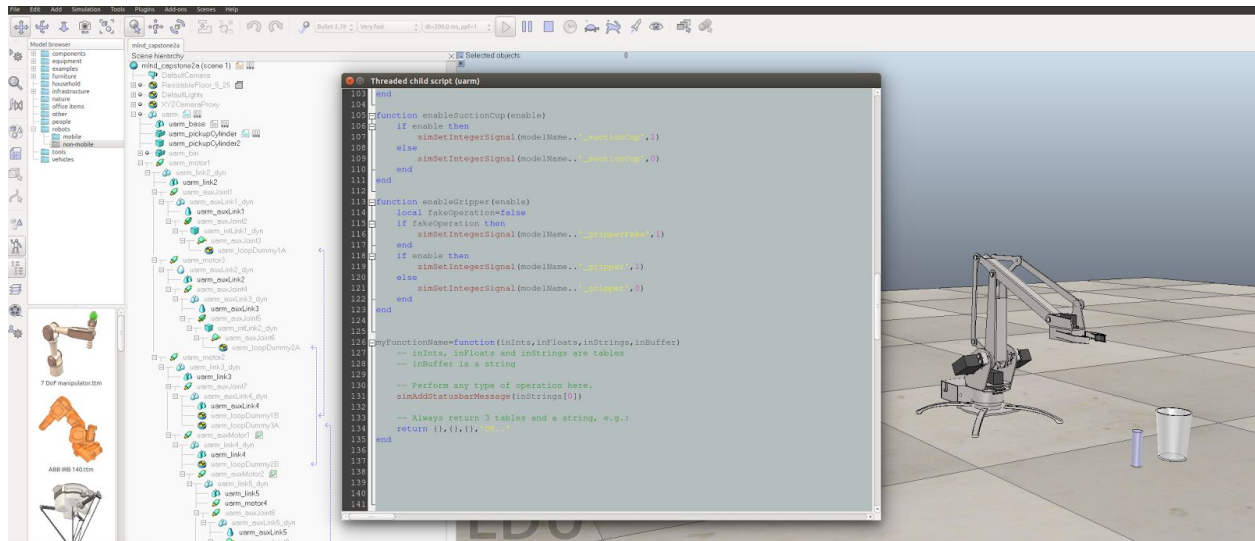


Fig: The V-REP environment

## Implementation

To start the simulation, one would invoke the main.py script with the argument '--train'

```
$ python main.py --train
```

Once training would end, one can test the final model by invoking main.py script with the argument '--test'

```
$ python main.py --test
```

The main loop starts executing episodes and gathering statistics. It checks for the conditions to stop the learning.

```

if train_mode:
    while episodes > 0:
        log_and_display('=====> Episode ' + str(episodes))
        agent.reset()
        total_reward, total_steps, success, total_explorations = agent.execute_episode_qlearn(config.NUM_MAX_ACTIONS)
        episode_num = config.NUM_EPISODES - episodes + 1
        # Reduce exploration rate with each episode
        agent.epsilon = np.maximum(0.0, config.EPSILON - episode_num * config.EPSILON_DECAY)

        stat_file = open(config.PLOT_FILE, 'a')
        stat_file.write("{0}, {1}, {2}, {3}, {4}, {5}\n".format(episode_num, success, total_reward,
                                                                total_steps, total_explorations, agent.epsilon))

        stat_file.close()
        agent.save_qtable()
        episodes -= 1

    if success and total_steps == config.MIN_ACTIONS_EXPECTED \
        and total_explorations == 0 and episode_num > config.MIN_EPISODES_TO_RUN:
        log_and_display('Optimal moves learnt. Terminating training. Now run agent with epsilon as 0.')
        break

```

Fig: Main loop in main.py

The execution of one episode consists of choosing an action, executing that action, and updating the Q-table.

```

def execute_episode_qlearn(self, max_steps: int):
    total_reward = 0
    total_steps = 0

    while max_steps > 0 and not self.env.is_goal_achieved() and not self.env.environment_breached:
        action_id = self.select_action(self.current_state_id)
        reward = self.execute_action(action_id)
        new_state_id = self.env.actionstate_curr['current_state_id']
        self.update_q_table(self.current_state_id, action_id, reward, new_state_id)
        self.current_state_id = new_state_id

        max_steps -= 1
        total_reward += reward
        total_steps += 1

    return total_reward, total_steps, self.env.is_goal_achieved(), self.total_explorations

```

Fig: Execution of one episode in agent.py

While choosing an action, we apply the epsilon-greedy approach. Epsilon is set in the config file. Also, it's value is decayed per episode run so that as time progresses, we decrease exploring.

```

def select_action(self, current_state_id):
    if np.random.uniform() < self.epsilon:
        log_and_display('Exploring...')
        self.total_explorations += 1
        action_id = np.random.choice(self.env.total_actions)
    else:
        log_and_display('Exploiting...')
        action_id = np.argmax(self.q_table[current_state_id])
    return action_id

def execute_action(self, action_id):
    action = self.env.actions[action_id]

    if action[0] == self.env.action_type1:
        log_and_display('Action: Moving claw ' + str(action[1]))
        return self.env.move_arm(action[1], action_id)
    elif action[0] == self.env.action_type2:
        log_and_display('Action: Engaging/Disengaging claw ' + str(action[1]))
        return self.env.enable_grip(action[1], action_id)

```

Fig: Choosing and executing action in agent.py

As shown below, the training of the arm would generate three files:

- log.txt: The execution log with details on how the Q-values were updated
- qtable.txt.npy: The Q-table dumped in file
- episodes.txt: Details of each episode

```

-rw-rw-r-- 1 anand anand 3.3M Aug 15 11:48 log.txt
-rw-rw-r-- 1 anand anand 290M Aug 15 11:48 qtable.txt.npy
-rw-rw-r-- 1 anand anand 55K Aug 15 11:48 episodes.txt

```

Fig: Output files

Following are important files in the project and their purpose.

Python file	Purpose
agent.py	Executes episodes, choses next action, receives reward, updates q-table, loads and saves q-table to file.
environment.py	Exposes the V-REP environment to the agent, allows setting of arm position and grip state, allows fetching position of object and arm, maintains state, calculates and returns reward for every action.
robot.py	Interfaces with V-REP environment using the V-REP python remove client scripts, connects to/disconnects from V-REP env, starts/stops simulation, allows setting position of arm and state of grip, allows fetching position of various actors in the scene.

reward_strategy.py	Calculates rewards based on current state.
config.py	Holds all configurable parameters.
utility.py	Holds misc utility functions.
main.py	Main entry into the code. When executed, initializes the environment and the agent, and starts the learning process.

Table: Files in the project

## Refinement

Some refinements were made since the first cut with each new insight into the problem. These refinements consisted of mostly fine tuning the configurable parameters.

- a. The minimum number of episodes that the agent should execute was increased from 1500 to 2500 to allow it to come up with a robust q-table
- b. Epsilon value was increased from 0.2 to 0.3 to allow more explorations initially with the aim that it would find winning actions faster
- c. Learn rate was increased from 0.3 to 0.5 to allow bigger impact of the error

---

```

9  # Number of episodes to run
10 NUM_EPISODES = 10000
11 MIN_EPISODES_TO_RUN = 2500
12 # Number of max actions to execute per episode (terminate episode if this number is exceeded)
13 NUM_MAX_ACTIONS = 100
14 # Least actions needed to achieve the goal
15 MIN_ACTIONS_EXPECTED = 5
16
17 # Initial Q value to be populated for every state/action pair
18 Q_INIT_VAL = 0.0
19 # Epsilon value (the portion of exploratory actions)
20 EPSILON = 0.3
21 # The amount by which epsilon will be reduced every episode (to reduce exploration as we start executing more episodes)
22 EPSILON_DECAY = EPSILON * 0.0001
23 # The discount rate in equation of Q-Value
24 DISCOUNT = 0.8
25 # The learn rate in equation of Q-Value
26 LEARN_RATE = 0.5
27
28 # File where Q table will be saved, loaded from. It's a numpy array.
29 Q_TABLE_FILE = 'qtables/qtable.txt.npy'
30 # File where per-episodic data will be saved: episode id, is success?, total rewards, total actions.
31 PLOT_FILE = 'qtables/episodes.txt'
32
33 # A filler state which represents an invalid state.
34 INVALID_STATE = [-100, -100, -100, False, -100, -100, -100]
35
36 # The step size to use to discretize the environment
37 UNIT_STEP_SIZE = 0.02
38 # A value to judge nearness
39 TOLERANCE = 0.01
40 # A finer value to judge nearness
41 TOLERANCE_FINER = 0.005
42
43 # Initial position of the arm (to be placed before any episode starts)
44 INIT_ARM_POSITION = [utility.rnd(-0.30), utility.rnd(-0.10), utility.rnd(0.14)]
45
46 # Dimension of the environment. [[xmin, xmax], [ymin, ymax], [zmin, zmax]]
47 ENV_DIMENSION = [[utility.rnd(-0.32), utility.rnd(-0.19)],
48                  [utility.rnd(-0.12), utility.rnd(-0.05)],
49                  [utility.rnd(0.00), utility.rnd(0.15)]]
50
51 # Some artificial delays to let V-REP stabilize after an action
52 SLEEP_VAL = 0.4
53 SLEEP_VAL_MIN = 0.3
54
55 # Distance between object and bin when object is in bin
56 CYLINDER_BIN_DISTANCE = 0.013
57
58 # Rewards
59 REWARD_TERMINATION = -10
60 REWARD_BAD_STEP = -3
61 REWARD_FIRST_SUCCESS = 5
62 REWARD_GOAL_ACHIEVED = 10




```

Fig: Configurable parameters (config.py)

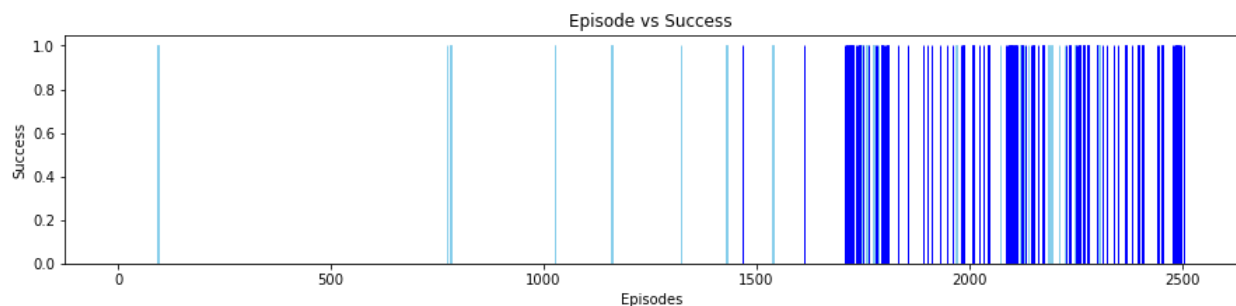
## IV. Results

## Model Evaluation and Validation

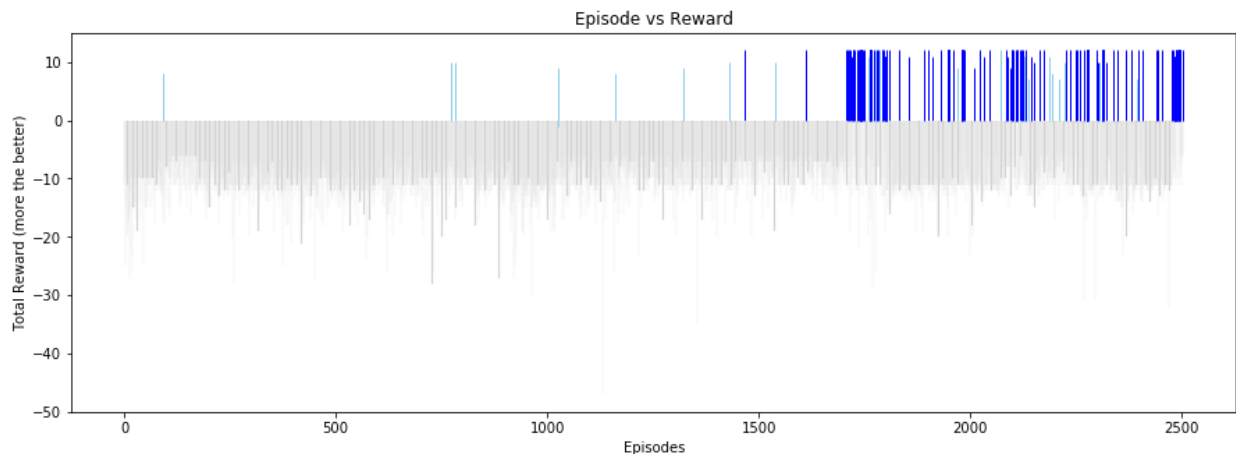
In this section, we look at the metrics that we described earlier. The simulation was allowed to run for 2500 episodes and data was collected. Also note that we used decaying epsilon-greedy approach, which is to say that rate of exploration decreased over each episode.

Legends	
	Episode ended successfully with full exploitation (desirable)
	Episode ended successfully with exploitation <i>and</i> exploration (good for learning)
	Episode ended with failure (environment went into an invalid state)

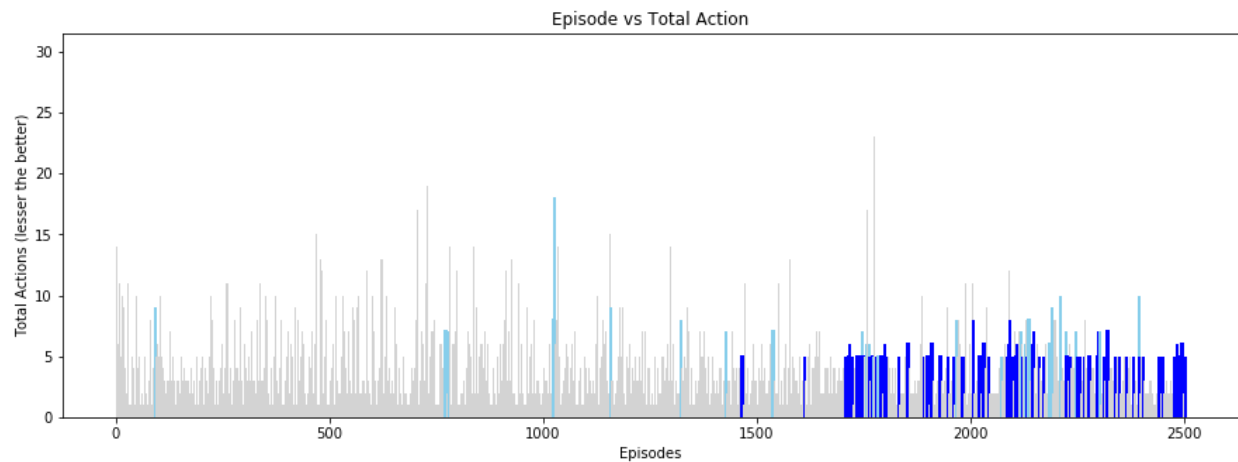
In the graph below, 1.0 suggests success (robot could complete the task) and 0.0 suggests failure. As can be seen, initial episodes mostly ended in failures. There were some successes below 1400 episodes. They are marked with sky blue color suggesting that it was due to exploratory steps. Post 1700 episodes, we see more frequent successes, suggesting (a) the model has gradually learnt the task (b) we are doing more exploitations than explorations.



In the graph below, we have plotted the reward each episode got. The maximum reward achieved was 12, and minimum was -47. The initial successful episodes received less than 12, mostly because they accomplished the task with more steps, which is penalized by default reward of -1.

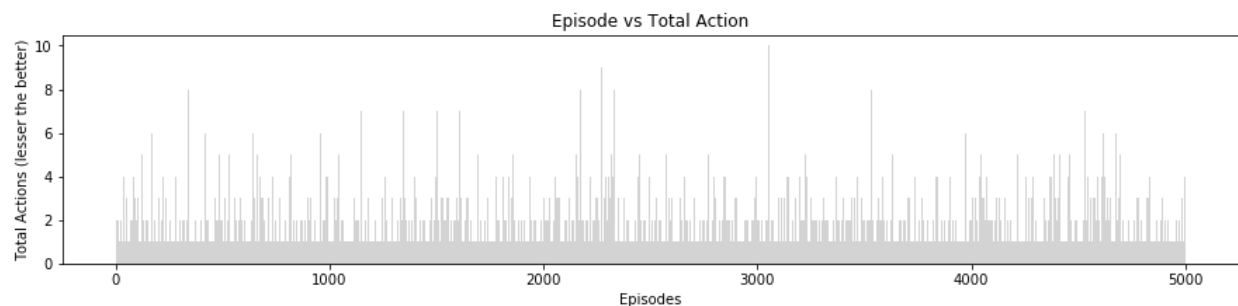
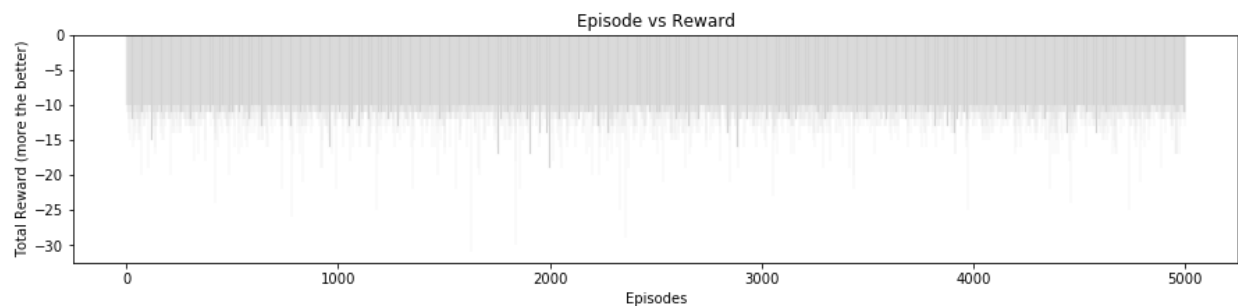


In the graph below, we have plotted the number of actions taken per episode. The initial successful episodes took more steps/actions than later ones. The task could be accomplished with minimum of 5 steps.



## Justification

The proposed benchmark was to try out with a uniformly random agent. Towards that, the simulation was run with full exploration and no exploitation. As was expected, all episodes were failures and all rewards were negative, as shown below.



Thus with our original solution, the agent could consistently accomplish the task in later episodes. Any failures or sub-optimal rewards in the later episodes were due to the exploratory actions.



## V. Conclusion

### Free Form Visualization

A short video of the training process can be seen here:

<https://youtu.be/yDugh3WH24M>

Initially the arm makes many mistakes but learns along the way. The various conditions which were considered for which reward/penalty was necessary, can be seen in the video.

### Reflection

The end to end solution involved:

1. Learning the V-REP environment, scene creation and the V-REP remote client APIs. This took considerable time, given that I had to learn this from scratch.
2. Designing the scene and writing LUA functions attached to the scene to act as hook into the framework. I used the `vrep.simxCallScriptFunction()` function call to invoke these functions. The V-REP online forum came in handy to clear my doubts around this.
3. Implementing the `RobotArm` class to interface with V-REP env. Writing the methods to detect if the object was being held by the gripper and to detect if the object is in the bin was specially tricky, I had to work with the coordinates to figure this out. Wrote a test script (`test_robot.py`) to test this out.
4. Implementing the Environment and Reward strategy. The reward strategy (both what events to consider for the reward and the values for them) evolved by observing the robot train.
5. Implementing the configurable Agent. Initially I had parts of the environment included in the agent. With further insights and refactoring, I could eventually segregate the two with clear demarcations.

Other than that,

- I had to design the reinforcement learning environment along with the solution, which was fun and challenging to work on.
- The training process was slow due to the fact that V-REP simulates real life robots and hence takes into account their mass, torque, speed of movement etc. It's rate of movements cannot be expedited beyond a certain limit.

### Improvement

Several improvements could be made to the solution:

- The training could have happened with the object kept at random locations. This would have made the robot arm learn to pick it from any location in the active space. This would have of course taken more time to train. I choose not to do it for time constraints.
- As can be seen, the state space can become very large as we increase the active space. To tackle this, we can either do function approximation, or implement DQN.
- Asynchronous methods could have been used to parallelize the training to decrease training time.

## References

- [1] <http://www.coppeliarobotics.com/>
- [2] [http://www.ias.tu-darmstadt.de/uploads/Publications/Kober\\_IJRR\\_2013.pdf](http://www.ias.tu-darmstadt.de/uploads/Publications/Kober_IJRR_2013.pdf)