Proposal towards

Udacity Machine Learning Engineer Nanodegree capstone project

By Anand Saha <anandsaha@gmail.com>
30th July 2017

1. Domain Background

Out of the three broad types of machine learning (unsupervised, supervised and reinforcement), reinforcement learning stands apart. While the first two learns from data, RL learns from experience. RL tends to be mimic the way living organisms interact and learn from their environment. The RL agent interacts with the environment by taking actions to maximize the cumulative reward obtained in the long term. Positive rewards encourage the agent to take desirable actions (in the direction of the goal state). Negative rewards act as deterrent towards undesirable actions and states.

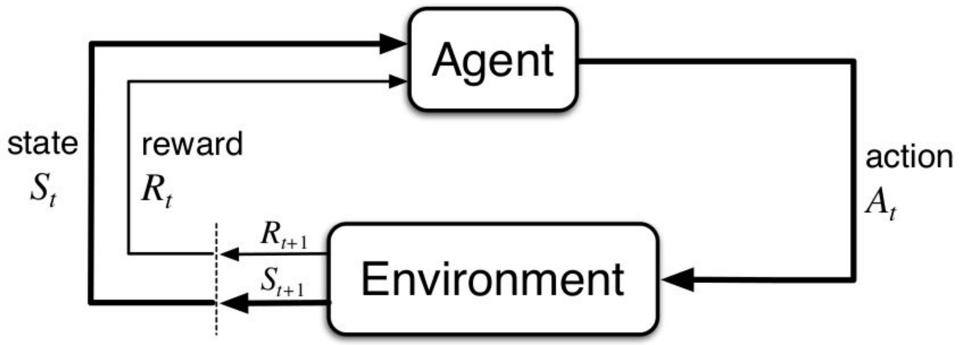


Fig 1: The reinforcement learning feedback loop (Sutton and Barto 2017)

The history, tools and techniques used to solve RL problems are well documented in the book "*Reinforcement Learning: An introduction*" by *Sutton and Barto (2008/2017)* [1].

The RL problem is well captured as a Markov Decision Process. Solutions to the MDP problem varies from those which need model of the environment (dynamic programming) to those which are model free (monte carlo and temporal difference learning) [1].

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]

1.1 Motivation

I am drawn to reinforcement learning by the range of problems it aims to solve and it's approach to doing so. It is foundational to solving Artificial General Intelligence. Recent advancements in beating the Go champion and Atari games have further energized the field. My interest lies in application of RL to training robots to accomplish tasks. The RL framework is well suited for such problems. I am also influenced by Pieter Abbeel's [8] work and talks.

2. Problem Statement

For my MLND Capstone project, I propose to explore the domain of reinforcement learning and apply it to the problem of training a robotic arm to accomplish a specific task.

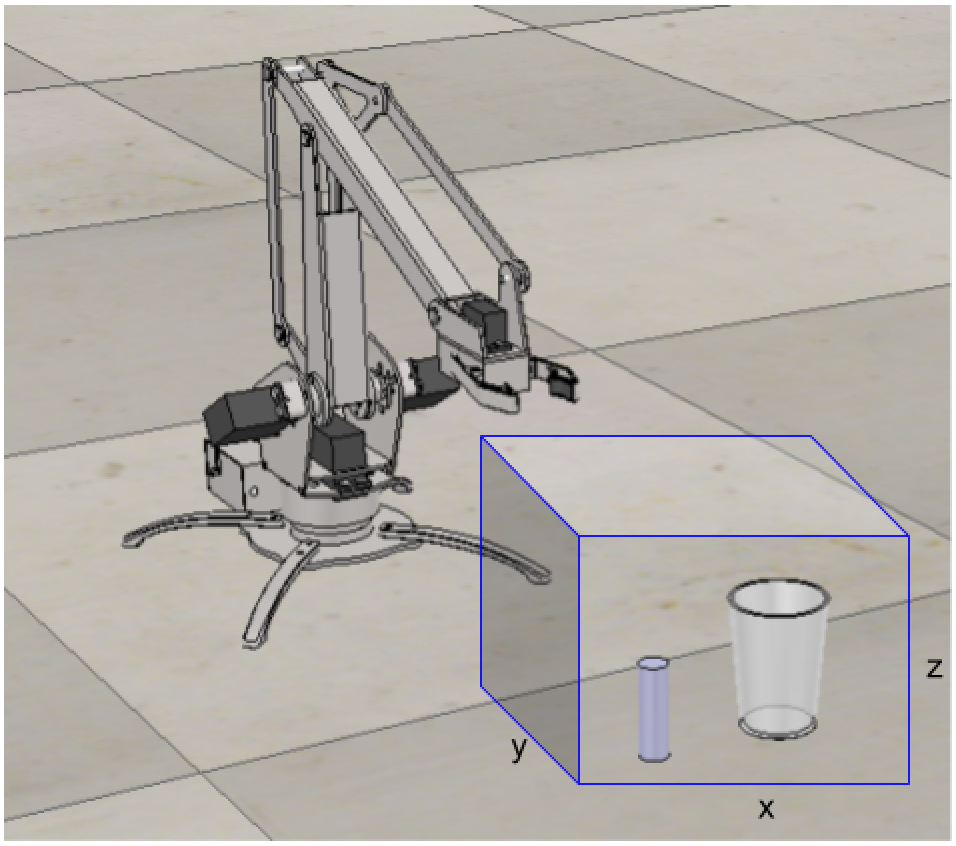


Fig 2: A 4 DOF robotic arm with gripper in the V-REP environment. The blue cube represents the maneuvering space.

In this project, the robotic arm will learn to grab an object (the cylinder in the above diagram) and put it in a desired location (in the bin in the above diagram).

The objective of the agent will be to grab the object and successfully put it in the bin using least possible actions.

The task is episodic and is considered finished if:

a. The arm hits and topples the object (failure)
b. The arm hits and topples the bin (failure)
c. The arm puts (drops) the object in the bin (success)

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.

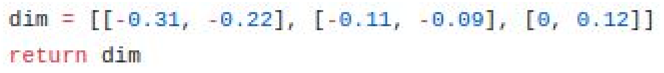
The arm will be simulated in the V-REP environment.

3. Datasets and inputs

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.

3.1 Inputs

The agent will be provided with the coordinates of the space where it can maneuver it's arm (the cubic space in figure 2).



```
dim = [[-0.31, -0.22], [-0.11, -0.09], [0, 0.12]]
return dim
```

The dimensions specify `[[x-min, x-max], [y-min, y-max], [z-min, z-max]]`.

Though in real life positions of the arm and the objects are in continuous space, I will use positions spaced out in discrete steps of 0.005 units. This will create discrete locations in the coordinate system where the arm can move to. This is enough for the gripper to hold the object and maneuver it. This will make the state space finite too.

The arm can reach any point in this space within a resolution of 0.005 units. For e.g., along the z-axis, there are 24 points where the gripper can be positioned ((0.12 - 0)/0.005). This will be fine tuned if needed.

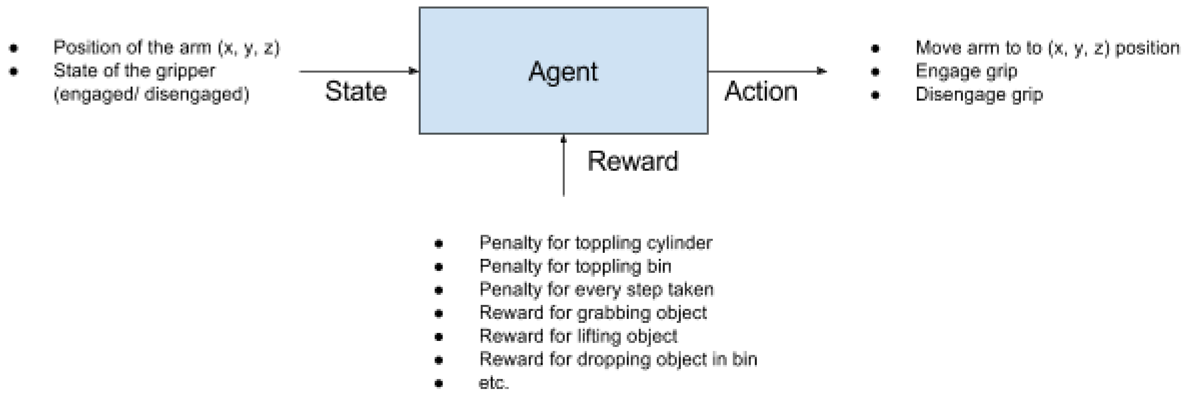
4. Solution statement

Classic Q-learning will be used for generating the optimal policy. It is an online, model-free and off-policy approach.

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

(From Wikipedia)

Appropriate state space, actions, reward function will be designed.



Position of the arm (x, y, z)
State of the gripper (engaged/ disengaged)
State
Agent
Action
Move arm to to (x, y, z) position
Engage grip
Disengage grip
Reward
Penalty for toppling cylinder
Penalty for toppling bin
Penalty for every step taken
Reward for grabbing object
Reward for lifting object
Reward for dropping object in bin
etc.

Fig. 3 The State-Reward-Action cycle

5. Benchmark model

While researching this project, I came across approaches to teach robotic arms to grasp objects (Google, Cornell, Google). These approaches use computer vision to learn the task. They focus on finer aspects like grasping objects of various shapes, sizes and textures.

I am using sensory data (position, gripper state) to accomplish the task mentioned in section 2. Hence using the above mentioned research will not be a good benchmark to compare to.

I propose to use a **uniformly random agent** 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.

6. Evaluation metrics

- **Success/failure per episode**: When an episode terminates, it will either have failed in it's goal or succeeded. Initially we will see failures and eventually success. 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 bot 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.

7. Project design

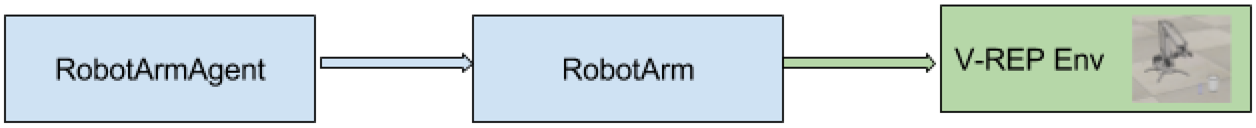


Fig 4: Object and communication diagram

7.1 The V-REP environment

As shown in figure 2, a scene will be created in the V-REP environment with three objects:

1. Uarm with gripper.ttm (controlled by agent)
2. A cylinder (object)
3. A bin (target)

The V-REP python remote client library will be used to interact with this scene.

7.2 The classes

The main logic will be contained in two classes:

- *RobotArm*: This class represents our Arm. It will be an interface to the virtual Arm and it's environment. Various functionalities exposed by it is shown in *figure 4*.
- *RobotArmAgent*: This is our agent. It will interface with the *RobotArm* class. It will be responsible for running episodes, which involves choosing and taking an action, collecting the reward and updating the Q-Table. It's details are shown in *figure 5*.

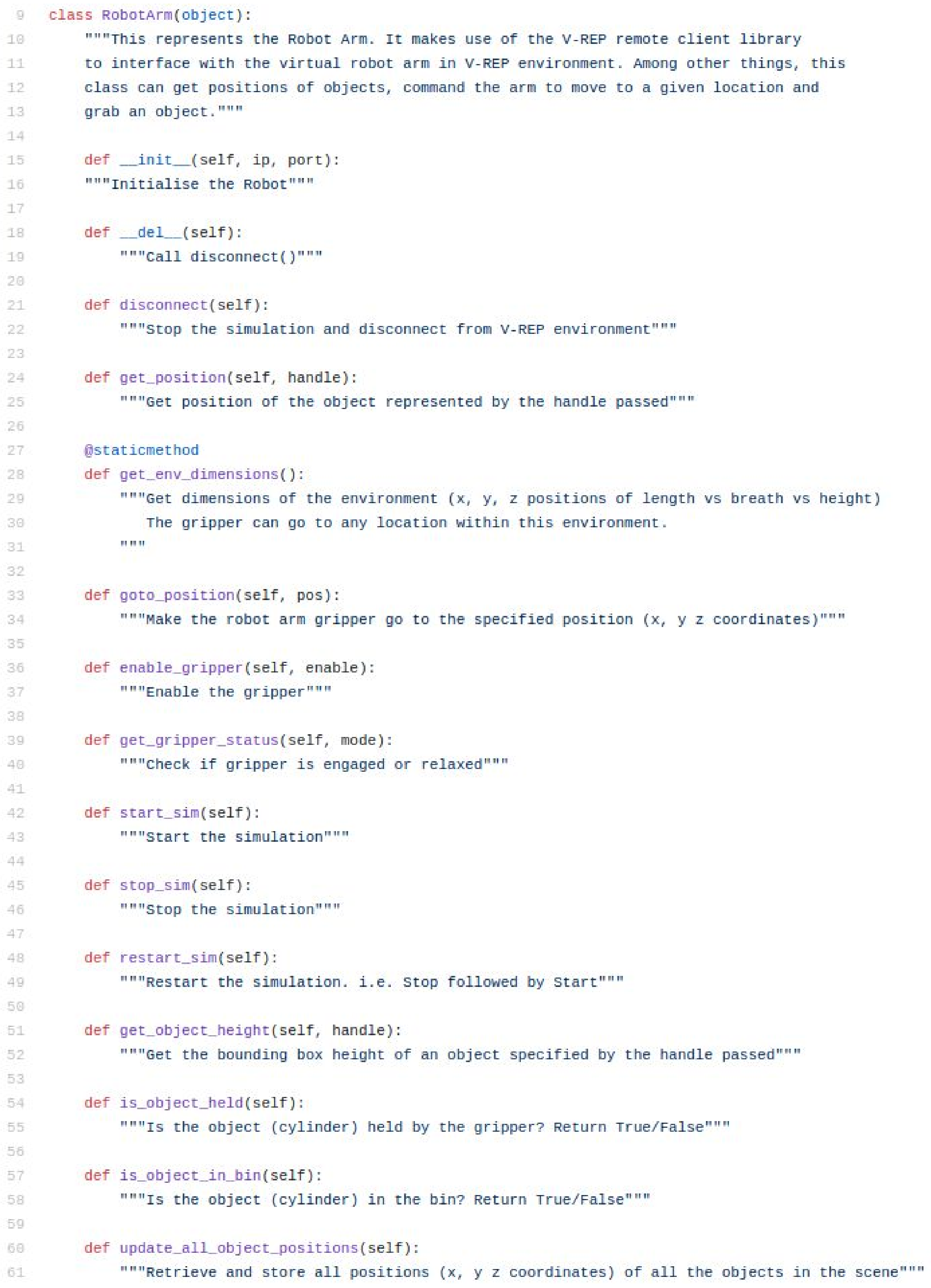
The sequence in which they will communicate is shown in figure 4.

(**NOTE**: These classes may get refactored along the way)

7.3 Software Stack

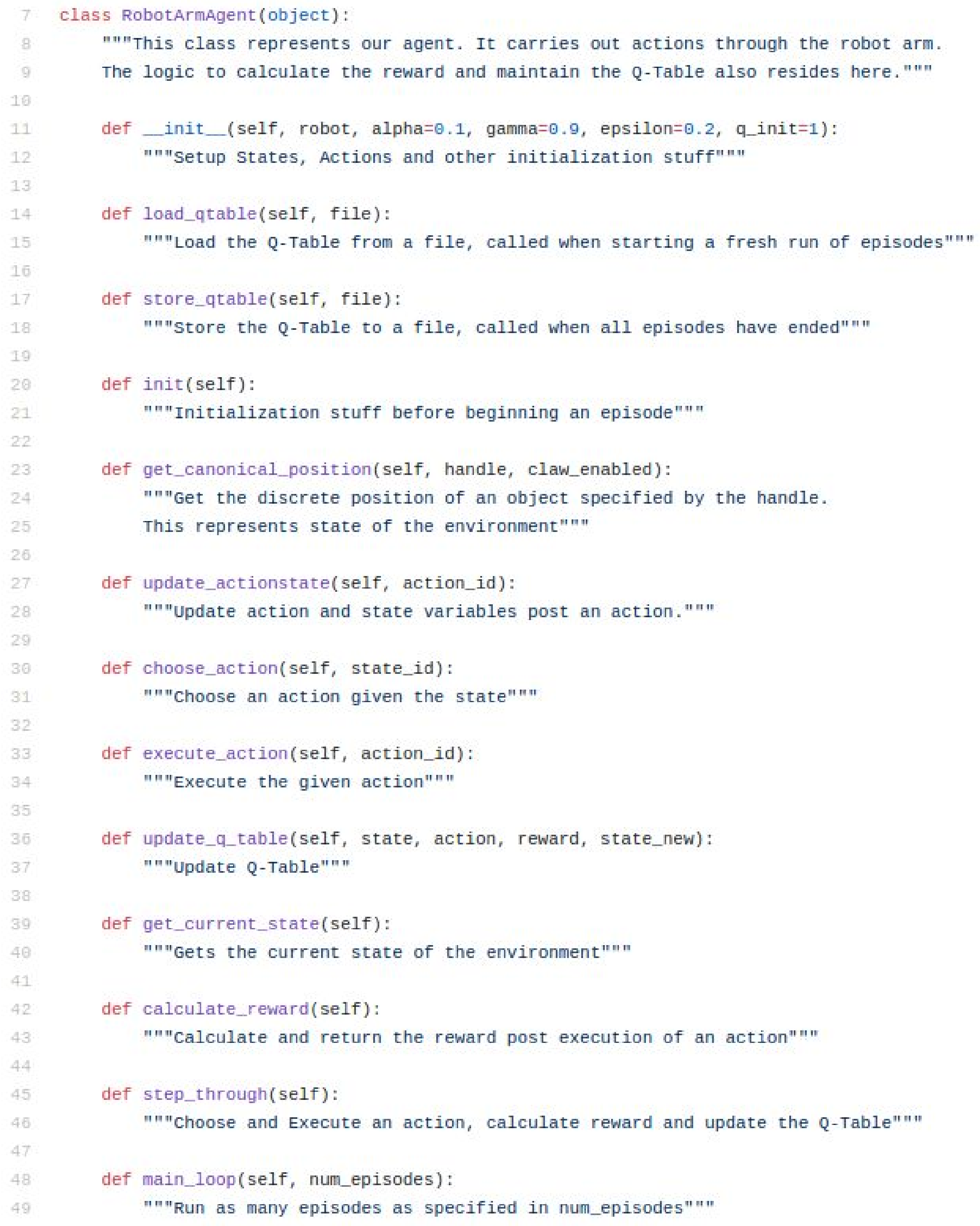
Apart from V-REP framework, the other requirements will be *python 3.5, matplotlib and numpy*.

The figures below show the class interfaces of RobotArm and RobotArmAgent.



```python
class RobotArm(object):
    """This represents the Robot Arm. It makes use of the V-REP remote client library
    to interface with the virtual robot arm in V-REP environment. Among other things, this
    class can get positions of objects, command the arm to move to a given location and
    grab an object."""

    def __init__(self, ip, port):
    """Initialise the Robot"""

    def __del__(self):
        """Call disconnect()"""

    def disconnect(self):
        """Stop the simulation and disconnect from V-REP environment"""

    def get_position(self, handle):
        """Get position of the object represented by the handle passed"""

    @staticmethod
    def get_env_dimensions():
        """Get dimensions of the environment (x, y, z positions of length vs breath vs height)
           The gripper can go to any location within this environment.
        """

    def goto_position(self, pos):
        """Make the robot arm gripper go to the specified position (x, y z coordinates)"""

    def enable_gripper(self, enable):
        """Enable the gripper"""

    def get_gripper_status(self, mode):
        """Check if gripper is engaged or relaxed"""

    def start_sim(self):
        """Start the simulation"""

    def stop_sim(self):
        """Stop the simulation"""

    def restart_sim(self):
        """Restart the simulation. i.e. Stop followed by Start"""

    def get_object_height(self, handle):
        """Get the bounding box height of an object specified by the handle passed"""

    def is_object_held(self):
        """Is the object (cylinder) held by the gripper? Return True/False"""

    def is_object_in_bin(self):
        """Is the object (cylinder) in the bin? Return True/False"""

    def update_all_object_positions(self):
        """Retrieve and store all positions (x, y z coordinates) of all the objects in the scene"""
```

Fig 5: The RobotArm class



```python
class RobotArmAgent(object):
    """This class represents our agent. It carries out actions through the robot arm.
    The logic to calculate the reward and maintain the Q-Table also resides here."""

    def __init__(self, robot, alpha=0.1, gamma=0.9, epsilon=0.2, q_init=1):
        """Setup States, Actions and other initialization stuff"""

    def load_qtable(self, file):
        """Load the Q-Table from a file, called when starting a fresh run of episodes"""

    def store_qtable(self, file):
        """Store the Q-Table to a file, called when all episodes have ended"""

    def init(self):
        """Initialization stuff before beginning an episode"""

    def get_canonical_position(self, handle, claw_enabled):
        """Get the discrete position of an object specified by the handle.
        This represents state of the environment"""

    def update_actionstate(self, action_id):
        """Update action and state variables post an action."""

    def choose_action(self, state_id):
        """Choose an action given the state"""

    def execute_action(self, action_id):
        """Execute the given action"""

    def update_q_table(self, state, action, reward, state_new):
        """Update Q-Table"""

    def get_current_state(self):
        """Gets the current state of the environment"""

    def calculate_reward(self):
        """Calculate and return the reward post execution of an action"""

    def step_through(self):
        """Choose and Execute an action, calculate reward and update the Q-Table"""

    def main_loop(self, num_episodes):
        """Run as many episodes as specified in num_episodes"""
```

Fig 6: The RobotArmAgent class

References:

1. http://incompleteideas.net/sutton/book/the-book.html
2. http://www.ias.tu-darmstadt.de/uploads/Publications/Kober_IJRR_2013.pdf
3. https://en.wikipedia.org/wiki/Reinforcement_learning
4. https://www2.informatik.uni-hamburg.de/wtm/ps/Yan_DAAAAM09.pdf
5. http://www.coppeliarobotics.com/
6. https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf
7. https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
8. https://people.eecs.berkeley.edu/~pabbeel/