Deep Reinforcement Learning for Robotics: Training a Robot Arm to Find Object of Interest

Smruti Panigrahi, Ph.D.
Udacity Robotics Nandegree

***Abstract*—In this paper, we explore the reinforcement learning (RL) and Deep RL algorithms and how they are used to train various agents. An integration of deep CNN model into the RL framework is adopted to train an agent to perform desired tasks based on imagery information received from a camera. The training is conducted in a Gazebo simulation environment where a robotic arm is tasked with reaching a desired target object. A Deep Q-Learning Network (DQN) agent is used along with reward functions to teach the robotic arm to carry out two primary objectives: (a) have any part of the robot arm touch the object of interest, with at least 90% accuracy, and (b) have only the gripper base of the robot arm touch the object, with at least 80% accuracy. Every camera frame at each simulation iteration, is fed into the DQN and the agent then makes a prediction and carries out an appropriate action. Incorporating LSTMs into the DQN, the network is trained using multiple past frames from the camera sensor instead of a single frame. In addition, the effect of tuning different parameters that influence the accuracy of the training outcome is discussed.**

Keywords—Reinforcement Learning, Deep RL, DQN, CNN, RNN, LSTM, Gazebo Simulation, ROS.

I. Introduction

There are essential two ways a robot can perform the assigned tasks. One is the traditional algorithm development for achieving certain tasks and the other is letting the robot to try various things and learn as it goes, just like humans and animals. The second part is basically know as reinforcement learning or RL. Whether the tasks of humans learning to walk, learning to ride a bicycle, or dogs learning to perform certain tricks, reinforcement learning is at the heart of it. At the heart of the reinforcement learning, the agent receives a positive reward if it performs as desires or else receives zero reward or a punishment for doing the wrong thing. Hence in order for an agent to learn effectively on par with human level intelligence [1], it is important for the agent to remember certain past experiences that resulted in positive or negative rewards. This way, the agent can apply the past experience to decide on its next action. This is where the Long-Short-Term-Memory (LSTM) comes in which uses both long-term and short-term memory to manage the past experience of the agent and uses this to perform future actions.

As a roboticist or a robotics software engineer, building our own robots is a very valuable skill and offers a lot of experience in solving problems in the domain. Often, there are limitations around building our own robot for a specific task, especially related to available resources or costs. That's where simulation environments are quite beneficial. Not only there is freedom over what kind of robot one can build, but also get to experiment and test different algorithms and scenarios with relative ease and at a faster pace. For example, one can have a simulated drone to take in camera data for obstacle avoidance in a simulated city, without worrying about it crashing into a building!

In this project, we are going to explore the basics of the reinforcement learning, deep reinforcement learning, and LSTM. We also discussed various examples where the Deep RL algorithm can be used to train an agent from scratch. We then use NVIDIA deep Q-learning network, to train a robot arm in a Gazebo environment to reach towards a goal and touch it without hitting the ground, and without hitting the goal object too hard. We optimize the model by tuning various hyper parameters involved in the Deep RL algorithm to reach at desired accuracy in a short period of time.

II. Background

The applications of reinforcement learning is diverse, ranging from self-driving cars to board games. For instance, one of the major breakthroughs in machine learning in the 1990s was TD-Gammon, an algorithm that uses RL to play backgammon on par with the best Backgammon players at the time. This algorithm advanced the theory of Backgammon by discovering the strategies that were previously unknown. Backgammon is a complicated game with 10^20 possible game states. More recently progress was made on a game that is much more complicated. DeepMind's [2] AlphaGo [3] is an AI agent trained to beat professional Go players. It's a popular belief that there are more configurations in the game Go than there are atoms in the universe. RL is also used to play games such as Atari Breakout. The AI agent is given no prior knowledge of what a ball is or what the controls do. It only sees the screen and its score. Then through interacting with the game, with testing out the various controls, it is able to devise a strategy to maximize its score. RL was also used to create a bot to top players in the online battle arena video game Dota.

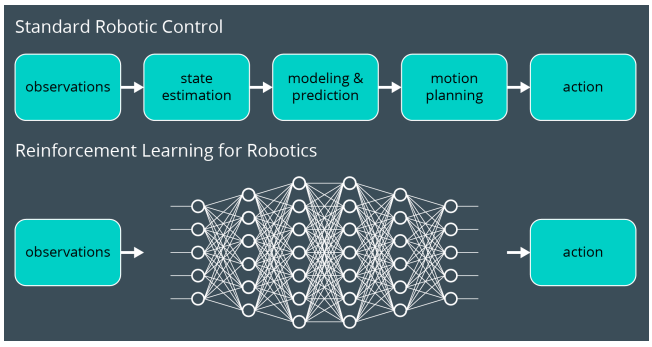


Fig. 1: Standard robotic control vs. RL-based robotic control

RL is also used in robotics. For instance it has been used to teach robots to walk. The idea is that the robot is given time to test out its new legs to see what works and what does not work for staying upright. Then an algorithm is created to help it learn from that gained experience, so that the robot will be able to walk as desired. RL is also used successfully in self-driving cars, ships, and airplanes. It's also been used in finance, biology, telecommunication, and inventory management among other applications.

Consider an "agent" who learns from trial and error, how to behave in an environment to maximize reward. But, what is reinforcement learning in general? The reinforcement learning framework is characterized by an agent learned to interact with its environment. This agent can be a human, a dog or even a self-driving car. Assume that time evolves in discrete timesteps. At the initial timestep, the agent observes the environment. One can think of the observations as a situation that the environment presents to the agent. Then, it must select an appropriate action in response. Then at the next timestep in response to the agent's action, the environment presents a new situation to the agent. At the same time, the environment gives the agent a reward, which provides some indication whether the agent has responded appropriately to the environment. Then the process continues where at each timestep the environment sends the agent an observation and reward. And in response the agent must choose an action. The flow of this framework is shown in Fig. 2.

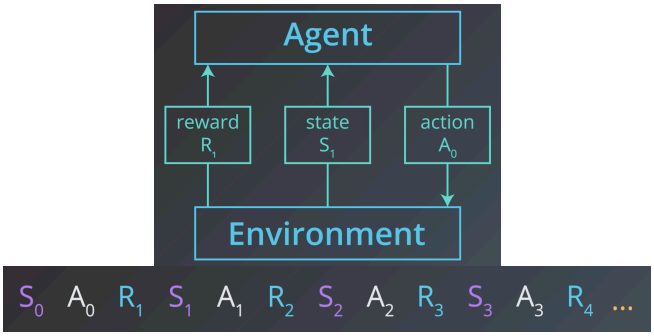


Fig. 2: Basic framework of a reinforcement learning model

In general, one does not need to assume that the environment shows the agent everything it needs to make well-informed decisions. However, knowing more about the environment, greatly simplifies the underlying mathematics. An assumption can be made that the agent is able to fully observe whatever state the environment is in. An observation received by the agent is generally referred to as the environment state.

The reinforcement algorithm steps are described here. At the very beginning at timestep zero, t_0, the agent first receives the environment state, which is denoted by S_0. Then, based on that observation/state the agent chooses an action, A_0 at the next timestep, t_1. As a direct consequence of the agent's choice of action, A_0, and the environments previous state, S_0, the environment transitions to a new state, S_1, and gives some reward R_1 to the agent (Fig. 2). The agent then chooses and action, A_1. At timestep, t_2, the process continues where the environment passes the reward in state. Then the agent responds with an action and so on, whereas the agent interacts with the environment. This interaction is manifested as a sequence of states, actions, and rewards as shown in Fig. 2. The reward will always be the most relevant quantity to the agent. To be specific, any agent has the goal to maximize expected cumulative reward or the sum of the rewards attained over all timesteps. In other words, it seeks to find the strategy for choosing actions with the cumulative reward is likely to be high. And the agent can only accomplish this by interacting with the environment. This is because at every timestep, the environment decides how much reward the agent receives. In other words, the agent must play by the rules of the environment. But through interactions the agent can learn those rules and choose appropriate actions to accomplish its goal. So for a reinforcement-learning problem, one has to specify the states, actions, and rewards, and decide the rules of the environment.

For instance, when one is teaching an agent to play a game, the interaction of the agent with the environment ends when the agent wins or looses. If we are teaching a car to drive itself, then the interaction ends if the car crashes. Not all reinforcement-learning tasks have well defined ending point, but those that do, are called episodic tasks. In this case, a complete sequence of interaction from start to finish is referred to as an episode. When the episode ends, the agent looks at the total amount of reward it received to figure out how well it did. It's then able to start from scratch as if it has been completely reborn into the same environment but now with the added knowledge of what happened in its past life. In this way, as time passes over its many lives, the agent makes better and better decisions. Once the agent spent enough time getting to know the environment, it should be able to pick a strategy where the cumulative reward is quite high.

There are two types of RL based on tasks, i.e. continuing task or episodic task. A task is an instance of the reinforcement learning (RL) problem.

- Continuing tasks are tasks that continue forever, without end.
- Episodic tasks are tasks with a well-defined starting and ending point.
    - In this case, a complete sequence of interaction is considered, from start to finish, as an episode.
    - Episodic tasks come to an end whenever the agent reaches a terminal state.

In other words, in the context of a game-playing agent, it should be able to achieve a higher score. Hence, episodic tasks are tasks with a well-defined ending point. The idea is that by playing the game many times, or by interacting with the environment in many episodes, the agent can learn to play the game better and better. It's important to note that this problem is exceptionally difficult, because the feedback is only delivered at the very end of the game. So, if the agent loses a game, it's unclear when exactly it went wrong: maybe the agent was so bad at playing that every move was horrible, or maybe instead it played beautifully for the majority of the game, and then made only a small mistake at the end. When the reward signal is largely uninformative in this way, the task suffers the problem of *sparse rewards*. There's an entire area of research dedicated to this problem.

The tasks that go on forever, without end, are called continuing tasks. An example of episodic tasks is a game of chess, where an episode finishes when the game ends. An example of a continuing task is an algorithm that buys and sells stocks in response to the financial market. In this continuing task case, the agent lives forever and it has to learn the best way to choose actions while simultaneously interacting with the environment. The algorithms in this case are more complex.

A. Markov Decission Process in Reinforcement Learning

In general, the reward hypothesis is that all goals can be framed as the maximization of expected cumulative reward.

The reward hypothesis can be modeled as a Markov Decision Process (MDP).

In general, the MDP is defined by a fine set of states, S, a finite set of actions A, and a finite set of rewards R along with the one-step dynamics of the environment $p(s',r|s,a)$ and the discount rate $\gamma \in [0,1]$ (Fig. 3). The state space S is the set of all nonterminal states. In continuing tasks (like the recycling task detailed in example provided in the next section), this is equivalent to the set of all states. In episodic tasks, $S+$ is used to refer to the set of all states, including terminal states. The action space A is the set of possible actions available to the agent. In the event that there are some states where only a subset of the actions are available, we can also use $A(s)$ to refer to the set of actions available in state $s \in S$.

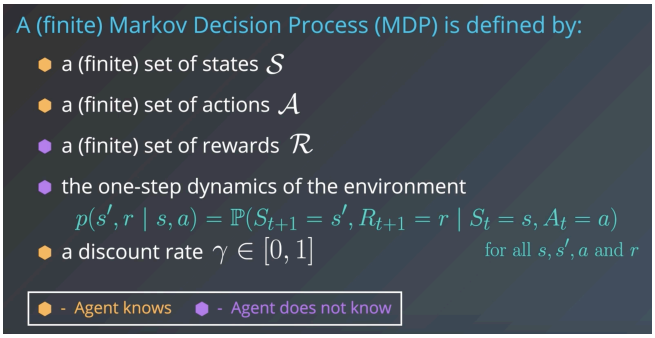


Fig. 3: Definition of a MDP

At an arbitrary time step t, the agent-environment interaction has evolved as a sequence of states, actions, and rewards $(S_0,A_0,R_1,S_1,A_1,\ldots,R_{t-1},S_{t-1},A_{t-1},R_t,S_t,A_t)$.

When the environment responds to the agent at time step $t+1$, it considers only the state and action at the previous time step (S_t,A_t). In particular, it does not care what state was presented to the agent more than one step prior. *In other words*, the environment does not consider any of $\{S_0,\ldots,S_{t-1}\}$. And, it does not look at the actions that the agent took prior to the last one. *In other words*, the environment does not consider any of $\{A_0,\ldots,A_{t-1}\}$.

Furthermore, how well the agent is doing, or how much reward it is collecting, has no effect on how the environment chooses to respond to the agent. *In other words*, the environment does not consider any of $\{R_0,\ldots,R_t\}$. Because of this, we can completely define how the environment decides the state and reward by specifying

$$p(s',r|s,a) \doteq \mathrm{P}(S_{t+1}=s', R_{t+1}=r \mid S_t=s, A_t=a)$$

for each possible $s',r,s,$ and a. These conditional probabilities are said to specify the one-step dynamics of the environment.

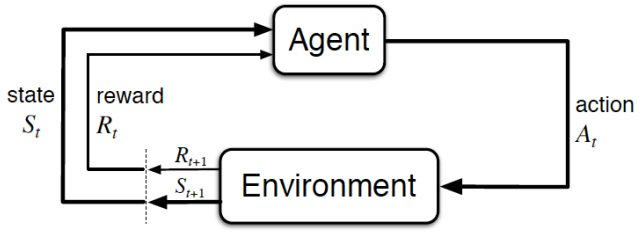


Fig. 4: The agent-environment interaction in RL [Sutton and Barto 2017]

In summary, the RL framework is composed of the following:

The Setting

- The RL framework is characterized by an agent learning to interact with its environment.
- At each time step, the agent receives the environment's state (*the environment presents a situation to the agent)*, and the agent must choose an appropriate action in response. One time step later, the agent receives a reward (*the environment indicates whether the agent has responded appropriately to the state*) and a new state.
- All agents have the goal to maximize expected cumulative reward, or the expected sum of rewards attained over all time steps.

Episodic vs. Continuing Tasks

- A task is an instance of the reinforcement learning (RL) problem.
- Continuing tasks are tasks that continue forever, without end.
- Episodic tasks are tasks with a well-defined starting and ending point.
  - In this case, we refer to a complete sequence of interaction, from start to finish, as an episode.
  - Episodic tasks come to an end whenever the agent reaches a terminal state.

The Reward Hypothesis

- All goals can be framed as the maximization of (expected) cumulative reward.

Cumulative Reward

- The return at time step t is

  $$G_t := R_{t+1}+R_{t+2}+R_{t+3}+\ldots$$

- The agent selects actions with the goal of maximizing expected (discounted) return. (*Note: discounting is covered in the next concept.*)

Discounted Return

- The discounted return at time step t is $Gt:=R_{t+1}+\gamma R_{t+2}+\gamma 2R_{t+3}+\ldots.$
- The discount rate γ is something that we set, to refine the goal that we have the agent.
  - It must satisfy $0 \leq \gamma \leq 1$.
  - If $\gamma=0$, the agent only cares about the most immediate reward.
  - If $\gamma=1$, the return is not discounted.
  - For larger values of γ, the agent cares more about the distant future. Smaller values of γ result in more extreme discounting, where - in the most extreme case - agent only cares about the most immediate reward.

MDPs and One-Step Dynamics

- The state space S is the set of all (*nonterminal*) states.

- In episodic tasks, we use $S+$ to refer to the set of all states, including terminal states.
- The action space A is the set of possible actions. (Alternatively, $A(s)$ refers to the set of possible actions available in state $s \in S$.)
- The one-step dynamics of the environment determine how the environment decides the state and reward at every time step. The dynamics can be defined by specifying $p(s',r \mid s,a) \doteq P(St+1=s',Rt+1=r \mid St=s,At=a)$ for each possible s',r,s,and a.
- A (finite) Markov Decision Process (MDP) is defined by:
  - a (finite) set of states S (or $S+$, in the case of an episodic task)
  - a (finite) set of actions A
  - a set of rewards R
  - the one-step dynamics of the environment
  - the discount rate $\gamma \in [0,1]$

B. RL Example

Let us consider a recycling robot that is tasked with collecting empty cans (Fig. 5). In this case, the robot is battery powered and is intended to be completely autonomous in terms of accomplishing its three "tasks", namely, wait, search, and recharge. The "state" of the robot is directly related to its battery charge level, i.e. either high or low depending on whether the robot's battery has enough charge left to accomplish its goal.

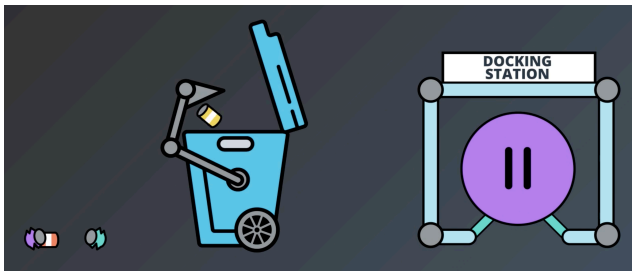


Fig. 5: Recycling robot tasked with collecting empty cans.

Say at an arbitrary time step t, the state of the robot's battery is high (S_t=high). Then, in response, the agent decides to search (A_t=search). In this case, the environment responds to the agent by flipping a theoretical coin with 70% probability of landing heads as depicted in Fig. 6.

- If the coin lands heads, the environment decides that the next state is high (S_{t+1}=high), and the reward is 4 (R_{t+1}=4).
- If the coin lands tails, the environment decides that the next state is low (S_{t+1}=low), and the reward is 4 (R_{t+1}=4).

In fact, for any state S_t and action A_t, it is possible to use the figure to determine exactly how the agent will decide the next state S_{t+1} and reward R_{t+1}.

In the example case in Fig. 3, for the case that S_t=high, and A_t=search, when the environment responds to the agent at the next time step,

- with 70% probability, the next state is high and the reward is 4. In other words,

  $p(\text{high},4 \mid \text{high},\text{search}) = P(S_{t+1}=\text{high},R_{t+1}=4 \mid S_t=\text{high},A_t=\text{search})=0.7$.

- with 30% probability, the next state is low and the reward is 4. In other words,

  $p(\text{low},4 \mid \text{high},\text{search}) = P(S_{t+1}=\text{low},R_{t+1}=4 \mid S_t=\text{high},A_t=\text{search})=0.3$.

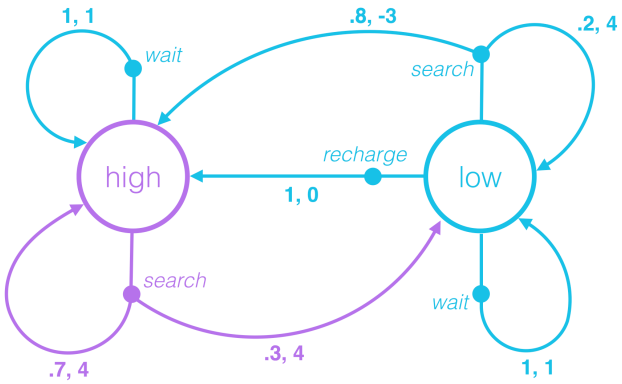


Fig. 6: RL example

C. Policies

The policy function represents a strategy that, given the value function, selects the action believed to yield the highest (long-term) reward. Often there is no clear winner among the possible next actions. For example, the agent might have the choice to enter one of four possible next states A,B,C, and D with reward values A=10, B=10, C=5 and D=5. So both A and B would be good immediate choices, but a long way down the road action A might actually have been better than B, or action C might even have been the best choice. It is worth exploring these options during training, but at the same time the immediate reward values should guide the choice. So how can we find a balance between exploiting high reward values and exploring less rewarding paths which might return high rewards in the long term? [15]

A clever way to proceed is to take choices with probability proportional to their reward. In this example the probability of taking A would be 33% (10/(10+10+5+5)), and the probability of taking B, C, and D would be 33%, 16%, and 16%, respectively. This random choice nature of the policy function is important to learning a good policy. There might exist effective or even essential strategies that are counter-intuitive but necessary for success.

For example, if we train a race car to go fast on a racetrack it will try to cut the corners with the highest speed possible. However, this strategy will not be optimal if we add competitors into the mix. The agent then needs to take into account these competitors and might want to take turns slower to avoid the possibility of a competitor overtaking or worse, a collision. Another scenario might be that cornering at very high speed will wear the tires much quicker resulting in the need for a pit-stop, costing valuable time.

If we want to have a specific outcome we can use both the policy and value functions to guide the agent to learn good strategies to achieve that outcome. This makes reinforcement learning versatile and powerful.

We train the policy function by (1) initializing it randomly, for example, let each state be chosen with

probability proportional to its reward—and initialize the value function with the rewards; that is, set the reward of all states to zero where no direct reward is defined (for example the racetrack goal has a reward of 10, off-track states have a penalty of -2, and all states on the racetrack itself have zero reward). Then (2) train the value function until convergence (see Figure 1), and (3) increase the probability of the action (moving from A to B) for a given state (state A) which most increases the reward (going from A to C might have a low or even negative reward value, like sacrificing a chess piece, but it might be in line with the policy of winning the game). Finally, (4) repeat from step (1) until the policy no longer changes.

The simplest kind of policies is the mapping from the set of environment states to a set of possible actions, i.e. π: $S \rightarrow A$. Think of the policy as a factory that takes any environment state as input and outputs any corresponding action that the agent can take. If the agent wants to keep track of its strategy, all it needs to do is to build this factory or to specify this mapping. This type of policy is known as a deterministic policy. It is most common to denote the policy as a Greek letter π.

Another type of policy is a stochastic policy. A stochastic policy is a mapping, π: S x $A \rightarrow [0,1]$. Stochastic policies allow the agents to choose actions randomly. $\pi(a|s) = P(A_t = a \mid S_t = s)$. A stochastic policy is defined as a mapping that accepts an environment state, s, and an action, a, as inputs and returns the probability that the agent takes action, a, while in state, s.

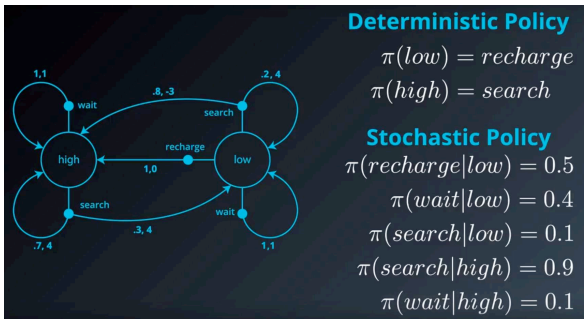


Fig. 7: Deterministic vs stochastic policy

Considering the example of the robot collection cans as shown in Fig. 5, the deterministic policy would specify, e.g., whenever the battery is low, recharge the robot, and if the battery is high, search for cans. Whereas, the stochastic policy would specify, whenever the battery is low then recharge the robot with 50% probability, wait for cans at its current location with 40% probability and otherwise search for cans with 10% probability. Whenever the battery is high, search for cans with 90% probability and otherwise wait for cans with 10% probability. The comparison is shown in Fig. 7. It is important to note that any deterministic policy can be expressed using the same notation as the stochastic policy.

In the grid-world example shown in Fig. 8, once the agent selects an action,

- it always moves in the chosen direction (contrasting general MDPs where the agent doesn't always have complete control over what the next state will be), and
- the reward can be predicted with complete certainty (contrasting general MDPs where the reward is a random draw from a probability distribution).

The policy π in Fig. 8 is given by $\pi(s_1)$=right, $\pi(s_2)$=right, $\pi(s_3)$=down, $\pi(s_4)$=up, $\pi(s_5)$=right, $\pi(s_6)$=down, $\pi(s_7)$=right, $\pi(s_8)$=right.

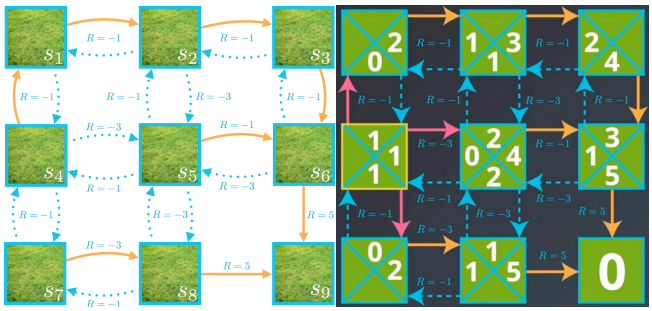


Fig. 8: (Left) A deterministic policy example shown in orange arrows. (Right) Possible value functions for all possible policies for the grid-world example.

Since s_9 is a terminal state, the episode ends immediately if the agent begins in this state. So, the agent will not have to choose an action (so, s_9 is not included in the domain of the policy), and $v_\pi(s_9)$=0. Then the value function for each state can be calculated going backwards from the terminal state and adding the terminal state value function to the reward of the previous step, i.e. $v_\pi(s_8) = R+v_\pi(s_9) = 5$. Calculating backwards for this particular deterministic policy, we get $v_\pi(s_7)$=2, $v_\pi(s_6)$=5, $v_\pi(s_5)$=4, $v_\pi(s_3)$=4, $v_\pi(s_2)$=3, $v_\pi(s_1)$=2, and $v_\pi(s_4)$=1. Fig. 8, shows possible value functions at each state for all possible policies.

In this simple example, we saw that the value of any state can be calculated as the sum of the immediate reward and the (discounted) value of the next state.

For a general MDP, we have to instead work in terms of an *expectation*, since it's not often the case that the immediate reward and next state can be predicted with certainty. Indeed, the reward and next state are chosen according to the one-step dynamics of the MDP. In this case, where the reward r and next state s' are drawn from a (conditional) probability distribution $p(s',r|s,a)$, the **Bellman Expectation Equation (for v_π)** expresses the value of any state s in terms of the *expected* immediate reward and the *expected* value of the next state:

$$v_\pi(s)=\mathrm{E}_\pi[R_{t+1}+\gamma v_\pi(S_{t+1})|S_t=s].$$

In the event that the agent's policy π is **deterministic**, the agent selects action $\pi(s)$ when in state s, and the Bellman Expectation Equation can be re-written as the sum over two variables (s' and r):

$$v_\pi(s)= \sum_{s'\in S^+,\, r\in R,\, a\in A(s)} p(s',r|s,\pi(s))(r+\gamma v_\pi(s'))$$

In this case, we multiply the sum of the reward and discounted value of the next state $(r+\gamma v_\pi(s'))$ by its corresponding probability $p(s',r|s,\pi(s))$ and sum over all possibilities to yield the expected value.

If the agent's policy π is **stochastic**, the agent selects action a with probability $\pi(a|s)$ when in state s, and the Bellman Expectation Equation can be rewritten as the sum over three variables (s', r, and a):

$$v_\pi(s)= \sum_{s'\in S^+,\, r\in R,\, a\in A(s)} \pi(a|s)p(s',r|s,a)(r+\gamma v_\pi(s'))$$

In this case, we multiply the sum of the reward and discounted value of the next state $(r+\gamma v_\pi(s'))$ by its corresponding probability $\pi(a|s)p(s',r|s,a)$ and sum over all

possibilities to yield the expected value. The Bellman Expectation Equation (for v_π) shows that it is possible to relate the value of a state to the values of all of its possible successor states.

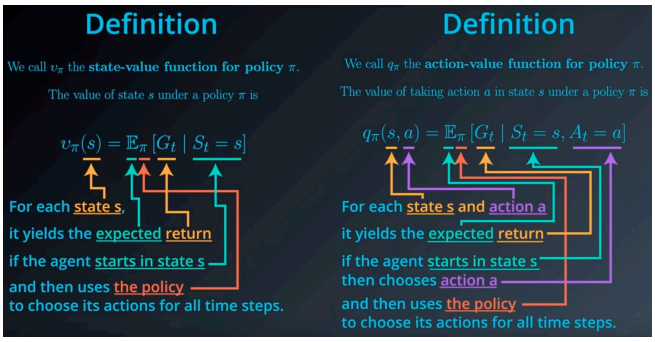


Fig. 9: State-value vs action-value function

D. *Optimal Policies*

By definition, a policy, $\pi' \geq \pi$, if and only if its state value function $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in S$. It is often possible to find two policies that cannot be compared. However, there will always be at least one policy that is better than or equal to all other policies, which is called an optimal policy which is guaranteed to exist but not necessarily unique. It is the optimal policy that the agent is searching for. It is the solution to the MDP and the best strategy to accomplish its goal. All optimal policies have the same value function denoted by v_{π^*} or v_*.

As show in Fig. 9, the state-value function, v_π, is defined for the value of state s under a policy π. For each state s, it yields the expected discounted return $E_\pi(G_t)$ if the agent starts in state s, and then uses the policy to choose its actions for all time steps. The action-value function q_π for policy π, is defined for the value of taking action a in state s under a policy π. While the state-values are the function of the environment state, the action-values are a function of the environment state s and the agent's action a. For each state, s, and action, a, it yields the expected return if the agent starts in state, s, then chooses action, a, and then uses the policy, π, to choose its actions for all time steps.

State-value function: $v_\pi(s) = E_\pi[G_t \mid S_t = s]$

Action-value function: $q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a]$

For the grid-world example shown in Fig. 8, in the case of the state-value function, we kept track of the value of each step with a number on the grid. For action-value function, however, we need four values for each state, each corresponding to a different action, with the exception of the terminal state as shown in Fig. 8. In each state, these four numbers correspond to the same state but different actions, i.e. the one on the top corresponds to action *up*, the one on the right corresponds to action *right*, the one on the bottom corresponds to action *down* and the one on the left corresponds to action *left*.

We will use "return" and "discounted return" interchangeably. For an arbitrary time step t, the discounted return is refer to $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} (\gamma^k R_{t+k+1})$, where $\gamma \in [0,1]$. In particular, when we refer to "return", it is not necessarily the case that $\gamma = 1$, and when we refer to "discounted return", it is not necessarily true that $\gamma < 1$.

An example comparing state-value and action-value function is shown in Fig. 10. It can be seen that the state-value (on the left) can be obtained from the action-value (on the right).

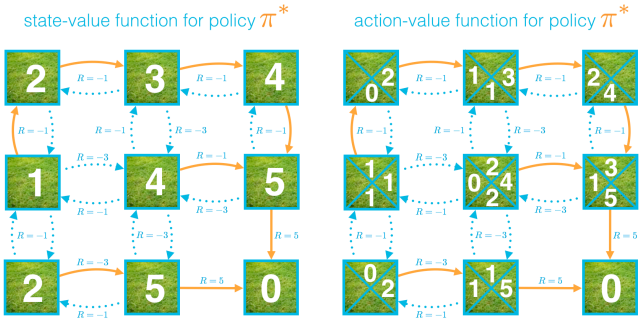


Fig. 10: State-value and action-value example.

Assuming that an agent knows the optimal action-value function and does not know the corresponding optimal policy it is possible to show that for each state we just need to pick the actions that yields the highest expected return as shown in Fig. 10. Hence, if the agent has the optimal action-value function, it can quickly obtain an optimal-policy, which is the solution to the MDP we are looking for.

If the state space S and action space A are finite, we can represent the optimal action-value function q_* in a table as shown in Fig. 11, where we have one entry for each possible environment state $s \in S$ and action $a \in A$.

The value for a particular state-action pair s, a is the expected return if the agent starts in state s, takes action a, and then henceforth follows the optimal policy π_*.

Once the agent has determined the optimal action-value function q_*, it can quickly obtain an optimal policy π_* by setting $\pi_*(s) = \text{argmax}_{a \in A(s)} q_*(s,a)$ for all $s \in S$. To see *why* this should be the case, note that it must hold that $v_*(s) = \max_{a \in A(s)} q_*(s,a)$.

q_*

	a_1	a_2	a_3
s_1	1	(2)	-3
s_2	-2	1	(3)
s_3	(4)	(4)	-5

Fig. 11: Example of some values for a hypothetical Markov decision process (MDP) (where $S = \{s1, s2, s3\}$ and $A = \{a1, a2, a3\}$).

In the event that there is some state $s \in S$ for which multiple actions $a \in A(s)$ maximize the optimal action-value function, an optimal policy can be constructed by placing any amount of probability on any of the (maximizing) actions. We need only to ensure that the actions that do not maximize the action-value function (for a particular state) are given 0% probability under the policy.

Towards constructing the optimal policy, we can begin by selecting the entries that maximize the action-value function, for each row (or state) for the MDP shown in Fig. 11. Thus, the optimal policy π_* for the corresponding MDP must satisfy:

- $\pi*(s_1)=a_2$ (or, equivalently, $\pi*(a_2|s_1)=1$), and
- $\pi*(s_2)=a_3$ (or, equivalently, $\pi*(a_3|s_2)=1$).

This is because $a_2=\text{argmax}_{a\in A(s1)}q*(s_1,a)$, and $a_3=\text{argmax}_{a\in A(s2)}q*(s_2,a)$. In other words, under the optimal policy, the agent must choose action a_2 when in state s_1, and it will choose action a_3 when in state s_2. As for state s_3, note that $a_1,a_2\in\text{argmax}_{a\in A(s3)}q*(s_3,a)$. Thus, the agent can choose either action a_1 or a_2 under the optimal policy, but it can never choose action a_3. That is, for $p,q\geq 0$, and $p+q=1$, the optimal policy $\pi*$ must satisfy:

- $\pi*(a_1|s_3)=p$,
- $\pi*(a_2|s_3)=q$, and
- $\pi*(a_3|s_3)=0$,

III. Q-Learning Algorithm

Q-learning [4] is a popular RL algorithm. We will work with some key features of the Q-learning algorithm and setup an agent in the OpenAI Gym, which provides an environment for the agent to interact with. The agent sends an action to the environment and the environment provides observations and reward in response.

One of the key ideas in solving the RL problem is that if we can figure out the optimal state-action value $q*$ then it becomes straightforward to choose the best policy $\pi*$. So we need to somehow calculate the best station-action value. Each possible state-action pair has a q value. If the agent knows what each true q value is for a given state, it will know which action is the best one to take. But, how does an agent learn what those values really are, since each q value is dependent upon some future reward? That's where the q-learning algorithm comes in.

Q-Learning is a model-free algorithm, which means it can explore an environment which may not be fully defined. The values for each state-action pair are estimated based on observations of that environment. More specifically Q-Learning is a TD or Temporal Difference learning approach, because state changes are learned with the assumption that they are sequential, or time-based. Here is the Q-Learning equation used for calculating the q values represented as a function of the state-action pairs.

$$Q(st,at)\leftarrow(1-\alpha)\cdot Q(s_t,a_t)+\alpha\cdot(r_t+\gamma\cdot\max_a Q(s_{t+1},a))$$

The Q-Learning algorithm is represented with an iterative equation that includes a **learning rate**(α), and a **discount factor**(γ). The learning rate is a value between 0 and 1 and represents the portion of new information that is incorporated into the q-value at each time step. The discount factor is also a value between 0 and 1 and represents the portion of the future rewards that influence the new q-value at each time step. In the equation above, $Q(s_t,a_t)$ is the old value, r_t is the reward, $\max_a Q(s_{t+1},a)$ is the estimate of the optimal future value, and $(r_t+\gamma\cdot\max_a Q(s_{t+1},a))$ is the learned value.

The Q-Learning algorithm, recursively updates or learns the q value for state-action pairs over multiple episodes, which we can think of as "experience". Each update is a combination of the previous estimate of the q value and a new learned value based on a current reward and an estimate of the next best value. The discount factor determines the relative importance of the future rewards compared to a current reward, and the learning rate determines the relative importance of new information during the learning process.

At the beginning of the learning, the q values are set randomly. Each episode results in an update to the Q values influenced by the q-values in possible next steps. The Q value closest to the goal has the highest value and after multiple episodes, the higher-value path propagates backwards through the entire state-action space. Ultimately, we want the q function to estimate or converge to the optimal result, so that we know what actions are optimal in all states? That is the optimal policy.

As the agent explores the environment and acquires experience from different state-action pairs, it converges on a policy of action for any given state it observes.

Q-Learning in its simplest form generally uses a table to keep track of all the discrete Q-values calculated for each state-action pair. In this scenario, the algorithm is iteratively run on multiple episodes until the "best action" for each state converges to a consistent choice. The values may continue to change slightly as the algorithm continues to iterate, but the choice of actions, or policy, does not.

The table in this diagram consists of five possible states that may be observed and three possible actions that the agent can take in response.

The number of states and actions are fixed, but at first, the agent doesn't know what action to take, so the table is initialized with zeros or, in some cases high values to encourage exploration. At the start of each episode, the agent is provided a starting observation, usually a random state.

When the agent knows its state, it can look at the values for all the possible actions it could take and choose the highest value. This kind of move is **exploitation** of the information already available. But is that really the right move? What if the other values are lower just because the agent hasn't had a chance to evaluate them? If that is the case, the agent might get "stuck" and completely miss the true optimal set of actions (the optimal policy). This is why **exploration** by the agent is important. A typical method in Q-Learning is to use the epsilon-Greedy ($\epsilon-Greedy$) exploration method:

$a_t = a_t*$ with probability $1-\epsilon$, or *random action* with probability ϵ.

The larger the epsilon value, the more frequently the agent chooses a random action instead of the action with the highest q-value in its table, a_t*. This improves the chances that a large part of the state-action space is explored and tested. Once convergence has been achieved, the trained agent can be run without the exploration method.

A. The Q-function

We've seen that the policy and value functions are highly interdependent: our policy is determined mostly by what we value, and what we value determines our actions. So maybe we could combine the value and policy functions? We can, and the combination is called the Q-function.

The Q-function takes both the current state (like the value function) and the next action (like the policy function) and returns the partial reward for the state-action pair. For more complex use cases the Q-function may also take more states to predict the next state. For example if the direction of

movement is important one needs at least 2 states to predict the next state, since it is often not possible to infer precise direction from a single state (e.g. a still image). We can also just pass input states to the Q-function to get a partial reward value for each possible action. From this we can (for example) randomly choose our next action with probability proportional to the partial reward (exploration), or just take the highest-valued action (exploitation).

However, the main point of the Q-function is actually different. Consider an autonomous car: there are so many "states" that it is impossible to create a value function for them all; it would take too long to compute all partial rewards for every possible speed and position on all the roads that exist in on Earth. Instead, the Q-function (1) looks at all possible next states that lie one step ahead and (2) looks at the best possible action from the current state to that next state. So for each next state the Q-function has a look-ahead of one step (not all possible steps until termination, like the value function). These look-aheads are represented as state-action pairs. For example, state A might have 4 possible actions, so that we have the action pairs A->A, A->B, A->C, A->D. With four actions for every state and a 10×10 grid of states we can represent the entire Q-function as four 10×10 matrices, or one 10x10x4 tensor. See Figure 3 for a Q-function which represents the solution to a grid world problem (a 2D world where we can move to neighboring states) where the goal is located in the bottom right corner.

B. *Q-learning Example*

Imagine an agent that moves along a line with only five discrete positions, [0, 4]. The agent can move left, right or stay put. If the agent chooses to move left when at position 0 or right at position 4, the agent just remains in place. The goal state is position 3, but the agent doesn't know that and is going to learn the best policy for getting to the goal via the Q-Learning algorithm. The environment will provide a reward of -1 for all locations except the goal state. The episode ends when the goal is reached. For this example, the parameters are very simple:

- Number of states = 5
- Number of actions = 3
- Q-table initialized to all zeros. (This "optimistic initialization" encourages exploration, because all the rewards are negative.)
- Learning rate of α=0.2
- Discount rate of γ=1.0
- Exploration rate of ϵ=0.0 (In other words, ϵ−Greedy is not implemented)

At the start of a new episode, the Q-table and agent are initialized as shown in Fig. 12.

The agent receives an **observation** and a **reward** from the environment: position=1, which is the "next" state, and reward=-1.0 which is based on the "last" action. The agent now knows s_0, a_0, r_0, and s_1 and can update the Q-table value for $Q(s_0,a_0)$. Using the Q-Learning equation from before and substituting our known values

$$Q(s_0,a_0)\leftarrow(1-0.2)\cdot Q(s_0,a_0)+0.2\cdot(r_0+\max_a Q(s_1,a))$$

We can find the *old value* for $Q(s_0,a_0)$ by looking it up in the table for state s_0=1and action a_0=*stay* which is a value of 0. To find the *estimate of the optimal future value*, $\max_a Q(s_1,a)$, we need to look at the entire row of actions for the *next* state, s_1=1 and choose the maximum value across all actions. They are all 0 right now, so the maximum is 0. Reducing the equation, we can now update $Q(s_0,a_0)$.

$$Q(s_0,a_0)\leftarrow-0.2$$

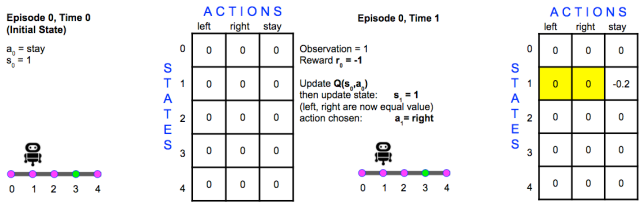


Fig. 12: Q-Learning Example (initial state on the left and the next timestep on the right).

At this step, an action must be chosen. The highest value action for position 1 could be either "left" or "right", since they are equal. In this example, "right" is chosen at random.

The cycle continues with a new observation and reward from the environment.

IV. Deep Reinforcement Learning

A. *OpenAI Gym*

Gym is a toolkit created by OpenAI [6] for the purpose of comparing Reinforcement Learning algorithms. It is not a collection of the many algorithms and agents, but rather a framework for the environments that those agents act within. Gym is open source and includes a wide variety of environment types including:

- Box2D [7] which includes LunarLander
- Classic control [8] such as CartPole and MountainCar
- Atari 2600 [9] games such as Breakout and SpaceInvaders
- MuJoCo [10] which includes the Humanoid robot
- Robotics [11] including FetchPickAndPlace and HandManipulate

More information about how OpenAI Gym is used to accelerate reinforcement learning (RL) research can be found in the OpenAI blog post [12].

B. *OpenAI Gym Installation Instructions*

More on OpenAI Gym can be found in the GitHub repository [13].

In order to install OpenAI Gym on a machine, the following minimal install steps must be followed. Start by adding libav [14], which provides cross-platform tools and libraries to convert, manipulate and stream a wide range of multimedia formats and protocols:

$ sudo apt-get install libav-tools

Then install OpenAI Gym**:**

$ git clone https://github.com/openai/gym.git

$ cd gym

$ pip install -e .

Once OpenAI Gym has been installed, obtain the code for the classic control tasks (such as 'CartPole-v0'):

$ pip install -e '.[classic_control]'

Finally, check the installation by running the *simple random agent* provided in the examples directory.

$ cd examples/agents

$ python random_agent.py

These instructions are derived from the README in the OpenAI Gym GitHub repository [13].

C. RL to Deep RL

Combining deep learning to reinforcement learning, the new algorithm called Deep RL can be used to train an agent more efficiently. We let the network (the agent in this case) learn the best way to solve the problem based on its experience rather than dictating to the system how best to solve the problem (Fig. 13). Deep RL involves the use of deep Q networks (DQN) and key ideas of Experience Replay and Fixed Q Targets.

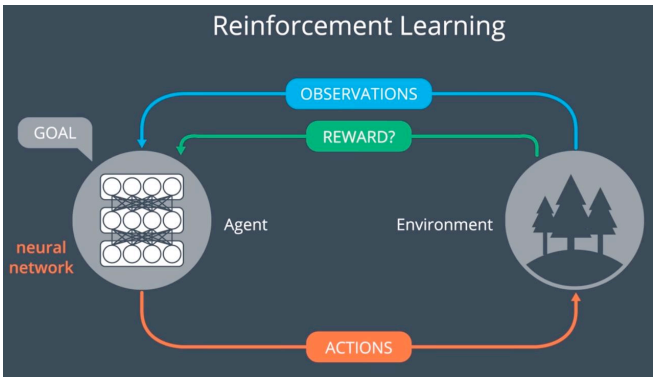


Fig. 13: Deep RL framework where the algorithm used by the agent for determining best actions to take in order to arrive at the goal is replaced by a deep neural network.

A key difference between RL and Deep RL is the use of a deep neural network (Fig. 14). Think of the collection of value-action pairs that define what actions an agent should take in any situation as a function of the observations that the agent receives from its environment. A neural network can be used to approximate this function because through its large quantity of parameters that can be "learned" through trial and error.

Deep RL is essentially an online version of the neural fitted Q value iteration [15]. Which uses the training of a Q-value function represented by a multi-layer perceptron. The DQN uses a rolling history of the past data via replay pool. By using the replay pool, the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations. This has the advantage that each step of the experience is potentially used in many weight updates. The other big idea is that use of a target network to represent the old Q-functions, which is used to compute the loss of every action during training. Why not use a single network? The issue with a single network is that at each step of the training, the Q-functions values change and then the value estimates can easily spiral out of control. These additions enable RL agents to converge, more reliably during training.

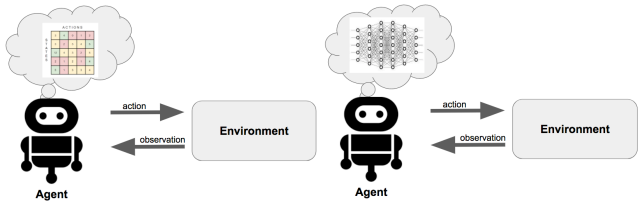


Fig. 14: RL (left) uses action-state table whereas Deep RL (right) uses neural network.

D. Deep Q-Network (DQN)

In 2015, DeepMind made a breakthrough by designing an agent that learned to play video games better than humans [1]. It is probably easy to write a program that plays Pong perfectly if we have access to the underlying game state, position of the ball, paddles, etc. However, this agent was only given raw pixel data, what a human player would see on a screen. Moreover, it learned to play a bunch of different Atari games, all from scratch. This agent is called a deep-Q network or DQN [5]. At the heart of this agent is a deep neural network that acts as a function approximate. We pass in images from our favorite video game one screen at a time, and it produces a vector of action values, with the max value indicating the action to take. As a reinforcement signal, it is fed back the change in game score at each rime step. In the beginning when the neural network is initialized with random values, the auctions taken are all over the place. However, over time it begins to associate situations and sequences in the game with appropriate actions and learns to actually play the game well.

Consider how complex the input space is. Atari games are displayed at a resolution of 210x160 pixels, with 128 possible colors for each pixel. This is still technically a discrete state space but very large to process as is. To reduce this complexity, we can convert the frames to gray scale, and scale-down the images to a square 84x84 pixel blocks. Square images allows us to use more optimized neural network operations on GPUs. In order to give the agent access to a sequence of frame, four frames stacked together are fed in, resulting in a final state space size of 84x84x4 (Fig. 15). There might be other approaches as well to deal with sequential data, but this was a simple approach that seems to work well.

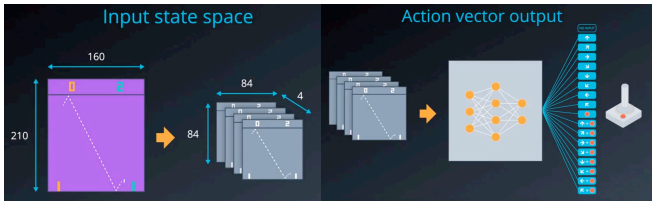


Fig. 15: (Left) Transformed input state space for DQN. (Right) Action vector output of the DQN

On the output side (Fig. 15), unlike a traditional reinforcement learning setup where only one Q-value is produced at a time, the deep Q-network is designed to produce a Q value for every possible action in a single forward pass. Without this, we would have to run the network individually for every action. Instead, we can now simply use this vector to take an action, either stochastically or by choosing the one with the maximum value.

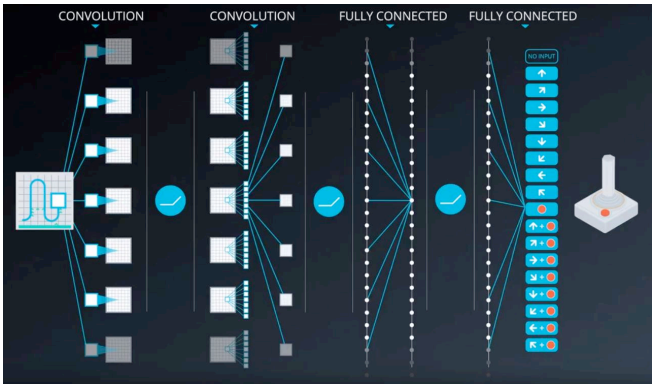


Fig. 16: The deep architecture of the DQN

These innovative input and output transformations support a powerful yet simple neural networks architecture under the hood. The screen images are first processed by convolutional layers. This allows the system to exploit spatial relationships, and spatial rule space. Also, since four frames and stacked and provided as input, these convolution layers also extract some temporal properties across those frames. The original DQN agent used three such convolutional layers with ReLU (regularized linear units) activations. They were followed by one fully-connected hiddel layer with ReLU activation, and one fully-connected linear output layer that produced the vector of action values. This same architecture (Fig. 16) was used for all the Atari games tested by DeepMind, but each game was learned from scratch with a freshly initialized network.

Training such networks takes a lot of data. But even then , it is not guaranteed to converge on the optimal value function. In fact, there are situations where the network weights can oscillate or diverge, due to the high correlation between actions and states. This can result in a very unstable and ineffective policy. In order to overcome these challenges several techniques are used that slightly modifies the base Q learning algorithm. Two of these techniques are (1) Experience Reply, and (2) Fixed Q Targets.

An autonomous vehicle may need to consider many states: every different combination of speed and position is a different state. However, most states are similar. Is it possible to combine similar states so that they have similar Q-values? Here is where deep learning comes into play. [16]

We can input the current point of view of the driver, an image, into a convolutional neural network (CNN) trained to predict the rewards for the next possible actions. Because images for similar states are similar (many left turns look similar), they will also have similar actions. For example, a neural network will be able to generalize for many left turns on a racetrack and even make appropriate actions for left turns that have not encountered before. Just like a CNN trained on many different objects in images can learn to correctly distinguish these objects, a network trained on many similar variations of left turns will be able to make fine adjustments of speed and position for each different left turn it is presented.

If we apply the Q-learning rule blindly, then the network will learn to do good left turns while doing left turns, but will start forgetting how to do good right turns at the same time. This is so because all actions of the neural network use the same weights; tuning the weights for left turns makes them worse for other situations. The solution is to store all the input images and the output actions as "experiences": that is, store the state, action and reward together.

After running the training algorithm for a while, we choose a random selection from all the experiences gathered so far and create an average update for neural network weights which maximizes Q-values (rewards) for all actions taken during those experiences. This way we can teach our neural network to do left and right turns at the same time. Since very early experiences of driving on a racetrack are not important, because they come from a time where our agent was less experienced, or even a beginner, we only keep track of a fixed number of past experiences and forget the rest. This process is termed Experience Replay.

Experience replay is a method inspired by biology. The hippocampus in the human brain is a reinforcement learning center in each hemisphere of the brain. The hippocampus stores all the experiences that we make during the day but it has a limited memory capacity for experiences and once this capacity is reached learning becomes much more difficult (cramming too much before an exam). During the night this memory buffer in the hippocampus is emptied into the cortex by neural activity that spreads across the cortex. The cortex is the "hard drive" of the brain where almost all memories are stored. Memories for hand movements are stored in the "hand area", memories for hearing are stored in the "hearing area", and so on. This characteristic neural activity that spreads outward from the hippocampus is termed sleep spindle. While there is currently no strong evidence to support it, many sleep researchers believe that we dream to help the hippocampus to integrate the experiences gathered over the day with our memories in the cortex to form coherent pictures [17].

Therefore, we see that storing memories and writing them back in a somewhat coordinated fashion is an important process not only for deep reinforcement learning but also for human learning. This biological similarity gives us a little confidence boost that our theories of the brain might be correct and that the algorithms we design are on the right path towards intelligence.

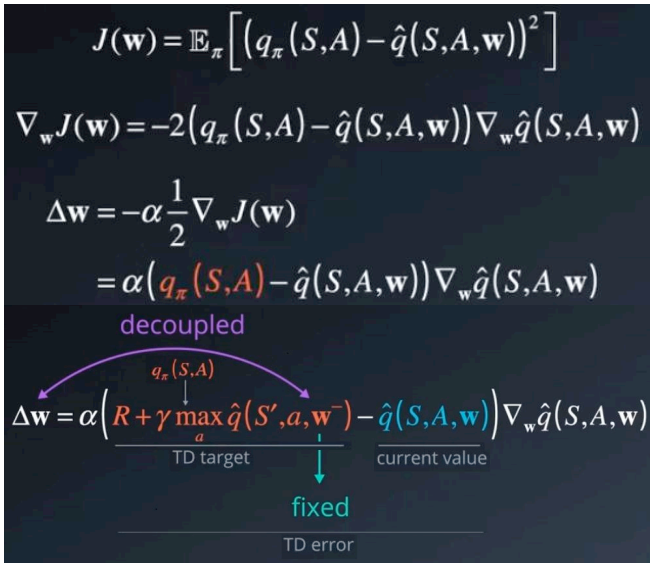


Fig. 17: The cost function for the Q-learning fixed-target update.

Experience reply helps us address one type of correlation, i.e. between consecutive experience tuples. The other kind of

correlation that Q-learning is susceptible to is called the Fixed-Q Targets. Since Q-learning is a form of Temporal Difference (TD) learning, where the reward (R) plus discount factor (γ) times the maximum possible value from the next state is called the TD target. And our goal is to reduce the difference between this target and the currently predicted Q-values. This difference is the TD error (Fig. 17).

The TD taget is supposed to be a replacement for the true value function $q_\pi(s,A)$ which is unknown. We originally used q_π to define a squared error loss, and differentiated that with respect to w to get our gradient descent update rule. Now, q_π is not dependent on our function approximation or its parameters, thus resulting in a simple derivative and update rule. However, our TD target is dependent on these parameters which means simply the true value function q_π with a target like this is mathematically incorrect. We can get away with it in practice because every update results in a small change to the parameters, which is generally in the right direction. If we set the learning rate $\alpha = 1$, and leap towards the target then we would likely overshoot and land in the wrong place. Also, this is less of a concern when we use a lookup table or a dictionary since Q-values are stored separately for each state action pair. But it can affect learning significantly when we use function approximation, where all the Q-values are intrinsically tied together through the function parameters. We may ask, "doesn't the experience replay take care of this problem?".

Experience replay addressees a similar but slightly different issue. In experience replay, we broke the correlation effects between consecutive experience tuples by sampling them randomly out of order. Here the correlation is between the target and the parameters we are changing, which s like chasing a moving target. In order to create a stable learning environment, some reward must be given at each step by fixing the function parameters used to generate our target. The fixed parameters indicated by a w^-, are basically a copy of w that we don't change during the learning step. In practice, we copy w into w^- and use it to generate targets while changing changing w for a certain number of learning steps. Then, we update w^- with the latest w, again learn for a number of steps and so on. This decouples the target from the parameters, makes the learning algorithm much more stable and less likely to diverge or fall into oscillations.

V. DQN Implementation and Optimization

The Q-learning algorithm specifies how to approximate the Q-function with a series of updates. The updated state-action pair can put into a table for lookup to find the optimal action. However, for realistic problems it may be intractable to create such a table for all possibilities. Instead, we need a function approximator, which is where the deep Q-network comes in.

In this project we will use OpenAI Gym to solve the CartPole problem with a DQN implementation. Before applying DQN, we need to learn about the python machine learning library PyTorch for defining and training deep neural networks. It has an intuitive interface for defining networks and provis strong GPU acceleration. After that, we will run our agent with the DQN algorithm in OpenAI Gym, this time with the pixels to actions paradigm. In other words, we will use screen captures of the game as inputs to DQN rather than specific cart and pole location observations as is used with Q-learning.

A. PyTorch

PyTorch [17] is an open source Python-based deep learning framework released in October 2016. It was inspired by the Torch framework [18], which was originally implemented in C with a wrapper in the Lua scripting language. PyTorch wraps the core Torch binaries in Python and provides GPU acceleration for many functions.

Most frameworks such as TensorFlow, Theano, and Caffe require static graphs to define networks. The network must first be built, then run. Any change in the network structure requires building from scratch. By contrast, PyTorch allows us to change the way our network behaves on the fly.

PyTorch is designed to be intuitive and linear. Code is executed in-line as Python is. This is a help in debugging as the stack-trace points to exactly where code was defined.

In summary, below are some of the ley features of PyTorch as a deep learning platform:

- PyTorch can perform tensor computations like numpy, but with stronger GPU acceleration.
- PyTorch builds computation graphs dynamically, which provides greater flexibility during development.
- PyTorch is designed to be intuitive, linear in throught, and easy to use. When one inputs a line of code it gets executed.

B. C/C++ API

Our focus is to use Deep RL to get a robot to learn a new task, in a 3D world in real-time. In order to do this efficiently, we need use a C++ API (application programming interface) to run the RL agent, which will set us up to run robotic platforms that can run much faster with compiled code using GPU acceleration. We will test the API with a simple game, and then implement our agent in a Gazebo world. The Deep RL API and examples used in this project are based on the Nvidia jetson-reinforcement open source project [19].

Why do we need a C/C++ API? The PyTorch DQN tutorial provides a good demonstration for how deep reinforcement learning works for problems that take sensor input and produce actions. However, to successfully leverage deep learning technology in robots, we need to move to a library format that can integrate with robots and simulators. In addition, robots require real-time responses to changes in their environments, so computation performance matters. In this section, we introduce an API in C/C++.

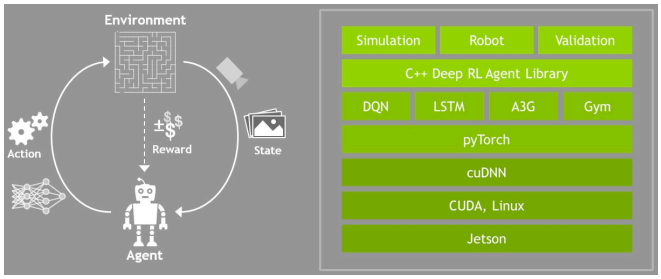


Fig. 18: NVidia Deep RL C/C++ API framework [20]

The API structure shown in Fig. 18 provides an interface to the Python code written with PyTorch, but the wrappers use Python's low-level C to pass memory objects between

the user's application and Torch without extra copies. By using a compiled language (C/C++) instead of an interpreted one, performance is improved, and speeded up even more when GPU acceleration is leveraged.

API Repository Sample Environments: The API repository provides instructions and files to build a few different RL agents and environments from source with PyTorch on either a Jetson or other GPU x86_64 system. In addition to OpenAI Gym samples similar to those already covered here (Cartpole), the repository contains the following demos:

- C/C++ 2D Samples
  - Catch (DQN text)
  - Fruit (2D DQN)
- C/C++ 3D Simulation
  - Robotic Arm (3D DQN in Gazebo)
  - Rover Navigation (3D DQN in Gazebo)

The purpose of building the simple 2D samples is to test and understand the C/C++ API as we move toward the goal of using the API for robotic applications in Gazebo such as the Robotic Arm. Each of these samples will use a DQN agent to solve problems.

The DQN agent: The repo provides a base rlAgent base class that can be extended through inheritance to implement agents using various reinforcement learning algorithms. We will focus on the dqnAgent class and applying it to solve DQN reinforcement learning problems.

Setting up the Deep RL agent: The agent is instantiated by the Create() function with the appropriate initial parameters. For each iteration of the algorithm, the environment provides sensor data, or environmental state, to the NextAction() call, which returns the agent's action to be applied to the robot or simulation. The environment's reward is issued to the NextReward() function, which kicks off the next training iteration that ensures the agent learns over time.

Below is list of the parameters that can be set up in the Create() function.

```
// Define DQN API settings
#define GAME_WIDTH  64          // Set an environment width
#define GAME_HEIGHT  64         // Set an environment height
#define NUM_CHANNELS 1          // Set the image channels
#define OPTIMIZER "RMSprop"     // Set a optimizer
#define LEARNING_RATE 0.01f     // Set an optimizer learning rate
#define REPLAY_MEMORY 10000 // Set a replay memory
#define BATCH_SIZE 32           // Set a batch size
#define GAMMA 0.9f              // Set a discount factor
#define EPS_START 0.9f          // Set a starting greedy value
#define EPS_END 0.05f           // Set a ending greedy value
#define EPS_DECAY 200           // Set a greedy decay rate
#define USE_LSTM true           // Add memory (LSTM) to network
#define LSTM_SIZE 256           // Define LSTM size
#define DEBUG_DQN false         // Turn on or off DQN debug mode
#define ALLOW_RANDOM true   // Allow RL agent to make random
choices
```

VI. Deep RL for Robot Simulation

In this project, we will get to explore and work with the environment that uses the C++ API and Gazebo. There are three main components to the gazebo file gazebo-arm.world in /gazebo/ folder in the DeepRL repo [20], which defines the environment:

- The robotic arm with a gripper attached to it.
- A camera sensor, to capture images to feed into the DQN.
- A cylindrical object or prop.

A. *Arm Plugin*

The robotic arm model, found in the gazebo-arm.world file, calls upon a gazebo plugin called the ArmPlugin. This plugin is responsible for creating the DQN agent and training it to learn to touch the prop. The gazebo plugin shared object file, libgazeboArmPlugin.so, attached to the robot model in gazebo-arm.world, is responsible for integrating the simulation environment, shown in Fig. 19, with the RL agent. The plugin is defined in the ArmPlugin.cpp file, also located in the gazebo folder.

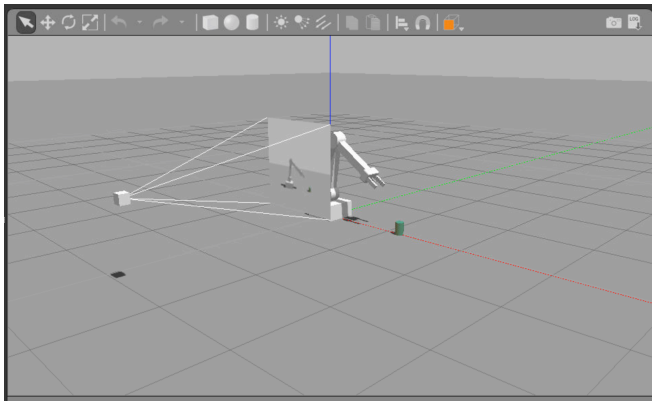


Fig. 19: Gazebo environment for the Deep RL agent.

The ArmPlugin.cpp file takes advantage of the C++ API discussed in the previous section. This plugin creates specific constructor and member functions for the class ArmPlugin defined in ArmPlugin.h. Some of the important methods, that we will require for the project, are discussed below:

ArmPlugin::Load(): This function is responsible for creating and initializing nodes that subscribe to two specific topics - one for the camera, and one for the contact sensor for the object. In gazebo, subscribing to a topic has the following structure:

```
gazebo::transport::SubscriberPtr sub = node->Subscribe("topic_name", call
back_function, class_instance);
```

Where, callback_function is the method that's called when a new message is received, and class_instance is the instance used when a new message is received. Documentation detailing this can be found in [21, 22]. For each of the two subscribers, there is a callback function defined in the file and is discussed below.

ArmPlugin::onCameraMsg(): This is the callback function for the camera subscriber. It takes the message from the camera topic, extracts the image, and saves it. This is then passed to the DQN.

ArmPlugin::onCollisionMsg(): This is the callback function for the object's contact sensor. This function is used to test whether the contact sensor, called my_contact, defined for the object in gazebo-arm.world, observes a collision with another element/model or not. Furthermore, this callback function can also be used to define a reward function based on whether there has been a collision or not.

ArmPlugin::createAgent(): Previously, for the fruit and catch samples, we created a DQN agent. The createAgent() class function serves the same purpose wherein we can create and initialize the agent. In ArmPlugin.cpp, the various parameters that are passed to the Create() function for the agent are defined at the top of the file, some of them are:

```
#define ALLOW_RANDOM true
#define DEBUG_DQN false
#define INPUT_WIDTH   512
#define INPUT_HEIGHT  512
#define INPUT_CHANNELS 3
#define OPTIMIZER "None"
#define LEARNING_RATE 0.0f
#define BATCH_SIZE 8
#define EPS_START 0.9f
#define EPS_END 0.05f
#define EPS_DECAY 200
#define GAMMA 0.9f
```

ArmPlugin::updateAgent(): For every frame that the camera receives, the agent needs to take an appropriate action. Since the DQN agent is discrete, the network selects one output for every frame. This output (action value) can then be mapped to a specific action - controlling the arm joints. The updateAgent() method, receives the action value from the DQN, and decides to take that action. There are two possible ways to control the arm joints:

- Velocity Control
- Position Control

For both of these types of control, we can increase or decrease either the joint velocity or the joint position, by a small delta value. We can experiment with several different approaches to define on what basis the joint velocity or position changes. For example, one such action can be based on whether the action value is odd or even.

ArmPlugin.cpp file includes the variable VELOCITY_CONTROL, set to false by default, which can be used to define whether we want to control joint velocities or positions.

ArmPlugin::OnUpdate(): This method is primarily utilized to issue rewards and train the DQN. It is called upon at every simulation iteration and can be used to update the robot joints, issue end of episode (EOE) rewards, or issue interim rewards based on the desired goal.

At EOE, various parameters for the API and the plugin are reset, and the current accuracy of the agent performing the appropriate task is displayed on the terminal.

B. Recurrent Neural Network (RNN)

The deep neural network architectures, specifically feed-forward networks, have the limitation that they do not have a memory element to them. They were trained using the current input only. RNNs [23] are a type of neural network that utilize memory, i.e. the previous state, to predict the current output as shown in Fig. 20.

RNNs have a wide range of applications, such as in natural language processing for machine translation, in computer vision for gesture recognition, speech recognition etc. But they also have certain limitations. RNNs are more effective when they are only trying to learn from the most recent information. For example, if we were trying to predict the last word in the sentence -

my mobile robot has four …,

an RNN would be able to predict the last word as wheels with high probability given the context. However, if the sentence was far longer and complicated, the RNN would struggle to maintain the context over each timestep and predict the last word with high probability.

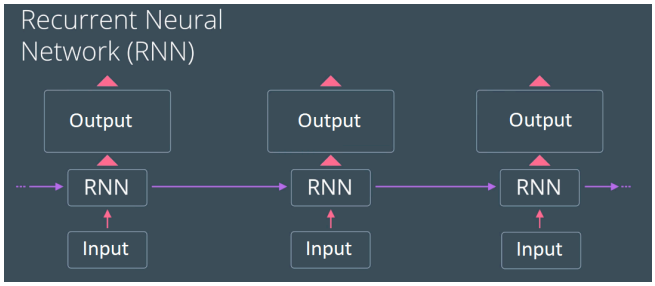


Fig. 20: RNN layout

This shortcoming of RNNs, where they are unable to learn from long-term dependencies, is handled by another architecture called Long Short Term Memory, or LSTM [24, 25] as discussed below.

C. Long Short Term Memory (LSTM)

The LSTM [26] architecture keeps track of both, the long-term memory and the short-term memory, where the short-term memory is the output or the prediction.

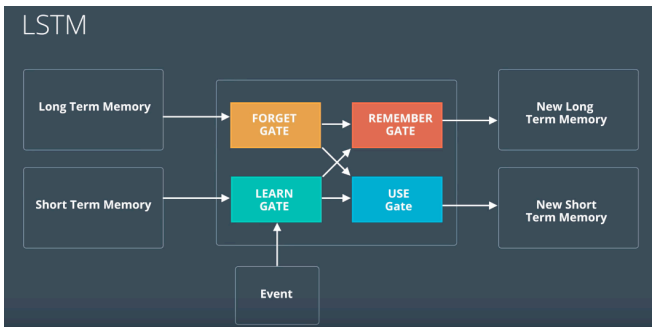


Fig. 21: LSTM architecture

The architecture (Fig. 21) of LSTM consists of four "gates" that carry out specific functions -

- Forget Gate - The long-term memory is input to this gate and any information that is not useful, is removed.
- Learn Gate - The short-term memory is input to this gate, along with the input (or event) to the LSTM at current timestep. It contains or outputs information that is recently learned and removes any non-useful information.
- Remember Gate - The output of the Forget Gate and the Learn Gate are fed into the Remember Gate, and it outputs an updated long-term memory.
- Use Gate - This gate uses the information from the Learn gate and the Forget Gate to make a prediction. This prediction acts as the short-term memory for the next timestep.

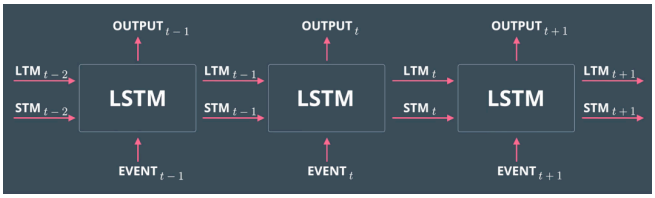


Fig. 22: Network with multiple LSTM nodes.

The image in Fig. 21 represents a single LSTM unit or cell. The image in Fig. 22 is an example of a network which depicts multiple LSTM nodes.

VII. Project Setup

For this project, our goal is to create a DQN agent and define reward functions to teach a robotic arm to carry out two primary objectives:

- Have any part of the robot arm touch the object of interest, with at least 90% accuracy.
- Have only the gripper base of the robot arm touch the object, with at least 80% accuracy.

Both of these objectives, have associated tasks that we will complete while working on this project.

To launch the project for the first time, run the following in the terminal of the desktop gui:

$ cd /home/RoboND-DeepRL-Project/build/x86_64/bin

$ ./gazebo-arm.sh

Gazebo can often take time to load. As a byproduct of that, we might see errors similar to the following -

[Err] [Scene.cc:2927] Light [sun] not found. Use topic ~/factory/light to spawn a new light.

These errors should go away once Gazebo loads up completely, and can be ignored. During the first launch of Gazebo, if it's taking too long, or if it throws errors and doesn't completely launch, then we need to stop the script and launch it again.

Once the gazebo environment loads up, we will see the robotic arm, a camera sensor, and an object in the environment. The gazebo arm will fall to the ground after a short while, and the terminal will continuously display the following message:

ArmPlugin - failed to create DQN agent

Since no DQN agent has been defined as of now, the arm isn't learning anything and has no input to control it. For this project, various tasks need to be completed in order for the robot arm to successfully learn to reach the object. These are mainly located in ArmPlugin.cpp file [27] and the following sections describe how to achieve the desired result.

A. *Subscribe to camera and collision topics*

The nodes corresponding to each of the subscribers have already been defined and initialized. We will have to create the subscribers in the ArmPlugin::Load() function as shown below (Fig. 23).

cameraSub = cameraNode-> Subscribe("/gazebo/arm_world/camera/link/camera/image", &ArmPlugin::onCameraMsg, this);

collisionSub = collisionNode-> Subscribe("/gazebo/arm_world/tube/tube_link/my_contact", &ArmPlugin::onCollisionMsg, this);

```
// Load
void ArmPlugin::Load(physics::ModelPtr _parent, sdf::ElementPtr /*_sdf*/)
{
    printf("ArmPlugin::Load('%s')\n", _parent->GetName().c_str());

    // Store the pointer to the model
    this->model = _parent;
    this->j2_controller = new physics::JointController(model);

    // Create our node for camera communication
    cameraNode->Init();

    /*
    / TODO - Subscribe to camera topic
    /
    */
    cameraSub = cameraNode->Subscribe("/gazebo/arm_world/camera/link/camera/image", &ArmPlugin::onCameraMsg, this);

    // Create our node for collision detection
    collisionNode->Init();

    /*
    / TODO - Subscribe to prop collision topic
    /
    */
    collisionSub = collisionNode->Subscribe("/gazebo/arm_world/tube/tube_link/my_contact", &ArmPlugin::onCollisionMsg, this);

    // Listen to the update event. This event is broadcast every simulation iteration.
    this->updateConnection = event::Events::ConnectWorldUpdateBegin(boost::bind(&ArmPlugin::OnUpdate, this, _1));
}
```


Fig. 23: Subcribing to camera and collision topics in the AmrPlugin's Load() function.

B. *Create the DQN Agent*

Refer to the API instructions to create the agent using the Create() function from the dqnAgent Class, in ArmPlugin::createAgent(). We pass the variable names defined at the top of the file to this function call (referring to the constructor in project_folder/c/dqnAgent.cpp file) as shown below (Fig. 24).

```
// CreateAgent
bool ArmPlugin::createAgent()
{
    if( agent != NULL )
        return true;


    /*
    / TODO - Create DQN Agent
    /
    */
    uint32_t NUM_ACTIONS = 2*DOF;   //Define the number of actions available.
    agent = dqnAgent::Create(INPUT_WIDTH, INPUT_HEIGHT, INPUT_CHANNELS, NUM_ACTIONS,
                             OPTIMIZER, LEARNING_RATE, REPLAY_MEMORY, BATCH_SIZE,
                             GAMMA, EPS_START, EPS_END, EPS_DECAY,
                             USE_LSTM, LSTM_SIZE, ALLOW_RANDOM, DEBUG);


    if( !agent )
    {
        printf("ArmPlugin - failed to create DQN agent\n");
        return false;
    }

    // Allocate the python tensor for passing the camera state
    inputState = Tensor::Alloc(INPUT_WIDTH, INPUT_HEIGHT, INPUT_CHANNELS);

    if( !inputState )
    {
        printf("ArmPlugin - failed to allocate %ux%ux%u Tensor\n", INPUT_WIDTH, INPUT_HEIGHT, INPUT_CHANNELS);
        return false;
    }


    return true;
}
```


Fig. 24: Creating the DQN agent in AmrPlugin's createAgent() function.

agent = dqnAgent::Create(INPUT_WIDTH, INPUT_HEIGHT, INPUT_CHANNELS, NUM_ACTIONS, OPTIMIZER, LEARNING_RATE, REPLAY_MEMORY, BATCH_SIZE, GAMMA, EPS_START, EPS_END, EPS_DECAY, USE_LSTM, LSTM_SIZE, ALLOW_RANDOM, DEBUG);

C. *Velocity or position based control of arm joints*

Previously, we discussed how the DQN output is mapped to a particular action, which, for this project, is the control of each joint for the robotic arm. In ArmPlugin::updateAgent(), there are two existing approaches to control the joint movements (Fig. 25). We can either use one of these approaches, or we can come up with our own solution.

- Velocity Control - The current value for a joint's velocity is stored in the array *vel* with the array lengths based on the number of degrees of freedom for the arm. If we choose to select this control strategy, the following line of code can be used to assign a new value to the variable velocity based on the current joint velocity and the associated delta value, actionVelDelta.

float velocity = vel[action/2] + ((action % 2 == 0) ? 1.0f : -1.0f) * actionVelDelta;

- Position Control - The current value for a joint's position is stored in the array *ref* with the array

lengths based on the number of degrees of freedom for the arm. If we choose to select this control strategy, the following line of code can be used to assign a new value to the variable joint based on the current joint position and the associated delta value, actionJointDelta.

```
float joint = ref[action/2] + ((action % 2 == 0) ? 1.0f : -1.0f) * actionJointDelta;
```

```
#if VELOCITY_CONTROL
    /*
    / TODO - Increase or decrease the joint velocity based on whether the action is even or odd
    /
    */
    // *** TODO - Set joint velocity based on whether action is even (increase by delta) or odd (decrease by delta). *** //
    float velocity = vel[action/2] + ((action % 2 == 0) ? 1.0f : -1.0f) * actionVelDelta;
    if(DEBUG){printf("Joint Velocity: %f\n", velocity);}

    if( velocity < VELOCITY_MIN )
        velocity = VELOCITY_MIN;

    if( velocity > VELOCITY_MAX )
        velocity = VELOCITY_MAX;

    vel[action/2] = velocity;

    for( uint32_t n=0; n < DOF; n++ )
    {
        ref[n] += vel[n];

        if( ref[n] < JOINT_MIN )
        {
            ref[n] = JOINT_MIN;
            vel[n] = 0.0f;
        }
        else if( ref[n] > JOINT_MAX )
        {
            ref[n] = JOINT_MAX;
            vel[n] = 0.0f;
        }
    }
#else
    /*
    / TODO - Increase or decrease the joint position based on whether the action is even or odd
    /
    */
    // *** TODO - Set joint position based on whether action is even (increase by delta) or odd (decrease by delta). *** //
    float joint = ref[action/2] + ((action % 2 == 0) ? 1.0f : -1.0f) * actionJointDelta;
    if(DEBUG){printf("Joint Position: %f\n", joint);}

    // limit the joint to the specified range
    if( joint < JOINT_MIN )
        joint = JOINT_MIN;

    if( joint > JOINT_MAX )
        joint = JOINT_MAX;

    ref[action/2] = joint;
#endif
```


Fig. 25: Defining joint velocity and joint position control.

To find out which joint's value to change (corresponding index to either of the arrays, ref or vel) we can define the index as a function of the variable action. This is helpful if we want to have a more complicated (more degrees of freedom) arm without having to define new set of conditions. The default number of actions, defined in DQN.py, is 3; and there are two outputs for every action - increase or decrease in the joint angles.

D. Reward for robot gripper hitting the ground

The next set of tasks are based on creating and assigning reward functions based on the required goals. There are a few important variables in relation to rewards -

- *rewardHistory* - Value of the previous reward, we can set this to either a positive or a negative value.
- *REWARD_WIN* or *REWARD_LOSS* - The values for positive or negative rewards, respectively.
- *newReward* - If a reward has been issued or not.
- *endEpisode* - If the episode is over or not.

Note that for each of the following, we have to decide what rewards to use, and whether or not it's a new reward and whether or not to trigger an end of episode.

In Gazebo's API [28], there is a function called GetBoundingBox() which returns the minimum and maximum values of a box that defines that particular object/model corresponding to the x, y, and z axes.

For example, for the bounding box defined for the robot's base, we will have the attributes box.max.x and box.min.x to obtain the minimum and maximum values of the box in reference to the x-axis.

```
        // get the bounding box for the gripper
        const math::Box& gripBBox = gripper->GetBoundingBox();
        const float groundContact = 0.05f;

        /*
        / TODO - set appropriate Reward for robot hitting the ground.
        /
        */
        // compute the distance to the goal
        const float distGoal = BoxDistance(gripBBox, propBBox);
        if(DEBUG){printf("distance('%s', '%s') = %f\n", gripper->GetName().c_str(), prop->model->GetName().c_str(), distGoal);}

        // issue reward based on distance from goal when robot hits ground
        bool checkGroundContact = (gripBBox.min.z <= groundContact || gripBBox.max.z <= groundContact) ? true : false;
        if(checkGroundContact)
        {

            if(DEBUG){printf("GROUND CONTACT, EOE\n");}
            rewardHistory = REWARD_LOSS;
            newReward     = true;
            endEpisode    = true;
            if(DEBUG){std::cout << "Ground Contact Detected. Reward: " << rewardHistory << "\n";}
        }
```


Fig. 26: Compute the distance of the gripper to the goal, and issue reward based on it when robot arm hits the ground.

Using the above, we can check if the gripper is hitting the ground or not, and assign an appropriate reward. The bounding box for the gripper, and a threshold value have already been defined for us in the ArmPlugin::OnUpdate() method.

```
// compute the distance to the goal
const float distGoal = BoxDistance(gripBBox, propBBox);
// issue reward based on distance from goal when robot hits ground
bool checkGroundContact = (gripBBox.min.z <= groundContact || gripBBox.max.z <= groundContact) ? true : false;
if(checkGroundContact)
{
    rewardHistory = REWARD_LOSS;
    newReward     = true;
    endEpisode    = true;
}
```

E. Issue interim reward based on the distance to the object

In ArmPlugin.cpp a function called "BoxDistance()" calculates the distance between two bounding boxes. Using this function, calculate the distance between the arm and the object. Then, use this distance to calculate an appropriate reward as well.

```
        /
        / TODO - Issue an interim reward based on the distance to the object
        /
        */
        // *** TASK-1 && TASK-2 *** //
        if(!checkGroundContact)
        {
            if( episodeFrames > 1 )
            {
                const float distDelta = lastGoalDistance - distGoal;

                // compute the smoothed moving average of the delta of the distance to the goal
                avgGoalDelta  = (avgGoalDelta * REWARD_ALPHA) + (distDelta * (1.0f - REWARD_ALPHA));

                // Computing reward based on distance from goal.
                rewardHistory = avgGoalDelta;   //TASK-1 && TASK-2
                newReward = true;
            }
            lastGoalDistance = distGoal;
        }
    }
```

Fig. 27: Issue an interim reward based on distance of the robot arm from the object.

A smoothed moving average of the delta of the distance to the goal is used as a reward function and is calculated as,

average_delta = (average_delta * alpha) + (dist * (1 - alpha));

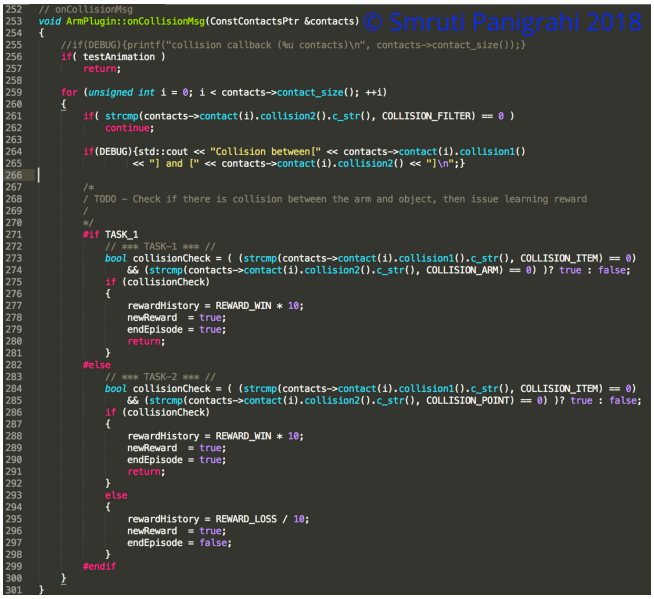
The following reward function will help us with both parts of the objectives of having any part of the robot arm as well as the robot gripper touch the object of interest, with at least a 90% accuracy.

```
// *** TASK-1 && TASK-2 *** //
if(!checkGroundContact)
{
    if( episodeFrames > 1 )
    {
            const float distDelta = lastGoalDistance - distGoal;
            // compute the smoothed moving average of the delta of the distance to the goal
            avgGoalDelta    = (avgGoalDelta * REWARD_ALPHA) + (distDelta * (1.0f – REWARD_ALPHA));
            // Computing reward based on distance from goal.
            rewardHistory = avgGoalDelta;    //TASK-1 && TASK-2
```

```
        newReward = true;
        }
        lastGoalDistance = distGoal;
    }
```

F. *Issue reward based on collision between the arm and the object and between the arm's gripper base and the object*

In the callback function onCollisionMsg, we can check for certain collisions. Specifically, we will define a check condition to compare if particular links of the arm with their defined collision elements are colliding with the COLLISION_ITEM or COLLISION_ARM or COLLISION_POINT (Fig. 28) and then assign appropriate rewards.




```
// onCollisionMsg
void ArmPlugin::onCollisionMsg(ConstContactsPtr &contacts)
{
    //if(DEBUG){printf("collision callback (%u contacts)\n", contacts->contact_size());}
    if( testAnimation )
        return;

    for (unsigned int i = 0; i < contacts->contact_size(); ++i)
    {
        if( strcmp(contacts->contact(i).collision2().c_str(), COLLISION_FILTER) == 0 )
            continue;

        if(DEBUG){std::cout << "Collision between[" << contacts->contact(i).collision1()
                << "] and [" << contacts->contact(i).collision2() << "]\n";}

        /*
        / TODO - Check if there is collision between the arm and object, then issue learning reward
        /
        */
        #if TASK_1
            // *** TASK-1 *** //
            bool collisionCheck = ( (strcmp(contacts->contact(i).collision1().c_str(), COLLISION_ITEM) == 0)
                || (strcmp(contacts->contact(i).collision2().c_str(), COLLISION_ARM) == 0) )? true : false;
            if (collisionCheck)
            {
                rewardHistory = REWARD_WIN * 10;
                newReward  = true;
                endEpisode = true;
                return;
            }
        #else
            // *** TASK-2 *** //
            bool collisionCheck = ( (strcmp(contacts->contact(i).collision1().c_str(), COLLISION_ITEM) == 0)
                || (strcmp(contacts->contact(i).collision2().c_str(), COLLISION_POINT) == 0) )? true : false;
            if (collisionCheck)
            {
                rewardHistory = REWARD_WIN * 10;
                newReward  = true;
                endEpisode = true;
                return;
            }
            else
            {
                rewardHistory = REWARD_LOSS / 10;
                newReward  = true;
                endEpisode = false;
            }
        #endif
    }
}
```

Fig. 28: Issue reward based on collision between the arm and the object and between the arm's gripper base and the object.

The name of the collision elements are defined in the gazebo-arm.world file. For example, for link 2, the collision element is called collision2.

At this stage, we test our project so far. Once in the project folder the following steps can be used to build and run the project.

```
$ cd build
$ make
$ cd x86_64/bin
$ ./gazebo-arm.sh
```

Once the environment/gazebo is loaded, we should notice the arm trying to move but it might not be learning. This brings us to the next task for the project, which is to tune various parameters such as hyperparameters, reward parameters, DQN API settings as discussed below.

G. *Tuning the hyperparameters*

For the project, every camera frame, at every simulation iteration, is fed into the DQN and the agent then makes a prediction and carries out an appropriate action. Using LSTMs as part of that network, we can train our network by taking into consideration multiple past frames from the camera sensor instead of a single frame.

The network in DQN.py has been defined such that we can include LSTMs into the network easily. In ArmPlugin.cpp, two variables have been defined that can be used to incorporate LSTMs in the DQN:

- USE_LSTM - This variable can be set to either true or false.
- LSTM_SIZE - Size of each LSTM cell.

The above variables can be treated as hyperparameters when training the RL agent for the project.

The list of hyperparameters that we can tune is provided in ArmPlugin.cpp file, at the top (Fig. 29). These hyperparameters along with DQN API settings, and reward parameters are tuned to obtain the required tranining accuracy. The results are discussed in the results and discussions sections.

VIII. Extra Challenge

The following iterms have not been implemented in this project and are reference for future work on complex situations where the object apears at random locations. In order for the robot arm to reach the object, we must modify its arm's rach as well.

A. *Object Randomization*

In the project, so far, the object of interest was placed at the same location, throughout. For this challenge, the object will instantiate at different locations along the x-axis. Following steps can be implemented to test our solution:

- In PropPlugin.cpp, redefine the prop poses in PropPlugin::Randomize() to the following:

```
pose.pos.x = randf(0.02f, 0.30f);
pose.pos.y = 0.0f;
pose.pos.z = 0.0f;
```

- In ArmPlugin.cpp, replace ResetPropDynamics(); set in the method ArmPlugin::updateJoints() with RandomizeProps();

B. *Increasing the Arm's Reach*

In the gazebo-arm.world file, the arm's base has a revolute joint. However, in the project, that was disabled to restrict the arm's reach to a specific axis. In this challenge, the object's starting location will be changed, and the arm will be allowed to rotate about its base. Following steps can be implemented to increase the arm's reach:

- In gazebo-arm.world, modify the tube model's pose to [0.75 0.75 0 0 0 0]
- In ArmPlugin.cpp, set the variable LOCKBASE to false.
- In ArmPlugin.cpp, replace RandomizeProps(); set in the method ArmPlugin::updateJoints() with ResetPropDynamics();

C. *Increasing Arm's Reach with Object Randomization*

This step implements both the object randomization with increased arm's reach as described in previous two subsections:

- In gazebo-arm.world, modify the tube model's pose to [0.75 0.75 0 0 0 0]

- In ArmPlugin.cpp, set the variable LOCKBASE to false.
- In ArmPlugin.cpp, replace ResetPropDynamics(); set in the method ArmPlugin::updateJoints() with RandomizeProps();
- In PropPlugin.cpp, redefine the prop poses in PropPlugin::Randomize() to the following:

```
pose.pos.x = randf(0.35f, 0.45f);
pose.pos.y = randf(-1.5f, 0.2f);
pose.pos.z = 0.0f;
```

IX. Results and Discussions

The robot arm is used to train a Deep Q-Learning Network (DQN) in a Gazebo simulation environment. The robot arm consists of 3 non-static joints that are controlled through either a velocity control or a position control command. The output of the DQN is mapped to the control of each joint of the robotic arm using the position control, due to superior learning performance. As described before the robot arm needs to be trained for two different tasks as described below.

- Have any part of the robot arm touch the object of interest, with at least 90% accuracy.
- Have only the gripper base of the robot arm touch the object, with at least 80% accuracy.

A. *Reward Functions*

The following reward functions were defined to train the robotic arm to reach each goal. For both the tasks shown above, the reward functions were the same, except for some changes in CollisionCheck. Here the collisionCheck is defined differently because for the first task the robot arm needs to touch the tube object whereas for the second task the girpper base needs to touch the tube object. This definition can be found below (Fig. 28).

```
#if TASK_1
    // *** TASK-1 *** //
    bool collisionCheck = ((strcmp(contacts->contact(i).collision1().c_str(), COLLISION_ITEM) == 0) && (strcmp(contacts->contact(i).collision2().c_str(), COLLISION_ARM) == 0) )? true : false;
    if (collisionCheck)
    {
        rewardHistory = REWARD_WIN * 10;
        newReward  = true;
        endEpisode = true;
        return;
    }
#else
    // *** TASK-2 *** //
    bool collisionCheck = ((strcmp(contacts->contact(i).collision1().c_str(), COLLISION_ITEM) == 0) && (strcmp(contacts->contact(i).collision2().c_str(), COLLISION_POINT) == 0) )? true : false;
    if (collisionCheck)
    {
        rewardHistory = REWARD_WIN * 10;
        newReward  = true;
        endEpisode = true;
        return;
    }
    else
    {
        rewardHistory = REWARD_LOSS / 10;
        newReward  = true;
        endEpisode = false;
    }
#endif
```

For the first task, a positive reward of REWARD_WIN*10 is issued when any part of robot arm collides with the tube object, after which the episode ends.

For the second task, positive reward of REWARD_WIN*10 is given only if the gripper base touches the object, triggering an end of episode (EOE). And if any other part of the robot arm touches the object, a negative reward of REWARD_LOSS / 10 is issued.

A negative reward of REWARD_LOSS was given when the robot gripper hits the ground and causing and end of episode (Fig. 26). An interim reward is issued based on the distance between the arm and object and was defined as a smoothed moving average of the delta of the distance to the goal as shown in Fig. 27.

A negative reward of REWARD_LOSS was issued for both tasks, if the camera image frame count exceeded the maxEpisodeLength (100), and triggering an end of episode (Fig. 29).

Fig. 29: Triggeting episode timeout.

```
// episode timeout
if( maxEpisodeLength > 0 && episodeFrames > maxEpisodeLength )
{
    rewardHistory = REWARD_LOSS;
    newReward  = true;
    endEpisode = true;
}
```

B. *Hyperparameters*

The following parameters were tuned in order to achieve the desired learning for the robot arm as described above. The hyper-parameter values were set the same for both the tasks as shown in Fig. 30.

- INPUT_WIDTH, INPUT_HEIGHT: The input height and width (of the image input) were reduced to 64x64 in order to reduce the computational complexity and memory usage compared to the original image size 512x512.
- NUM_ACTIONS: The number of actions was set to 2*DOF, as the robot arm had 3 joints (DOF = 3) and each of them could move in two directions.
- OPTIMIZER: The RMSprop optimizer was the best optimizer to achieve the required learning accuracy.
- LEARNING_RATE: The learning rate was set to 0.03 as the performance was optimum for this value.
- REPLAY_MEMORY: The replay memory was set to 10,000 in order to allow enough stored information for the model to train on.
- BATCH_SIZE: The batch size was set to 32 for both tasks.

```
// Choose TASK-1 or TASK-2
#define TASK_1 false        // TASK-1: true // TASK-2: false

// Turn on velocity based control
#define VELOCITY_CONTROL false  // TASK-1 && TASK-2: false
#define VELOCITY_MIN -0.2f
#define VELOCITY_MAX  0.2f

/*
// Define DQN API Settings
*/
#define INPUT_CHANNELS 3
#define ALLOW_RANDOM true
#define DEBUG_DQN false
#define GAMMA 0.9f
#define EPS_START 0.9f       // TASK-1 && TASK-2: 0.9f
#define EPS_END 0.01f        // TASK-1 && TASK-2: 0.01f
#define EPS_DECAY 300        // TASK-1 && TASK-2: 300

/*
// TODO - Tune the following hyperparameters
*/
#define INPUT_WIDTH   64
#define INPUT_HEIGHT  64
#define OPTIMIZER "RMSprop"
#define LEARNING_RATE 0.03f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 32
#define USE_LSTM true
#define LSTM_SIZE 256

/*
// TODO - Define Reward Parameters
*/
#define REWARD_WIN  1000.0f
#define REWARD_LOSS -100.0f
#define REWARD_ALPHA 0.2f

// Define Object Names
#define WORLD_NAME "arm_world"
#define PROP_NAME  "tube"
#define GRIP_NAME  "gripper_middle"

// Define Collision Parameters
#define COLLISION_FILTER "ground_plane::link::collision"
#define COLLISION_ITEM   "tube::tube_link::tube_collision"
#define COLLISION_POINT  "arm::gripperbase::gripper_link"
#define COLLISION_ARM    "arm::link2::collision2"
```


Fig. 30: Define DQN API settings and hyperparameters.

- USE_LSTM: The LSTM implementation was enabled to take advantage of the previous learning experience by taking into consideration multiple past frames from the camera sensor instead of a single frame.
- LSTM_SIZE: The size of each LSTM cell was set to 256 to reach the required target learning accuracy.
- EPS_START: This is the initial exploration probability. This value was set to 0.9 for both the tasks. Initially, the robot must explore more; hence a higher exploration at the beginning of learning will make the agent more robust.
- EPS_END: This is the final exploration probability. This value was set to very low value of 0.01 so that at the end robot will make less exploration (random moves), and more exploitation i.e. make decisions based on previously obtained results.
- EPS_DECAY: This is the greedy decay rate of the exploration. This value was set to 300 for both the tasks.
- INPUT_CHANNELS: This value was set to 3.
- GAMMA: This is the discounting factor for DQN agent and the value was set to 0.9 for both the tasks.
- ALLOW_RANDOM: This value is set to *true*, in order to allow the RL agent to make random choices.

C. *Results*

The above hyperparameters, DQN API settings, reward parameters, and reward functions both the tasks were performed at high accuracy.

For TASK-1, any part of the robot arm was able to touch the goal, at 90% accuracy after 230 episodes as shown in Fig. 31. This accuracy increased to 95% after 480 episodes.

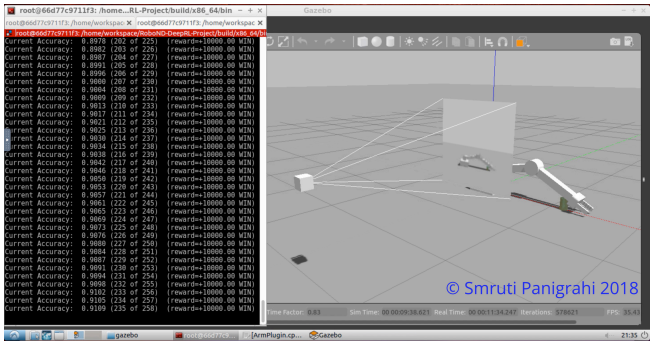


Fig. 31: The arm of the robot touching the tube object. The accuracy of the RL agent reached 90% after 230 episodes.

For TASK-2, any part of the robot arm was able to touch the goal, at 80% accuracy after 85 episodes as shown in Fig. 32. This accuracy increased to 90% after 320 episodes.

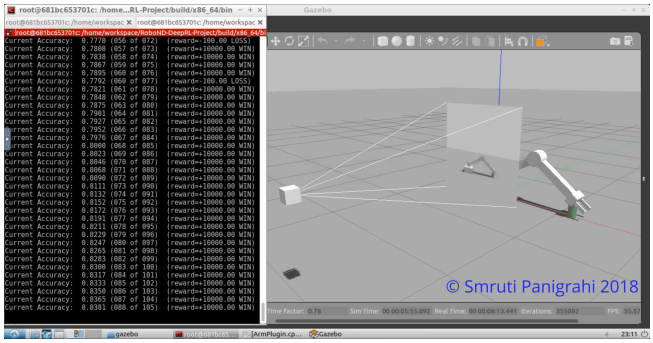


Fig. 32: The gripper of the robot touching the tube object. The accuracy of the RL agent reached 80% after 85 episodes.

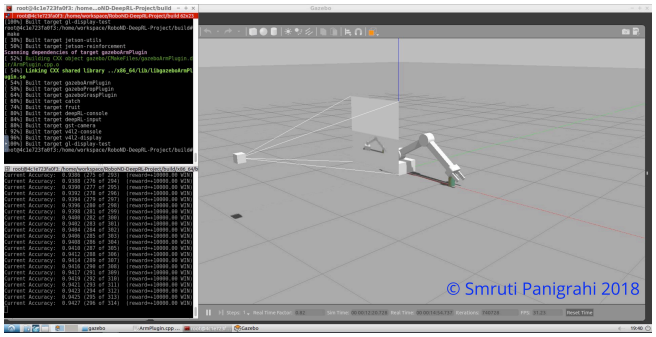


Fig. 33: The gripper of the robot touching the tube object. The accuracy of the RL agent reached 94% after 300 episodes.

The learning speed depends heavily on the initial rewards the agent receives. If the robot keeps making mistakes and receiving negative rewards, it takes the agent extremely long time to explore other moves and reach the desired accuracy. Hence, if the robot starts learning with high rewards starting from the very beginning, then it takes very less number of episodes to acieve high learning accuracy. This however, comes at a disadvantage of less exploration and high exploitation. An axample of this is shown in Fig. 33 below,

where for TASK-2, the robot agent achieved 85% accuracy after 55 episodes and 94% accuracy after 300 episodes.

The video of the robot arm and the robot gripper touching the goal object (the green tube) can be found in /docs/videos folder.

X. Conclusion

A robot arm was used as an agent that needed to learn multiple tasks. Using the Deep RL algorithm using Deep Q-Learning Network, along with LSTM for memorizing past learning, the Robot arm was able to perform the task of touching the goal tube object at more than 90% desired accuracy. Though the learning was robust enough for the chosen hyperparameters, it is possible to increases the speed of achieving that accuracy. The DQN API Settings could be changed to decrease exploration and increase exploitation to help teach the robot/agent to arrive at high accuracy quicker. Along with DQN API settings change, decreasing the learning rate and the increasing the batch size could help in the robust learning. The LSTM cell size could be increased to capture diverse learning memory.

For the future work, other types of motion control can be explored, essentially including the velocity control for smoother movement and integrating that with the position based control. Further tuning the hyper parameters and rewarding strategies can be considered to achieve the accuracy goal faster. Other future work would involve randomizing the goal object location and modifying the robot arm's reach as discussed in Section VIII. Additionally, incorporating gripper control in order for the robot arm to reach for the goal and grasp and release the object with high accuracy. This would have practical applications in sorting and inventory tracking.

Acknowledgment

The author would like to thank Udacity Inc. for providing some basic directions for DeepRL/DQN parameter tuning.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. *Nature*, *518*(7540), p.529.

[2] Beattie, C., Leibo, J.Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A. and Schrittwieser, J., 2016. Deepmind lab. *arXiv preprint arXiv:1612.03801*.

[3] https://deepmind.com/research/alphago/

[4] Van Hasselt, H., Guez, A. and Silver, D., 2016, February. Deep Reinforcement Learning with Double Q-Learning. In *AAAI* (Vol. 2, p. 5).

[5] https://sites.google.com/a/**deepmind**.com/dqn

[6] https://openai.com

[7] https://gym.openai.com/envs/#box2d

[8] https://gym.openai.com/envs/#classic_control

[9] https://gym.openai.com/envs/#atari

[10] https://gym.openai.com/envs/#mujoco

[11] https://gym.openai.com/envs/#robotics

[12] https://blog.openai.com/envs/openai-gym-beta

[13] https://github.com/openai/gym.git

[14] https://www.libav.org

[15] Riedmiller, M., 2005, October. Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning* (pp. 317-328). Springer, Berlin, Heidelberg.

[16] https://devblogs.nvidia.com/deep-learning-nutshell-reinforcement-learning/

[17] https://pytorch.org/

[18] http://torch.ch/

[19] https://github.com/dusty-nv/jetson-reinforcement

[20] https://github.com/udacity/RoboND-DeepRL-Project

[21] http://gazebosim.org/tutorials?tut=topics_subscribed

[22] http://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/classgazebo_1_1transport_1_1Node.html#a13a67ebd4537a0057ae92f837bb3042f

[23] https://www.wetube.com/watch?v=UNmqTiOnRfg

[24] https://www.wetube.com/watch?v=iX5V1WpxxkY

[25] https://www.wetube.com/watch?v=WCUNPb-5EYI

[26] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[27] https://github.com/udacity/RoboND-DeepRL-Project/blob/master/gazebo/ArmPlugin.cpp

[28] http://osrf-distributions.s3.amazonaws.com/gazebo/api/2.2.1/classgazebo_1_1physics_1_1Model.html#a27cf5a6ec66f6a5c03ebf05ff9592c9a

[29] https://www.watermarquee.com