HANDLING CONTROL OPERATIONS OF FETCH ROBOTIC SIMULATOR USING DEEP REINFORCEMENT LEARNING

A PROJECT REPORT

Submitted By

GAJESH S.	**312215104086**
BHASKAR V.	**312215104016**
LAXMAAN B.	**312215104018**

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

SSN COLLEGE OF ENGINEERING

KALAVAKKAM 603110

ANNA UNIVERSITY :: CHENNAI - 600025

April 2019

ANNA UNIVERSITY : CHENNAI 600025

BONAFIDE CERTIFICATE

Certified that this project report titled **"HANDLING CONTROL OPERATIONS OF FETCH ROBOTICS SIMULATOR USING DEEP REINFORCEMENT LEARNING"** is the *bonafide* work of "**GAJESH S. (312215104086)**, **BHASKAR V. (312214104016)**, and **LAXMAAN B. (312214104018)**" who carried out the project work under my supervision.

DR. CHITRA BABU
HEAD OF THE DEPARTMENT
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

DR. R S. MILTON
SUPERVISOR
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

Place:
Date:

Submitted for the examination held on. . . . . .

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENTS

We thank GOD, the almighty for giving me strength and knowledge to do this project. We would like to thank and deep sense of gratitude to our guide **Dr. R S. MILTON**, Professor, Department of Computer Science and Engineering, for his valuable advice and suggestions as well as his continued guidance, patience and support that helped us to shape and refine our work.

My sincere thanks to **Dr. CHITRA BABU**, Professor and Head of the Department of Computer Science and Engineering, for her words of advice and encouragement and I would like to thank our project Coordinator **Dr. S.SHEERAZUDDIN**,Associate Professor, Department of Computer Science and Engineering for his valuable suggestions throughout the project.

We express our deep respect to the founder **Dr. SHIV NADAR**, Chairman, SSN University. We also express our appreciation to our **Dr. S. SALIVAHANAN**, Principal, for all the help he has rendered during this course of study.

We would like to extend our sincere thanks to all the teaching and non-teaching staffs of our department who have contributed directly and indirectly during the course of our project work. Finally, I would like to thank our parents and friends for their patience, cooperation and moral support throughout our life.

GAJESH S. **BHASKAR V.** **LAXMAAN B.**

ABSTRACT

Dealing with sparse rewards in an environment is a very challenging process that has caused many Reinforcement Learning models problems because the exploration of the environment does not yield much information about the goal. Therefore, unless we know beforehand the results of a set of actions, we cannot effectively learn to navigate in the environment. Neural networks based algorithms such as deep Q-learning and actor-critic networks work well in such environments because of their ability to learn data representations over time, but they take a lot of training to converge. We aim to utilize this aspect of Neural Networks and combine a Deep Neural Network with Reinforcement Learning to create a system similar to the aforementioned models and train a Robotic Manipulator to perform some basic control operations. Further, we make use of an Hindsight Experience Replay to optimize the process. We investigate the effect of adding demonstration data to our model and show how our optimizations affect our model.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Traditionally, Machine Learning uses some collection of data to learn a trend or a pattern. It fails when there is a lack of data. This is where the significance of reinforcement learning comes in, as it does not use a dataset per se.

The idea behind reinforcement learning is to provide an agent with an environment, so that it explores and learns to do the specified tasks. We reward the agent based on how well it performs the task. Here, the performance of the agent depends on how well it has explored the environment and how it chooses the action to take.

Dealing with sparse rewards is one of the biggest challenges in Reinforcement Learning. The current algorithms involved hardcoding the goal based tasks. By implementing an architecture with a combination of deep networks and Reinforcement learning, the policies can be learnt which can prevent hardcoding the goals and therefore create a system that can be extrapolated to solve real tasks in the world.

In our project, we explore a method to help speed up convergence in sparse environments. We aim to demonstrate this on the task of manipulating the fetch robotic manipulator. In particular, we run experiments on four different tasks: pushing, sliding, pick-and-place, and reaching a point, in each case using only binary rewards indicating whether or not the task is completed.

In the next Chapter, we discuss Reinforcement Learning and relevant algorithms. In Chapter 3, a brief overview of Deep Neural Networks is given with all the

relevant concepts. In Chapter 4, we take a look at the related work that has been done by other scholars. In Chapter 5, we develop the architecture of the system and our proposed methodolgy in solving the problem. In Chapter 6, we go into detail the methods that were used to solve the problem and how it was implemented. Next, we publish the results for the control operations and how it compares with a generic model without HER algorithm. Finally, we conclude our work and discuss the possible applications of our project and the various avenues that it can be explored upon.

CHAPTER 2

REINFORCEMENT LEARNING

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. This reward may be obtained at the end of every 'episode' or at the end of the process.

Reinforcement learning involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). Finding a balance between exploration and exploitation forms the major chunk of the work the model needs to do.

The environment is typically formulated as a Markov Decision Process (MDP), as many reinforcement learning algorithms utilize dynamic programming techniques. The main difference between the classical dynamic programming methods and reinforcement learning algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP. The basic workflow of any RL model is shown in Figure 2.1.

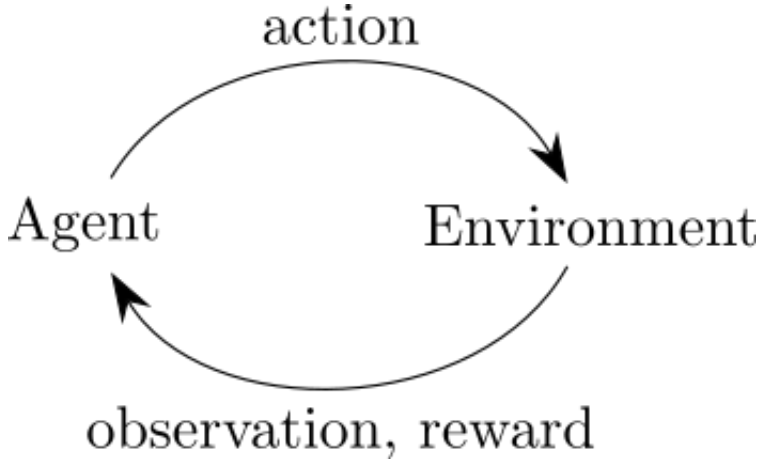

FIGURE 2.1: Abstract-Workflow of a RL model

2.1 Components of a Reinforcement Learning Problem

2.1.1 Agent

An Agent is a model created by the user. Over time, the agent learns to perform a specific task or action that we desire. This is achieved by observing the environment,and taking an action based on this observation. In our project,our goal is to create an agent that is capable of performing the aforementioned tasks in an intelligent manner thereby, over time, it is capable of taking the correct actions that lead to the control operations being solved.

2.1.2 Environment

An environment is the surroundings the agent interacts with.It consists of the agent and all the constituent components which include the robot, the puck,and the table.The main purpose of an environment is to provide the agent with

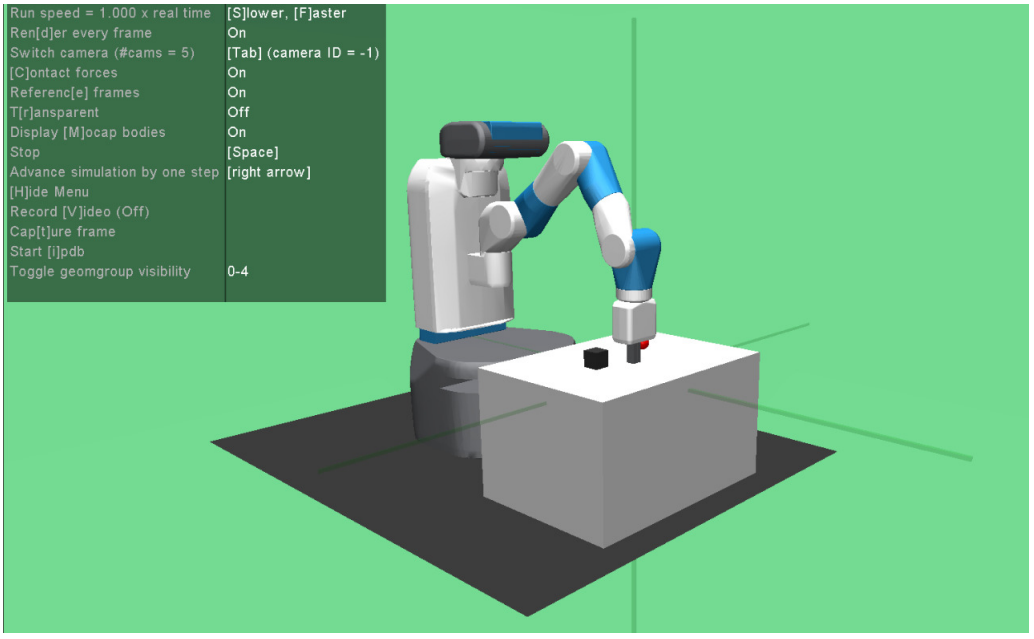

FIGURE 2.2: Environment

observations. Subsequently, it provides rewards based on the actions supplied by the agent. Figure 2.2 is an example of an environment.

2.1.3 States and Observations

A state $\boldsymbol{s}$ is a complete description of the state of the world. There is no information about the world which is hidden from the state. An observation $\boldsymbol{o}$ is a partial description of a state, which may omit information.

In deep RL, we almost always represent states and observations by a real-valued vector, matrix, or higher-order tensor. For instance, a visual observation could be represented by the RGB matrix of its pixel values; the state of a robot might be represented by its joint angles and velocities.

When the agent is able to observe the complete state of the environment, we say that the environment is fully observed. When the agent can only see a partial observation, we say that the environment is partially observed.

2.1.4 Action Spaces

Different environments allow different kinds of actions. The set of all valid actions in a given environment is often called the action space. Some environments, have discrete action spaces, where only a finite number of moves are available to the agent. Other environments, like where the agent controls a robot in a physical world, have continuous action spaces. In continuous spaces, actions are real-valued vectors.

This distinction has some quite-profound consequences for methods in deep RL. Some families of algorithms can only be directly applied in one case, and would have to be substantially reworked for the other.

2.1.5 Policies

A policy is a rule used by an agent to decide what actions to take. It can be deterministic, in which case it is usually denoted by μ:

$$a_t = \mu(s_t)$$

or it may be stochastic, in which case it is usually denoted by π:

$$a_t \sim \pi(.|s_t)$$

Because the policy is essentially the agents brain, its not uncommon to substitute the word policy for agent, eg saying The policy is trying to maximize reward.

In deep RL, we deal with parameterized policies: policies whose outputs are computable functions that depend on a set of parameters (eg the weights and biases of a neural network) which we can adjust to change the behavior via some optimization algorithm.

We often denote the parameters of such a policy by θ or ϕ, and then write this as a subscript on the policy symbol to highlight the connection:

$$a_t = \mu_\theta(s_t)$$

$$a_t \sim \pi_\theta(.|s_t)$$

2.1.6 Trajectories

A Trajectory τ, also known as an **episode** or a **rollout**, is a sequence of states and actions in the world.

$$\tau = (s_0, a_0, s_1, a_1, ...)$$

The very first state of the world, s_0, is randomly sampled from the start-state distribution, sometimes denoted by ρ_0:

$$s_0 \sim \rho_0(.)$$

State transitions are governed by the natural laws of the environment, and depend on only the most recent action, a_t. They can be either deterministic,

$$s_{t+1} = f(s_t, a_t)$$

or stochastic,

$$s_{t+1} \sim P(\cdot|s_t, a_t).$$

Actions come from an agent according to its policy.

2.1.7 Reward and Return

The reward function R is critically important in reinforcement learning. It depends on the current state of the world, the action just taken, and the next state of the world:

$$r_t = R(s_t, a_t, s_{t+1})$$

although frequently this is simplified to just a dependence on the current state, $r_t = R(s_t)$, or state-action pair $r_t = R(s_t, a_t)$.

The goal of the agent is to maximize some notion of cumulative reward over a trajectory, but this actually can mean a few things. Well notate all of these cases

with $R(\tau)$, and it will either be clear from context which case we mean, or it wont matter (because the same equations will apply to all cases).

One kind of return is the finite-horizon undiscounted return, which is just the sum of rewards obtained in a fixed window of steps:

$$R(\tau) = \sum_{t=0}^{T} r_t.$$

Another kind of return is the infinite-horizon discounted return, which is the sum of all rewards ever obtained by the agent, but discounted by how far off in the future theyre obtained. This formulation of reward includes a discount factor $\gamma \in (0,1)$:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t.$$

2.1.8 Value Functions

Its often useful to know the value of a state, or state-action pair. By value, we mean the expected return if you start in that state or state-action pair, and then act according to a particular policy forever after. Value functions are used, one way or another, in almost every RL algorithm.

Four main value functions are:

- The On-Policy Value Function, $V^{\pi}(s)$, which gives the expected return if you start in state s and always act according to policy π:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)\,|s_0 = s]$$

- The On-Policy Action-Value Function, $Q^{\pi}(s,a)$, which gives the expected return if you start in state s, take an arbitrary action a (which may not have come from the policy), and then forever after act according to policy π:

$$Q^{\pi}(s,a) == \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)\,|s_0 = s, a_0 = a]$$

- The Optimal Value Function, $V^{*}(s)$, which gives the expected return if you start in state s and always act according to the optimal policy in the environment:

$$V^{*}(s) = \max_{\pi} \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)\,|s_0 = s]$$

- The Optimal Action-Value Function, $Q^{*}(s,a)$, which gives the expected return if you start in state s, take an arbitrary action a, and then forever after act according to the optimal policy in the environment:

$$Q^{*}(s,a) = \max_{\pi} \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)\,|s_0 = s, a_0 = a]$$

2.2 RL Algorithm Taxonomy

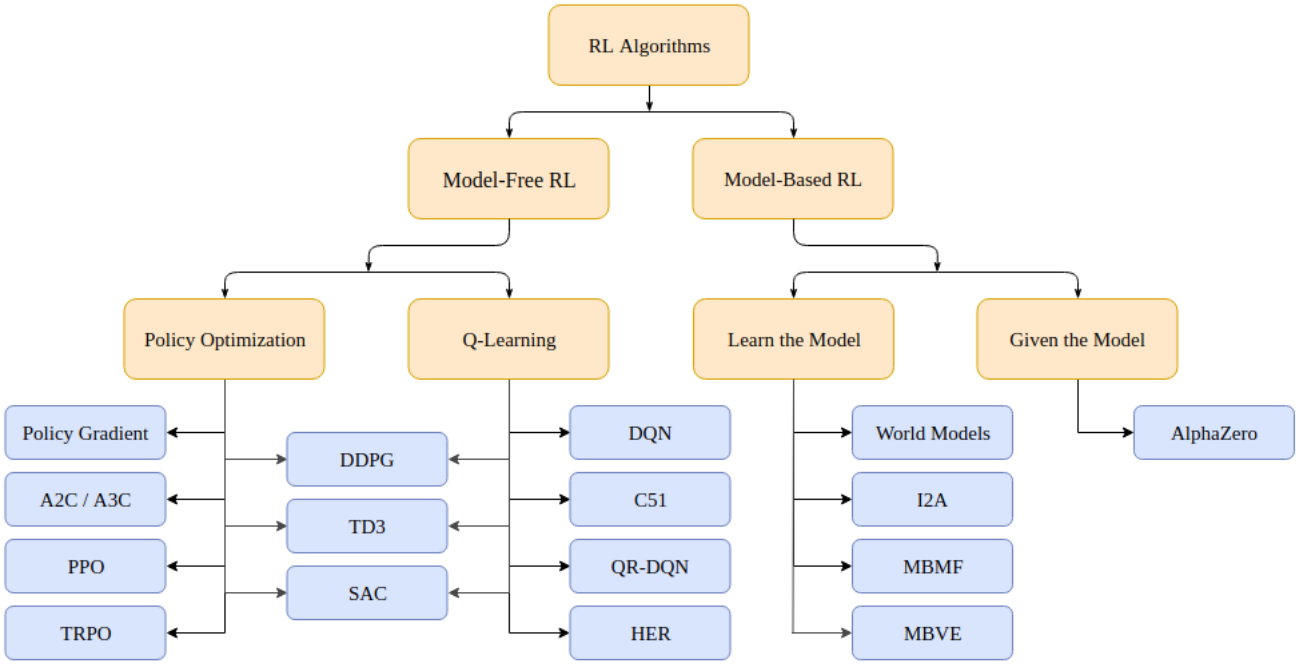

FIGURE 2.3: Classification of RL Algorithms

One of the most important branching points in an RL algorithm(Figure 2.3) is the question of whether the agent has access to (or learns) a model of the environment. A model is a function which predicts state transitions and rewards.

The main advantage to having a model is that it allows the agent to plan by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. Agents can then distill the results from planning ahead into a learned policy.

The main downside is that a ground-truth model of the environment is usually not available to the agent. If an agent wants to use a model in this case, it has to learn the model purely from experience, which creates several challenges. The biggest challenge is that bias in the model can be exploited by the agent, resulting

in an agent which performs well with respect to the learned model, but behaves sub-optimally in real environments.

Algorithms which use a model are called model-based methods, and those that dont are called model-free. While model-free methods forego the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune.

2.2.1 Model-Free RL

Model-free RL consists of two main approaches:

- **Policy Optimization.** Methods in this family represent a policy explicitly as $\pi_\theta(a|s)$. They optimize the parameters θ either directly by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$. This optimization is almost always performed on-policy, which means that each update only uses data collected while acting according to the most recent version of the policy. **e.g. Asynchronous Advantage Actor Critic (A3C)**

- **Q-Learning.** Methods in this family learn an approximator $Q_\theta(s,a)$ for the optimal action-value function, $Q^*(s,a)$. Typically they use an objective function based on the Bellman equation. This optimization is almost always performed off-policy, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained. **e.g. Deep Q-Learning (DQN)**

- **Interpolating Between Policy Optimization and Q-Learning.** policy optimization and Q-learning are not incompatible (and under some circumstances, it turns out, equivalent), and there exist a range of algorithms that live in between the two extremes. Algorithms that live on this spectrum are able to carefully trade-off between the strengths and weaknesses of either side. **e.g. Deep Deterministic Policy Gradients (DDPG)**

2.2.2 Model-Based RL

Unlike model-free approaches, the list of model-based approaches is far from exhaustive. A few examples include:

- **Background: Pure Planning** he most basic approach never explicitly represents the policy, and instead, uses pure planning techniques like model-predictive control (MPC) to select actions. In MPC, each time the agent observes the environment, it computes a plan which is optimal with respect to the model.

- **Expert Iteration**. A straightforward follow-on to pure planning involves using and learning an explicit representation of the policy, $pi_\theta(a|s)$. The agent uses a planning algorithm (like Monte Carlo Tree Search) in the model, generating candidate actions for the plan by sampling from its current policy.

CHAPTER 3

DEEP NEURAL NETWORKS

A deep neural network (DNN) is an artificial neural network (ANN) with multiple layers between the input and output layers. DNNs are used in applications based on learning data representations like Computer Vision, Speech Recognition. Recently, they have also been used in Reinforcement Learning.

3.1 DNN Architecture

DNNs follow a layered architecture. It has an input layer, an output layer, and some hidden layers. Each layer comprises of nodes, which carry a value. The input layer consists of neurons carrying information input to the system and the output layer consists of neurons carrying information after processing the input.

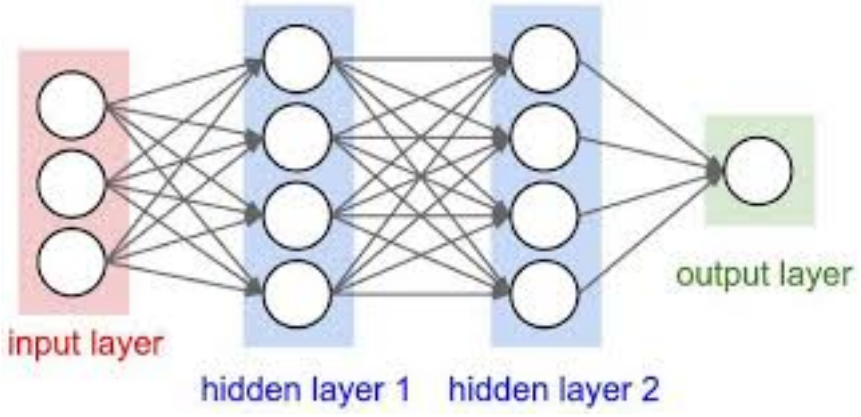

FIGURE 3.1: A simple Deep Neural Network with 2 hidden layers

DNNs are typically feed forward networks in which data flows from the input layer to the output layer without looping back. The DNN assigns random numerical

values, or ”weights”, to connections between them. The weights and inputs are multiplied and return an output between 0 and 1. If the network did not accurately recognize a particular pattern, an algorithm would adjust the weights.

3.2 Input Layer

The Input layer is the very first layer in any DNN architecture. It takes in the input that is fed by the user for their application. This maybe in the form of numerical values, or images in the case of Convolutional Neural Networks. The main purpose of an Input Layer is to take in the user inputs or dataset and feed it forward to the other layers in the network. Usually, the input layer is not modified and the input is passed forward as and when it is fed to the network.

3.3 Hidden Layer(s)

The Hidden layer is the intermediate layer between the Input and Output layer. There may be several hidden layers depending on the users preference and application. The hidden layers consists of nodes which combines input from the data with a set of coefficients, or weights, that either amplify or dampen that input, thereby assigning significance to inputs with regard to the task the algorithm is trying to learn.These input-weight products are summed and then the sum is passed through a nodes so called activation function, to determine whether and to what extent that signal should progress further through the network to

affect the ultimate outcome, say, an act of classification. If the signals passes through, the neuron has been activated.

3.4 Output Layer

The final layer of a DNN is usually the output layer, which comprises neurons which output the prediction made by the model. Further, an activation function is applied on the output node, so as to adjust the output to within a desired range.

3.5 Activation Functions

A Rectified Linear unit(ReLU) activation function, when applied upon a node, returns max(0,x) where x is the input value. The main feature of a ReLU function is that negative input values are removed, as the function outputs 0 if the input value is less than zero. When the input is positive, the function just outputs the input therefore RELUs result in much faster training for large networks due to faster back propagation.

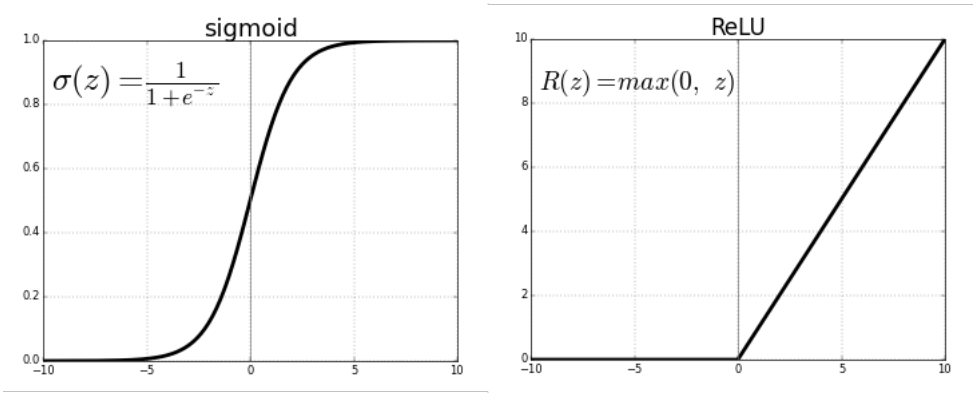

FIGURE 3.2: ReLU activation function

3.5.1 Tanh Activation

The Hyperbolic Tangent Activation (tanh) activation function, when applied upon a node, returns a value in range (-1,1). Traditional activation functions like sigmoid activation functions can cause a neural network to get stuck during training. This is due in part to the fact that if a strongly-negative input is provided to the logistic sigmoid, it outputs values very near zero.Since neural networks use the feed-forward activations to calculate parameter gradients, this can result in model parameters that are updated less regularly than we would like, and are thus stuck in their current state. Since tanh outputs a value in range (-1,1), strongly negative inputs to the tanh will map to negative outputs. Additionally, only zero-valued inputs are mapped to near-zero outputs. These properties make the network less likely to get stuck during training.

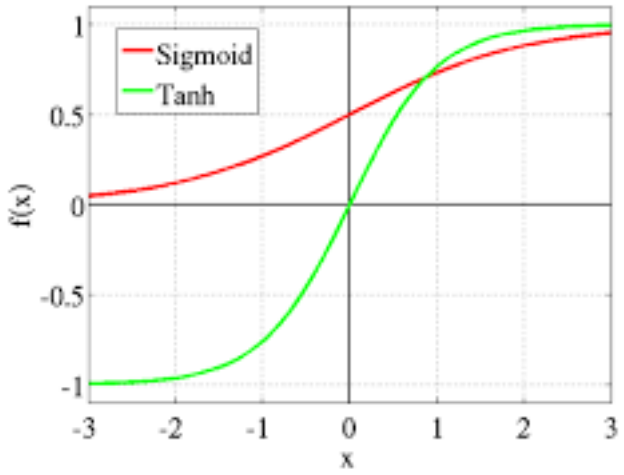

FIGURE 3.3: Tanh Activation function

CHAPTER 4

LITERATURE SURVEY

4.1 Deep Q-Learning

DeepMind has applied Deep Q-Learning(DQN) [5] [4] to play Atari games, using the raw pixel values. They overcome the problems in traditional Q-Learning method, which is being unable to accurately predict the Q values of unvisited states. The, they do by using a neural network instead of the Q table. However, it still hasn't overcome the problem of continuous action spaces.

4.2 Continuous Control with Deep Reinforcement Learning

Table based methods such as Q-learning fails when the action space provided by the environment is continuous. A method to achieve control over a continuous action space was proposed by Lillicrap et.al, where an actor-critic algorithm [3] based on the deterministic policy gradient was designed to operate over continuous action spaces.

Based on Deep Q-Learning and hyper-parameter tuning, the algorithm robustly solves more than 20 simulated physics tasks, including classic problems such as CartPole, Swing-Up, Dexterous Manipulation, Legged Locomotion and Car Driving. Further their algorithm is able to find policies whose performance is

competitive with those found by a planning algorithm with full access to the dynamics of the domain and its derivatives,for many of the tasks the algorithm can learn policies end-to-end: directly from raw pixel inputs.

4.3 Hindsight Experience Replay (HER)

The performance of Neural Networks in Reinforcement Learning are highly depend highly on the reward function engineered for the environment. The reward function must be very carefully defined in this case, making it difficult to scale for larger and more complicated environments.

Andrychowicz et. al proposed a method of using replay buffers[1], where the model is taught to converge faster by creating intermediate goal states. They store state transitions from a series of experiences, and replay them with a set of arbitrary goals. This approach works when training is done with an off-policy algorithm such as Deep Deterministic Policy Gradients, Deep Q-Learning, or SDQN. HER has been proven to speed up convergence in case of binary rewards, and is shown to do a better job than adding count based exploration.

4.4 Demonstrations

Another optimization on the training of the system is to provide the system with sample demonstrations [8] of success scenarios, as proposed by Andrychowicz et. al. It draws the parallel of learning by watching. Robustness is introduced by training from these demonstrations only for a small time, after which the agent

starts exploring. This method makes sure not to falsely provide large Q values. The approach has proven to dominate over baselines provided sufficient examples, and still learn in case of small amounts of demonstration data.

4.5 Curiosity

Reinforcement learning algorithms rely on carefully engineering environment rewards that are extrinsic to the agent. However, annotating each environment with hand-designed, dense rewards is not scalable. Pathak et.al [9] proposed 'Curiosity' a type of intrinsic reward function which uses prediction error as a reward signal without the need for an extrinsic reward signal.They tested it in environment of no extrinsic rewards, of sparse extrinsic rewards. The curiosity driven models performed better than the models with only extrinsic reward signals in 70 percent of the cases.

CHAPTER 5

PROPOSED SYSTEM

The Fetch Robotic Environment provides us controls as continuous values ranging from -1 to 1 for each of the parameters. Algorithms such as Q-Learning fail in this case as there are infinite values each control can take. Discretizing this is not a good idea as it restricts movements if the samples are too big and generates too many possible actions for each parameter otherwise.

Due to this reason, we look for table free approaches that can process continuous action space. One such algorithm is Deep Deterministic Policy Gradients.

5.1 Deep Deterministic Policy Gradients

DDPG combines the ideas of Actor Critic Networks, Deterministic Policy Gradients and Deep Q Networks. In this, we replace the Q-function of Q-learning with a neural network, as used in Deep Q-learning. This changes the Q-function as

$$Q^{\pi}(s,a) \approx Q(s,a,w)$$

where w is the weights defined by the neural network. This modifies the loss function as

$$\frac{\partial L(\theta)}{\partial \theta} = \frac{\partial Q(s,a,w)}{\partial a}\frac{\partial a}{\partial \theta}$$

where the policy parameters θ can be updated via stochastic gradient ascent.

5.2 Actor Critic Networks

The Actor-Critic Algorithm is essentially a hybrid method to combine the policy gradient method and the value function method together. The policy function is known as the actor, while the value function is referred to as the critic. Essentially, the actor produces the action a given the current state of the environment s, while the critic produces a signal to criticizes the actions made by the actor.

The actor runs the episode by providing the set of actions. The cumulative rewards obtained are used to tune the critic so as to update the expected rewards. These changes are then propagated to the actor for training. We use a constant τ so that we do not lose out on past experiences, that a part of it is preserved.

Loss Function for Critic Updation:

$$L = \frac{1}{N}\sum_{i}(y_i - Q(s_i, a_i|\theta^Q))^2$$

Policy Gradient Formula:

$$\nabla_\theta J = \frac{\partial Q^\theta(s,a)}{\partial a}\frac{\partial \mu(s|\theta)}{\partial \theta}$$

Network Updation Formula:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

Data: Initial policy parameters θ ;

Q function Parameters ϕ;

Empty Replay Buffer D

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ;

Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$ Initialize replay buffer R **for** episode = 1, M **do**

  Initialize a random process N for action exploration;

  Receive initial observation state s1;

  for t = 1, T **do**

    Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise;

    Execute action at and observe reward r_t and observe new state s_{t+1};

    Store transition (s_t, a_t, r_t, s_{t+1}) in R;

    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R;

    Set $y_i = r_i + \gamma \times Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q\prime})$;

    Update critic by minimizing the loss: ;

    $L = \frac{1}{N}\sum_i (y_i Q(s_i, a_i|\theta^Q))^2$;

    Update the actor policy using the sampled policy gradient:;

    $\bigtriangledown_{\theta^\mu} J \approx \frac{1}{N}\sum_i \bigtriangledown a_Q(s,a|\theta^Q)|s = s_i, a = \mu(s_i) \bigtriangledown_{\theta^\mu}(s|\theta^\mu)|s_i$;

    Update the target networks: $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$;

    $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$;

  end

end

Algorithm 1: Deep Deterministic Policy Gradients

5.2.1 Actor Critic Network Architecture

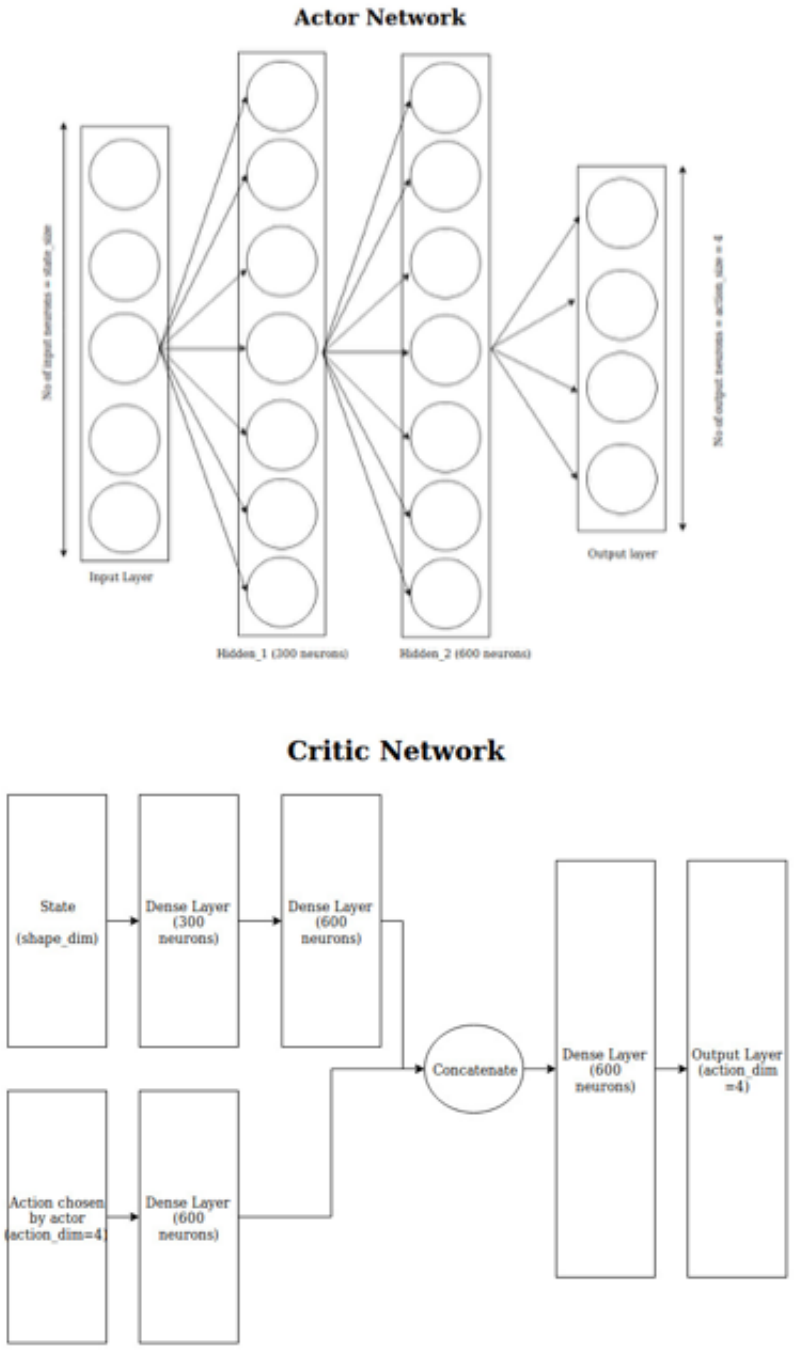

FIGURE 5.1: Caption

5.3 Architecture of the Model

5.3.1 Critic Network

Input Layer 1a (from state space)	state size = number of input parameters in state space
Hidden Layer 1a	Fully Connected Layer of 300 neurons connected to Input Layer 1
Activation 1a	ReLU
Input Layer 1b (from action space)	state size = number of input parameters in action space
Hidden Layer 1b	Fully Connected Layer of 600 neurons connected to Input Layer 2
Activation 1b	linear
Hidden Layer 2	concatenation of Hidden Layer 1a activation and Hidden Layer 1b Activation
Hidden Layer 3	Fully Connected Layer of 600 neurons connected to Hidden Layer 2
Activation 3	ReLU
Output Layer	number of neurons = size of action space
Output Activation	linear

5.3.2 Actor Network

Input Layer	Same as size of State space
Hidden Layer 1	Fully Connected Layer of 600 neurons
Activation 1	ReLU
Hidden Layer 2	Fully Connected Layer of 600 neurons
Activation 2	ReLU
Output Layer	Same as size of action space
Output Activation	Tanh

5.4 Hindsight Experience Replay

OpenAI published a paper in Hindsight Experience Replay(HER), where they explain in detail about how it overcomes the problem of sparse rewards by setting intermediate goals. HER formalizes what humans do intuitively. It's never stagnant. Even though our model may not have achieved desired target, we have atleast achieved another one , i.e A "new state". So we assume , this was our goal state to begin with instead of what we set out to achieve initially. By doing this substitution, the reinforcement learning algorithm can obtain a learning signal since it has achieved some goal; even if it was not the one that we meant to achieve originally. If we repeat this process, we will eventually learn how to

achieve arbitrary goals, including the goals that we really want to achieve.

Data: An off-policy RL algorithm A, . e.g. DQN, DDPG, NAF, SDQN

A strategy S for sampling goals for replay, . e.g. $S(s_0,...,s_T) = m(s_T)$

A reward function $r : SxAxG \rightarrow R$. . e.g. r(s, a, g) = [fg(s) = 0]

Initialize A (neural networks) ;

Initialize replay buffer R;

for episode = 1, M **do**

  Sample a goal g and an initial state s0;

  for t = 0, T 1 **do**

    Sample an action a_t using the behavioral policy from A;

    $a_t \leftarrow \pi_b(st||g)$ || denotes concatenation;

    Execute the action at and observe a new state s_{t+1};

  end

  for t = 0, T 1 **do**

    $r_t \leftarrow r(s_t, a_t, g)$;

    Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in R;

    Sample a set of additional goals for replay G

    $\leftarrow S(currentepisode)$ **for** $g' \in G$ **do**

      $r' \leftarrow r(s_t, a_t, g')$;

      Store the transition $(st||g, a_t, r_t, s_{t+1}||g')$ in R

    end

  end

  for t = 0, N **do**

    Sample a minibatch B from the replay buffer R;

    Perform one step of optimization using A and minibatch B;

  end

end

Algorithm 2: Hindsight Experience Replay (HER)

We can think of it as a short circuit. It breaks complicated tasks into simpler ones, and attempts to solve them. For example, in the pick and place environment, the manipulator first learns to move, then to pick the block, then to move the block to the right place. This offers a significant improvement over the original algorithm, where only the episodes performing all the tasks correctly are used for training. We will discuss more in details along with the results.

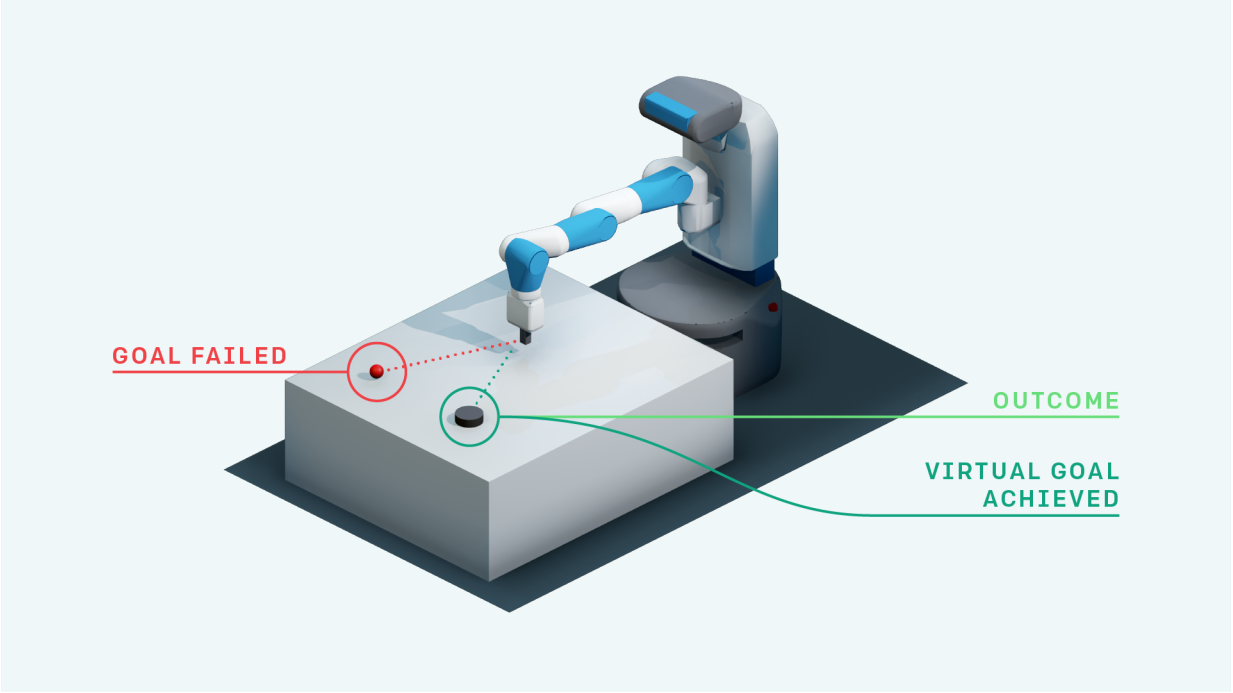

FIGURE 5.2: Idea behind HER

5.5 Demonstrations

In the event of extremely distant horizons, exploration may take a very long time to go over all possible actions and from them, identify potential actions in the optimal policy. For this reason, demonstrations of the task to be learnt can be used to overcome challenges in exploration as listed in [8] A human expert playing an

iteration or a program that performs the task with the same degree of skill is used. We shall refer to both as expert.

A Demonstration Buffer

A secondary buffer R_D similar to the replay buffer R is maintained. All (s_t, a_t, s_{t+1}) transitions from N episodes of expert demonstrations are stored in R_D. R_D is the demonstration buffer.

B Exploration from Demos

When sampling a mini batch to train the Actor-Critic Networks, we additionally sample N_D samples from R_D. These N_D samples have a separate loss function

$$L_{BC} = \sum_{n=1}^{N_D} (a_n - \pi_\theta(s_n))^2$$

This is a behaviour cloning loss, common in imitation learning scenarios.

5.6 Curiosity

A Curiosity Module

The curiosity module consists of a forward model. The forward model represents the function

$$\hat{s_{t+1}} = f(s_t, a_t)$$

where $s_t\hat{}_{+1}$ is the predicted next state upon choosing action a_t at state s_t.

B Training Curiosity Module

The forward model is allowed to train on data collected from N episodes. In each episode, the action is randomly sampled from the action space. At the end of the training, the forward model becomes adept at predicting the outcomes of random actions in the environment.

C Intrinsic Reward

The curiosity metric, the intrinsic reward is expressed as:

$$r_t^i = \frac{\eta}{2}||s_t\hat{}_{+1} - s_{t+1}||_2^2$$

which is the L2 normalization of the difference between the predicted next state and the actual next state.

The reward at time t is the sum of extrinsic reward r_t^e and intrinsic reward r_t^i

$$r_t = r_t^e + r_t^i$$

This increases the reward for taking actions that result in states that surprise the forward model, by causing a large prediction error.

CHAPTER 6

IMPLEMENTATION

6.1 OpenAI Gym Environment

Open AI's Gym functions as the environment back-end for reinforcement learning. A gym environment provides several functionalities such as:

- **gym.make("env") :** Returns an environment called "env"
- **env.reset() :** Provides the initial state for an episode.
- **env.step(action) :** Returns a tuple consisting of (next_state, reward, episode_done)

Gym environments abstract the state transition function $\delta(s,a) \rightarrow s'$ as well as the reward function $R(s,a) \rightarrow r'$ from the user and return the values (s', r') through the **env.step(a)** function. The environments employed in order to develop the Fetch Control Module are :

- **FetchReach-v1**: Move Fetch to a goal position.
- **FetchPush-v1**: Push a block to the goal position.
- **FetchSlide-v1**: Slide an object to a goal position.
- **FetchPickAndPlace-v1**: Lift a block in air.

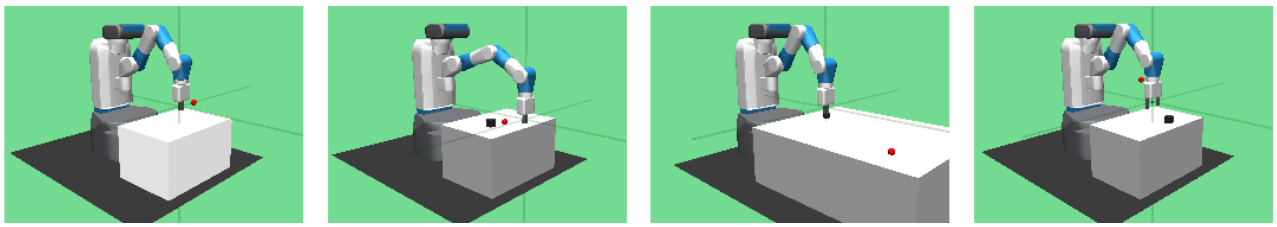

FIGURE 6.1: FetchReach, FetchPush, FetchSlide, and FetchPickAndPlace

The state space is given by a set of 17 to 25 parameters (numbers vary with the environment).

```
laxmaan@laxmaan-HP-Pavilion-Notebook: ~/Desktop/FYP/FinalYearProject
File  Edit  View  Search  Terminal  Help
laxmaan@laxmaan-HP-Pavilion-Notebook:~/Desktop/FYP/FinalYearProject$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import gym
>>> import gym.spaces
>>>
>>> env=gym.make('FetchPush-v1')
>>>
>>> env.action_space.low, env.action_space.high
(array([-1., -1., -1., -1.], dtype=float32), array([1., 1., 1., 1.], dtype=float32))
>>> env.observation_space
Dict(achieved_goal:Box(3,), desired_goal:Box(3,), observation:Box(25,))
>>>
>>> obs= env.reset()
>>>
>>> obs
{'observation': array([ 1.34193113e+00,  7.48903354e-01,  4.13631365e-01,  1.29687432e+00,
        6.35830019e-01,  4.24702091e-01, -4.50568034e-02, -1.13073335e-01,
        1.10707256e-02, -2.06467383e-06,  1.87039390e-03, -8.82449686e-08,
        1.35761490e-07,  3.97194063e-15, -2.96711930e-07, -5.50441164e-05,
        4.69134018e-05,  5.03907166e-08, -7.75241793e-08,  2.51278755e-18,
        2.94776838e-07,  5.50428586e-05, -8.56360696e-08,  5.26545712e-07,
        6.00978493e-05]), 'achieved_goal': array([1.29687432, 0.63583002, 0.42470209]), 'desired_goal': array([1.26744531, 0.87070661, 0.42469975])
}
>>>
>>> env.render()
Creating window glfw
>>> env.step(env.action_space.sample())
({'observation': array([ 1.34479779e+00,  7.59976704e-01,  4.21934440e-01,  1.29687432e+00,
        6.35830018e-01,  4.24737381e-01, -4.79234652e-02, -1.24146686e-01,
        2.80294100e-03,  0.00000000e+00,  0.00000000e+00, -5.03884423e-08,
        7.75206804e-08, -6.39615925e-17, -2.36728771e-03, -1.24638951e-02,
       -4.24163406e-03,  2.85323085e-08, -4.38958592e-08, -3.74911595e-19,
        2.36728661e-03,  1.24638944e-02,  4.26829962e-03,  8.05874181e-05,
        3.22572783e-03]), 'achieved_goal': array([1.29687432, 0.63583002, 0.42473738]), 'desired_goal': array([1.26744531, 0.87070661, 0.42469975])
}, -1.0, False, {'is_success': 0.0})
>>>
>>>
```

FIGURE 6.2: Input-States of our model: Fetch Push Environment

6.2 System Requirements

The program was implemented in python3.6, using Tensorflow. The gym environment was available via the gym library. This can be installed using pip (pip install gym). Additionally, the Fetch environment requires that MuJoCo dependency has to be installed.

The operating system we used was Ubuntu 16.04. Additionally we used a GPU to improve training speed. Also, we used the Anaconda Installation of Python3.6

6.2.1 Installing MuJoCo

MuJoCo offers a free trial for 30 days. It also provides students with a free license for a year. These can be availed in the official page. Following this,

- Download mjpro150 from the MuJoCo site.
- Create a file .mujoco in your home directory and extract the contents to this directory.
- Run getid_linux.sh to determine your computer id.
- Apply for the key using the using the account id and computer id.
- You will recieve an email with a file containing the licence key. Add the file to .mujoco/mjpro150/bin
- add .mujoco/mjpro150/bin to LD_LIBRARY_PATH and restart the bash terminal

6.3 Actor-Critic Networks

The Actor-Critic Networks are used to approximate the policy

$$\mu_\theta(s_t) = a_t$$

(which provides the action given a state) and the Q function

$$Q_\phi(s,a) = E[r(s,a) + \max_{a'} Q_\phi(s',a')]$$

(which provides the expected return by choosing action 'a' at state 's')

The actor network approximates μ_θ where θ represents the trainable parameters in the actor network. Similarly, the critic approximates Q_ϕ and ϕ represents its trainable weights.

The actor and critic train and converge together. The gradients from the critic are used to train the actor.

6.4 Replay Buffer

The replay buffer stores every step of every episode as a tuple (s,a,r,s',G) where r is the reward obtained upon reaching state s' from state s by choosing action a in trying to achieve goal G.

The replay buffer is sampled periodically (in our implementation, it is sampled after every episode). The sampled batch containing N_R samples is used to train the critic and the actor network.

6.4.1 Hindsight Experience Replay

To implement HER, we store each episode twice in the replay buffer. However, the second time we store the episode, we store the steps as (s, a, r, s', G') where G' is the goal achieved in the last state of the episode.

6.5 Demo Buffer

In order to overcome exploratory issues, especially given sparse rewards, a buffer containing state transitions from demonstration episodes is maintained in addition to the replay buffer.

These demonstrations consist of episodes where a human expert or carefully crafted code supplies actions to be taken. These actions are assumed to be of high quality and are used to guide the agent in learning to approximate μ_θ.

Along with the N_R samples from the replay buffer, N_D samples from the demo buffer are also considered as part of the batch used to train the actor-critic.

A behaviour cloning loss L_{BC} is used as auxiliary loss on the demo samples alone when training the actor network.

$$L_{BC} = \sum_{n=1}^{N_D} (a_n - \mu_\theta(s_n))^2$$

CHAPTER 7

RESULTS

We have discussed in detail the agent that attempts to intelligently learn and solve the control operations of the fetch robot simulation. We have combined a complex Actor-Critic Network model with Hindsight Experience Replay, that significantly improves the performance of the model as seen below.

7.1 Fetch Reach Environment

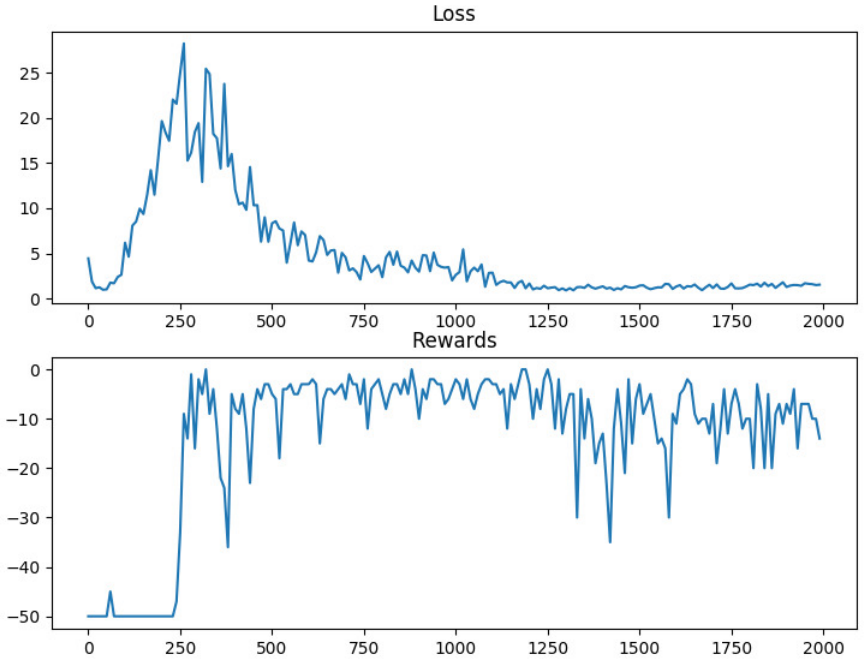

FIGURE 7.1: Fetch Reach with HER

We can see that initially, our loss function peaks to about 25, but over the course of a 250 iterations, the loss function begins to smoothen and ultimately begins

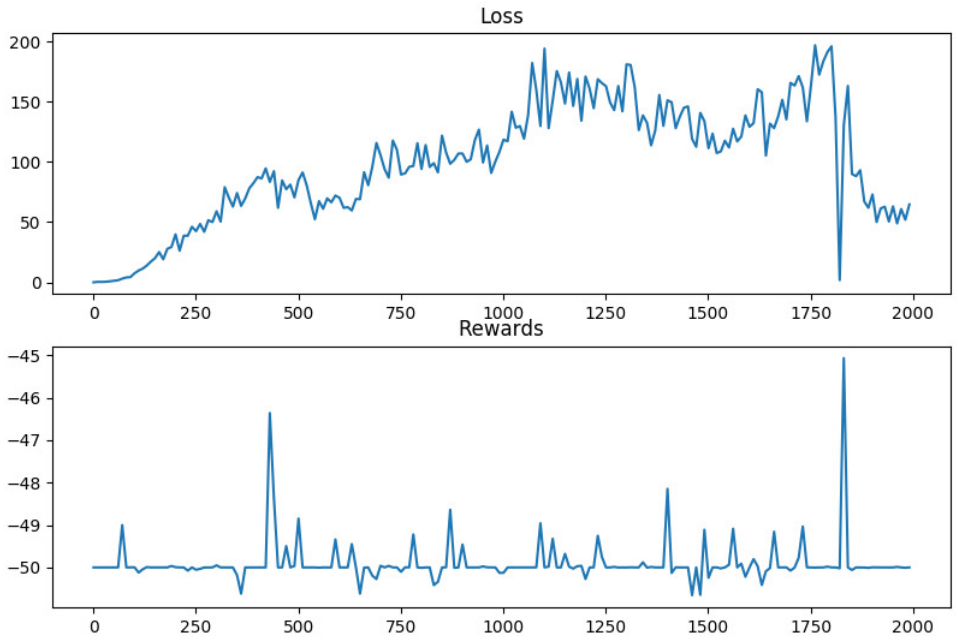

FIGURE 7.2: Fetch Reach without HER

to converge to a value nearer to 0,which is ideal. This improvement is further solidified by the huge improvements in the reward function of the model with HER algorithm. At the onset, the reward takes the maximum possible value of -50 as the model takes random actions in the environment which naturally do not converge without much learning. Therefore, continuous rewards of -1 accumulate to give a total of -50.

However, over time, the model learns to take correct decisions and slowly starts performing better which is reflected by better reward accumulation. Ultimately, the model converges as rewards stabilize closer to a value of 0. We can clearly see the difference in the performance of the model with and without using HER, as seen by the loss and reward values below. While the graph converged to a reward value closer to 0, when using HER, the rewards do not cross a cumulative value of -45

in the other, which is clearly worse. Further, the loss function does not stabilise at all. Rather, it continues to incur very low reward values, even after 2000 iterations.

7.2 Fetch Pick and Place

We can see a similar trend in the pick and place operation. However, convergence took much longer than in the case of fetch reach (nearly 9000 episodes). This is because the task was much more complicated, compared to reaching a specified point. Below is the graph containing the rewards and losses over the episodes. In this case, we agent picks up the block and drops it at the goal. This leads to low losses in case of goals being on the ground but otherwise higher losses. In either case, we can observe an initial spike in the loss function. This is because of using HER algorithm.

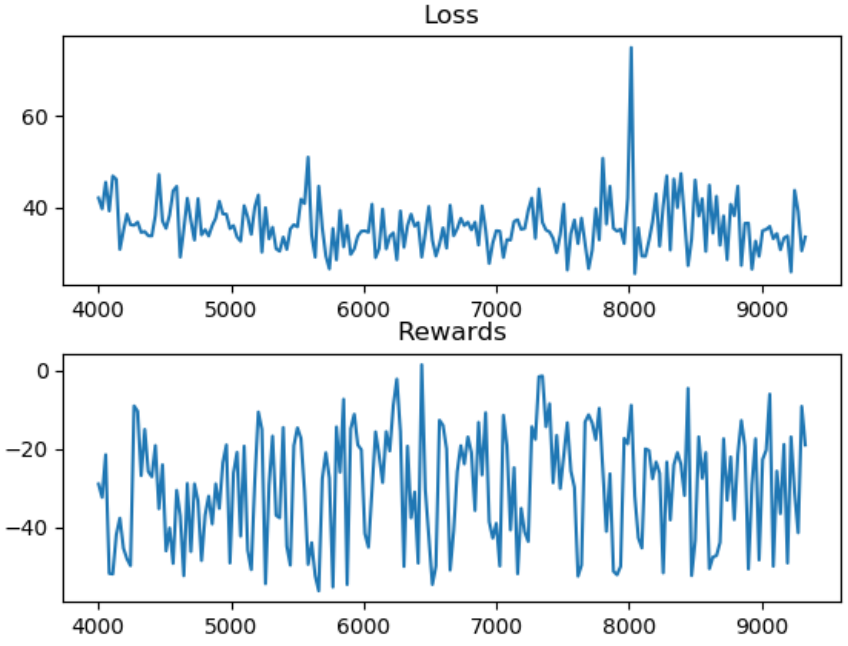

FIGURE 7.3: Fetch Pick and Place using HER

7.3 FetchPush

For the case of FetchPush, we have evaluated our model against success rate and maximum success rate as the metric. Note that while in the previous two models, we plotted Loss and Rewards against the iteration number, in these models, we plot success rate against the epoch number. An epoch consists of 50 iterations. The success rate is calculated for every epoch as the percentage of iterations in that epoch which where the goal was achieved.

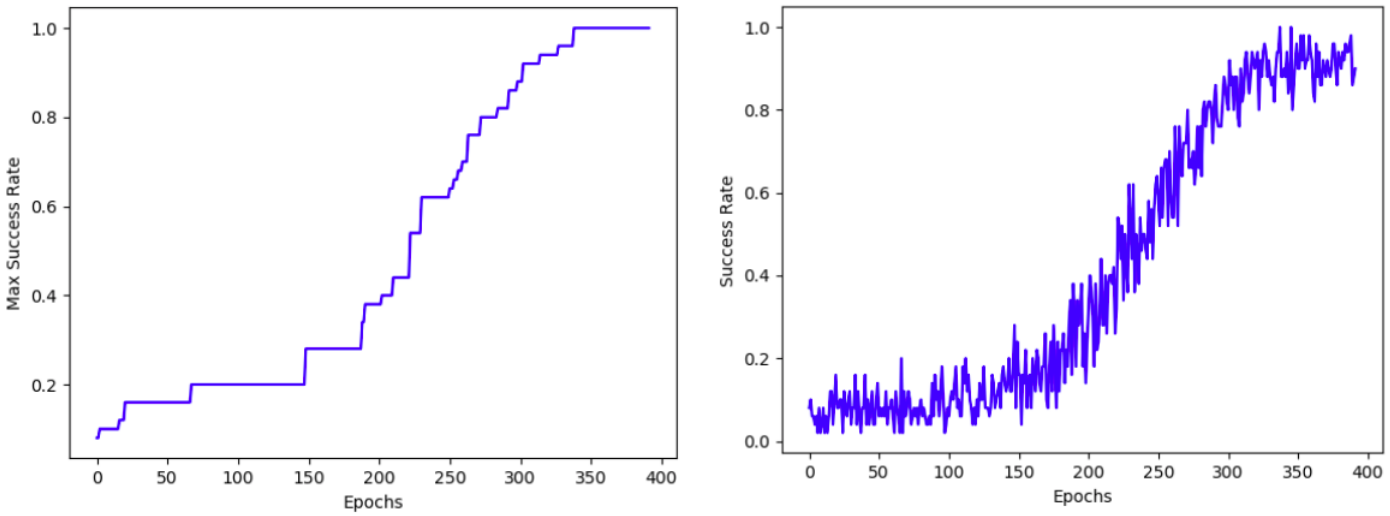

FIGURE 7.4: Fetch-Push Max success rate with HER (left),Fetch-Push success rate with HER (right)

7.4 FetchSlide

Similar to FetchPush, success rate and maximum success rate were used as the metrics to evaluate our model. The model was allowed to train over 200 epochs and as seen in the graph below (Fig 7.5), the model gradually gets better at performing the task of sliding an object to a desired position. The model learns to get better at performing the task over time by interacting with the environment and continuously evaluating the reward it gets for each step taken in the action space. Over time, as the model takes the correct action, it gets rewarded better which leads to an increase in success rate.

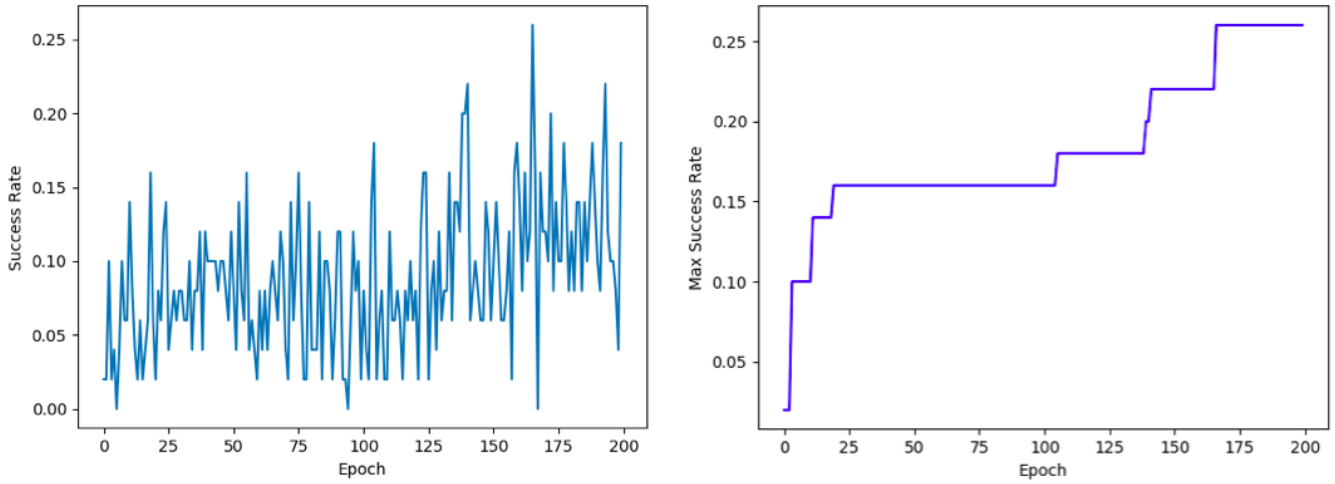

FIGURE 7.5: Fetch-Slide using HER - Success Rate(left), Fetch-Slide using HER - Max Success Rate(right)

CHAPTER 8

CONCLUSION AND FUTURE WORK

Though HER is a promising way towards learning complex goal-based tasks with sparse rewards like the environments that we have used here, there is still some room for improvement. Currently, each of the control operations have separate, well defined environments that are provided to us by OPEN AI. Each of these environments define the rewards for respective environments and the agent is positively reward only if the action specific to that environment is undertaken. For example, Fetch-Reach has it's own software environment defined by the API. Only if the robotic arm is able to perform the 'Reach' operation is the agent rewarded, any other action,including, the other control operations result in a negative reward being assigned to that particular action. Similarly, in a Fetch Pick and Place environment, the agent must be able to pick and place the puck at a designated position.

If the agent performs the action 'Fetch-Reach' in the Pick and Place environment, a negative reward will be assigned to that action. Therefore, it is easy to see that the agent must only perform the action that corresponds to a particular environment. Hence, we have trained separate models for each of the control operations that we wish to solve.

Ultimately, the project can be extrapolated to create a 'Super-agent': A singular Agent capable of selecting any of the control operations and performing it. This super-agent establishes a Hierarchical structure to the project.Rather than the individual models solving the control operations that it is assigned to, we can modify the output of a super-agent in such a manner that it selects any of the possible control operations to perform, which in turn uses our predefined model to solve that particular action.

Another potential application of a super-agent would be to recursively perform the same operation which in turn could have myriad applications. For example, consider the Fetch Pick and Place control operation. Assume, an environment requiring stacking of blocks. We can have a super-agent decide actions in a higher level and low level agents implement the pick and place operation sequentially.

In addition, there are several research requests made by OpenAI on HER algorithm that could fine tune the algorithm and help it refine models better. Since, HER is used extensively in most of the Deep RL applications, just a small change in the algorithm performance can lead to huge advancements in pre existing applications and systems. Some of the potential improvements that could be made include:

- Automatic hindsight goal creation: We currently have a hard-coded strategy for selecting hindsight goals that we want to substitute. This could be learnt in the future.

- RL with very frequent actions: Current RL algorithms are very sensitive to the frequency of taking actions. In our continuous control domain, inconsistent exploration and necessity to propagate information backwards, leads to performance going to zero as the action space goes high. A method to devise an efficient RL algorithm which can retain its performance even when the frequency of taking actions goes to infinity can be researched upon.

- Faster information propagation: Our agent makes use of a target network to stabilize training. However, since changes need time to propagate, this will limit the speed of training and we have noticed in our experiments that it is often the most important factor determining the speed of DDPG+HER learning. It would be interesting to investigate other means of stabilizing training that do not incur such a slowdown.

- HER + HRL: It would be interesting to further combine HER with a recent idea in hierarchical reinforcement learning (HRL). Instead of applying HER just to goals, it could also be applied to actions generated by a higher-level policy. For example, if the higher level asked the lower level to achieve goal A but instead goal B was achieved, we could assume that the higher level asked us to achieve goal B originally.

REFERENCES

1. Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, J., (2017) 'Hindsight Experience Replay', Advances in Neural Information Processing Systems, Vol. 30, pp. 5048–5058

2. Gao, Y., Xu, H., Lin, J., Yu, F., Levine, S., and Darrell, T. (2018) 'Reinforcement Learning from Imperfect Demonstrations', In the proceedings of the International Conference on Machine Learning, arXiv preprint arXiv:1802.05313.

3. Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, .D (2016) 'Continuous Control with Deep Reinforcement Learning', International Conference on Learning Representations arXiv preprint arXiv:1509.02971.

4. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis,D. (2015) 'Human-level control through deep reinforcement learning' Nature, Vol. 518, pp. 529533.

5. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013) 'Playing Atari with Deep Reinforcement Learning', In the proceedings of Neural Information Processing Systems Conference, Deep Learning Workshop, arXiv preprint arXiv:1312.5602.

6. Mnih, V., Badia, A., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K., (2016) 'Asynchronous Methods for Deep

Reinforcement Learning', In the proceedings of the International Conference on Machine Learning, arXiv preprint arXiv:1602.01783.

7. Nachum, O., Gu, S., Lee, H., and Levine, S., (2018) 'Data-Efficient Hierarchical Reinforcement Learning', In the proceedings of Neural Information Processing Systems Conference arXiv preprint arXiv:1805.08296.

8. Nair, A., McGrew, B., Andrychowicz, M., Zaremba, W., and Abbeel, P., (2018) 'Overcoming Exploration in Reinforcement Learning with Demonstrations', In the proceedings of IEEE International Conference on Robotics and Automation, pp. 6292-6299.

9. Pathak, D., Agrawal, P., Efros, A., and Darrell, T., (2017), 'Curiosity-driven Exploration by Self-Supervised Prediction', In the proceedings of the International Conference on Machine Learning arXiv preprint arXiv:1705.05363.

10. Plappert, M., Houthooft, R., Dhariwal, P., Sidor, S., Chen, R., Chen, X., Asfour, T., Abbeel, P., and Andrychowicz, M., (2018),'Parameter Space Noise for Exploration', In the proceedings of the International Conference on Learning Representations, arXiv preprint arXiv:1706.01905.