Virtual Robot Climbing using Reinforcement Learning

A Project Presented to

The Faculty of Department of Computer Science

San Jose State University

In Partial Fulfilment of

the Requirements for the Degree

Master of Science

By

Ujjawal Garg

December 2018

© 2018

Ujjawal Garg

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Master's Project Titled

Virtual Robot Climbing using Reinforcement Learning

By

Ujjawal Garg

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2018

Dr. Christopher Pollett	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Katerina Potika	Department of Computer Science

ABSTRACT

Virtual Robot Climbing using Reinforcement Learning

By Ujjawal Garg

Reinforcement Learning (RL) is a field of Artificial Intelligence that has gained a lot of attention in recent years. In this project, RL research was used to design and train an agent to climb and navigate through an environment with slopes. We compared and evaluated the performance of two state-of-the-art reinforcement learning algorithms for locomotion related tasks, Deep Deterministic Policy Gradients (DDPG) and Trust Region Policy Optimisation (TRPO). We observed that, on an average, training with TRPO was three times faster than DDPG, and also much more stable for the locomotion control tasks that we experimented. We conducted experiments and finally designed an environment using insights from transfer learning to successfully train an agent to climb slopes up to 36°.

ACKNOWLEDGEMENTS

I would like to express my sincerest gratitude to Dr. Christopher Pollett for his patience and his pertinent guidance throughout the duration of my project. I consider myself to be extremely fortunate to have had an opportunity to work with someone as brilliant as him.

I am also grateful to my committee members, Dr. Robert Chun and Dr. Katerina Potika for providing their valuable feedback and guidance.

And finally, I thank my wonderful friends and parents for always having my back and making my journey every bit memorable.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Problem Statement

One of the long-standing goals of Artificial Intelligence is to achieve an intelligent machine which is able to perform any intellectual task that a human is capable of. This is sometimes referred to as Artificial General Intelligence (AGI). AGI models the human intellect in a software such that, given an unfamiliar task, the software is able to come up with a solution to perform the task. Physical control tasks like walking, opening a door, etc. are performed naturally by the human mind but still remain as one of the biggest tasks that we have not yet been able to model for practical use in robotics. These tasks give rise to highly complex control challenges, making it impractical to program all the aspects of this problem by hand. However, recent interest and advancements in the field of Deep Reinforcement Learning have been a tremendous boon for robotics research. The goal for this project is to use Deep Reinforcement Learning to train a quadruped agent to learn climbing on a variety of slope based environments. The final trained agent is shown to be performing remarkably well and was able to climb slopes of up to 36°.

Advancements in the field of robotics will enable humans to build more capable robots so that they can be used in society. These robots might be used to replace humans for dangerous tasks like mountain exploration and rescue missions. According to [1], the K2 mountain range has a fatality rate as high as 25% and nearly 300 people have died on Mount Everest [2] in the last 100 years. With the advancements in technology and robotics, we have the potential to deploy intelligent robotic systems for these kinds of explorations. Such deployment evades the need for humans to perform life-threatening exploration and avoid the loss of human life. Another use case for robotic intelligence in terrain exploration can be for interplanetary exploration. On November

26, 2018, NASA successfully landed project Insight [3] onto the Martian surface. However, the Insight probe is stationary and cannot move around the Martian surface. Thus, it is able to collect the data only at the site of the landing. Such extra-terrestrial expeditions cost billions of dollars and thus we need to make the most efficient use of the resources available. With current advancements in reinforcement learning methods, we have the potential to train a system that can learn to navigate the slopes of various Martian terrain and climb even the most treacherous terrains easily. This can help in exploring the uncharted challenging terrains of Mars. There are numerous other strenuous tasks that can be fulfilled by intelligent robotics advancements. With the current advancements in intelligent systems ranging from autonomous cars to smarter gaming bots, we need systems that can explore possibilities beyond human capabilities.

The Google DeepMind project published a paper [4] and video [5] showing simulated agents trained to navigate through a set of challenging terrains. These agents were trained using reinforcement learning. Initially, the goal was to similarly train a humanoid simulation to successfully climb a rock wall. However, it became apparent that such a task was infeasible given the current constraints on computation resources and tools available. In fact, the task of *generalized* object grasping remains a fundamental problem that is not easy to solve. Works like [6] show promising results but require feedback in terms of visual input or other sensory data [7] [8]. These works also require working on a physical robot instead of a simulation and require a significant amount of domain knowledge. Thus, we decided to reduce the scope of the project to train an agent to climb a slope-based environment.

1.2 Related Work

Related work in climbing robots have considered climbing as a problem of gait control [9], or made use of dedicated grasping effectors [10], [11] on a physical robot. Other works consider the climbing problem as a planning problem [12]. Even though recent works show remarkable progress in highly complex control tasks, to our knowledge this type of task as a physical control problem has not been solved using only the Reinforcement Learning algorithms. In [8], they train a humanoid simulation to perform a variety of control tasks through imitation learning using the Motion Capture (MoCap) data available from [13]. One of the tasks is to train the agent to climb a set of stairs. However, such MoCap or visual data may not be available for any custom designed robot, and generating the data itself is a difficult task as it requires specialized domain knowledge.

Moreover, most research work is focused on training to perform a specific task at hand and the trained models fail to work when the parameters of the tasks are changed. Recent advances in the field of transfer learning [14] [15] [16] for neural networks have focused on improving the training time and performance of the trained models. These techniques usually work by learning a source task first and then using the trained representation to speed up the learning of the target task. Adopting these transfer learning techniques for robotics can prove to be very useful because it can help in reducing the time and efforts spent during the exploration phase of the training. However, applying these techniques to the Reinforcement Learning algorithms is a difficult task [17]. This is because specific aspects of the source and target problem need to be matched to make the transfer successful. Moreover, other transfer learning issues like negative transfer, etc. are observed in Reinforcement Learning adopted frameworks as well.

1.3 Contribution

The primary contribution of this project is to evaluate the effectiveness of two Deep Reinforcement Learning algorithms, namely Trust Region Policy Optimisation (TRPO) [18] and Deep Deterministic Policy Gradient (DDPG) [19] algorithms, to train an agent to learn the locomotion behaviors needed to climb a slope based environment. We also observed that utilizing the concept of transfer learning, we can significantly improve the effectiveness of the trained agent. We designed an environment, that had the concept of transfer learning built into the environment itself. The environment was designed as a series of sequential climbing tasks with increasingly steeper slopes. In our experiments, using such an environment has shown a significant increase in the effectiveness and efficiency of the training.

The project report is organized into chapters as follows: Chapter 2 defines the common concepts and terms used in Reinforcement Learning. Chapter 3 define the design and implementation of our project and the setup we used to conduct the experiments. Chapter 4 describes the experiments we conducted and their results. Finally, in Chapter 5 we have our conclusion.

CHAPTER 2
Background

In this chapter, we provide background information related to reinforcement learning and other related concepts. This background information is crucial to understand the algorithms and techniques used in the project. Section 2.1 defines the RL framework, and Section 2.2 provides definitions for common terms and concepts used in RL. Section 2.3 introduces Q-Learning, which is a simple learning algorithm for RL. Section 2.4 defines how Deep Learning is used to augment RL techniques and presents the two state-of-the-art algorithms evaluated in this project. Finally, in Section 2.5, the concept of transfer learning and how it can be used to improve the training process in RL.

2.1 Reinforcement Learning

Unlike supervised learning, where we use some training data as input, in reinforcement learning, we only have access to an environment and an agent (aka actor) which can perform only a specific set of actions. The goal for Reinforcement Learning is to train this agent to automatically determine the ideal sequence of actions to perform for maximum efficiency in solving the required task for the given environment. The environment is assumed to be a Markov Decision Process (MDP). An MDP is a process which can be defined using a set of states and a transition function. This transition function defines the probability of moving from one state to another. For any given environment in Reinforcement Learning, we assume that we do not have any knowledge of the transition function and the set of states. The agent needs to *explore* the environment and gain knowledge about the states and the transitions. As the training progresses, the agent starts to learn about the environment and can use this knowledge to take actions which are more favorable. This is known as *exploitation*. Fig. 1 shows the reinforcement learning framework [20].

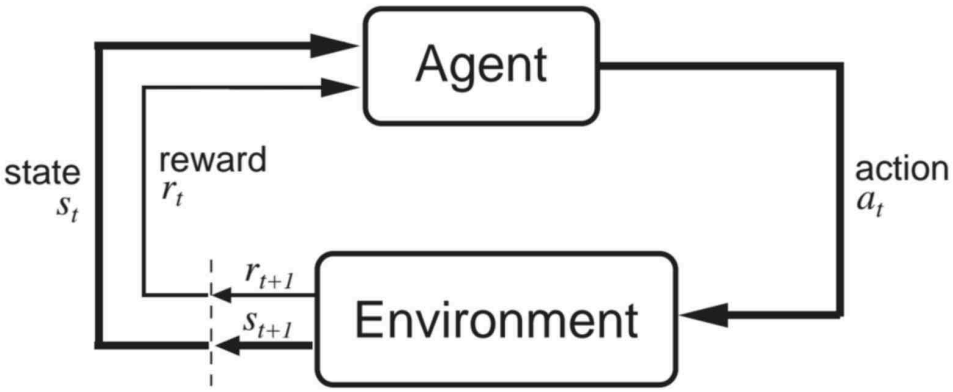

Figure 1: Reinforcement Learning framework

2.2 Formal Definitions

State Space: Let $S = \{s_1, s_2, \ldots, s_N\}$ be the set of possible states in the environment. This is known as *state space*.

Action Space: Let $A = \{a_1, a, \ldots, a_M\}$ be the set of actions that the agent can perform. This is known as an *action space*.

Transition Probability: Let the agent perform action $a \epsilon A$ while it is in state $s \epsilon S$. Let the new state it arrives be $s' \epsilon S$. Then the *transition probability* is denoted as $T(s'|s, a)$.

Reward Function and Reward: As mentioned earlier, the agent learns from the results of its actions. To compare the results, it needs a metric. This metric is termed as a *reward*. If an agent performs action $a \epsilon A$ while in state $s \epsilon S$, then the reward received from this environment is denoted by $R\ (s, a)$. The function which models the reward values for a particular environment is known as *reward function.* This function is always contextual and depends on the task being solved in the environment.

Policy: A *policy*, often denoted by π_θ, is a rule or mapping from each state $s \epsilon S$, to an optimal action $a \epsilon A$ for the given environment θ. An optimal policy is one in which if we follow the actions determined by such a policy, then the total expected reward will be maximum. A *stochastic policy* gives a probability distribution for the next optimal action, and is denotes by $\pi_\theta(a|s) = \mathrm{P}[a|s;\theta]$. A *deterministic policy*, on the other hand, gives a single action that should be taken.

Value and Q-Value: The reward for performing an action gives us information about only its immediate effect. A *value,* often denoted by $V^\pi(s)$, on the other hand, provides us information regarding the advantage of being in state s in the long run. The Q-value, denoted by $Q^\pi(s,a)$, represents the total amount of reward the agent can expect for performing action a in state s, if the agent follows policy π to determine its future actions. For a given policy π, the value equation is given as:

$$V^\pi(s) = E_\pi\{\sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1} | s_t = s\}$$

Here, $E_\pi[x]$ denotes the expected value of random variable x, if policy π is follows by the agent. Also, γ is the discount factor and r is the immediate reward.

Model-Based and Model-free: A reinforcement learning algorithm may be model-based or model-free. A *model-based* algorithm tries to learn the transition function T and the reward function R to create a model of the given environment. This model can then be used to determine optimal actions. On the other hand, a *model-free* algorithm does not try to learn anything regarding the underlying dynamics of the environment. Instead, it estimates either the value function or the policy and uses them to determine the optimal actions.

Off-policy and On-policy: A reinforcement learning algorithm which estimates the policy may be either off-policy or on-policy. As mentioned earlier, while training the agent, it needs to explore the environment. An *off-policy* algorithm will choose the actions from a behaviour policy that is different from the policy being trained on. While an *on-policy* algorithm will use the same policy for exploration and exploitation.

2.3 Q-Learning

Q-Learning [21] is a model-free, off-policy learning algorithm, where we model the value function as a Bellman equation, and the reward for performing an action a is calculated as the instant reward for entering the new state plus the maximum discounted future reward that can be obtained in the new state. This equation is given as:

$$Q(x, a) = R_x(a) + \gamma \, max_{a'} Q(x', a')$$

Here, Q is the function that we want to learn, R_x is the instant scalar reward for performing action a in state x, γ is the discount factor, and x' is the next state. The estimated Q-values are stored in a look-up table known as Q-table. As the algorithm trains the agent, this table is updated with values which are more correct. The discount factor serves two purposes. First, it makes future rewards worth less than the immediate rewards. Second, it ensures that the sum is finite and the model can converge.

A simple Tic-Tac-Toe agent was implemented to gain a better understanding of the Q-Learning algorithm. For training the agent, two python scripts were created. The first script was used to train the agent against a programmed bot and generated a model file which contained the Q-table. The second script simulated the gameplay against a human player and was used for evaluation. The agent was trained using the first script for 50000 random gameplays. This training generated more than 5000 entries in the Q-table. During the evaluation, the agent performed well and won 14 out of 15 games on an average. Fig. 2 shows the result of this evaluation.

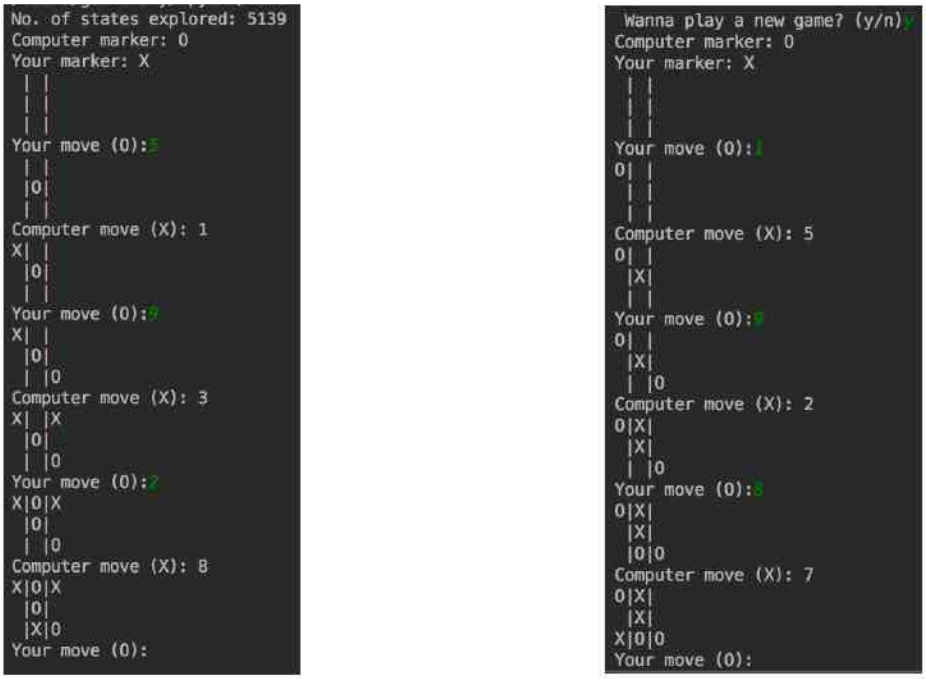

Figure 2: Evaluation of trained Tic-Tac-Toe agent

2.4 Deep Reinforcement Learning

The Q-Learning algorithm using look-up table works well for the Tic-Tac-Toe example because there are only 5812 legal states and our agent explored most of these states. However, for a more complex task like a chess game such exploration and creation of a look-up table is not practical, both in terms of time and space constraints. Deep Reinforcement Learning (DRL) uses the recent advancements and interest in Deep Learning to solve this problem. In [22], Deep Q-Learning (DQN) algorithm is used to train a model to play Atari video games. In DQN, instead of lookup tables, a neural network like a Convolutional Neural Network (CNN) is used to estimate the Q-value function. Such function approximators are the core concept of Deep learning.

One additional issue with traditional Q-Learning techniques is that the underlying data distribution changes as agent learns new behaviors. DQN solves this problem using a technique known as experience replay. In this technique, instead of using data only from current iteration, a randomly sampled mini-batch from previous

iterations is used to update the weights in the network. This technique allows us to make the most efficient use of previous experience.

While DQN solved the issue of high observation spaces, it is not practical for physical control tasks like grasping, object manipulation etc. This is due to the fact that these tasks have continuous and high-dimensional action spaces, and DQN can only handle discrete and low-dimensional action spaces. Policy Iteration methods with deep function approximators like Trust Region Policy Optimisation (TRPO) [18] have shown remarkable results for a variety of physical control tasks. These methods learn to estimate the policy directly instead of the Q-value function. In addition, a combination of policy and value estimation like Deep Deterministic Policy Gradients (DDPG) [19] are also quite commonly used to train models for such tasks.

2.4.1 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is a model-free, off-policy, actor-critic approach based on the DPG (Deterministic Policy Gradient) [23] method. In an actor-critic approach, the actor represents the policy function and specifies the action to be performed given the current state of the environment. The critic represents the value function which specifies the resultant reward and produces a signal error to criticize the actions made by the actor. There are two network instances of actor and critic. The first instances, known as target policies, are the networks that are being trained. The other two instances, known as behavior policies, are used to generate the trajectory. The deterministic part comes from the fact that while the behavior policies are still stochastic, the target policies are deterministic.

In algorithms like DQN, we use a stochastic policy and learn to select the action based on the following equation:

$$a_t = max_a \, Q^*(\emptyset(s_t), a; \, \theta)$$

But this equation is not practical for continuous action spaces. Using a deterministic policy allows us to use the equation:

$$a_t = \mu(s_t|\theta^\mu)$$

Similar to DQN, a replay buffer R is used to sample the episode data from previously generated trajectories. This data is then used for training. Also, a random noise process $\mathcal{N}$ is used to ensure sufficient exploration:

$$a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$$

The final algorithm is shown in Algorithm 1. Here the algorithm runs for $M\ X\ T$ time-steps. For each time-step, the algorithm generates the trajectories or transitions according to the current policy. To do so, the agent performs an action based on the current actor policy μ and exploration noise $\mathcal{N}$ and then stores the resulting transitions in a replay buffer R. The noise process selected for our experiments is the Ornstein-Uhlenbeck process.

At the end of each time-step, the algorithm updates the networks using a randomly sampled mini-batch from the replay buffer R. In [23], Silver, et al. proved that the equation given in (1.1) is the *policy gradient*, i.e., we will get the maximum expected reward as long as we update the model parameters using the gradient formula given in the equation.

Algorithm 1: Deep Deterministic Policy Gradient

Initialization:
Initialize critic $Q(s, a|\theta^Q)$ and actor $\mu(s_t|\theta^\mu)$ with random 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

Training:
for episode 1 to M **do:**
Initialize a random noise process $\mathcal{N}$
Receive initial observation state s_1
for time-step 1 to T **do:**
Generate Trajectories:
Execute action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$
Receive reward r_t and new state s_{t+1}
Store transition $(s_t, a_t, s_{t+1}, a_{t+1})$ in R
Update networks:
Sample a random mini-batch of length N from R:
$(s_i, a_i, s_{i+1}, a_{i+1})$, $1 \leq i \leq N$
Get the resultant value of state s_i according to target critic Q':
$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\, \theta^{Q'})$
Get the resultant value of state s_i according to critic Q:
$x_i = Q(s_i, a_t|\, \theta^Q)$
Update critic Q by minimizing loss L:
$L = \frac{1}{N} \sum_i (y_i - x_i)^2$
Update the actor μ using the policy gradient:
$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$ **(1.1)**
Apply soft-update to the target networks:
$\theta^{Q'} \leftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'}$
$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1 - \tau)\theta^{\mu'}$

2.4.2 Trust Region Policy Optimization

Trust Region Policy Optimization (TRPO) is a model-free, on-policy, actor-critic approach based on Natural Policy Gradient (NPG) [24] method. The policy being trained is a stochastic policy. Thus, TRPO can work on both continuous and discrete action spaces. The major contribution of this algorithm is that it uses insights from optimization theory which provides guaranteed monotonic improvement. In particular, the algorithm uses a *trust region* [25] to constraint the update size during training. This constraint size is determined by the Kullback-Leibler (KL) divergence between the old policy and the new policy. I.e.,

$$E\left[D_{KL}(\pi_{\theta_{old}}(.|s)\middle|\middle|\pi_{\theta_{new}}(.|s)\right] \leq \delta,$$

where,

D_{KL} is the KL divergence,

δ is the constraint region

TRPO uses the advantage estimation $\hat{A}_t$ to define the policy gradient $\hat{g}$ as:

$$\hat{g} = \hat{\mathrm{E}}_t\left[\nabla_\theta log\pi_\theta(a_t|s_t)\hat{A}_t\right]$$

where,

$$A = Q(s,a) - V_\pi(s)$$

However, the advantage estimate can be noisy and is prone to making the loss function explode or implode. Schulman, et al., proved that the objective function can be modified to:

$$\max_\theta \hat{\mathrm{E}}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right], \text{ constrained to } \hat{\mathrm{E}}_t\left[\overline{D_{KL}}(\pi_{\theta_{old}}(.|s) \,||\, \pi_{\theta_{new}}(.|s))\right] \leq \delta$$

where,

$\overline{D_{KL}}$ is the *mean* KL divergence over the state space.

To calculate the gradient direction, we need to calculate $\widehat{H}_k^{-1}\hat{g}_k$, where $\widehat{H}_k$ is the Hessian product or Fisher Information Matrix. Calculating the inverse Hessian is computationally very expensive. So, TPRO uses the conjugate gradient [26] to approximate the inverse. The final algorithm is given as Algorithm 2.

Like the DDPG algorithm, TRPO also maintains two network instances for actors and critics each. The first actor-critic pair of networks belongs to the current policy that we want to refine. The second pair belongs to the policy that we last used to collect the samples.

Algorithm 2. Trust Region Policy Optimisation

Initialization:

Initialize the actor & critic networks corresponding to initial policies and the constraint size δ.

Training:

for k = 0 to M **do**

Generate the rollout set $\mathcal{D}_k$ with N trajectories using policy $\pi_k = \pi(\theta_k)$

Estimate the advantage function $\hat{A}_k$ for all time-steps in $\mathcal{D}_k$

Calculate the policy gradient $\hat{g}_k$ and KL divergence Hessian $\widehat{H}_k$

Use conjugate-gradients to obtain $x_k \approx \widehat{H}_k^{-1}\hat{g}_k$

Compute the step size $\Delta_k \approx \sqrt{\frac{2\,\delta}{x_k^T \widehat{H}_k x_k}}\, x_k$

$\theta_{k+1} = \theta_k + \Delta_k$

2.4 Transfer Learning

Deep learning techniques have shown remarkable success in learning a very accurate mapping from inputs to outputs. Achieving this success requires a huge amount of training and hyperparameter tuning. In transfer learning, we seek to transfer the knowledge learned from pre-trained models to a new task. For example, researchers have spent hundreds of hours to train Convolutional Neural Networks (CNNs) on

ImageNet [27] data-set so that the trained model can do various image classification tasks. In [28], the authors use such previously trained CNNs to speed up the learning process and achieve state-of-the-art results for different data-sets as well.

One approach to apply transfer learning is to freeze the weights of the pre-trained network in all the layers except the output layer. The training is done keeping only the output layer trainable. The number of outputs in the layer and the training data is modified according to the new task. This approach has proved to be very effective in speeding up the training process for similar tasks in Deep Learning.

Traditional techniques for transfer learning works very well on CNNs but adopting them for Reinforcement Learning is a difficult task. In [29], Hierarchal learning is suggested as one of the approaches to adopt transfer learning for Reinforcement Learning. In this approach, a single large task is divided into a series of sequential sub-tasks. As model learns these sub-task, it will start to move towards achieving success in large task as well. In this project, we use a similar approach to model our learning environment to facilitate Reinforcement Learning.

CHAPTER 3
Design and Implementation

In this chapter, we provide the design and implementation details about our project. Section 3.1 describes the tools and libraries used for the experiments. Section 3.2 defines how we used these libraries to create the environments for training the agents. In Section 3.3, we provide details about the algorithms we used. Finally, in Section 3.4, we describe the cloud machine that was used to conduct the experiments.

3.1 Libraries Setup

To conduct the experiments, we used the OpenAI *gym* [30] [31] and *baseline* [32] libraries to define the environment and train the models. Both these libraries are open source and implemented in Python 3 programming language. The *gym* library is a framework for reinforcement learning which provides a uniform interface for building reinforcement learning environments, and running the agents. The *baseline* library contains standard implementations for popular reinforcement learning algorithms. In particular, we use the baseline implementations of *trpo_mpi* and *ddpg* algorithms to train our models. Using the standardised framework and algorithms will make our results replicable. Both the gym and baseline libraries were installed from GitHub source using *pip*.

The gym library uses the MuJoCo physics engine [34] [35] to render the environment and simulate the actions and physical systems. This physics engine is implemented in C++ programming language. The gym library uses *mujoco-py* [33], which provides the python bindings for the MuJoCo physics engine. At the time of this writing, the MuJoCo library required a paid license, which costs $500 for personal non-commercial use. However, MuJoCo also offers a one-month trial license and a free student license for one year. The trial license can be used on 3 different machines, while the student license can be used only on one machine. Initially, we used the trial license to evaluate the library on our local machine (a core-i5 MacBook) and on a Google

Compute Engine (GCE) instance. Once we were satisfied that this setup was working we obtained a student license and registered it for our GCE instance.

3.2 Gym Environment Creation

For each experiment, we created a gym environment. Each gym environment requires an XML file, which represents the MuJoCo model of the simulation. The XML file format is known as MJCF and contains the data about the joints configurations, actuator and gear information, and how the different parts of the robot are connected together. Each MJCF file contains an XML tree created by nested *body* elements. Each body element has *geoms* attached to it, which define the geometric shape and structure of the body. Fig. 3 shows a snippet of MJCF XML file for a one-legged hopper agent. For each joint, an actuator element defines the motor object to control the joint. The XML file is parsed using the mujoco-py library for rendering and physics simulation. Some experiments required the XML file to be dynamically generated. For such experiments, we used the *elementTree* python module to generate the XML file.

```
<body name="torso" pos="0 0 1.25">
  <joint armature="0" axis="1 0 0" damping="0" limited="false" name="rootx" pos="0 0 0" stiffness="0" type="slide"/>
  <joint armature="0" axis="0 0 1" damping="0" limited="false" name="rootz" pos="0 0 0" ref="1.25" stiffness="0" type="slide"/>
  <joint armature="0" axis="0 1 0" damping="0" limited="false" name="rooty" pos="0 0 1.25" stiffness="0" type="hinge"/>
  <geom friction="0.9" fromto="0 0 1.45 0 0 1.05" name="torso_geom" size="0.05" type="capsule"/>
  <body name="thigh" pos="0 0 1.05">
    <joint axis="0 -1 0" name="thigh_joint" pos="0 0 1.05" range="-150 0" type="hinge"/>
    <geom friction="0.9" fromto="0 0 1.05 0 0 0.6" name="thigh_geom" size="0.05" type="capsule"/>
    <body name="leg" pos="0 0 0.35">
      <joint axis="0 -1 0" name="leg_joint" pos="0 0 0.6" range="-150 0" type="hinge"/>
      <geom friction="0.9" fromto="0 0 0.6 0 0 0.1" name="leg_geom" size="0.04" type="capsule"/>
      <body name="foot" pos="0.13/2 0 0.1">
        <joint axis="0 -1 0" name="foot_joint" pos="0 0 0.1" range="-45 45" type="hinge"/>
        <geom friction="2.0" fromto="-0.13 0 0.1 0.26 0 0.1" name="foot_geom" size="0.06" type="capsule"/>
      </body>
    </body>
  </body>
</body>
```

Figure 3: A MJCF body element for one-legged hopper agent

Apart from the XML file, the *gym* environment is implemented as a python class which extends the *MujocoEnv* parent class defined in the gym framework. Fig. 4 shows the signature and brief description of the methods for such a class file. Each environment needs to implement the *step* method, which takes the action to be performed as input, performs the action using MuJoCo library, and returns the

calculated reward based on the new state of the environment. Each environment is registered with gym framework with a specific name so that it can be used for training.

```
def __init__(self):
    """
    Specify the xml file for model and load using self.__init__(model_path
    Implement this in each subclass.
    """

def step(self, a):
  """Run one timestep of the environment's dynamics. When end of
     episode is reached, you are responsible for calling `reset()`
     to reset this environment's state.
     Accepts an action and returns a tuple (observation, reward, done, inf
     Args:
        action (object): an action provided by the environment
     Returns:
        observation (object): agent's observation of the current environme
        reward (float) : amount of reward returned after previous action
        done (boolean): whether the episode has ended, in which case furth
        info (dict): contains auxiliary diagnostic information (helpful fo
    """

def reset_model(self):
    """
    Reset the robot degrees of freedom (qpos and qvel).
    Implement this in each subclass.
    """
```

Figure 4: Signature of a MuJoCo environment python class

3.3 Baseline algorithms

Both the *trpo_mpi* and *ddpg* algorithms in the baseline library takes some parameters that define how the algorithm will train the model. The *network* argument passed to the algorithm defines the neural network to be used for actors and critics. Tensorflow library is used to implement the networks. For our experiments, we use a multi-layer perceptron (MLP) for both actors and critics. Unless otherwise specified, we use an MLP with 2 fully-connected hidden layers, having 64 units in each layer. The *tanh* function is used for activation. The input and output layers for actor and critic are different. For actors, the input layer is the observations from the environment and the output layer provides the actions to be performed. For critics, the input layer is the the

observations from the environment and the last hidden layer takes actions taken by agent as additional inputs. The final output of the citric is a signal value which is used to make the training updates.

Fig. 5 shows the complete network architecture for actor and Fig. 6 shows the network architecture for Critic. The number of observable units and the number of actions is determined by the environment.

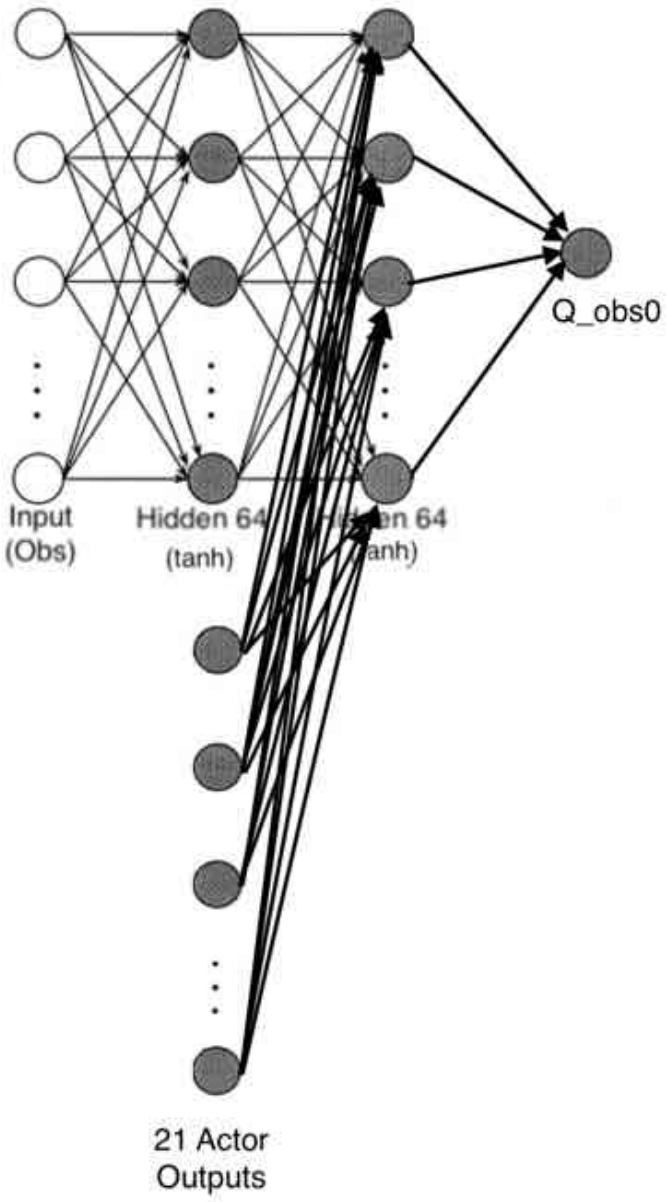

Figure 5: Neural Network for Critic

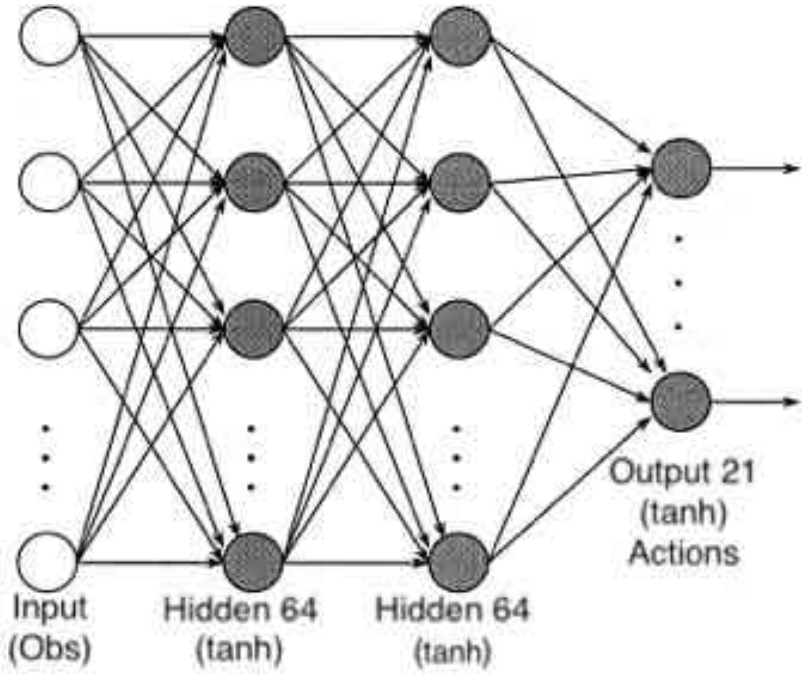

Figure 6: Neural Network architecture for Actor

3.4 Cloud Setup

Apart from a few initial experiments, most of the experiments were run on a Google Compute Engine (GCE) instance with 24 cores and 20GB memory. Both of the algorithms available in the baseline library have support for parallel computation using OpenMP framework. This enables us to make full utilization of the available computing resources.

CHAPTER 4
Experimental Results

In this chapter, we provide details about each experiment we conducted and show the results and observations. For each trial, the agent was trained inside the registered gym environment for a fixed number of simulation time-steps. After training, we evaluate the agent and observe the results achieved in each experiment. We use these observations to design the next trial.

4.1 Hopper

The aim of this experiment was to familiarize our self with the libraries and the training procedure. For this experiment, we decided to train a two-dimensional one-legged robot [36] to hop forward as fast as possible. It is one of the simplest physical control task available in the gym framework. There are only 3 joints to be controlled and the state space has 20 variables. We conducted two experiments using the DDPG and TPRO algorithms respectively. The training was done for 216 epochs using DDPG algorithm and 591 epochs using TRPO algorithm. At the time of this experiment, the baseline library used epochs instead of time-steps to specify the training duration. This has been now changed to accept time-steps as it provides more consistent training times. Both the experiments were done on an Apple MacBook-Pro 2015 laptop and training took a little more than 10 hours for both experiments run together. The following reward function was used:

$$reward = v_{fwd} - 10^{-3}\left|\left|u\right|\right|^2 + 1.0$$

Here,

v_{fwd} = forward velocity

u = vector of joint torques

f_{imp} = impact forces

In the above equation, the addition of a constant offset value of 1.0 is used to encourage longer episodes. Otherwise, the training will lead to a policy that ends episodes as quickly as possible. The torque vector penalty is to discourage big changes, and the forward velocity provides the actual reward for that time-step. These values are the defaults for OpenAI gym *Hopper-v2* environment. Fig. 7 and Fig. 8 show the simulation images for an episode after training using DDPG and TRPO algorithms respectively.

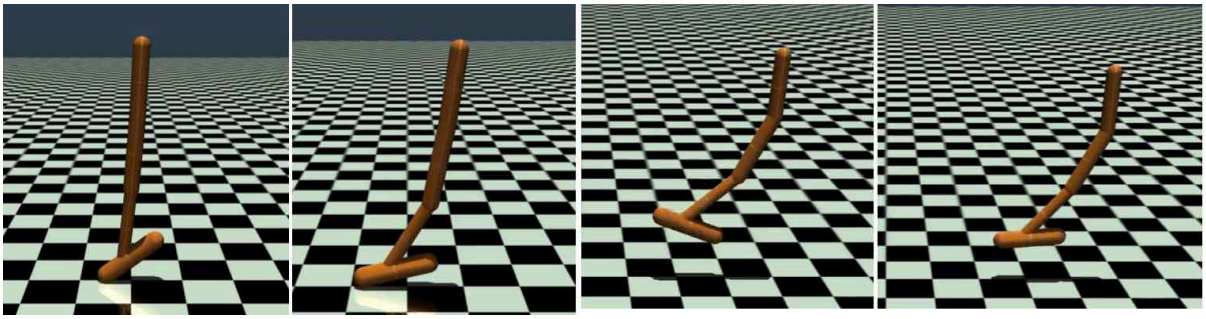

Figure 7: Simulation images after training hopper using TRPO for 591 epochs

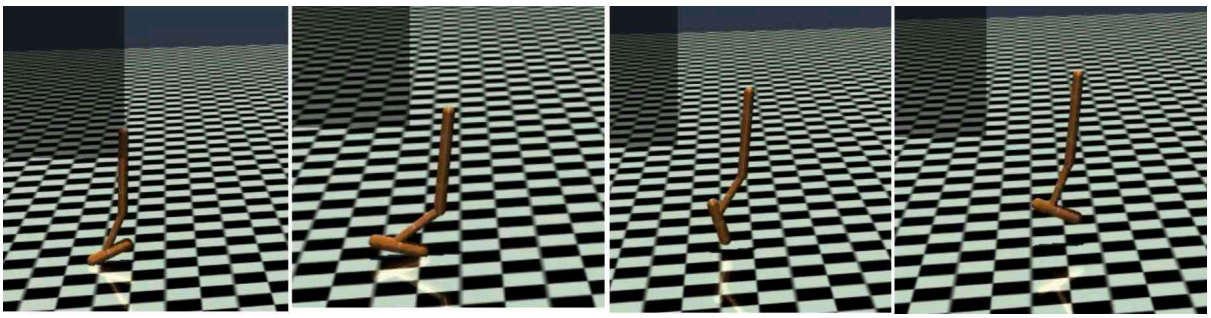

Figure 8: Simulation images after training hopper using DDPG for 216 epochs

It was observed that both models trained the simulation to hop successfully and move forward. The model trained using DDPG was able to reach larger distance but was slower to move forward as compared model trained using TRPO. Also, training done using the DDPG algorithm was observed to be slower as it only reached 216 epochs in 10 hours, while the TRPO algorithm covered 591 epochs.

4.2 Humanoid-Rock-Climbing

The aim of this set of experiments was to train a humanoid simulation to climb a wall that simulated the rock-climbing. For this experiment, we created an environment with a wall that had various types of rock-climbing *holds*. [37] describes different hand configurations for rock-climbing holds. Fig. 9 shows an instance of the environment. MuJoCo does not support creating objects of arbitrary shapes. However, it does allow us to include *meshes* in our environment. The mesh files for rock-climbing holds were found from websites like [38]. Usually, such files are made for 3-D printing and required some pre-processing using the *MeshLab* tool to make them compatible with the MuJoCo engine.

Figure 9: MuJoCo simulation of a wall with rock-climbing holds

We realized early on that training the simulation to grasp the holds would be too complex using the resources we had. So, we tried to simulate the grasping using various types of joint constraints in the MuJoCo physics engine, namely the *distance*, *weld* and *equality* constraints. Out of these, only the *weld* joint was successful in sticking the

humanoid to the wall. However, creating dynamic *weld* joints was not possible using the MuJoCo library. So, we decided to train the humanoid simulation using the constraint that if contact forces between the wall and the humanoid agent's *hands* or *legs* are greater than a specified threshold, we set the weld joint constraint to enabled, else the constraint was disabled. The design for the humanoid agent was taken from OpenAI's *Humanoid-v2* [39] gym environment. The training was done on the GCE instance as described in Section 3, using TRPO for 12 hours and using DDPG for 42 hours. The following reward function was used:

$$reward = v_{up} - 10^{-1}\left|\left|u\right|\right|^2 - 10^{-5}\left|\left|f_{imp}\right|\right|^2 + 5.0 - p_{glue} - p_{dist}$$

Here,

v_{up} = upward velocity

u = vector of joint torques

f_{imp} = impact forces

p_{glue} = glue penalty (5.0 if number of weld joints less than 2 else 0)

p_{dist} = penalty based on distance from rock wall

In the above equation, the addition of a constant offset value of 5.0 is to encourage longer episodes. Otherwise, the training will lead to a policy that ends episodes as quickly as possible. The torque vector penalty is to discourage big changes and impact force penalty is used to make the simulation more realistic. The glue penalty and distance penalty were added to discourage agent from falling down. The agent was trained for using both TRPO and DDPG algorithms. Fig. 10 and Fig. 11 shows the simulation images for an episode for both the trained agents respectively.

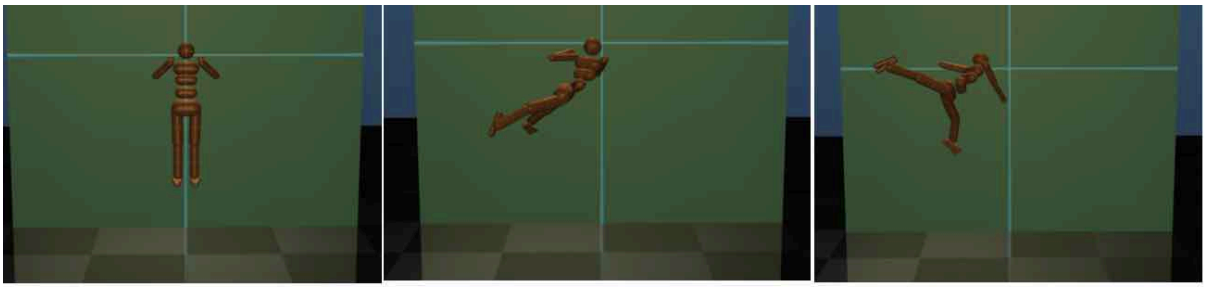

Figure 10: Simulation images after training humanoid-RockClimb using TRPO for 1M time-steps

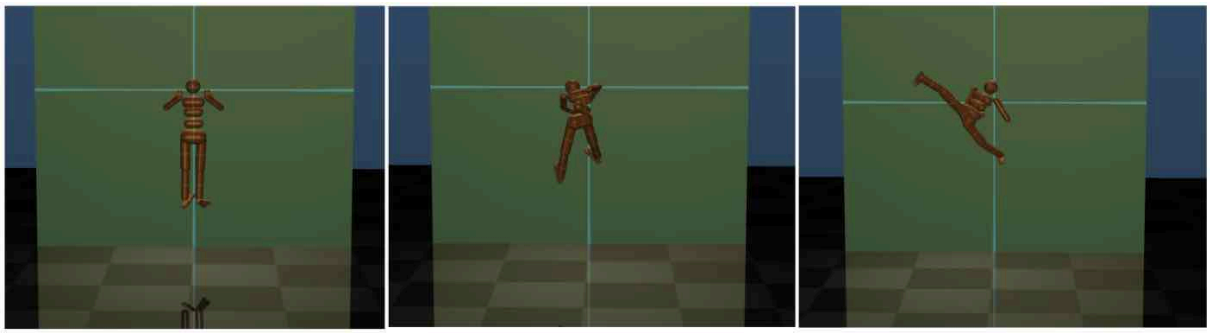

Figure 11: Simulation images after training humanoid-RockClimb using DDPG for 1M time-steps

As can be seen, the trained model only learned to stick to the wall and move legs upwards. It might be possible that training for longer duration could have led to better results. We also tried other configurations by increasing the gear values in the actuator inside the MJCF model and by modifying the reward function but were unable to train the agent to climb. Based on the negative results from this experiments, we decided to take a step back and decided to replicate DeepMind's result for humanoid walking.

4.3. Humanoid Walking

The aim of this experiment was to train a humanoid simulation to walk using the *Humanoid-v2* [39] gym environment. Two experiments were conducted using the DDPG and TRPO algorithms respectively. The model training with TPRO was done for 1 million time-steps. While for DDPG, the training was done for 0.5 million time-steps. Both the experiments were done on the GCE instance described in Section 3. The following reward function was used:

$$reward = v_{fwd} - 10^{-1}\left|\left|u\right|\right|^2 - 10^{-5}\left|\left|f_{imp}\right|\right|^2 + 5.0$$

Here,

v_{fwd} = forward velocity

u = vector of joint torques

f_{imp} = impact forces

In the above equation, the addition of a constant offset value of 5.0 is to encourage longer episodes. Otherwise, the model will lead to a policy that ends episodes as quickly as possible. The torque vector penalty is to discourage big changes and impact force penalty are used to make the simulation more realistic. These values are the defaults for OpenAI gym Humanoid-v2 Environment. The episodes were terminated when the center of mass of the body falls below 1.0m or above 2.0m. The model training for TRPO took a little more than 12 hours. Over the whole course of that training, data from 200,000 episodes were collected. On the other hand, training using DDPG took 32 hours and ran 50,000 episodes in total. Fig. 12 and Fig. 13 shows the initial and final images for the model after training with DDPG and TRPO respectively.

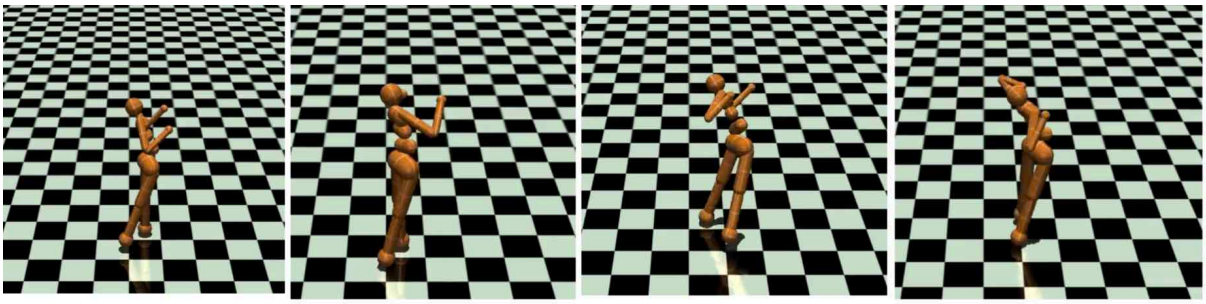

Figure 12: Simulation images after training human-walk using TRPO for 1M time-steps

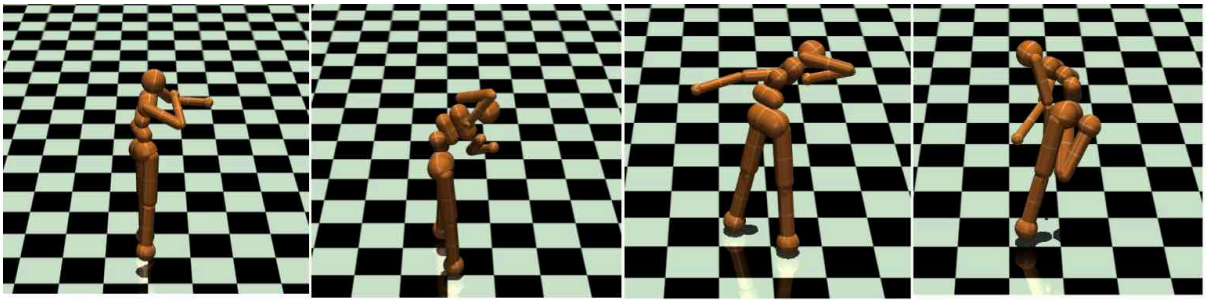

Figure 13: Simulation images after training human-walk using DDPG for 0.5M time-steps

As can be seen, the trained model was not able to learn to walk with either of these algorithms and kept falling down. This could be explained by the fact that the humanoid model has a huge observation and action space. There are 47 state dimensions in the humanoid model and each joint has 17 actuated degrees of freedom. Exploring such a large state space will require a lot more training and computing resources. Furthermore, training using DDPG was observed to be much slower as compared to TRPO.

4.4. Ant Walking

The aim of this experiment was to make a quadrupedal simulation to walk using the *Ant-v2* [40] gym environment. This simulation looks visually similar to an ant. Two experiments were conducted using DDPG and TRPO algorithms respectively. Both the models were trained for 1 million time-steps on the GCE instance as described before. The training took a little less than 10 hours using the TRPO algorithm. Using the DDPG algorithm, the training time was a little over 31 hours. Following reward function was used in both the experiments:

$$reward = v_{fwd} - 0.5 * ||u||^2 - 0.5 * 10^{-5} ||f_{imp}||^2 + 1.0$$

Here,

v_{fwd} = forward velocity

u = vector of joint torques

f_{imp} = impact forces

The values for impact costs and survival reward offset were obtained from OpenAI gym Ant-v2 environment. The episodes were terminated when the center of mass of the body fell below 0.2m or above 1.0m. Fig. 14 and Fig. 15 shows the initial and final images of the model after training with DDPG and TRPO respectively.

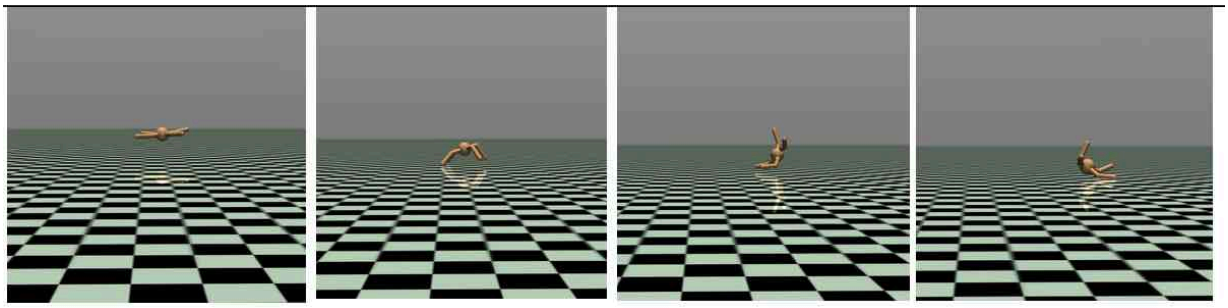

Figure 14: Simulation images after training ant-walk using DDPG for 1M time-steps

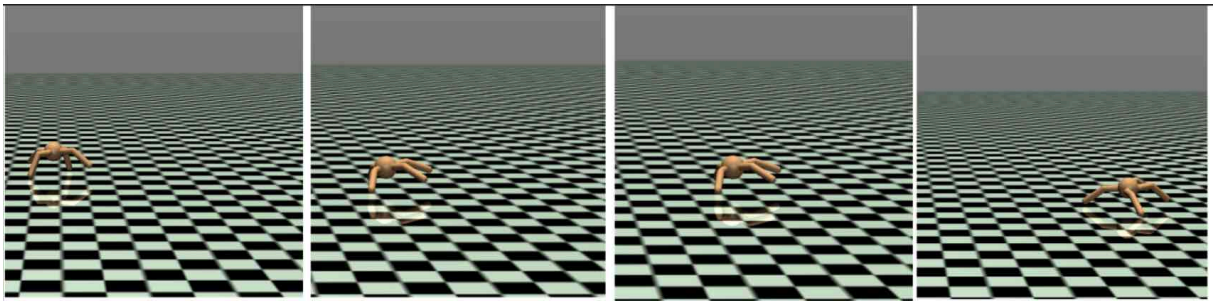

Figure 15: Simulation images after training ant-walk using TRPO for 1M time-steps

As can been seen, TRPO performs much better than DDPG for training the model. In fact, when the training was terminated, the DDPG model only learned to make small jumps, instead of walking, and the agent kept flipping on its back. This can be explained by the fact that since DDPG is an off-policy algorithm, it takes sample data from greedy exploration while exploring the possible solution state space. Since jumping gives a quick reward, it contains much more data from those actions. TRPO, on the other hand, explores within the specified *trust region* only. Thus, training with TRPO is much more stable than DDPG. We also observed that, on an average, training with TRPO was three times faster than DDPG. Even after just 0.5 million time-steps of TRPO training, the model was behaving fairly well. Based on the success of this experiment over the previous ones, we realized that training a humanoid agent is a too complex task due to its much larger state space. Thus, we decided to run further experiments on the ant environment only.

4.5. Ant Steps climbing

The aim of this experiment was to teach the ant simulation to climb a set of steps. Two experiments were conducted with step sizes 0.5cm, and 0.25cm respectively. The episodes were terminated when the simulation had reached the top or when the torso of the body (sphere) got in contact with anything. The reward function was modified to consider upward velocity v_{up} as well.

$$reward = v_{up} + v_{fwd} - 0.5 * ||u||^2 - 0.5 * 10^{-5} ||f_{imp}||^2 + 1.0$$

Here,

v_{up} = upward velocity

v_{fwd} = forward velocity

u = vector of joint torques

f_{imp} = impact forces

The values for impact costs and survival rewards were obtained from OpenAI gym Ant-v2 environment. The model training with TRPO was done for 1 million time-steps, which took a little more than 10 hours. The DDPG model was trained for 0.5 million time-steps and took a little more than 25 hours. Fig. 16-19 shows the initial and final images after training.

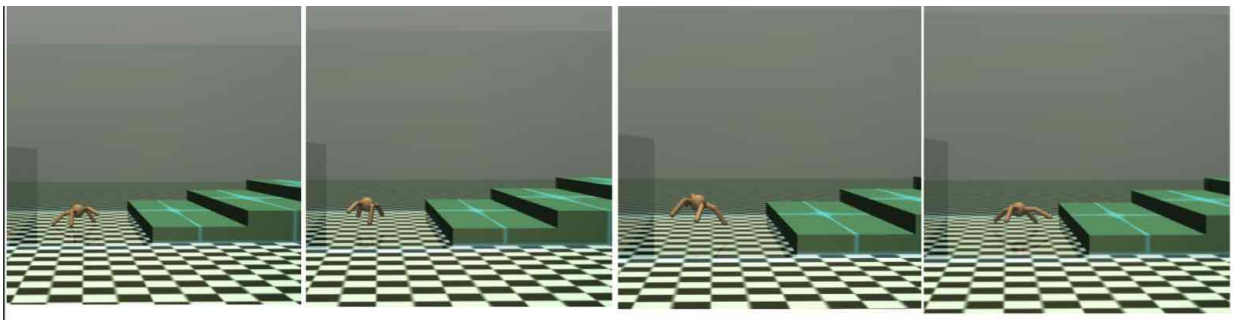

Figure 16: Simulation images after training ant-steps using TRPO for 1M, time-steps with step size 0.5cm

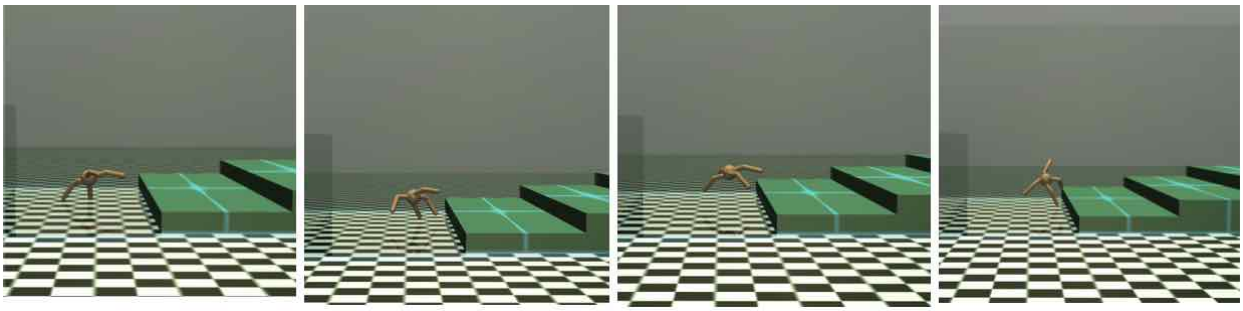

Figure 17: Simulation images after training ant-steps using DDPG for 0.5M time-steps with step size 0.5cm

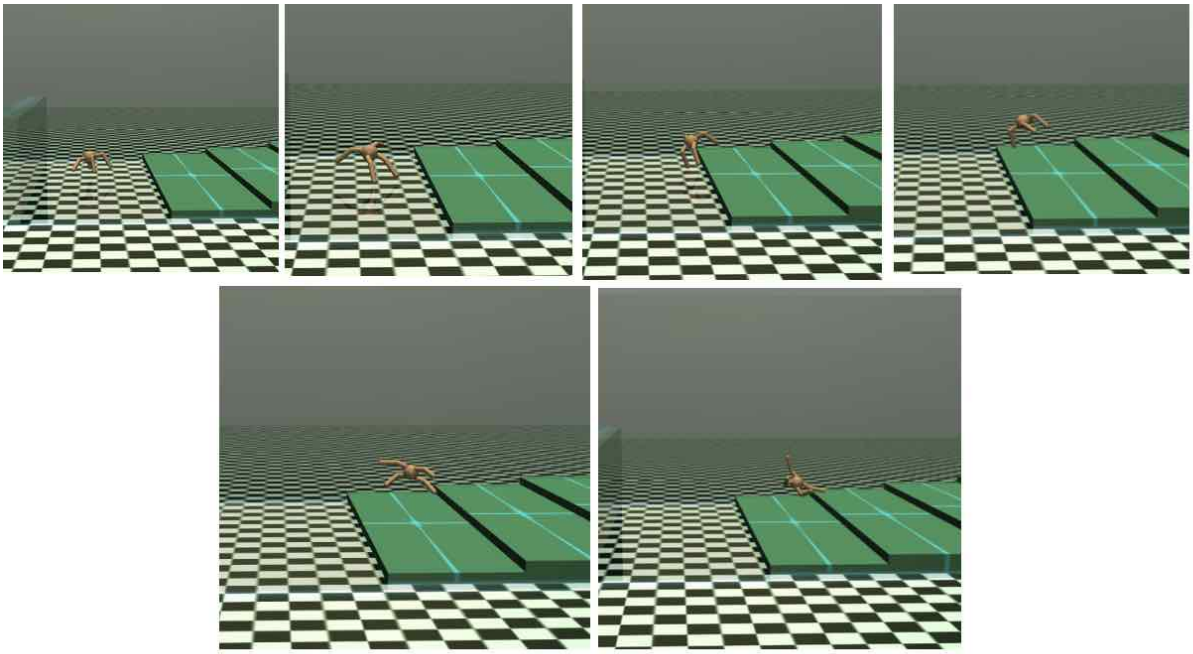

Figure 18: Simulation images after training ant-steps using DDPG for 0.5M time-steps with step size 0.25 cm

As can be seen, both the models failed in both the experiments to climb even one step. The models only learned to make the agent jump, and not walk or climb. Also, the model trained using DDPG was observed to be much less stable and more prone to flipping on its back. To solve this problem, we decided to run another experiment by changing the terminating condition so that episodes will terminate when all four legs are in the air. This was done to prevent the jumping behavior. This experiment was done for step size 0.25cm only and the model was trained using TRPO for 0.5 million time-steps. Fig. 20 shows the result of this experiment. In this experiment, model learned to stop jumping, but still did not learn to climb the step.

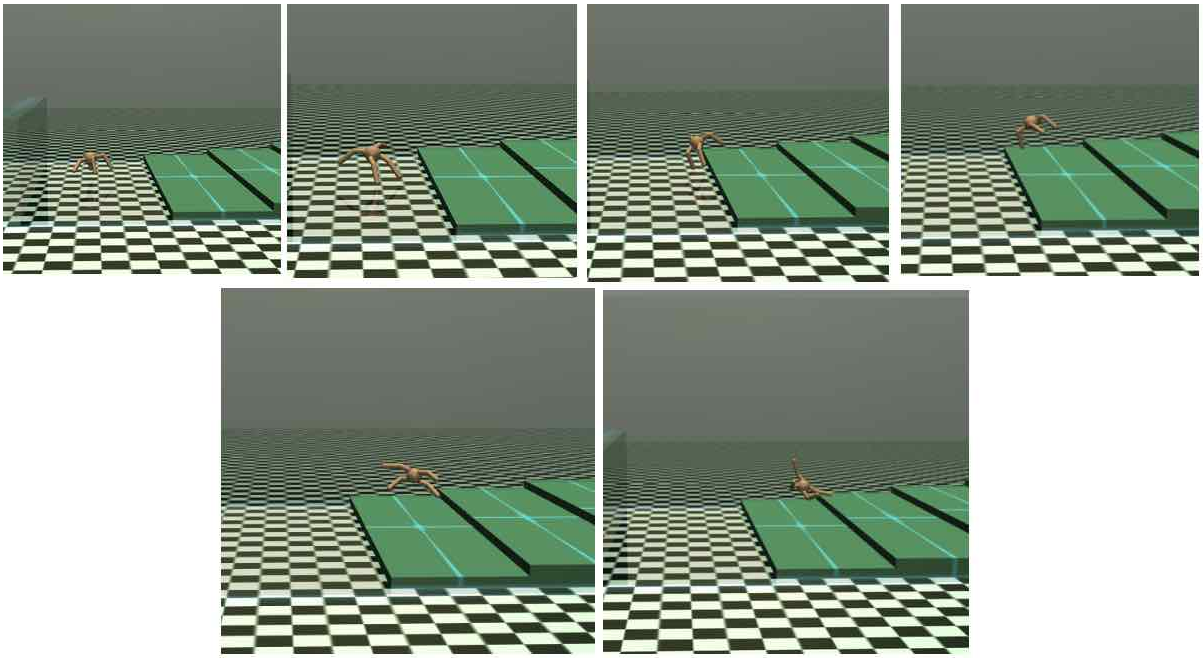

Figure 19: Simulation images after training ant-steps using TRPO for 1M time-steps with step size 0.25 cm

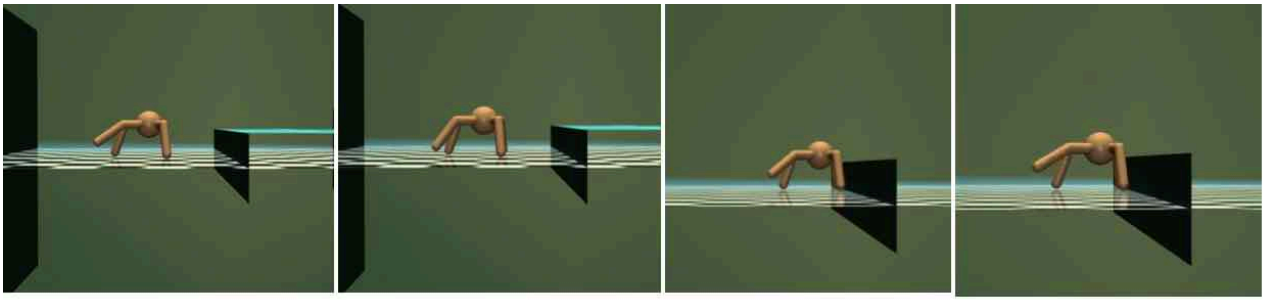

Figure 20: Simulation images after training ant-steps using TRPO for 0.5M time-steps with step size 0.25 cm with modified terminating condition

Based on these observations, since it did not achieve the determined goal for any of these experiments, we concluded that this task may be too complex in this environment. It was realized that we need to make changes to the environment itself so that it is feasible to train the model. Also, it takes a long time to run these experiments and we had limited computing resources in terms of Google GCE. So, we decided to drop the DDPG experiments and focus on only TRPO as the latter was observed to be much more stable and faster.

4.6. Ant Slope Climbing

Based on our previous results, we decided to modify the environment to have a slope instead of steps. Three experiments were conducted with slopes equal to 9°, 18°, 27° respectively. Once again, the final goal for all the experiments was to reach the top. The episodes were terminated when the simulation had reached the top or when the torso of the body got in contact with anything. Also, the friction coefficient value was set to 1.5. This value was chosen by conducting a set of early experiments. The reward function used was the same as in Experiment 4.5. The model was trained for 10 million time-steps for each experiment. Each experiment took a little less than 78 hours. Fig. 21-23 shows the initial and final images for the three experiments.

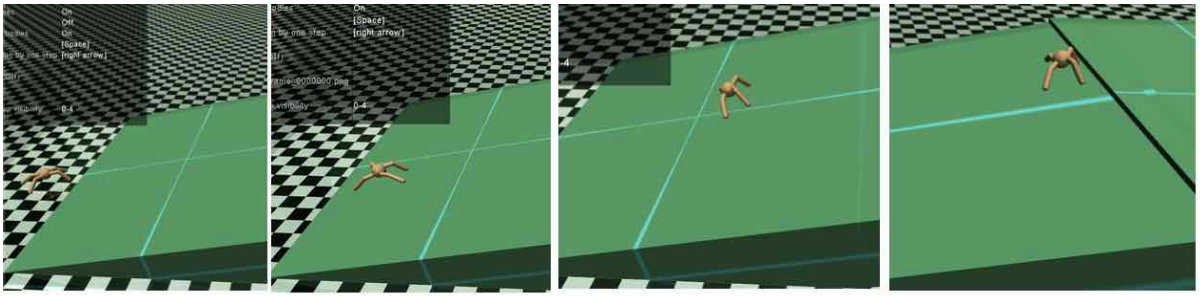

Figure 21: Simulation images after training ant-slope using TRPO for 10M time-steps for slope equal to 9°

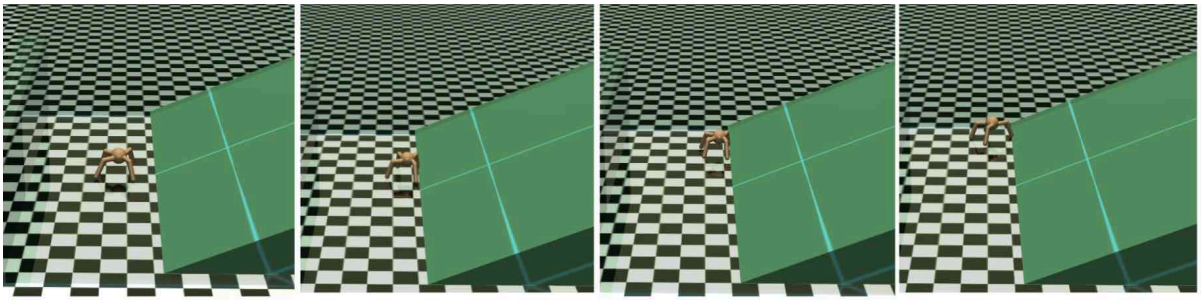

Figure 22: Simulation images after training ant-slope using TRPO for 10M time-steps for slope equal to 18°

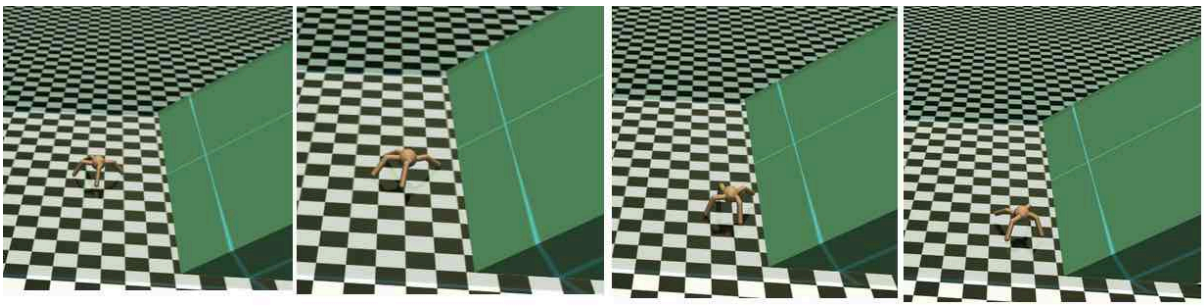

Figure 23: Simulation images after training ant-slope using TRPO for 10M time-steps for slope equal to 27°

As can be seen, the model trained for slope equal to 9° works well and is able to climb the slope successfully. However, it fails for environments with greater slopes and the simulation just gets as close to the slope and moves back and forth near it. It was observed during the training that for higher slopes the agent was not able to stick its legs on the slope and kept toppling down. Based on these observations, we realized that the agent needs some way to stick its legs to the slope. Increasing the friction coefficient could have helped, but it was not considered because having such a large value is not reasonable according to physics.

Additional experiments which used soft joint constraints like *weld* joint and *distance* joint in MuJoCo physics engine were conducted, but did not give the desired results.

4.7. Ant Grooved Slope Climbing

For this experiment, we modified the previous environments to have grooves on the slope. These grooves facilitated the required friction and can be used by the agent to stick its legs on the slope. Here, experiments were conducted with slopes equal to 18° and 27° respectively. The terminating conditions and other parameters were kept the same as in the previous experiment. The model was trained for 5 million time-steps for each experiment, and each experiment took a little over 22 hours. Fig. 24 and Fig. 25 shows the initial and final images for the two experiments.

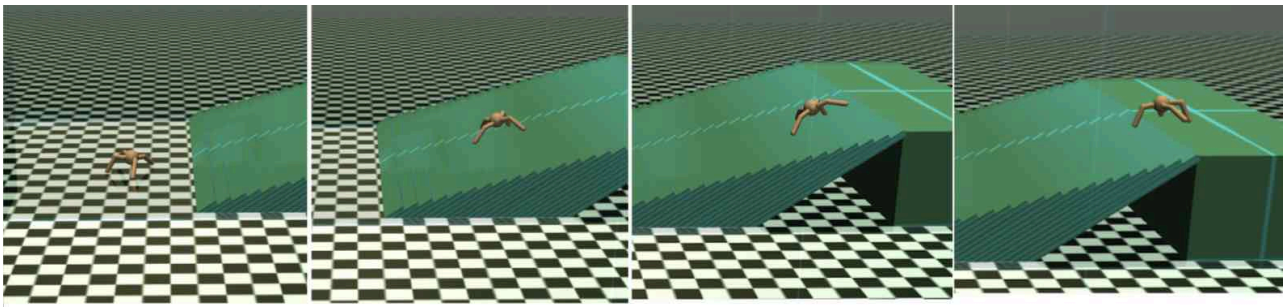

Figure 24: Simulation images after training ant-grooved-slope using TRPO for 5M time-steps for slope equal to 18°

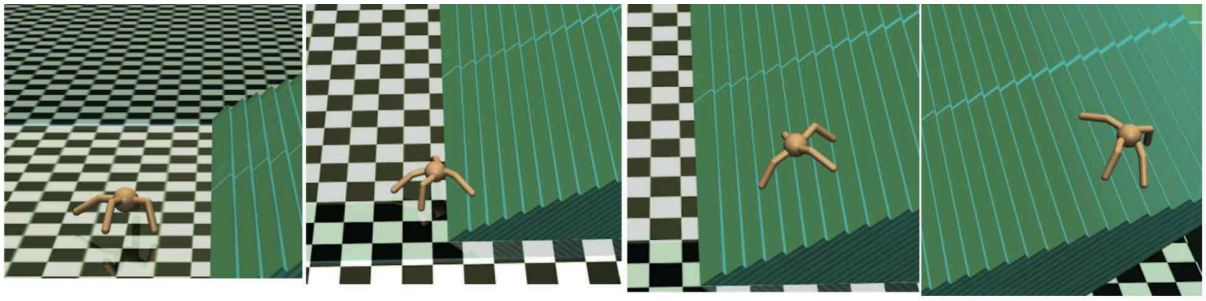

Figure 25: Simulation images after training ant-grooved-slope using TRPO for 5M time-steps for slope equal to 27°

As can be seen, the model trained with TRPO works well for 18°. However, the models trained for greater slope environments are only able to climb a few steps. It is possible that training the model for longer duration might work.

One interesting observation was made when we ran the agent trained for 18° slope environment on the environment with 27° slope. It was observed that the trained model was able to successfully climb the harder environment as well. This result is shown in Fig. 26.

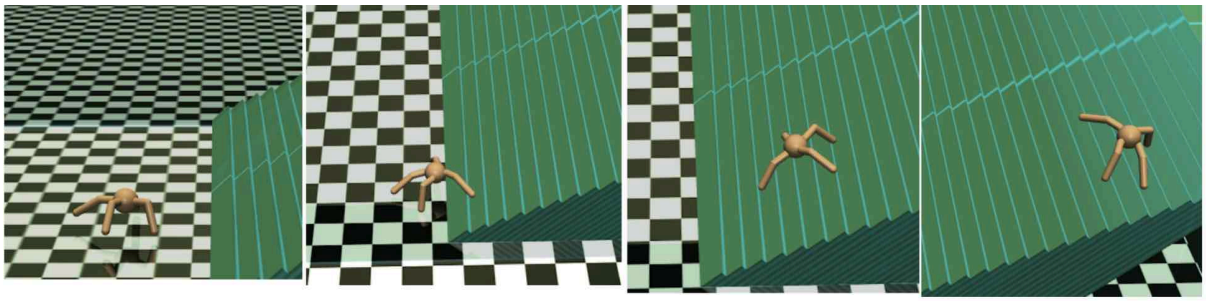

Figure 26: Simulation images after training ant-grooved-slope-18° using TRPO for 5M time-steps and running on environment with slope equal to 27°

Based on this observation, we realized that since the tasks are quite similar, the policy learned for an easy task may be used as the base to learn a more difficult task. This observation led us to explore transfer learning and we decided to design the next experiment to facilitate such transfer of knowledge.

.

4.8. Ant Grooved Multiple Slopes Climbing

For this experiment, we built a large environment with increasingly difficult slopes after every few intervals. The intuition here is that the main task is to climb as high as possible, while sub-tasks would be to climb each slope. In particular, there were five slopes 9°, 18°, 27°, 36°, and 45° in the environment. Each consecutive slope had a flat surface of 6m length between them. Two experiments were conducted using slightly different reward functions. To consider random initializations, we conducted two trials for both these experiment. Each trial was run for 10 million iterations, with initial weights as random. The trial which performed best for each set of experiment was considered. Also, the max episode length was set to 5000 for both trials. The reward functions and other parameters were kept the same as in the previous experiment.

For the first set of experiments, both forward and upward velocity were considered in the reward function. For the second set of experiments, only the forward velocity was considered in the reward function. Figure 16 and Figure 17 shows the initial and final images of an episode after training for both these experiments.

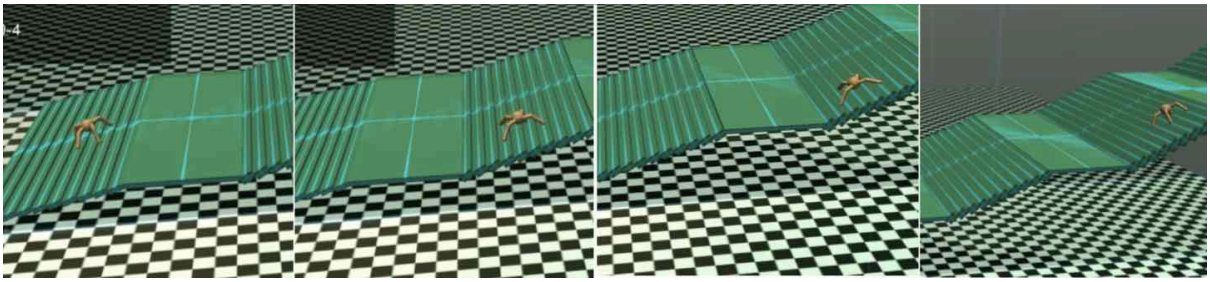

Figure 27: Simulation images after training ant-grooved-multiple-slopes using TRPO for 10M time-steps considering both upward and forward velocity

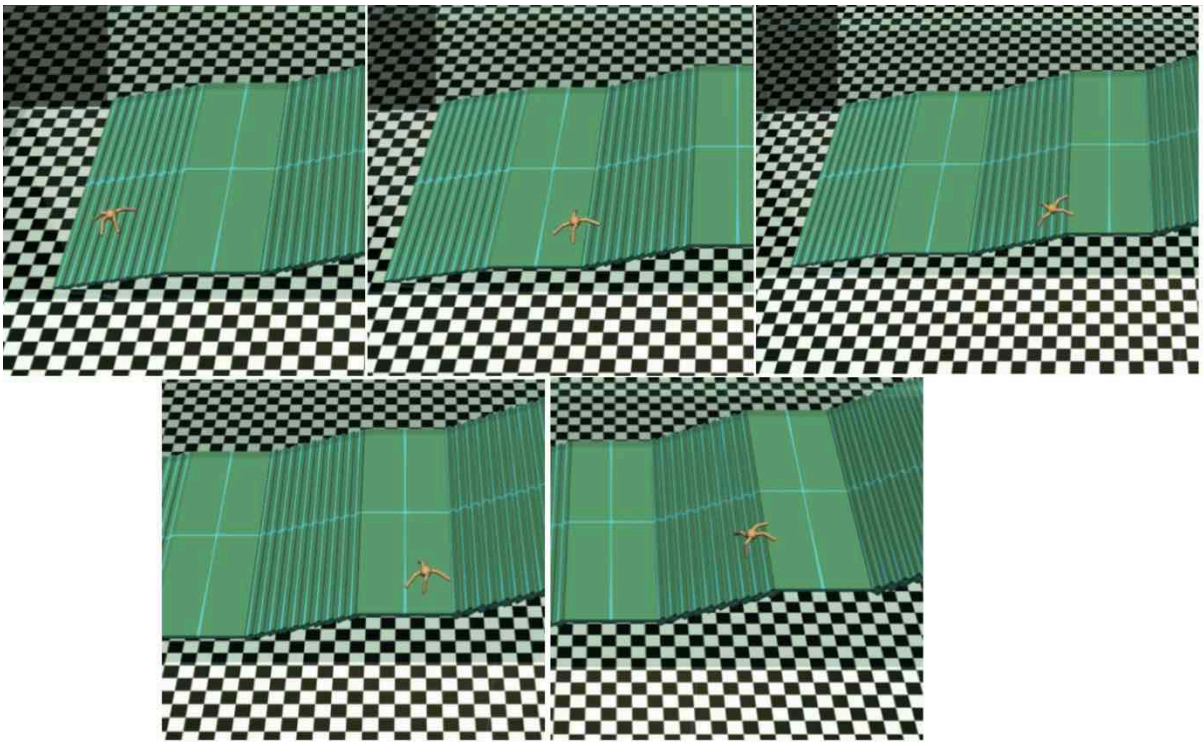

Figure 28: Simulation images after training ant-grooved-multiple-slopes using TRPO for 10M time-steps considering only forward velocity

When trained using only forward velocity, the agent was able to climb the slopes till 27°. After reaching the end of 27° slope, it was not able to proceed further. However, when trained using both forward and upward velocity, it was able to climb till 36°.

Also, instead of saving just the final model, the intermediate models were saved every 100 iterations, along with the max height and the max distance achieved per episode. Figure 18 shows the graphs for the rate of training in both experiments as episode count versus the maximum heights and distances achieved per episode. As can be seen, considering both the forward velocity as well as upward velocity results in a faster training of the model. Max distance traveled was 41.7m was for the second experiment. This distance means that simulation was able to reach the start of the 45° slope. While for the first experiment, a max distance of 34.5m was observed. This means that simulation was able to reach the ending point of the 27° slope.

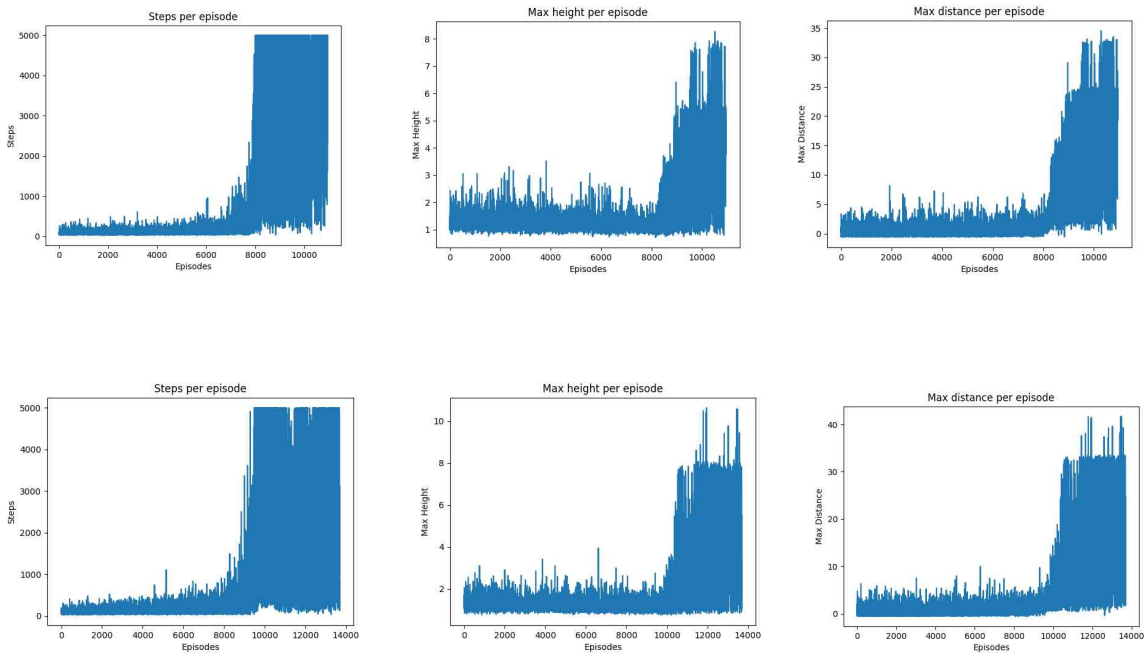

Figure 29: Graphs showing the rate of training as episode count versus the steps, heights and distances per episode

4.9. Ant Grooved Multiple Slopes Climbing with Potholes

For this experiment, we wanted to test the agent trained in Experiment 4.8 on learning new task. So, we introduce potholes (depressions) on the slopes inside the environment and trained the model to avoid the potholes. We used a combination of transfer learning and residual learning techniques to train the agent with additional objective that it should learn to avoid potholes. Using transfer learning, we used the previously trained model as base, and only made the weights of output layer as trainable. In other words, we freeze the initial layers which correspond to the pre-trained model so that the weights received from the pre-trained model are not modified. This way, we preserve the intermediate representation of the environment learned previously. Similar to residual learning technique, we also provide direct observations from the environment to the trainable layers. Thus, the trainable part of the network gets a combined input of both the learned representation of pre-trained model, and also the direct observations from the environment.

For this experiment, the reward function and all other parameters were kept the same as in the previous experiment. The model was trained for 10 million time-steps using TRPO algorithm. Fig. 30 shows the simulation images for an episode after training.

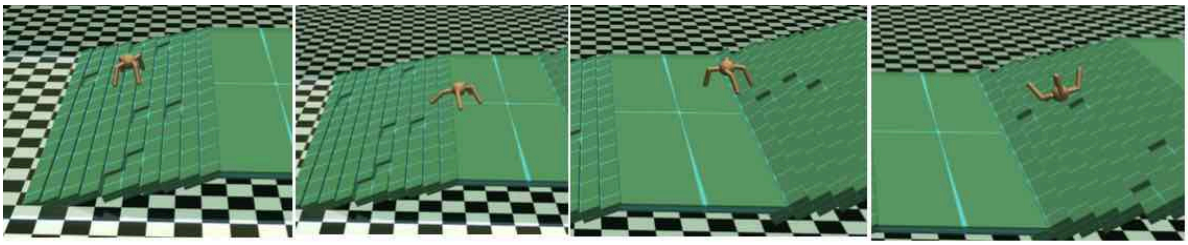

Figure 30: Simulation images for model trained using TPRO for Experiment 4.9

As can be seen, the trained model was not able to learn to move further than the second slope (18°). We expected that using knowledge from previously trained model, we might achieve good results. However, model was not able to learn from the knowledge.

CHAPTER 5
Conclusion and Future Work

The goal of this project was to evaluate the potential of Reinforcement Learning to train a robot simulation to climb slope based environments. For this purpose, we identified and explored two state-of-the-art algorithms for Deep Reinforcement Learning, DDPG and TRPO. We evaluated the performance and effectiveness of these algorithms to train the agents for this task. We observed that, on an average, training with TRPO was three times faster than DDPG, and also much more stable for the locomotion control tasks that we experimented.

We conducted multiple experiments to train an agent to climb the steep slope using TRPO algorithm. We found that the it takes a lot of time and resources to properly train the agent. Although that agent was able to learn climbing actions for slopes till 18°, it failed to train properly for steeper slopes. We used insights from transfer learning to improve our results. We observed that if we design our environment to facilitate transfer learning, the agents were trained much more effectively and efficiently. Our final trained agent was able to climb slopes till 36°. This proved that transfer learning applied to reinforcement learning acts fruitful in effective results.

One way to improve this project would be to combine the results for our project and the potential of Convolutional Neural Networks (CNNs) for image recognition. CNNs are already used in DQNs to train agents to play Atari games. The agents trained in this project take observations in terms of force vectors applied on its body. If we can somehow supply visual simulation information from MuJoCo engine, we can use a CNN to generate a visual representation of the environment. This has the potential to train better models.

References

[1] AdventureStats - by Explorersweb. (n.d.). Retrieved December 1, 2018, from http://www.adventurestats.com/tables/k2routes.shtml

[2] How To Remove Dead Bodies From Mount Everest? (n.d.). Retrieved December 1, 2018, from https://explorersweb.com/2018/05/11/how-to-remove-dead-bodies-from-mount-everest/

[3] Mars New Home 'a Large Sandbox' – NASA's InSight Mars Lander. (n.d.). Retrieved December 1, 2018, from https://mars.nasa.gov/news/8395/mars-new-home-a-large-sandbox/?site=insight

[4] Heess, N., Sriram, S., Lemmon, J., Merel, J., Wayne, G., Tassa & Silver, D. (2017). Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*.

[5] Emergence of Locomotion Behaviours in Rich Environments. (2017, July 14). Retrieved December 1, 2018, from https://youtu.be/hx_bgoTF7bs

[6] Quillen, D., Jang, E., Nachum, O., Finn, C., Ibarz, J., & Levine, S. (2018). Deep Reinforcement Learning for Vision-Based Robotic Grasping: A Simulated Comparative Evaluation of Off-Policy Methods. *arXiv preprint arXiv:1802.10264*.

[7] Ju, E., Won, J., Lee, J., Choi, B., Noh, J., & Choi, M. G. (2013). Data-driven control of flapping flight. *ACM Transactions on Graphics (TOG)*, *32*(5), 151.

[8] Merel, J., Tassa, Y., Srinivasan, S., Lemmon, J., Wang, Z., Wayne, G., & Heess, N. (2017). Learning human behaviors from motion capture by adversarial imitation. *arXiv preprint arXiv:1707.02201*.

[9] Nagakubo, A., & Hirose, S. (1994, May). Walking and running of the quadruped wall-climbing robot. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on* (pp. 1005-1012). IEEE.

[10] Sitti, M., & Fearing, R. S. (2003, September). Synthetic gecko foot-hair micro/nano-structures for future wall-climbing robots. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on* (Vol. 1, pp. 1164-1170). IEEE.

[11] Yano, T., Suwa, T., Murakami, M., & Yamamoto, T. (1997, September). Development of a semi self-contained wall climbing robot with scanning type suction cups. In *Intelligent Robots and Systems, 1997. IROS'97., Proceedings of the 1997 IEEE/RSJ International Conference on* (Vol. 2, pp. 900-905). IEEE.

[12] Libeau, B., Micaelli, A., & Sigaud, O. (2009, May). Transfer of knowledge for a climbing virtual human: A reinforcement learning approach. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on* (pp. 2119-2124). IEEE.

[13] Carnegie Mellon University - CMU Graphics Lab - motion capture library. (n.d.). Retrieved December 2, 2018, from http://mocap.cs.cmu.edu/

[14] Torrey, L., & Shavlik, J. (2010). Transfer learning. In Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques (pp. 242-264). IGI Global.

[15] Hoo-Chang, S., Roth, H. R., Gao, M., Lu, L., Xu, Z., Nogues, I., ... & Summers, R. M. (2016). Deep convolutional neural networks for computer-aided detection: CNN architectures, dataset characteristics and transfer learning. *IEEE transactions on medical imaging*, *35*(5), 1285.

[16] Weiss, K., Khoshgoftaar, T. M., & Wang, D. (2016). A survey of transfer learning. *Journal of Big Data*, *3*(1), 9.

[17] Taylor, M. E., & Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, *10*(Jul), 1633-1685.

[18] Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015, June). Trust region policy optimization. In International Conference on Machine Learning (pp. 1889-1897).

[19] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

[20] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

[21] Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279-292.

[22] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[23] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014, June). Deterministic policy gradient algorithms. In ICML.

[24] Kakade, S. M. (2002). A natural policy gradient. In Advances in neural information processing systems (pp. 1531-1538).

[25] Sun, W., & Yuan, Y. X. (2006). Optimization theory and methods: nonlinear programming (Vol. 1). Springer Science & Business Media.

[26] Hestenes, M. R., & Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems (Vol. 49, No. 1). Washington, DC: NBS.

[27] Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009, June). Imagenet: A large-scale hierarchical image database. In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on (pp. 248-255). Ieee.

[28] Sharif Razavian, A., Azizpour, H., Sullivan, J., & Carlsson, S. (2014). CNN features off-the-shelf: an astounding baseline for recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition workshops (pp. 806-813).

[29] Taylor, M. E., & Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. Journal of Machine Learning Research, 10(Jul), 1633-1685.

[30] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. arXiv preprint arXiv:1606.01540.

[31] Openai. (n.d.). Openai/gym. Retrieved December 1, 2018, from https://github.com/openai/gym

[32] Openai. (n.d.). Openai/baselines. Retrieved December 1, 2018, from https://github.com/openai/baselines

[33] Openai. (n.d.). Openai/mujoco-py. Retrieved December 2, 2018, from https://github.com/openai/mujoco-py/

[34] Todorov, E., Erez, T., & Tassa, Y. (2012, October). Mujoco: A physics engine for model-based control. In Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on (pp. 5026-5033). IEEE.

[35] MuJoCo. (n.d.). Retrieved December 1, 2018, from https://www.roboti.us/index.html

[36] Hopper. (n.d.). Retrieved December 1, 2018, from https://gym.openai.com/envs/Hopper-v2

[37] Watts, P. B. (2004). Physiology of difficult rock climbing. European journal of applied physiology, 91(4), 361-372.

[38] Thingiverse.com. (n.d.). Rock Wall hold 1.0 by Jeremy007007. Retrieved December 1, 2018, from https://www.thingiverse.com/thing:34331/#collections

[39] OpenAI. (n.d.). Humanoid-v2. Retrieved December 1, 2018, from https://gym.openai.com/envs/Humanoid-v2/

[40] OpenAI. (n.d.). Ant-v2. Retrieved December 1, 2018, from https://gym.openai.com/envs/Ant-v2/