ROCK-WALL CLIMBING AI

CS 297 Report

Presented to

Dr. Chris Pollett

Department of Computer Science

San Jose State University

In Partial Fulfilment

Of the Requirement for the Class

CS297

By

Ujjawal Garg

May 2018

TABLE OF CONTENTS

Introduction

Reinforcement Learning is a field of Artificial Intelligence that has gained a lot of attention in recent years. In July 2017, Silver et.al published a paper [1] and video [2] showing a simulated humanoid body trained to navigate through a set of challenging terrains, using reinforcement learning. This project will use a similar approach to train a simulated humanoid body to climb a rock wall. The process of rock climbing is more complex than walking because the viable solution space is much more constrained, but the search space is just as large.

Unlike supervised learning, where we use some training data as input, in reinforcement learning we only have access to an environment where each actor or agent can perform a set of specific actions. This environment is considered as a MDP (Markov Decision Process), where we do not know the states we can visit and also the transition function from each state is not known. Each action performed has a reward that depends on the new state and previous state. The goal of reinforcement learning is to learn this reward function, which is generally referred to as policy π. Using this reward function, an agent can choose the actions which leads to best rewards.

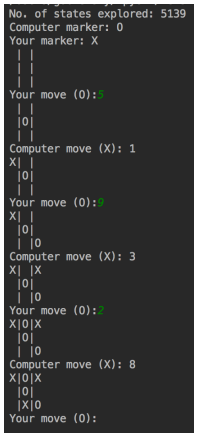
The final goal of this semester is to have learned the fundamentals of reinforcement learning and do research on existing literature. Through this, I will have developed a baseline architecture of my final project, which I can then tweak and train to train the simulation model.

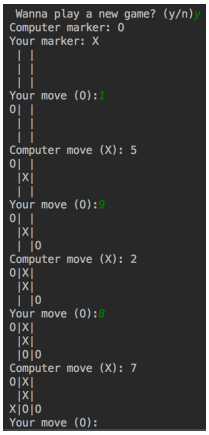
I. Deliverable 1: Q-Learning implementation for Tic-Tac-Toe

The objective of this deliverable was to develop a python module for an agent and a bot that uses the Q-Learning technique to train the agent on how to play Tic-Tac-Toe game. Q-Learning [3] is an off-policy learning algorithm, where we model the value function as a Bellman equation, where the reward for performing 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 solved by creating a table of Q-values. 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 reward of value of performing action a in state x, γ is the discount factor, and x' is the next state. There are two scripts in the module: The first *Q_Learning_Tic_Tac_Toe.py* uses a simple bot to train the agent, and the output is a model file that contains the Q-values in a table. The second file *Tic-Tac-Toe-Player.py* uses the model generated by first to play against a human player. Value of γ is taken as 0.1. Training the agent for 50000 games created 5000 entries in the Q-table on average. Considering there are only 5812 legal states in Tic-Tac-Toe, the agent performed very well and won 14 out of 15 games on average.





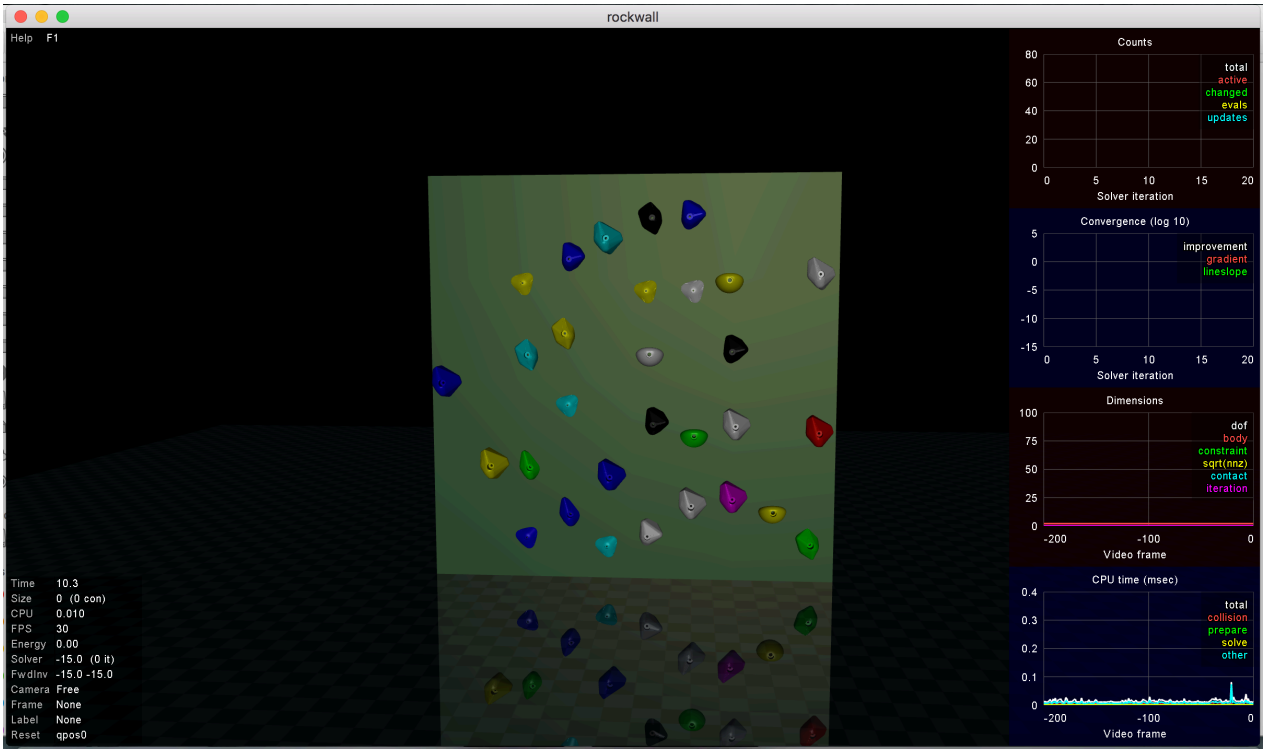
II. Deliverable 2: MuJoCo Rock Wall Generator

The goal of this deliverable was to develop a python script to generate an environment in which our humanoid simulation can learn to climb the rock wall. There are several libraries and packages available for this purpose that provides the physics and simulation capabilities. Out of these two packages: MuJoCo [4] and Bullet Physics [5] stand out, mainly due to their compatibility with OpenAI Gym. OpenAI Gym [6] has used MuJoCo since its beginning. However, MuJoCo requires a paid license, although it is free for students. The support for pybullet was added with introduction of Roboschool framework which was supposed to replace Gym, but it has not been updated for almost a year. So, we decided to use MuJoCo with student license. The final outcome of this deliverable was a python script to generate an xml in MJCF format which can be run via the MuJoCo simulator.

The MJCF file contains an XML tree created by nested *body* elements. The generated world contains a single wall that has different types of rock climbing holds placed in a semi-random fashion. The rock wall is specified as a *geom* element of type *box* within the main body. The rock-climbing holds are specified as *geom* element of type *mesh*. The mesh files for the holds were obtained from sites like thingverse.com [7], where people make these files for 3-D printing. To make the pattern semi-random, we use the following method:

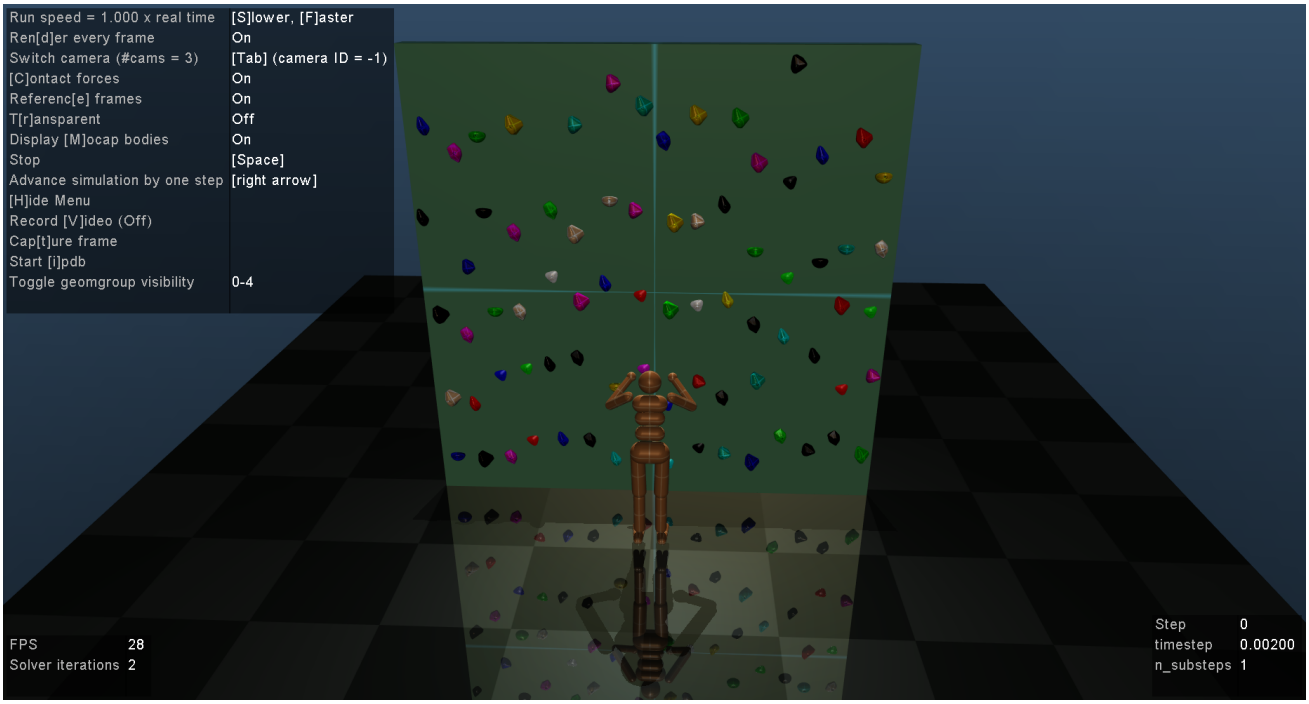
1. Divide the wall into tiles of size 1x1, and traverse the tiles from bottom left to top right.
2. With 50% probability place a hold at a tile, and shift it along x-axis by 6% on average.
3. Set y-position of each hold 0.2 distance above the hold below it.

To generate the xml file, the *etree* python library was used. This xml file could then be supplied to the MuJoCo simulator to view the result. The figure below shows a sample output of the deliverable.



III. Deliverable 3: RockClimbEnv Gym Environment

The objective of this deliverable was to explore the OpenAI gym environment, and to develop a gym environment for generating a random rock wall. This OpenAI gym environment can then be used to train a humanoid on how to climb the rock wall. One of the main benefit of using OpenAI Gym is that it will be much easier for someone else to replicate my work and build upon it. The environment is shown in the picture below.



This gym environment provides a framework where we can choose an action for the Humanoid. This action is in the form of value for 24 joint motors, each in range [-1, 1]. Changing these values enables the movement of humanoid. Actions are drawn randomly from the action space. Based on action performed, and resulting new state agent is given a reward. This reward depends on the z position of the humanoid body.

III. Deliverable 4: DDPG Algorithm

The goal of this deliverable was to choose a policy gradient algorithm and implement it. Algorithms like Q-Learning and DQN (Deep Q-Networks) can work only on discrete and low-dimensional action spaces. For this reason, they are not suitable for continuous control tasks like robot or humanoid movements, etc. Policy gradient methods with deep function approximators have shown remarkable success in continuous control tasks. Unlike value-based methods, here we learn and optimize for policy function directly. For this deliverable, I implemented the DDPG [8] (Deep Deterministic Policy Gradient) policy gradient algorithm.

DDPG is a model-free, off-policy, actor-critic approach based on the DPG (Deterministic Policy Gradient) method. Model-free means that the underlying dynamics of the environment are unknown and are learned by exploring. Off-policy means that actions are chosen from a behaviour policy that is different from the policy being trained. 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. The deterministic part comes from the fact that we training to learn the Deterministic policy. In algorithms like DQN, we can use stochastic policy and learn select the action based on the following eq.:

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

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

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

Also, similar to the €–greedy approach used in DQN, a noise process $\mathcal{N}$ is used to ensure sufficient exploration:

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

The complete algorithm is shown below:

Algorithm 1 DDPG algorithm

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 $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t 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 $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    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:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    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 for
end for

Conclusion

During this semester, I have completed the preliminary steps required towards CS298. I know what needs to be worked on during next semester. I have developed and explored the OpenAI Gym and Baselines libraries, which I will be using to perform the training. I have researched the previous work that has been done in this field. In detail, I have implemented the Q-Learning and DDPG algorithms, and trained a humanoid simulation to take the first step towards walking. Through these, I have gained knowledge and exposure to deep reinforcement learning concepts.

For next semester, I need to determine how to setup the rewards inside the gym environment that I have created. Currently, the reward is based on the z position of the humanoid body. However, a better way could be setup incremental reward such that goal changes from one hold to another as the humanoid starts climbing. In this way, the solution space that needs to be explored can be reduced. Further, I will need to try different configurations for the hyper-parameters while training the simulation. Also, based on the experiments that I did using the baseline algorithms, I might need to use a HPC cluster to do the training. Currently, on my MacBook Pro, it took 3-4 hours for 500 iterations of DDPG algorithm. After 500 iterations, humanoid only learned to take one step. For more complex tasks like rock-climbing, much more training would be required.

References

[1] D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, A. . S. M. Eslami, M. Riedmiller and D. Silver, "Emergence of Locomotion Behaviours in Rich Environments," *arXiv,* 2017.

[2] DeepMind, "Emergence of Locomotion Behaviours in Rich Environments," 2017. [Online]. Available: https://youtu.be/hx_bgoTF7bs.

[3] P. Dayan and C. Watkins, "Q-Learning".

[4] E. Torodov, T. Frez and Y. Tassa, "MuJoCo: A physics engine for model-based control".

[5] "Bullet Physics SDK," [Online]. Available: https://github.com/bulletphysics/bullet3.

[6] OpenAI, "OpenAI Gym," [Online]. Available: https://gym.openai.com/.

[7] "Rock Wall hold 1.0," [Online]. Available: https://www.thingiverse.com/thing:34331/#collections.

[8] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra, "Continuous control with deep reinforcement learning," 2015.