AI Based Running Simulation

Zeyang Bao, Timothy Yong

Rutgers University, New Brunswick

RUTGERS Aresty Research Center for Undergraduates

Abstract

Nowadays, the usage of robotics is becoming more widespread for different applications and environments. There is a need for low-cost robotics to become more useful in larger settings, and a variety of methods can be applied to accomplish this task, the most successful of them being reinforcement learning (RL). The default RL algorithm used by the OpenAI for robotic environments is Proximal Policy Optimization (PPO), which has relatively good model convergence, but easily converges to a local optimum if the entropy coefficient is not parameterized correctly. We had hoped to use a new algorithm named Chained Q-Learning (CQL), which samples the dependency between different joints in order to reduce the action space, in order to quickly first find a semi-global optimum before optimizing with another policy gradient algorithm.

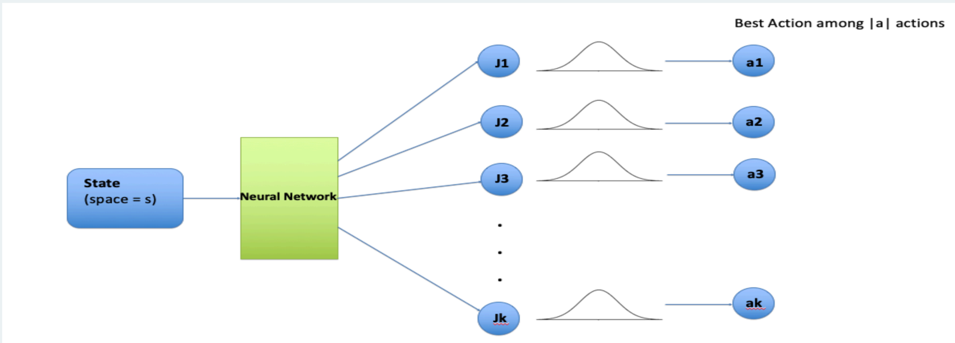
Background/Intro

At first we implement the traditional PPO algorithm, we find that the model converges faster but it is really likely to converge at local optimum. Then we do experiment from the most easy RL algorithm DQN and evolve it to a different form **CQL (Chained Q-Learning)**, which train the model for each joint separately and reduce the running time.

Objectives

#of joints: ***n***

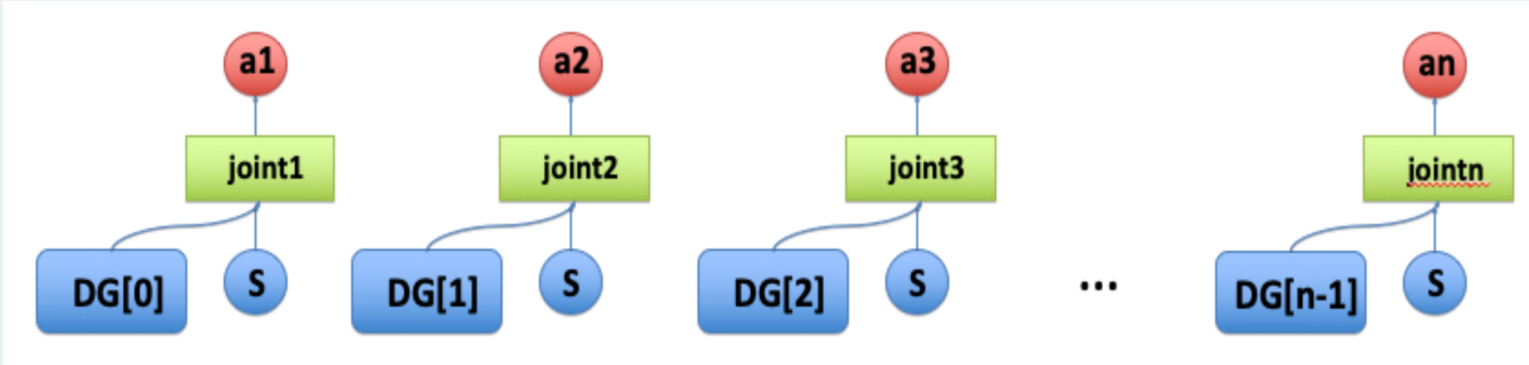
action space: |***a***|



Basically, the action space for this model |a|^n, so many action need to be considered at the same time. Cause longer time to converge.

Methods

We want to reduce the convergence time so we sample the dependency for each joint



DG respresents **Dependence Graph** which is a list:
[[a_d0], [a_d1], … [a_d(n-1)]], Dp[i] represents the list of action that jointi+1 depends.

We get the action by :
actions ~ QValue(joint, dp)
Each input: State and other joints' action by ***dp***
Then the total action space becomes :
|a|*n reduced a lot from the |a|^n

Algorithm 1: Chained Q-Learning

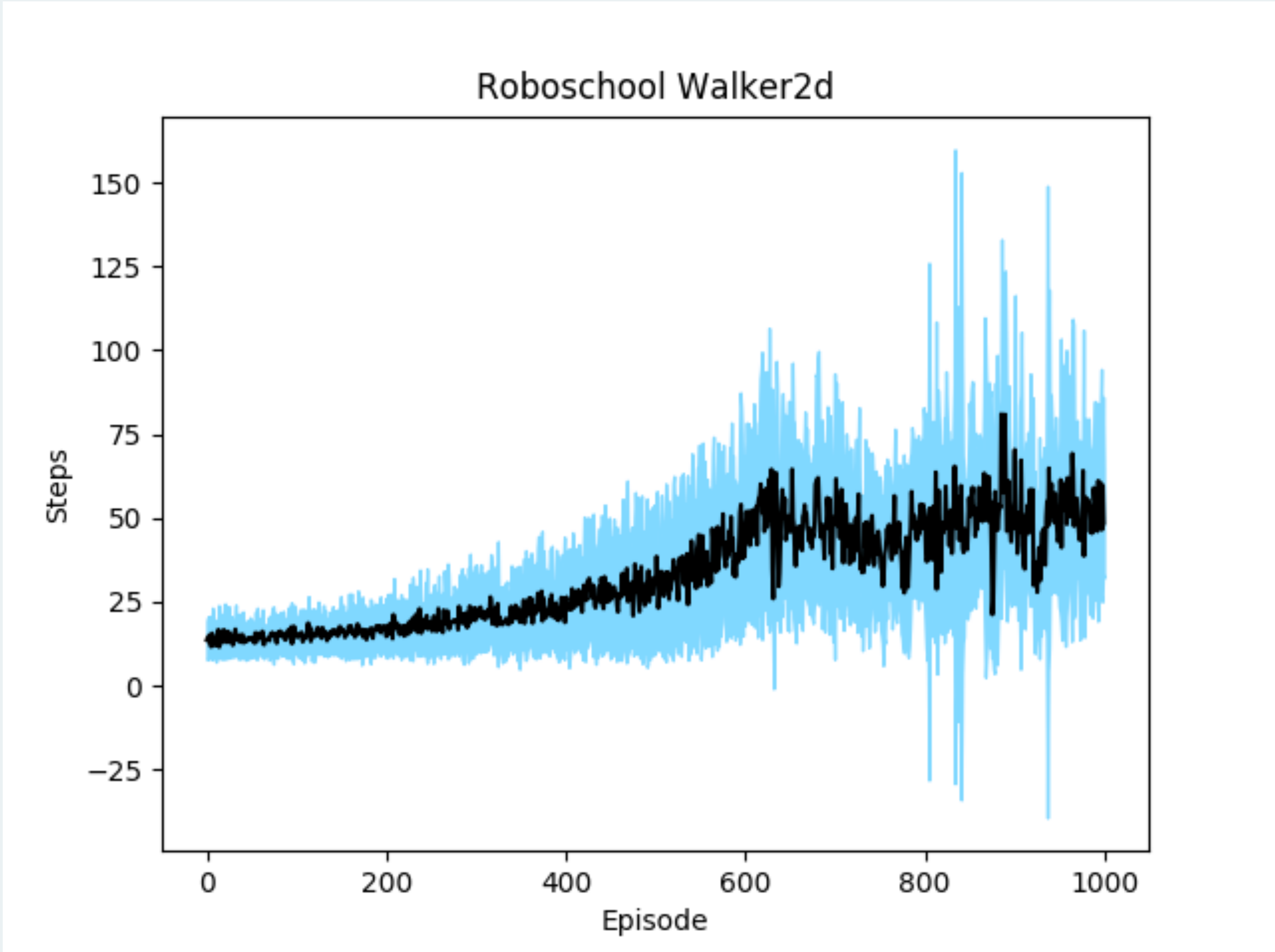
Result: Write here the result
Initialize replay memory D to capacity L;
Initialize dependency graph DG to size n;
for *epsilon = 1, M* **do**
  Initialize sequence $s_1 = x_1 and preprocessed sequenced \Phi(s_1)$;
  for *t=1, T* **do**
    $r_t = 0$
    for *i=0, n* **do**
      With probability ϵ select a random action a_t;
      otherwise select $a_t = max_a Q^*(\Phi(s_t), a, dg[i]; \theta)$;
      Execute action a_t in emulator and observe reward r_{ti};
      $r_t + = r_{ti}$
    end
    Set $S_{t+1} = S_t, \ [a_t], x_{t+1}$ and preprocess $\Phi_{t+1} = \Phi_{s_{t+1}}$;
    Store transition $(\Phi_t, [a_t], r_t, \Phi_{t+1})$ in D;
    Sample random minibatch of transitions $(\Phi_j, [a_j], r_j, \Phi_{j+1})$ from D;

$$Set \qquad y_j = \begin{cases} r_j & for\ terminal \Phi_{j+1} \\ r_j + \gamma max_{a'} Q(\Phi_j, [a_j]; \theta) & for\ non-terminal\ \Phi j + 1 \end{cases} \tag{1}$$

    Perform a gradient descent step on $(y_j - Q(\Phi_j, [a_j]; \theta))^2$;
  end
end

Results

(Test environment: Roboschool, walker2d)



In our machine (64 GB memory and two 1080Ti), DQN runs out of memory and crashed.
The most important advantage of CQL is that it reduces the action space, which saves the memory and improves the memory efficiency when training. By using CQL, we reduces the running time for each step although the number of each step is similar.

Future Directions

In the future, we may consider how to reduce the steps to converge and how can we find a good dependency graph to reach the global optimum faster.