Deep Neural Networks for Policy Representation in Continuous Action domains

Sanjay Thakur
260722338

Sumana Basu
260727568

Abstract

Practical problems of reinforcement learning have continuous state and action space. This makes them unfit to be solved by tabular methods. In this project we used neural networks as function approximators to map our states to actions. We used policy gradient method REINFORCE to solve 3 locomotion tasks of MuJoCo. However, our solutions get stuck to local optima. Better exploration strategies with random restarts of the neural network will solve this problem.

1. Introduction

Reinforcement learning (RL) is the area of machine learning that deals with the problem of sequential decision making to maximize some notion of cumulative reward. An RL agent interacts with the environment and learns an optimal policy that maximizes the long term reward. The optimal policy tells the agent which action to take from each state to solve the decision problem. To solve problems with low dimensional and finite environment space, tabular approaches are sufficient. But all practical problems have high dimensional and continuous observation and action space. To apply tabular solution methods, first the state and action space need to be discretized. Coarse discretization of action would produce poor performance and fine discretization would make the number of states and actions exponentially large, making it infeasible to solve them using tabular methods.

To handle continuous state and action space, we need function approximation techniques. Function approximations are dependant on the feature mapping. But optimal feature mapping varies with every environment. Generalized function approximators such as neural networks, decision-trees or instance based methods are required to handle this issue (Richard S. Sutton).

With the recent advances of deep learning, feature representation using deep neural networks have revolutionized the capability and scalability of reinforcement learning algorithms. Success of Atari games and AlphaGo are most notable example of it (Silver (2016)). Deep neural network also saves us from tedious task-specific manual feature engineering. Feature representation using deep neural networks coupled with control based algorithms on continuous action tasks is capable of solving complex tasks such as 3D-locomotion and

manipulation.

In this project we have used a deep neural network for feature representation and implemented policy gradient method REINFORCE with baseline to solve three Multi-Joint dynamics with Contact (MuJoCo) environments, namely Swimmer, Hopper and Ant. The goal of these tasks are to move forward as quickly as possible. Besides being multi-dimensional, these tasks require a lot of exploration to learn to move forward without getting stuck at local optima. It happens because there exists a local optima of staying at the origin or diving forward slowly.

2. Environment Description

MuJoCo is an environment that foster research in robotics, mechanics, graphics and animation where fast and accurate simulation is required. It supports a few basic tasks like cart-pole balancing, mountain car; locomotion tasks swimmer, hopper, ant, simple humanoid, full humanoid; partially observable tasks and hierarchical tasks. In this project we have concentrated on few of the locomotion tasks.

2.1 Swimmer

Swimmer is the simplest of all locomotion environments since this robot does not flip over (eg. ant) or fall down (eg. humanoid). This gives the robot a better opportunity to explore. This is a planar robot with 3 links and 2 actuated joints. It is assumed to be in a viscous fluid in which the robot is trying to move forward by exerting force to change joint angles, joint velocities, centre of mass etc. The environment has 8 dimensional continuous observation space and 2 dimensional continuous action space. Reward is given by,

$$r(s, a) = v_x - 0.005||a||_2^2 \tag{1}$$

where, v_x = forward velocity
No termination condition is applied (Duan et al. (2016)).

2.2 Hopper

Hopper is planar mono-pod robot having 4 rigid links for torso, upper leg, lower leg and a foot and 3 joints (Lillicrap et al. (2015)). This environment is more complex than swimmer since the agent must learn hopping without falling. This demands for more exploration than the previous environment. Hopper has 11 dimensional observation space and 3 dimensional action space. Dimension of observation space correspond to joint angles, joint velocities, centre of mass etc. Reward is given by,

$$r(s, a) = v_x - 0.005||a||_2^2 + 1 \tag{2}$$

where, the last term is a bonus for not falling over.
Episode terminates when $z_{body} < 0.7$ (where z_{body} is the z-coordinate of the body), or when $|\theta_y| < 0.2$, (where θ_y is the forward pitch of the body) (Duan et al. (2016).

2.3 Ant

Ant has 4 legs, 1 torso and 8 joints. So, in total in has 13 dimensional observation space and Ant is the toughest of all the locomotion environments we successfully tried because of high dimensional observation space. It has 111 dimensional observation space and 8 dimensional action space. Reward is given by,

$$r(s, a) = v_x - 0.005||a||_2^2 - C_{contact} + 0.05 \tag{3}$$

where, $C_{contact}$ penalizes for contact to ground.
Episode terminates when $z_{body} < 0.2$ or $z_{body} > 1.0$

3. Algorithm

We used REINFORCE with baseline on a deep neural network (Lillicrap et al. (2015)) to solve control problem on high dimensional, continuous-control locomotion tasks.

The vanilla REINFORCE is a policy gradient technique that rely upon optimizing parametrized policies with respect to the expected return (long-term cumulative reward) by gradient descent. As a result, it does not suffer from many of the problems that have been marring traditional reinforcement learning approaches such as the lack of guarantees of a value function, the intractability problem resulting from uncertain state information and the complexity arising from continuous states actions.

However, vanilla REINFORCE has high variance and hence it might take huge number of convergence steps to converge itself to the optimal solution. To solve this problem, we used a critic whose purpose is to evaluate how good the policy is with the current parameters (weights) and then guide the parameter updates in that direction. But introducing a critic would make our policy gradient an approximate policy gradient step. Hence, extra care is to be taken while choosing critic since an approximate policy would introduce bias in the system, which might not allow the algorithm to solve the problem in hand. We used the unbiased estimate of our advantage function to serve as our critic. The advantage function is shown in equation 4, and the equation for its unbiased estimate is formulated in equation 5. The proof of relation between them is given in 6.

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \tag{4}$$

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s) \tag{5}$$

$$
\begin{aligned}
E_{\pi_\theta}[\delta^{\pi_\theta}|s,a] &= E_{\pi_\theta}[r + \gamma V^{\pi_\theta}(s')|s,a] - V^{\pi_\theta}(s) \\
&= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\
&= A^{\pi_\theta}(s,a)
\end{aligned} \tag{6}
$$

As a result, our updates to the weights become what is given as in equation 7

$$
\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\delta_\theta^\pi] \tag{7}
$$

We followed the algorithm 1 for our implementation

Algorithm 1: Unbiased REINFORCE with baseline using Deep Neural Network

Input : Observation from the environment
Output: Magnitude of action in each action dimension

```
Initialize critic parameter w
Initialize actor parameter θ
Initialize current_state as the initial environment's state
while true do
    current_action = actionSuggestedbyNN(current_state, θ)
    next_state, reward = takeActionOnEnvironment(current_state, current_action)
    δ^{π_θ} = r + γV^{π_θ}(s') − V^{π_θ}(s)
    doGradientAscent(θ, ∇_θ log π_θ(s,a) δ_θ^π)
    doGradientDescent(w, δ_θ^π)
    current_state = next_state
end
return
```

4. Model

We used two deep neural networks and a linear model to implement algorithm 1. The first neural network is to make the actor learn to take actions, second neural network for the critic to estimate state values, and the linear model for parameterising the variance that was needed to take continuous actions. Since our actions are continuous in nature, we adopted Gaussian policy to sample actions which can be formulated as 8.

$$
a \sim \mathcal{N}(\mu(s), \sigma^2) \tag{8}
$$

Table 1 shows the various components of the two deep neural networks. The linear model simply combined the input states to its weights. The loss function for training the linear

model is same as that of the neural network for the actor. The gradient of the loss function for the actor and the variance estimate can be formulated as equation 9.

$$\nabla_\theta \pi_\theta(s, a) = \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) \tag{9}$$

Table 1: The implemented architectures

	Input Units	Hidden Units	Output Units	Loss function
Actor	Observation Dimension	(100, 50, 25)	Action Dimension	Equation 9
Critic	Observation Dimension	(100, 50, 25)	Action Dimension	Least square
Variance	Observation Dimension	-	Action Dimension	Equation 9

We used the Tensorflow (Abadi et al. (2015)) backend to create three of our models.

5. Results

For all the three algorithms, return improved quickly upto a point(figure 1,2, and 3). It signifies that our agent is learning. However, video logs of these environments revealed that for all these environments, learning got stuck into local optima which although were better than random, left a lot of scope for improvement. This has been described in detail in the following subsections.

5.1 Swimmer

As depicted in figure 1, Sum of rewards did not increase over episodes. It got saturated pretty soon and did not improve after that. On looking into the videos, we observed that the agent learnt to generate an initial momentum in the forward direction and then used its inertia to keep moving forward without taking any action. Thus the agent accumulated positive reward as it kept on moving forward, but this is definitely not what we intended it to do in the first place. It could attempt to make forward moves all the time and could gain even better rewards.

5.2 Hopper

As figure 2 suggests, in case case also the the agent learnt a local optimal solution. It learnt how to dive forward slowly every single time gaining some positive reward in this process. However, again this is not what we wanted the agent to learn. We wanted it to keep on hopping without falling over for as long as it can.

5.3 Ant

For the first few iterations, the ant always ended up flipping on its back. But over time it learned how to not flip over. But it stopped learning anything after that and it was apparently it local optimal solution (figure 3)

Figure 1: Swimmer : Rewards over Episodes

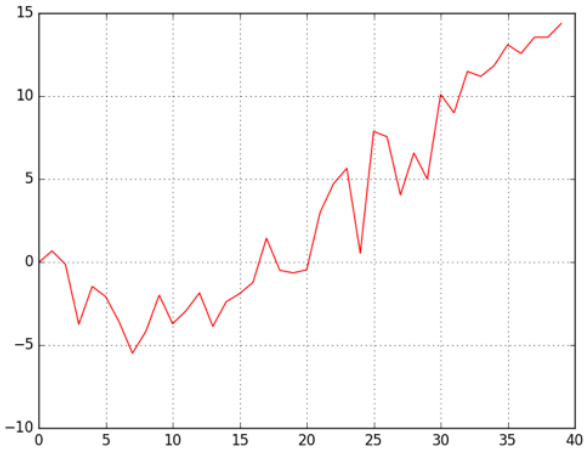
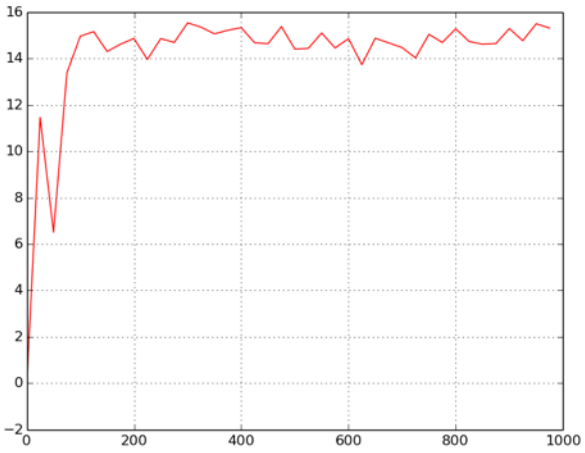
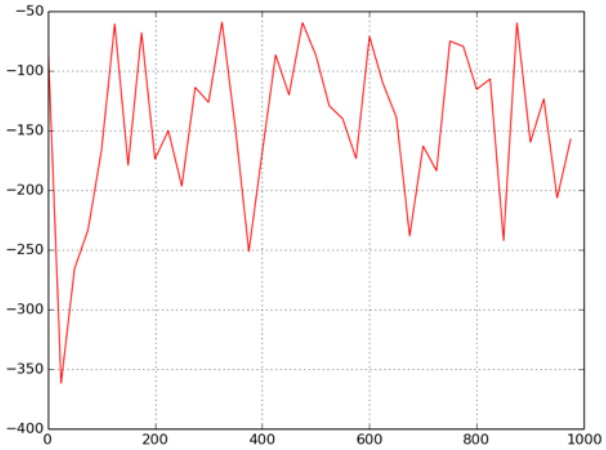


Figure 2: Hopper : Rewards over Episodes



We believe that these unexpected behavior of agents calls for better exploratory implementations which falls under the future scope of our project (section 7)

Figure 3: Ant : Rewards over Episodes



6. Relevance

Deep learning techniques save us from the tedious work of task-specific manual feature-engineering. Policy gradients have been a boon to many applications especially in the field of robotics as it pursues end to end learning. Together, they have given unprecedented amount of learning capabilities in domains where the environment contains continuous actions with high dimensions.

Continuous control domains were challenging to solve as discretizing them was not a robust and reliable solution. Granular discretization makes the problem infeasible to solve owing to the curse of dimensionality and discretizing it coarsely might make the algorithm miss out on essential information.

7. Discussion and Future Scope

The results section explained how we our algorithm got stuck in local optimas in all the environments. Some of the ways to overcome the shortfalls of our process are listed below:

1. Use random restarts.

2. Use gradient-free methods.

3. Use techniques like DDPG that inserts noise in the policies so that it doesn't stop exploring.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control, 2016. URL `http://arxiv.org/abs/1604.06778`.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2015. URL `http://arxiv.org/abs/1509.02971`.

Satinder Singh Yishay Mansour Richard S. Sutton, David McAllester. Policy gradient methods for reinforcement learning with function approximation.

Huang A. Maddison C. J. Guez A. Sifre L. Van Den Driessche G. Schrittwieser J. Antonoglou I. Panneershelvam V. Lanctot M. et al. Silver, D. Mastering the game of go with deep neural networks and tree search, 2016.