Application of Deep Reinforcement Learning On Robotic Arms

Zhewei Song, Xuexian Li, Northeastern University, Boston, MA, 02115

li.xuex@husky.neu.edu, song.zhew@husky.neu.edu

Abstract

The focus of this work is to enumerate the various approaches and algorithms that center around application of reinforcement learning in robotic manipulation tasks. Earlier methods utilized specialized policy representations and human demonstrations to come up with generalized policies. Subsequently, high dimensional non-linear function approximators like neural networks have been used to learn policies from scratch. Several novel and recent approaches have also embedded control policy with efficient perceptual representation using deep learning. This has led to the emergence of a new branch of dynamic robot control system called deep reinforcement learning. This work embodies a survey of the most recent algorithms, architectures and their implementations in simulations and real world robotic platforms. The gamut of Deep Reinforcement architectures are based on continuous action space algorithms. Further, the continuous action space algorithms are divided into stochastic continuous action space and deterministic continuous action space algorithms. Along with elucidating an organization of the Deep Reinforcement Learning algorithms this work also manifests some of the state of the art applications of these approaches in robotic manipulation tasks.

Introduction

Reinforcement learning methods have been applied to range of robotic control tasks. However, practical real-world applications of reinforcement learning have typically required significant additional engineering beyond the learning algorithm itself: an appropriate representation for the policy or value function must be chosen so as to achieve training times that are practical for physical hardware. In this work, we show that a recently proposed deep reinforcement learning algorithms based on off-policy training of deep Q-functions, can be extended to learn complex manipulation policies from scratch, without user-provided demonstrations, and using only general-purpose neural network representations that do not require task-specific domain knowledge.

One of the central challenges with applying direct deep reinforcement learning algorithms to real-world robotic platforms has been their apparent high sample-complexity. We demonstrate that, contrary to commonly held assumptions, recently developed on-policy A3C algorithm, off-policy deep Q-function based algorithms such as the Deep Deterministic Policy Gradient algorithm(DDPG) and Normalized Advantage Function algorithm(NAF) can achieve training times that are suitable for a simulated 2-dimentional robotic environment. We also demonstrate that we can place our objects in environment randomly and let robotic arm to scratch it to right place.

The main contribution of this project is a demonstration of asynchronous advantage actor-critic, NAF, DDPG algorithms performance on robotic arms manipulation tasks. Our technical contribution consists of build a static, dynamic environment for the experiment, algorithm implementation, optimize both environment and neural net by doing feature engineering and reward engineering. Our experiments also evaluate the benefits of deep neural network representations for several manipulation tasks, including scratch fixed objects, scratch moving objects, move it to right place. Our simulated experiments show that our approach can be used to train robots learn objects classification task from scratch using only general-purpose neural network representations and without any human demonstration.

Background

As we discussed above, the process of arms moving during the exploration is a continuing case. Therefore, we are going to apply policy gradient algorithms like DDPG to solve this problem. The deterministic policy gradient algorithm(DPG) is derived from its counter-part stochastic policy gradient algorithm and is dependent of a similar deterministic policy gradient theorem.

In continuous action spaces, greedy policy improvement step[1]. As a result, it is more computationally tractable to update the policy parameters in the direction of the gradient of the Q function.

$$\theta^{k+1} = \theta^k + \alpha \mathrm{E}[\Delta_\theta Q(s, \mu_\theta(s)]$$

Here, $\mu_\theta(s)$ is the deterministic policy, α is the learning rate and θ are the policy parameters. Chain can be applied to the above equation in order to get the policy gradient equation:

$$\theta^{k+1} = \theta^k + \alpha \mathrm{E}[\Delta_\theta \mu_\theta(s) \Delta_a Q(s, \mu_\theta(s)]$$

The above update rule can be incorporated into a deep neural network architecture where the policy parameters are updated using stochastic gradient ascent. The gradient of policy parameters is the product of the gradient of Q value with respect to action and the action with respect to the policy parameters. This is also the basis of DDPG(deep deterministic policy gradient algorithm) which performs better than any other continuous action algorithms. We would also applied NAF and A3C on this robotic arms manipulation tasks[2]. We are using Tensorflow to do deep network architecture with 2-dimention state space consisting of the joint angles and the end effector pose. The reward function for the reaching task is the distance

between the end effector and the object, whereas for stuff classification the reward was the distance to the right area boundary.

Related Work

Since the action space is continuous space, algorithm that using policy gradient like PPO, A2C and ACER can also be applied to this problem. A2C stands for Advantage-Actor-Critic. It is a synchronous variant of A3C algorithm. Base on OpenAI Research[3], A2C have equal performance as A3C. ACER is short of Sample Efficient Actor-Critic with Experience Replay. It is a Actor-Critic model with replay buffer which is stable and performs well on challenging environments. But ACER is much complicated and need extra work on replay buffer and off-policy update. PPO stands for Proximal Policy Optimization which use a new objective function.

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t]$$

This objective implements a way to do a Trust Region update which is compatible with SGD, and simplifies the algorithm by removing the KL penalty and need to make adaptive updates.[4] All those three methods are relatively new methods, so as newbie in reinforcement learning and tensorflow, we choice some classical methods to solve this problem.

Project Description

This project's task is defined by ourselves, so we need to architect our own environment for this task. This environment is supposed to be robust for our RL algorithms. After environment architecture, the first thing we need to consider is our agent's actions in this environment are continuous or discrete. This is an essential difference. Not every RL method is suitable for continuous and discrete actions. For this task, we think continuous actions can simulate robotics arms actions better. There are some methods like Policy gradient, A3C, PPO, NAF, DDPG suitable for continuous actions. The following lists how we choose some of them to deploy into our environment:

Environment Architecture

There are some basic functions should be concerned in our env.py:

```
env.reset()
env.render()
env.step(a)
env.state_dim
env.action_dim
env.action_bound
```

1) Initialize our static environment with pyglet

We can use pyglet package to visualize our environment, so that we can view our agent's every step action during training process. In our environment, we created a red color robotics arm with two parts, a blue color square object and a grey color target area.

2) Construct and test dynamic environment

This time we combine arms movement part and visualization part to test our dynamic environment. Firstly, we define two part of arm's length, two nodes swing angles are actions that the arm choose at each time. We can use this information to compute two joints coordinates and finger's coordinates. After that we can define arms update rules each step.

3) State, *Actions and Rewards*

We all know that state, actions and rewards are three key factors in reinforcement learning. They are all defined in environment. After experiments, we figure out nine-dimension states input, two-dimension actions selection, increasing reward until finger touch goal and target area:

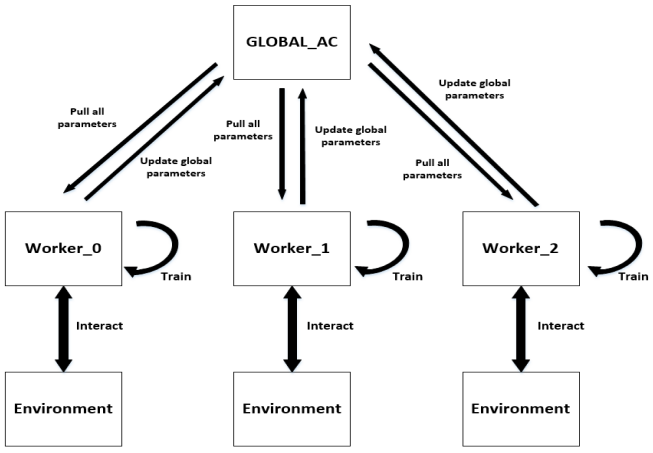
State: second joint's coordinate, finger's coordinate, distance between finger and goal, distance between second joint and goal, the time duration of finger touching on the goal.

Action: two joints rotation angles.

Reward: the closer finger approaches to the goal, the higher reward agent gets during an episode.

A3C Algorithms

A3C stands for Asynchronous Advantage Actor-Critic. We use Tensorflow to implement this algorithm and teach the robot arm to catch object that show up randomly on the field. In the course, we learn how the DQN work, but unlike DQN, A3C utilizes multiple agent in order to learn more efficiently. There is a global network and multiple worker agents in A3C. Each of these agents interacts with it's own copy of the environment at the same time as the other agents are interacting with their environments. Since the experience of each agent is independent, A3C do not need a replay buffer to train the network.



Figure[1]

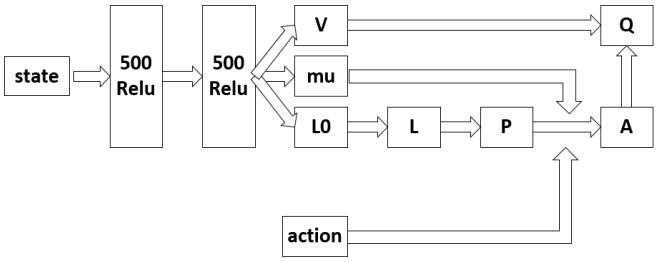
Including global network, each of those agents have an actor network to predict the policy which is a set of action and a critic network to estimate the value function. In this project both the actor and critic network have two fully connected layers and take the state as input. Since the critic network need to "correct" the actor network, we set the critic network to have a larger learning rate. Then A3C use advantage function to judge the action, but we won't determining the Q values in A3C, we use the discounted returns as the estimate of Q to generate an estimate of the advantage. Also we use entropy to encourage exploration. So this is the objective function we use in this model.

$$\nabla_{\theta'} J(\theta) = \frac{1}{T}\sum_{t}^{T} \nabla_{\theta} \log \pi(a_t \mid s_t;\theta)\Big(\sum_{i=1}^{n} \gamma^{i-1} r_{t+1} + v(s_{t+n}) - v(s_t)\Big) + \beta \nabla_{\theta} H(\pi(s_t;\theta))$$

The A3C algorithm begins by constructing the global network. This network will not contain the loss function because we do not need to train this network. Next, create a set of worker agents. In this project we create 8 worker agents because my cpu have 8 core. Each of those agents have their own network and environment. Then each of those workers are run on a separate processor thread. Each worker then interacts with its own copy of the environment and have a buffer to collects experience. Once the agent have 10 experience in the buffer or the environment return done equal to True, we use those experience to calculate value and policy losses. A worker then use these losses to calculate the gradients and then update the global network parameters. In this way, the global network is constantly being updated by different agents.

NAF Algorithms

NAF stands for Normalized Advantage Function. NAF is an improvement of DQN which allow as to use off-policy Q-learning in a continuous action space with deep neural networks. This model represent the Q-function Q(x,u) in Q-learning in such a way that its maximum, $\text{argmax}_u Q(x,u)$, can be determined easily and analytically during the Q-learning update.[5] Unlike A3C which use actor to predict action and critic to predict value, NAF use one network to estimate both value and action. So this is the structure of NAF.



Figure[2]

Also, this model use advantage function, just like Dueling-DQN to judge the action:

$$A(x,u \mid \theta^A) = -\frac{1}{2}(u - \mu(x \mid \theta^{\mu}))^T P(x \mid \theta^P)(u - \mu(x \mid \theta^{\mu}))$$

$P(x \mid \theta^P)$ is a state-dependent, positive-define square matrix which is parametrized by $P(x \mid \theta^P) = L(x \mid \theta^P)L(x \mid \theta^P)^T$ where $L(x \mid \theta^P)$ is a lower-triangular matrix whose entries come from a linear output layer of a neural network.[5]

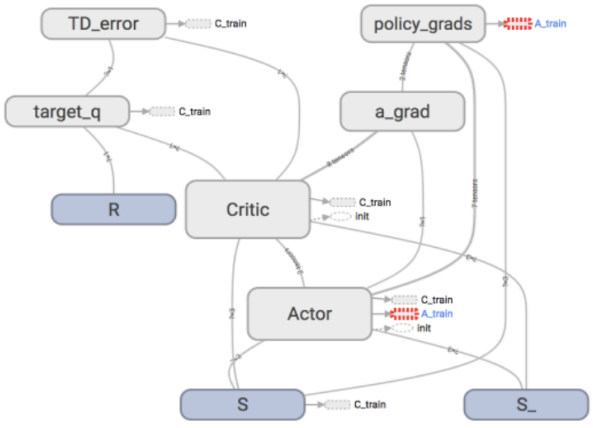
The NAF algorithm begin with construct two networks, just like the DQN, we also use target network to stable the value function. Each of these networks have two fully connected layers with 500 neurons. Then we create a replay buffer and use the annealing epsilon to control the exploration. After that the agent start to interact with the environment and store the experience into the replay buffer. After some steps of interaction we use the Adamoptimizer to minimize the loss function and train the agent.

DDPG Algorithms[6]

It is not possible to straightforwardly apply Q-learning to continuous action spaces, because in continuous spaces finding the greedy policy requires an optimization of at at every timestep; this optimization is too slow to be practical with large, unconstrained function approximators and nontrivial action spaces. Instead, here we used an actor-critic approach based on the DPG algorithm (Silver et al., 2014). The DPG algorithm maintains a parameterized actor function $\mu(s|\theta\ \mu)$ which specifies the current policy by deterministically mapping states to a specific action. The critic Q(s, a) is learned using the Bellman equation as in Q-learning. The actor is updated by following the applying the chain rule to the expected return from the start distribution J with respect to the actor parameters:

$$\nabla_{\theta^{\mu}} J \approx \mathbb{E}_{s_t \sim \rho^{\beta}}[\nabla_a Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t|\theta^{\mu})}] = \mathbb{E}_{s_t \sim \rho^{\beta}}[\nabla_a Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s=s_t}]$$

Silver et al. (2014) proved that this is the policy gradient, the gradient of the policy's performance. As with Q learning, introducing non-linear function approximators means that convergence is no longer guaranteed. However, such approximators appear essential in order to learn and generalize on large state spaces. DDPG here is to provide modifications to DPG, inspired by the success of DQN, which allow it to use neural network function approximators to learn in large state and action spaces online. We refer to the algorithm Deep DPG.



Figure[3]

As in DQN, DDPG used a replay buffer to address these issues. The replay buffer is a finite sized cache R. Transitions were sampled from the environment according to the exploration policy and the tuple (st, at, rt, st+1) was stored in the replay buffer. When the replay buffer was full the oldest samples were discarded. At each timestep the actor and critic are updated by sampling a minibatch uniformly from the buffer. Because DDPG is an off-policy algorithm, the replay buffer can be large, allowing the algorithm to benefit from learning across a set of uncorrelated transitions.

Directly implementing Q learning with neural networks proved to be unstable in many environments. Since the network Q(s, a|θ^Q) being updated is also used in calculating the target value, the Q update is prone to divergence. DDPG's solution is similar to the target network used in (Mnih et al., *2013)* but modified for actor-critic and using "soft" target updates, rather than directly copying the weights. We create a copy of the actor and critic networks, $Q'(s, a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$ respectively, that are used for calculating the target values. The weights of these target networks are then updated by having them slowly track the learned networks: $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$ with $\tau \ll 1$. This means that the target values are constrained to change slowly, greatly improving the stability of learning. This simple change moves the relatively unstable problem of learning the action-value function closer to the case of supervised learning, a problem for which robust solutions exist.

Experiments

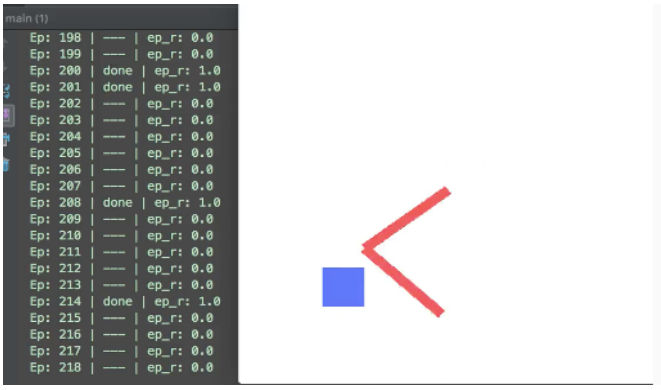
In this project, we plan to realize three stages robotics arm manipulation tasks. They are touch fixed objects, touch random location objects and place random location objects to target area.

First Stage Task:

Problem: At this stage, our goal is try to make the arm touch a fixed object in environment. However, after we start use algorithms to train an evaluated neural net, we find two problems during the training process by observing our visualization window:

- Episodes ended with arm swinging, cannot identify the time when the finger touch the object.
- At the end of training, performance of agent doesn't converge.

Only if we visualize every step(Figure[2]) movement as an animation window, we can figure out the problems.
Visualization is important during reinforcement process.



Figure[2]

Solutions: We assume some reasons that could cause these problems:

- We may modify our environment?
- Neural net capacity?
  Features selection?
  Reward setting?

A. Fix arm swing

For arms the swing problem, we should find a way to make finger stay at goal object some time. So we create a counter called on_goal in env.step(), if finger stay at goal area more than 50 step, done=True.

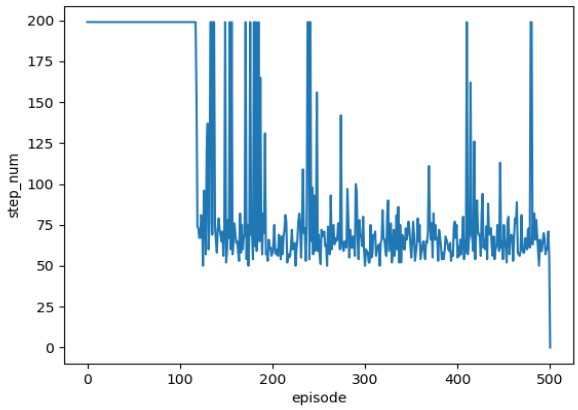
B. Reward Engineering:

We notice that if we give reward continuously, it would be better than give sparsely. If agent can only get reward when it finally reach the goal, it is hard to learn during the training. This single reward might disappear in large size of episode data easily. If agent can get reward signal at every step and these rewards scaled into different value. Sparse reward problem can be easily solved. At this phase, build a good reward function is important.
In our environment, the closer the finger to goal, the higher reward the agent can get. Furthermore, if finger stay at goal area, reward r+=1.

C. Feature Engineering

If we spend some time on finding useful features. Convergence can be improved more. Feature is a key factor in reinforcement learning. If features we use can show learning environment perfectly, arms can learn quickly from these valuable features.
Previously, we only use two part of arms rotation angle as states. From learning status, we conclude that this information cannot provide whole profile of state. We tried add distance between two arms joint, finger and the goal into our state space. Now our state dimension increases to 9.

After we did feature engineering and reward engineering, performance improved. We use the number of steps that arms used to touch the goal as evaluation standard and visualize it:



Figure[3]

At the beginning of training, we can see that agent need 200 to converge. After 100 episodes, the number of converged steps decrease to 60-80 steps.

Second Stage Task:

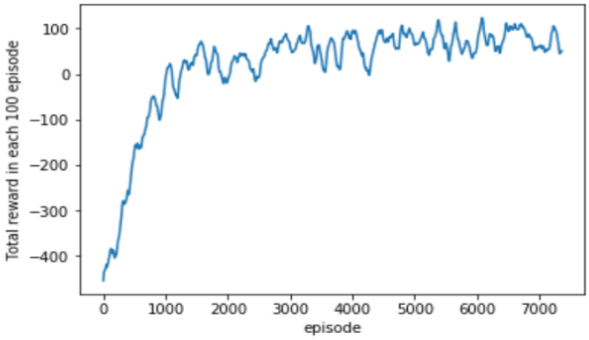
Problem: At this stage, our goal is to make robotics arms learn how to capture any places objects. I modified our environment to make the goal object appears randomly during the training process. We tried three RL algorithms on this task:

A. Experiments on A3C

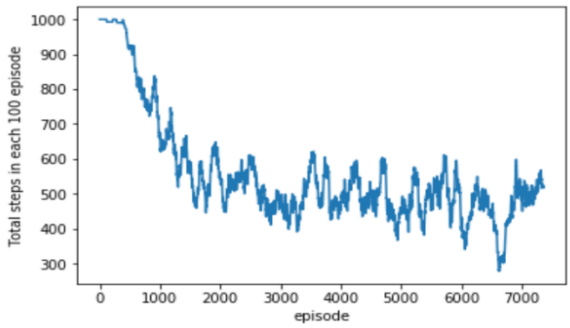
First we choose A3C to run the experiment. In this experiment we train the model certain number of episodes and then observe

the reward of each episode and how many steps each episode take. Since the environment won't return done equal to True until the robot arm catch the object, the episode could never terminate in the exploration episode. So we set the max steps of each episode to 1000. The reward per episode and the steps per episode can show whether the agent learn how to catch the object. If the agent did learn how to catch the object, the reward per episode should increase and the steps per episode should decrease along with the increase of episode.

First of all, we set the max episodes equal 2000, set the training frequency equal 10 and set gamma equal 0.8 to urge the agent to find and catch the object. A3C agent finish all episode in less than 20 minutes but the result show that this agent is barely converge. The graph of reward per 100 episodes and steps per 100 episodes shows that the agent learn something but there is still a potential to learn better. So we add one new global parameter GLOBAL_SUCCESS_EPISODE to count how many episodes the agent succeed in catching the object. Then we set the experiment terminate condition to the successful episode count equal to 5000 and rerun the training process. Then this is the result of the experiment.



Figure[4]

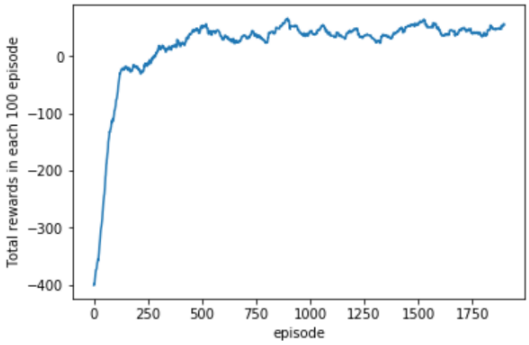


Figure[5]

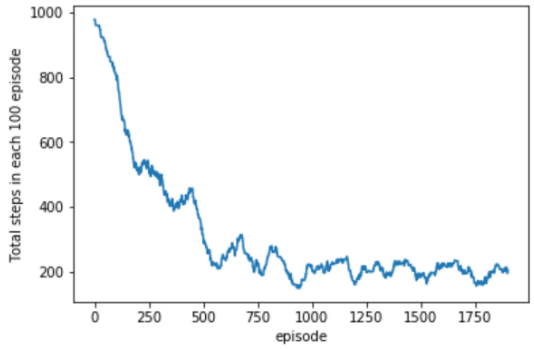
It take 34 minutes to run 7463 episodes and get 5000 successful episodes. Although we smooth the line using 100 episodes window, the steps per 100 episodes curve still swing around 500. So we visualize the environment and manual change the object location and see whether the robot arm can still catch the object. The result is the robot arm can catch the object at certain location while the other the robot arm fail to catch the object. In my opinion, the reason is though we have 8 independent worker agents, it is still no enough to keep the IID situation. In order to improve the performance, we should use more worker in the same time.

B. Experiment on NAF

Then we use NAF model to run the experiment. Also we set the max episodes equal 2000, gamma equal 0.8 and training frequency equal 10. The result is the agent run all the episodes in a ridiculous long time. Also from the visualization of train process, we found that the robot arm need to first learn how to swing the arm, that is to say the agent need to learn how to output the action in the same direction. Otherwise the arm is just shaking around the initial position and use up all the exploration steps. The is why the result is also bad. In order to reduce the time of training and improve the performance of the agent, we modified the get_action function. In the exploration steps, instead of randomly choose action from the [-1, 1], which are the action bound, the agent randomly choose action from either [-1,0] or (0, 1). In this way the agent have a fix direction to move the robot arm. Then we reduce the training frequency to 100, train the agent every 100 episodes. After that we rerun the experiment. The NAF agent finish training in 16 minutes and the result is better.



Figure[6]

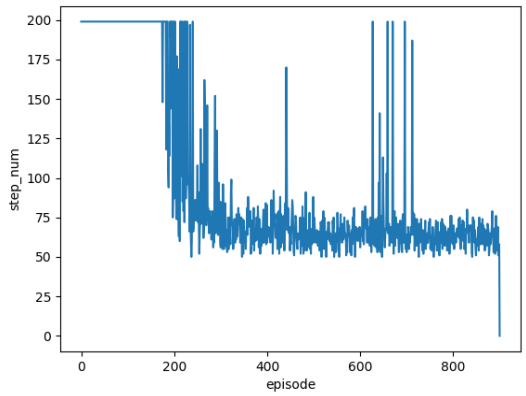


Figure[7]

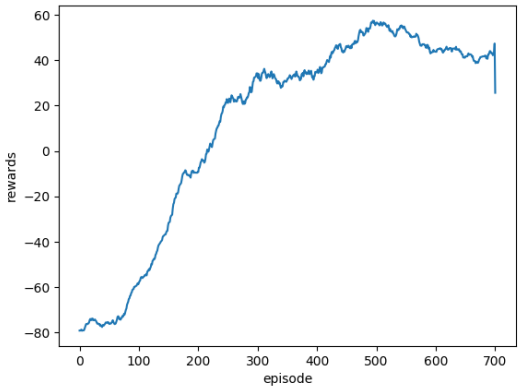
Compared with A3C agent, the curve of steps per 100 episodes of NAF agent have a much smoother curve, and converge to a lower number of steps. After the exploration steps the steps of each 100 episodes quickly drop and converge to 200. Also the NAF agent do not need a very long time to train and get a better result that A3C. So NAF algorithm is a better solution than A3C for this robot arm problem. The visualization of NAF agent using the trained parameters also confirm this, though the robot arm have some vibration, it can catch the object wherever we place.

C. Experiments on DDPG

With experiences from implementing NAF and A3C on this task, we save much time on DDPG's nets parameter modification. Because DDPG integrate advantages from both A3C and NAF. In actor and critic nets, DDPG use reply buffer like DQN to store previous episode memory. For fixed object scratch, we use 100 neurons in first hidden layer and memory size of target net is 10000. In this task, we try to increase neurons to 300 and memory size to 30000. In this way, after test, our arms become more robust and can capture any place object moved by mouse.



Figure[8]



Figure[9]

As we can see from result, DDPG perform better than NAF and A3C. For random goal scratch task, it only uses 200 episodes to converge. And our max_step in every episode is defined as 200 which means DDPG has higher capacity on this task.

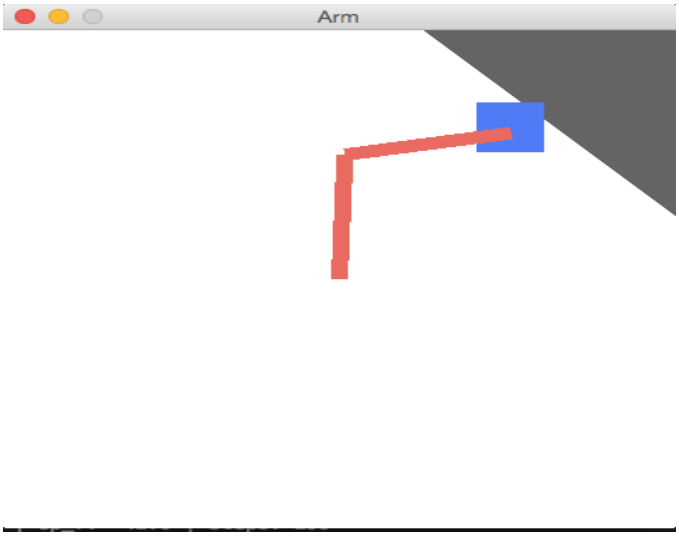
Final Stage Task:

Problem: At this stage, we want to upgrade our goal to make arms move random objects in environment to a certain area.

Solution: At the beginning, we tried to let our agent learn "scratch objects" and "place to right area" tasks at the time. However, finally we find it is hard to define our done step, state and reward space if we have two phases tasks in one environment.

Therefore, we separated them into two individual tasks. After arms finish the first task, our goal would switch to certain area. Second task for arms is try to approach to a certain area, just like how it learned from pervious task. We trained another DDPG nets and after agent finish the first task it uses second train net to choose actions. In this way, agent can easily complete a two phases task.

Figure[10]



Conclusion

In all three experiment, A3C agent perform the worst. It needs a lot of time to train and the result is poor. NAF agent perform better. It still need some time to train but the result is much better than A3C agent. DDPG agent perform the best. It is both efficient and effective. It only needs 800 episodes training and use the least steps per episodes. In this project we learn how to apply reinforcement learning method on continuous action space environment. There is two ways to modify classical Q-learning method so that it could be applied to continuous action space. First, we could build two network, one of these output the action and the other output the state value or action value. That is how actor-critic network work. Second way is that we could build one network and output both action and state value. That is how NAF network work. In order to improve the A3C agent, we can try to run more workers at the same, or we can add a replay buffer to the model. For the NAF agent, we use Cholesky decomposition to calculate the $P(x \mid \theta^{P})$, actually we could try identity covariance, diagonal covariance or square covariance. Also we could try to apply batch normalize method and observe the result.

Algorithmic ideas, theories and implementation details of several deep reinforcement learning algorithms have been delineated in detail. It can be concluded that for the purpose of robotic manipulation continuous action domain algorithms are the most fruitful and applicable. Further, it can be observed that there is a trend towards exploration of sample efficient and time efficient algorithms, having solved both continuous state and action space problems. Breakthroughs in these domains will have significant impacts in the field of robotics learning.

Also, as demonstrated from current state of the art in DRL, the approaches fail to handle complex policies[7]. A reason

could be that complicated policies require more samples to learn and even a sophisticated reward function. This observation high- lights a void in RL in robotics. There is a need to learn highly complicated reward functions and methods to represent highly skilled behaviors and skills. This are of Inverse reinforcement learning needs to be paid more attention while learning policies using DRL. After all, complexity of the reward function is proportional to the policy complexity.

References

[1] Lever, G.,2014. Deterministic policy gradient algorithms.
[2] Gu, S., Holly, E., Lillicrap, T. and Levine, S., 2016. Deep Reinforcement Learning for Robotic Manipulation. arXiv preprint arXiv: 1610.00633.
[3] OpenAI Baselines: ACKTR & A2C. Available: https://blog.openai.com/baselines-acktr-a2c/
[4] Proximal Policy Optimization. Available: https://blog.openai.com/openai-baselines-ppo/#content
[5] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, Sergey Levine. "Continuous Deep Q-Learning with Model-based Acceleration." Available: https://arxiv.org/pdf/1603.00748.pdf
[6] Lillicrap,2016. Continuous control with deep reinforcement learning. arXiv:1509.02971v5.
[7] Smruti Amarjyoti., 2017. Deep reinforcement learning for robotic manipulation-the state of the art. arXiv:1701.08878v1.