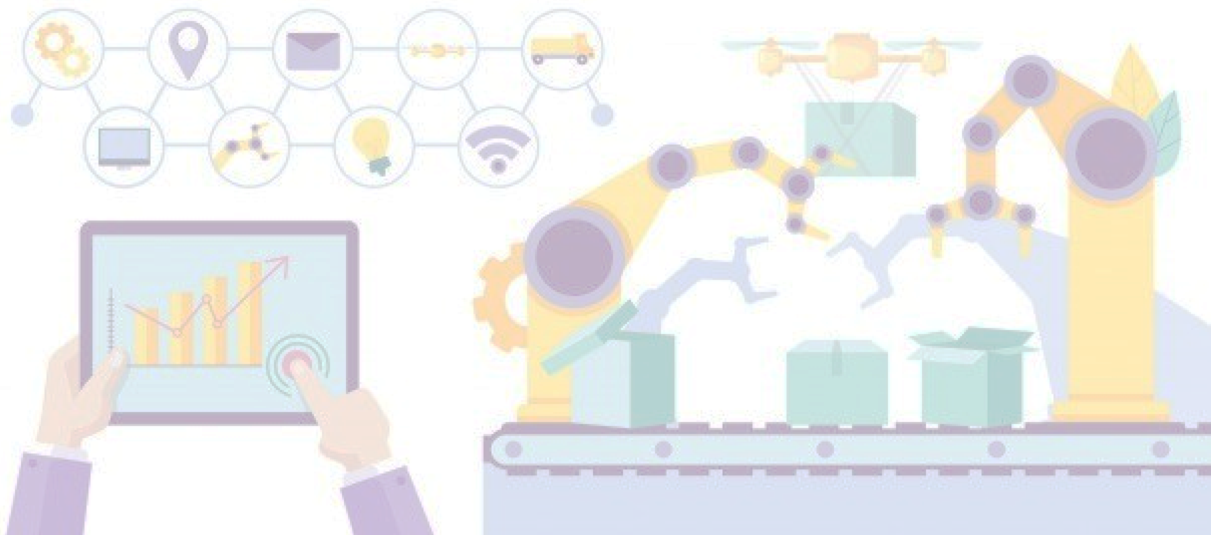
AI61009: Artificial Intelligence for Manufacturing
Indian Institute of Technology Kharagpur

AI-BASED MOTION PLANNING OF AN ARTICULATED ROBOT ARM IN AN ACTIVITY



Mentor: Prof C.S. Kumar
Group 11 Team Members
Abhranil Chandra(19MF10002)
Shreya Khare(19ME10062)
Rishabh Roy(19ME10097)
Pratham Nigam(19CS10047)

Introduction

In recent years, computational frameworks have been developed for generating open-loop motion plans for manipulation tasks in a deterministic environment. The emphasis in these tasks is on the kinematic and dynamic interactions between the robot(s) and the environment. In this report, we present an articulated robot arm. The objective is to train this arm, which has no prior experience, to reach a random point in 3D space by trial and error learning. Trial and Error Learning formally known as Reinforcement Learning has been the very reason for massive improvements in robot learning and manipulation over the last couple of years. Famous examples include the BRETT Robot from the University of California, Berkeley (https://engineering.berkeley.edu/brett/) and the Shadow Dexterous Hand from Open AI (https://openai.com/blog/learning-dexterity/). Automated robot learning enables robot manipulation and motion planning that has lots of applications in the manufacturing sector, medical surgery, and high stake crisis handling, to name a few.

Problem Statement and Possible Solutions

The problem statement is to do AI-based motion planning of an articulated robot arm for an activity. The action space used here is continuous with X, Y, Z coordinates. Deep Q-Network will inevitably fail here unless we discretize the action space which will, in turn, make it harder to train the agent as rewards are sparse and any information loss in the form of discretization will lead to a highly unstable training scheme which might not even converge. Thus we use the Deep Deterministic Policy Gradient(DDPG) algorithm.

We have the Sparse Discrete Reward Setting(-1 and 0 only). Supervision signal is particularly difficult to propagate in sparse reward settings and thus learning which action is better than the other becomes very difficult particularly in high dimensional robot manipulation tasks. It's almost guaranteed that the agent will get -1 in the beginning episodes and if the value function is not updated the agent will get stuck in a local minima and training will become very slow. So it's imperative to devise some method to acquire skills even from the failed steps and episodes. Thus we use Hindsight Experience Replay(HER) as it randomly replaces the goal with what the robot achieves in some steps from the replay buffer thus leading to more rewards and at least some learning which helps the agent learn faster even in sparse settings.

Solution Used

We have used Deep Deterministic Policy Gradient(DDPG) ALgorithm. It is an Actor-Critic based algorithm that uses the actor which is a policy-based network to optimize the actions based on the current state. The critic is a value-based network that optimizes the value prediction(Q) based on the current state and the actor's actions. We use the standard neural implementations of both the actor and the critic.

We have used Hindsight Experience Replay(HER). HER substitutes the goal state (in our experiments the position of goal point) with an achieved goal(the actual position that the robot arm reaches after the action) in the replay buffer and uses this as training. This ensures that in sparse reward settings where the probability of getting positive rewards is really less and thus learning becomes difficult, at least the agent keeps learning something from even the failed episodes. ("You never lose. You either win or learn."). We have only sampled given achieved goals and the desired goal, but one can also sample different goals to make the algorithm learn faster.

Tech Stack Used

Primary Package Requirements:

- Python 3.9.7
- Tensorflow 1.15.0 (to build and train neural architectures)
- OpenAI Gym 0.20.0 (provides the environment for training the agent)
- MuJoCo(free-mujoco-py) and mujoco_py (physics simulation engine)

Hardware Used:

- Nvidia Tesla P100 GPU
- 15 GB RAM
- 77 GB Disc Space

We used Google Colaboratory to do all the training and experiment work. The total training time taken was about 7 hours 30 minutes.

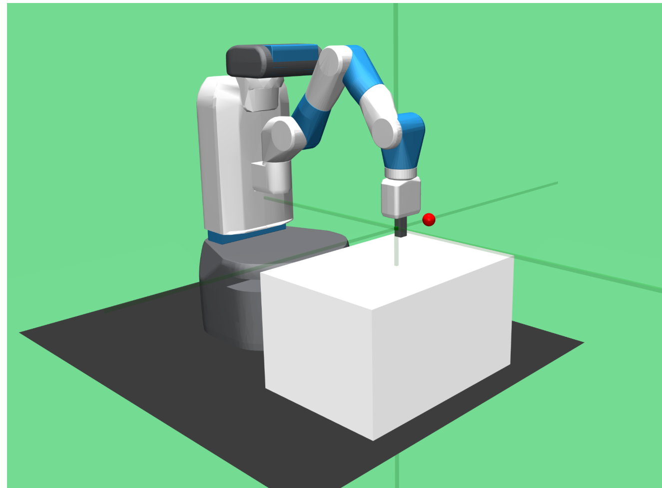
Environment Used

OpenAI Gym Simulated Robotics Environment:

We use the 'FetchReach-v1' environment using the Fetch research platform. The manipulation tasks contained in the Robotics environments are significantly more difficult than the MuJoCo continuous control environments also available in Gym, all of which are now easily solvable using recently released algorithms like PPO. Furthermore, the Robotics environments use models of real robots and require the agent to solve realistic tasks. It uses the MuJoCo physics simulator. In the FetchReach-v1 environment, the agent has to move its end-effector to the desired goal position. It is a goal-based task for example reaching the desired position in the reacher task. The environment by default uses a sparse reward of -1 if the desired goal was not yet achieved and 0 if it was achieved (within some tolerance). This is in contrast to the shaped rewards used in the old set of Gym continuous control problems.
A variant with dense rewards for each environment is also included. However, that sparse rewards are more realistic in robotics applications.

Robot Before Training

The robot shows mindless behaviour before training as it fails to reach the red point.



Data Generated from the Environment Simulator:

- Environment observations(10 float values)
    - 3D coordinate of the gripper, linear velocity, gripper velocity, etc.
    - 3D coordinate of the goal object, rotation, linear velocity, etc.
    - Achieved and Desired Goals(3 float values each)
- Reward per step(0 if succeed; -1 if fail)
- Episode Complete(True/False)
- Lowest Episode Score(-200 per episode)

Installations needed for Execution

Installations:

- 3D graphics library(libgl1-mesa)
- C/C++ extension loading library(libglew)
- Off-screen rendering extension of Mesa(libosmesa6)
- Gym
- free-mujoco-py(mujoco is now license-free after being acquired by DeepMind and can be run on Colab)
- Tensorflow 1.15

Steps to run:

- Connect session and in runtime type change Hardware accelerator to GPU.
- Restart the environment once after all the package installations are done.
- Run each cell one after the other as the actor-critic classes are defined first and need to be executed first before we start training.

Algorithms

- **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 $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode=1, M **do**
    Initialize a random process N for action exploration
    Receive initial observation state s_1
    for t=1, T **do**
        Select action $a_t=\mu(s_t|\theta^\mu)+N_t$ according to the current policy and exploration noise
        Execute action a_t and observe reward r_t and observe the new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in R
        Sample a 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 $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\Theta^Q))^2$ loss:

        Update the actor policy using the sampled policy gradient:
$$\nabla_{\Theta^\mu} J \approx \frac{1}{N}\sum \nabla_a Q(s, a|\Theta^Q)|_{s=s_i;a=\mu(s_i)} \nabla_{\Theta^\mu} \mu(s|\Theta^\mu)_{s_i}$$
        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

- **Hindsight Prioritized Experience Replay:**

Initialize value function Q
Initialize Є-greedy policy P
Initialize replay buffer M
while not converged **do**
  Sample goal G and get initial state S
  while S is not the terminal state **do**
    Select action A according to P with respect to Q
    Execute A, get reward R, and next state S'
    Store transition t=(S, A, R, S', G) into M
    Sample a set G of additional goals for replay

for G’∈G do

Compute new reward R’

Store transition (S, A, R’, S’, G’) into M

end

Sample batch B from M following ω

Update Q with B

S ← S’

end

end

Code snippets of Actor Network

Network Architecture

- Inputs are the state observations from environment.
- 3 layer fully connected neural network.
- Normalize the inputs and batch-normalization between dense layers.
- Output layer activation function is tanh as then actions will be bound between (-1, 1).
- The Actor uses gradients from Critic and optimizes actions and states.

```
def actor_model(self):
    inputs = tf.placeholder(tf.float32, [None, self.state_size])
    x = tc.layers.layer_norm(inputs, center=True, scale=True, begin_norm_axis=0)
    h1 =  tf.layers.dense(x, 400, activation = tf.nn.relu )
    h1 = tc.layers.layer_norm(h1, center=True, scale=True)
    h2 = tf.layers.dense(h1, 300, activation = tf.nn.relu )
    h2 = tc.layers.layer_norm(h2, center=True, scale=True)
    k_init = tf.random_uniform_initializer(minval=-0.003, maxval=0.003)
    out = tf.layers.dense(h2, self.action_size, activation = tf.nn.tanh,kernel_initializer = k_init)
    scaled_out = tf.multiply(out, self.action_bound)

    return inputs, out, scaled_out
```

- The Actor uses gradients from Critic and optimizes actions and states.
- Thus the Critic gradients are combined with the Actor gradients and then passed into the optimizer
- We use the Adam optimizer with learning rate 1e-3.
- The Actor Network also creates a target network for time step =1 to compute the temporal difference between different steps.

```
with tf.variable_scope('actor_target_net'):
    self.input_target_actor, self.target_out_, self.target_scaled_out = self.actor_model()

self.ac_target_pram = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = 'actor_target_net')
#print(len(self.ac_params))


self.update_target_in = [self.ac_target_pram[i].assign ( tf.multiply(self.ac_target_pram[i], 0) + tf.multiply(self.ac_params[i],1) ) for i in range(len(self.ac_target_pram))]
self.update_target = [self.ac_target_pram[i].assign ( tf.multiply(self.ac_target_pram[i], 1-self.tau) + tf.multiply(self.ac_params[i],self.tau) ) for i in range(len(self.ac_target_pram))]


self.critic_grad = tf.placeholder(tf.float32,[None, self.action_size], name = 'critic_grad')

self.actor_grad = tf.gradients(self.scaled_out, self.ac_params, -self.critic_grad)
self.actor_gradients = list(map(lambda x: tf.div(x, self.batch_size), self.actor_grad))

self.loss = tf.train.AdamOptimizer(self.lr).apply_gradients(zip(self.actor_gradients, self.ac_params))
```

Code snippets of Critic Network

Network Architecture

- The Critic uses a single dense layer(size = 200) after the state input.
- It has another input branch of the action inputs
- Finally these two are concatenated and passed through another dense layer(size = 300)
- Weights are randomly initialized using uniform distribution data.
- Finally we use Adam optimizer on mean-squared-error loss.

```
def build_net(self):
    inputs = tf.placeholder(tf.float32, [None, self.state_size])
    x = tc.layers.layer_norm(inputs, center=True, scale=True, begin_norm_axis=0)

    action = tf.placeholder(tf.float32, [None, self.action_size])
    h1 = tf.layers.dense(x, 200, activation = tf.nn.relu)
    h1 = tc.layers.layer_norm(h1, center=True, scale=True)
    h11 = tf.layers.dense(h1, 200,activation = tf.nn.relu)
    a1 = tf.layers.dense(action, 200)

    h1_ = tf.concat([h11,a1],axis = 1)
    h1_ = tc.layers.layer_norm(h1_, center=True, scale=True)

    h2 = tf.layers.dense(h1_, 300, activation=tf.nn.relu)
    h2 = tc.layers.layer_norm(h2, center=True, scale=True)
    k_init = tf.random_uniform_initializer(minval=-0.003, maxval=0.003)
    out_cr = tf.layers.dense(h2, 1,kernel_initializer=k_init)
    return inputs, action, out_cr
```

- Again a target network for time step =1 to compute the temporal difference between different steps.
- The network also calculates a gradient between the output(Q) and actual action.
- We the pass this gradient to the Actor Network.

```
with tf.variable_scope('critic_net'):
    self.input_critic, self.action_critic, self.value,  = self.build_net()
self.cr_prams = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = 'critic_net')

with tf.variable_scope('target_critic_net'):
    self.input_target_critic, self.action_target_critic, self.target_value = self.build_net()
self.target_cr_prams = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = 'target_critic_pram')

self.update_critic_target_in = [self.target_cr_prams[i].assign ( tf.multiply(self.target_cr_prams[i], 0) + tf.multiply(self.cr_prams[i],1) ) for i in range(len(self.target_cr_prams))]

self.update_critic_target = [self.target_cr_prams[i].assign ( tf.multiply(self.target_cr_prams[i], 1 - self.tau) + tf.multiply(self.cr_prams[i], self.tau) ) for i in range(len(self.target_cr_prams))]

self.pred = tf.placeholder(tf.float32, [None, 1], name= 'pred_value')
self.loss = tf.reduce_mean(tf.square(self.pred - self.value))
self.optimize = tf.train.AdamOptimizer(self.lr).minimize(self.loss)
self.comment_grad = tf.gradients(self.value, self.action_critic)
```

Code snippets of training using HER

- Experience is stored in a deque
- When sufficient sample trajectories have been saved in the deque, more than the current batch size, the code samples a mini-batch of previous experience/trajectories to train the networks.
- The critic is trained first with state, action and discounted reward.
- Then the actor is trained with gradient from the critic.
- HER also samples random past trajectories and replaces actual end state with episode end state achieved, so that some gradients pass to the actor and training can happen faster, particularly in this sparse reward setting.
- Since we sample episodes generated from a past policy to update the new policy DDPG+HER is a good example of off-policy reinforcement learning algorithm.

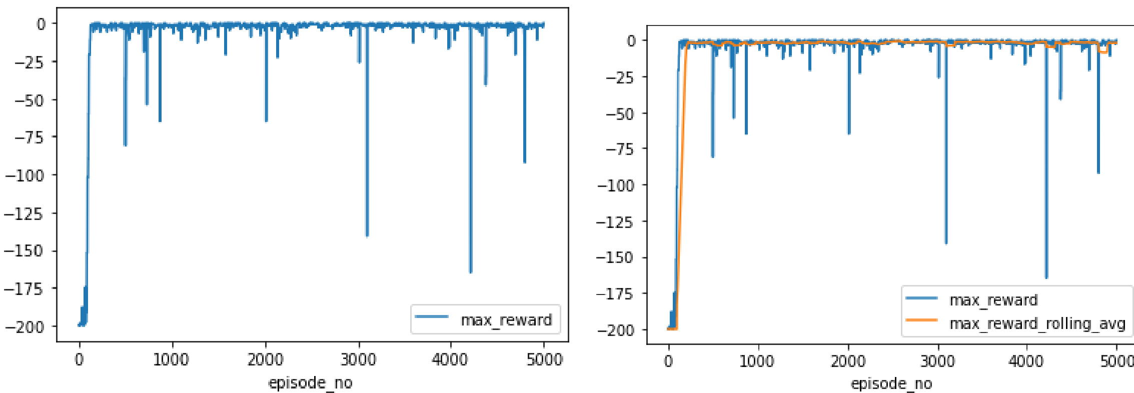
```
store_sample(s,a,r,d,info,s2)
#replay_memory.append((s,a,r,d,s2))
s = s2
R += r_2
if batch_size < len(replay_memory):
    minibatch = random.sample(replay_memory, batch_size)
    s_batch, a_batch,r_batch, d_batch, s2_batch = [], [], [], [], []
    for s_, a_, r_, d_, s2_ in minibatch:
        s_batch.append(s_)
        s2_batch.append(s2_)
        a_batch.append(a_)
        r_batch.append(r_)
        d_batch.append(d_)
    s_batch = np.squeeze(np.array(s_batch),axis=1)
    s2_batch = np.squeeze(np.array(s2_batch),axis=1)
    r_batch=np.reshape(np.array(r_batch),(len(r_batch),1))
    a_batch=np.array(a_batch)
    d_batch=np.reshape(np.array(d_batch)+0,(128,1))
    #print(d_batch)
    a2 = ac.get_action_target(s2_batch)
    #print(a2.shape)
    v2 = cr.get_val_target(s2_batch,a2)
    #print(v2.shape)
    #for
    tar= np.zeros((128,1))
    for o in range(128):
        tar[o] = r_batch[o] + gamma * v2[o]
    #print(tar.shape)
    cr.train_critic(s_batch,a_batch,tar)
    #print(loss_cr)

    a_out = ac.get_action(s_batch)
    kk = cr.get_grad(s_batch,a_out)[0]
    #print(kk)
    ac.train_actor(s_batch, kk)
    cr.update_critic_target_net()
    ac.update_target_tar()
    #exit()
```

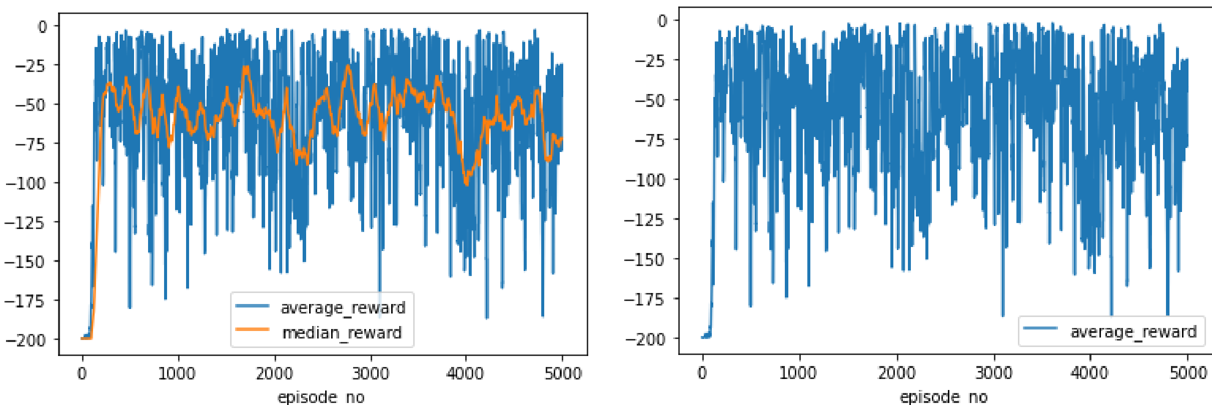
Results of the experiment and Hyperparameters used

We trained the agent for 5000 episodes each of length 200 steps and it took 7 hour 30 minutes to train.

- Its visible clearly that initially the robot fails totally for the first 250 episodes.
- Then as the experience is stored in replay memory and is used to update the current policy the agent learns rapidly within a span of the next 250 episodes.
- Finally we see that the agent achieves an average maximum reward of almost 0(the maximum possible per episode) and its learning saturates and it succeeds in most steps.



- The median cumulative reward per episode also steadily increases and plateaus out as the training ends. This proves that our agent has successfully learned to take correct actions mostly over the span of 200 steps per episode thus learning complex manipulation tasks from scratch simply by trial and error reinforced supervision.



We also tried to experiment with some of the hyperparameters like the batch size, network depth, learning rate etc.

- We used a batch size of 128 as mentioned in the DDPG paper.
- Adam optimizer learning rate was 1x10^-3
- Discount factor of 0.99
- Maximul episodes was 5000
- Per episode length was 200

Conclusion and future work

- DDPG+HER is the recipe for the successful training of many complex environments having continuous action space.
- DQN fails miserably here as we are using a continuous action space environment..
- A larger batch size leads to better generalization and faster convergence.
- Using a Replay Buffer to store episode data is a very important step in any off-policy RL algorithm. This is the reason we could leverage HER in the first place.
- The training is still stochastic, increasing episode lengths and the number of episodes helps.
- To stabilize training we may try Prioritized Experience Replay(PER), wherein we more frequently replay transitions with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error.