Aim: To make an Automatic Pick and Place Manipulator using Deep Reinforcement Learning.

Theory

1. Deep Reinforcement Learning

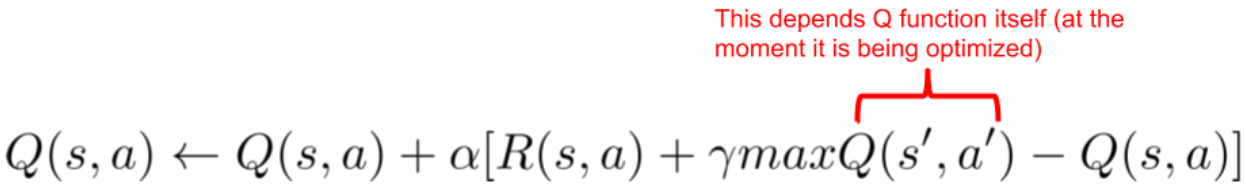
Deep reinforcement learning (DRL) uses deep learning and reinforcement learning principles in order to create efficient algorithms that can be applied on areas like robotics, video games, finance and healthcare. Implementing deep learning architecture (deep neural networks) with reinforcement learning algorithms (Q-learning, actor critic etc.), a powerful model (DRL) can be created that is capable to scale to previously unsolvable problems. That is because DRL usually uses raw sensor or image signals as input as can be seen in DQN for ATARI games and can receive the benefit of end-to-end reinforcement learning as well as that of convolutional neural networks.

2. Deep Deterministic Policy Gradients (DDPG)

DDPG uses four neural networks: a Q network, a deterministic policy network, a target Q network, and a target policy network.

The Q network and policy network is very much like simple Advantage Actor-Critic, but in DDPG, the Actor directly maps states to actions (the output of the network directly the output) instead of outputting the probability distribution across a discrete action space

The target networks are time-delayed copies of their original networks that slowly track the learned networks. Using these target value networks greatly improve stability in learning. Here's why: In methods that do not use target networks, the update equations of the network are interdependent on the values calculated by the network itself, which makes it prone to divergence. For example:



3. Hindsight Experience Replay (HER)

To understand what HER does, let's look at in the context of a Pick and Place Manipulator, a task where we need to learn to pick a box from one position and place it at a target position **(red sphere)**. Our first attempt very likely will not be a successful one. Unless we get very lucky, the next few attempts will also likely not succeed. Typical reinforcement learning algorithms would not learn anything from this experience since they just obtain a constant reward (in this case: -1) that does not contain any learning signal.

The key insight that HER formalizes is what humans do intuitively: Even though we have not succeeded at a specific goal, we have at least achieved a different one. So why not just pretend that we wanted to achieve this goal to begin with, instead of the one that we set out to achieve originally? By doing this substitution, the reinforcement learning algorithm can obtain a learning signal since it has achieved some goal; even if it wasn't the one that we meant to achieve originally. If we repeat

this process, we will eventually learn how to achieve arbitrary goals, including the goals that we really want to achieve.

This approach lets us learn how to pick the box from one position and put it in another position on or off the table even though our reward is fully sparse and even though we may have never actually hit the desired goal early on. θe call this technique Hindsight Experience Replay since it replays experience (a technique often used in off-policy RL algorithms like DQN and DDPG) with goals which are chosen in hindsight, after the episode has finished. HER can therefore be combined with any off-policy RL algorithm (for example, HER can be combined with DDPG, which we write as "DDPG + HER").

Technology used: Python, NumPy, PyTorch, OpenAI's Gym, MuJoCo

Procedure

1. Randomly initialize Critic Network $Q(s,a|\theta^Q)$ and Actor $M(s|\theta^M)$ with θeight θ^Q and θ^M
2. Initialize target network Q' and M' with θeights $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{M'} \leftarrow \theta^M$
3. Initialize Hindsight Experience Replay Sampler R
4. For episode = 1, E do
    a. Initialize a Random Process N for Action Exploration
    b. Receive Initial Observation State s1
    c. For t = 1,T do
        i. Select action $a_t = M(s_t|\theta^M) + 'N_t$ according to the current policy and exploration noise
        ii. Execute action a_t and observe reward r_t and observe new state s_{t+1}
        iii. If the new state is not the desired state, assume that it is now the desired state
        iv. Store transition (s_t,a_t,r_t,s_{t+1}) in R
        v. Sample a random minibatch of N transitions (s_t,a_t,r_t,s_{t+1}) from R
        vi. Set $y_i = r_i + G.Q'(s_{i+1},M'(s_{i+1}||\theta^{M'})\theta^{Q'})$ (G is the discount factor hyperparameter)
        vii. Update Critic by minimizing the loss: $L = (1/N).\Sigma(y_i - Q(s_i,a^i|\theta^Q))^2$
        viii. Update the actor policy using the sampled policy gradient

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

Here, μ = M in above equation

        ix. Update the target networks
            1. $\theta^{Q'} \leftarrow r.\theta^Q + (1-r)\theta^{Q'}$
            2. $\theta^{M'} \leftarrow r.\theta^M + (1-r)\theta^{M'}$
        x. end for
    d. End for
5. End

Code

Actor and Critic Networks

```
class actor(nn.Module):
	def __init__(self, env_params):
		super(actor, self).__init__()
		self.max_action = env_params['action_max']
		self.fc1 = nn.Linear(env_params['obs'] + env_params['goal'], 256)
		self.fc2 = nn.Linear(256, 256)
		self.fc3 = nn.Linear(256, 256)
		self.action_out = nn.Linear(256, env_params['action'])

	def forward(self, x):
		x = F.relu(self.fc1(x))
		x = F.relu(self.fc2(x))
		x = F.relu(self.fc3(x))
		actions = self.max_action * torch.tanh(self.action_out(x))

		return actions

class critic(nn.Module):
	def __init__(self, env_params):
		super(critic, self).__init__()
		self.max_action = env_params['action_max']
		self.fc1 = nn.Linear(env_params['obs'] + env_params['goal'] + env_params['action'], 256)
		self.fc2 = nn.Linear(256, 256)
		self.fc3 = nn.Linear(256, 256)
		self.q_out = nn.Linear(256, 1)

	def forward(self, x, actions):
		x = torch.cat([x, actions / self.max_action], dim=1)
		x = F.relu(self.fc1(x))
		x = F.relu(self.fc2(x))
		x = F.relu(self.fc3(x))
		q_value = self.q_out(x)

		return q_value
```

HER Sampling

```
import numpy as np

class her_sampler:
	def __init__(self, replay_strategy, replay_k, reward_func=None):
		self.replay_strategy = replay_strategy
		self.replay_k = replay_k
		if self.replay_strategy == 'future':
			self.future_p = 1 - (1. / (1 + replay_k))
```

```
            else:
                self.future_p = 0
            self.reward_func = reward_func

    def sample_her_transitions(self, episode_batch, batch_size_in_transitions):
            T = episode_batch['actions'].shape[1]
            rollout_batch_size = episode_batch['actions'].shape[0]
            batch_size = batch_size_in_transitions
            # select which rollouts and which timesteps to be used
            episode_idxs = np.random.randint(0, rollout_batch_size, batch_size)
            t_samples = np.random.randint(T, size=batch_size)
            transitions = {key: episode_batch[key][episode_idxs, t_samples].copy() for key in
    episode_batch.keys()}
    # her idx
            her_indexes = np.where(np.random.uniform(size=batch_size) < self.future_p)
            future_offset = np.random.uniform(size=batch_size) * (T - t_samples)
            future_offset = future_offset.astype(int)
            future_t = (t_samples + 1 + future_offset)[her_indexes]
            # replace go with achieved goal
            future_ag = episode_batch['ag'][episode_idxs[her_indexes], future_t]
            transitions['g'][her_indexes] = future_ag
            # to get the params to re-compute reward
            transitions['r'] = np.expand_dims(self.reward_func(transitions['ag_next'], transitions['g'],
    None), 1)
            transitions = {k: transitions[k].reshape(batch_size, *transitions[k].shape[1:]) for k in
    transitions.keys()}

            return transitions
```

Output

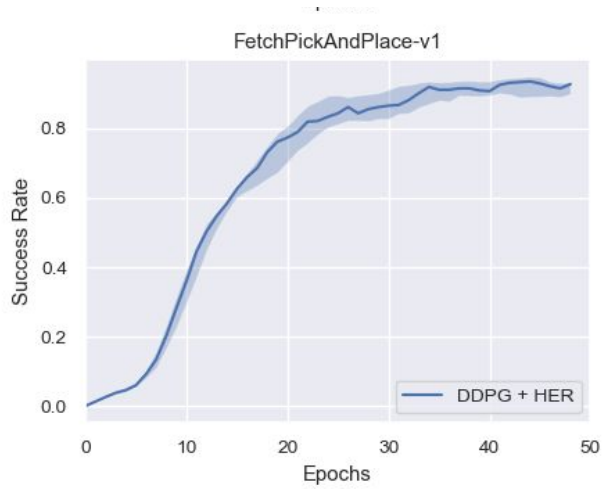


Fig: Success Rate VS Epochs Graph for DDPG+HER Algorithm

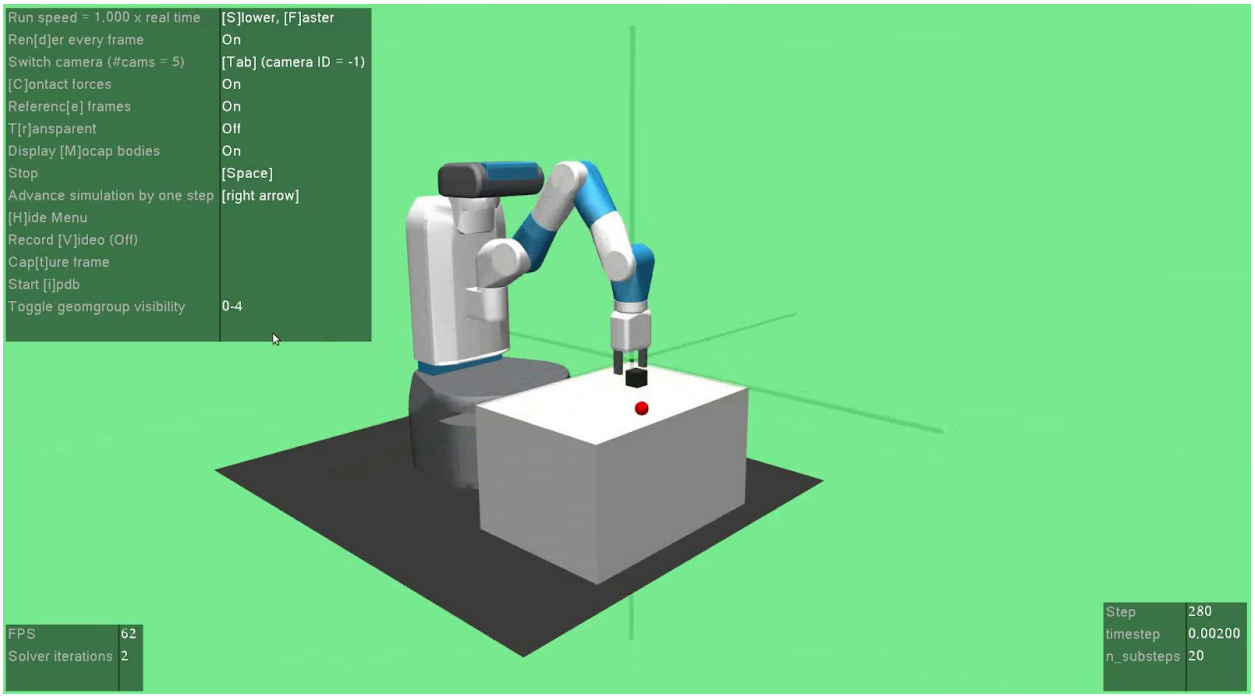


Fig: Robot in Lift Position

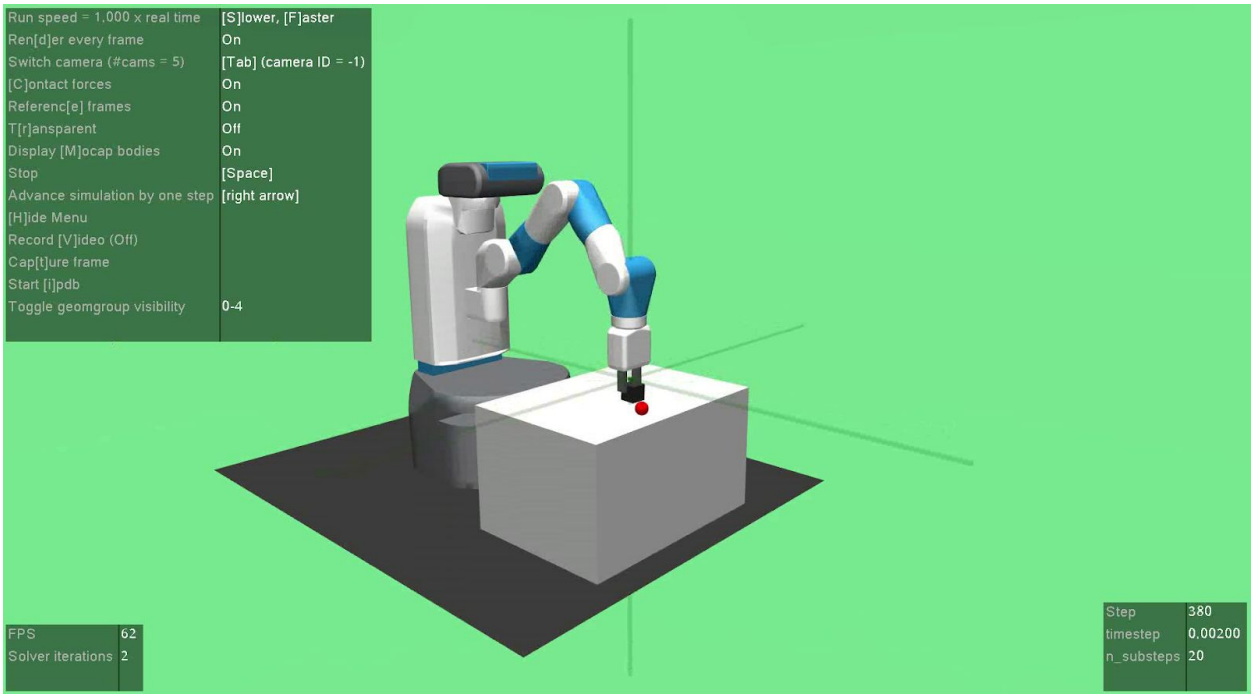


Fig: Robot in intermediate position grasping the block

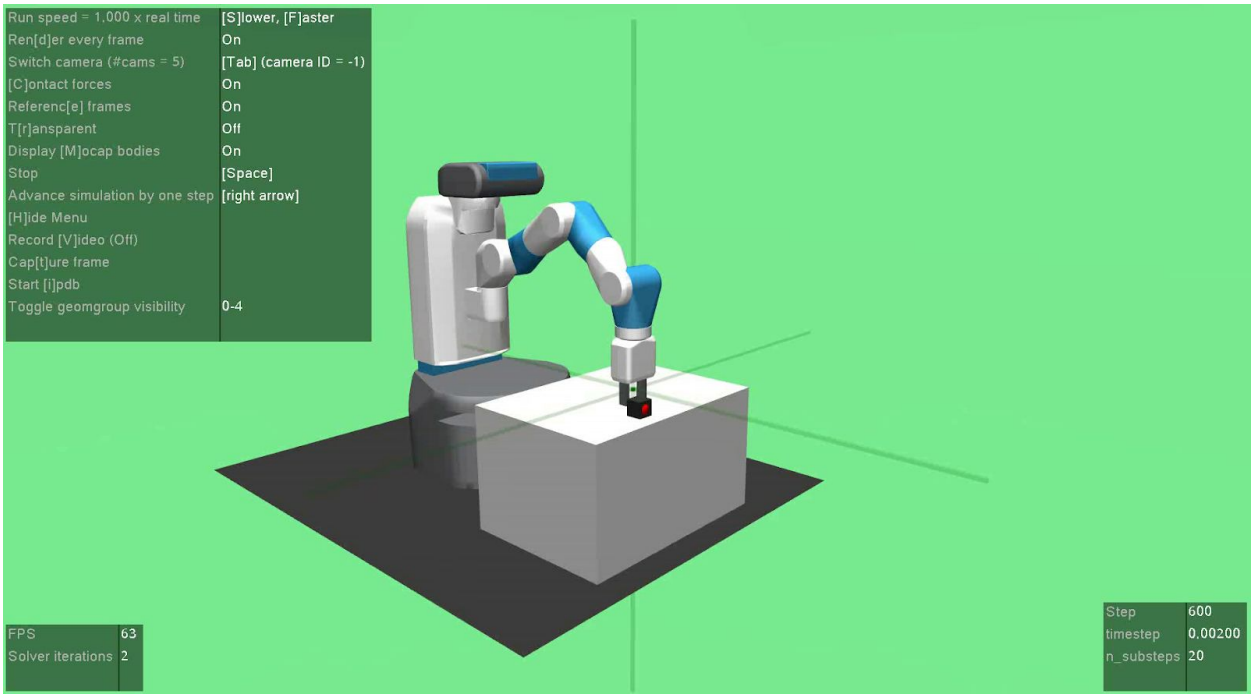


Fig: Robot placing the block on target position

Conclusion: Successfully trained a Robotic Manipulator to pick a block from initial position and place it at the target position using Deep Deterministic Policy Gradients with Hindsight Experience Replay.