# Deep Reinforcement Learning for Robotic Manipulation

**Tony Qin**
Carnegie Mellon University
Pittsburgh, PA 15213
tcq@andrew.cmu.edu

**Prateek Parmeshwar**
Carnegie Mellon University
Pittsburgh, PA 15213
pparmesh@andrew.cmu.edu

**Sachin Vidyasagaran**
Carnegie Mellon University
Pittsburgh, PA 15213
svidyasa@andrew.cmu.edu

**Arun Thomas Varughese**
Carnegie Mellon University
Pittsburgh, PA 15213
athomasv@andrew.cmu.edu

May 9, 2020

The code for this project can be found at https://github.com/sachin-vidyasagaran/11785-Project

## 1 Introduction

Robotic Arms have grown in popularity in the recent past, and with a growing need for quick, accurate and repeatable object manipulation in a wide range of fields, the need for intelligent arm manipulation techniques has never been more pressing. There already exist many analytical and iterative techniques to manipulating robot arms to achieve desired configurations, otherwise known as Inverse Kinematics. In this project, we decided to explore the possibility of using Deep Reinforcement Learning to instead train an arm to move to any desired configuration.

In Reinforcement Learning, we try to design agents to take the actions in an environment to maximize some cumulative reward signal. This environment is typically formulated as a Markov Decision Process, where at each time step the agent receives an observation, receives a reward, and takes an action. The figure below describes this relationship [1].

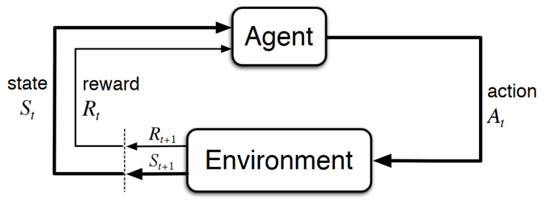


Figure 1: Markov Decision Processes in Reinforcement Learning

In the case of this project, the robot or robot arm is considered to be the agent, and the surrounding area in front of it within its work-space is considered to be the environment. The robotic arm can execute "actions" in this environment by driving its actuators to move within the environment, and this in turn changes the state of the environment. Depending on the location of the gripper relative to the goal location, the robot receives a reward. These could be in the form of binary sparse rewards where the reward is positive when the gripper is at the goal location and negative otherwise, or dense rewards where the reward is a function of the euclidean distance between the gripper position and the goal position.

The goal of the agent, therefore, is to learn the optimal action to take at every timestep, given an initial state and desired goal state (known as a policy), that maximizes the cumulative reward earned over the episode.

### 1.1 Environment

In order to develop a deep reinforcement learning algorithm, involving simulating an environment, running experiments, training a model and validating results, the **OpenAI Gym** toolkit is a very useful resource. The "fetch" environments available in Gym consist of a Fetch Robotics robot arm in front of a platform, as well as additional objects in some cases. A figure of such an environment (FetchPickAndPlace-v1) can be seen below.

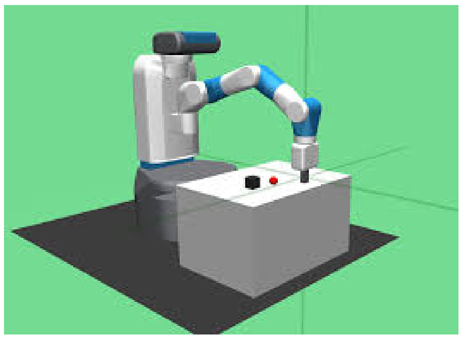


Figure 2: The FetchPickAndPlace Environment

For the different fetch environments, the state observations may include the cartesian position and linear velocity of the robot's gripper. If an object is present, such as in the FetchPickAndPlace environment, the object's cartesian position and rotation in euler angles are also available, along with its linear and angular velocities and its position and linear velocities relative to gripper.

### 1.2 DDPG

DDPG[2] is an algorithm that is built on Deterministic Policy Gradient. It is a model free approach that combines Deep Q networks and Deterministic Policy Gradient. DDPG works well in control problems that need complex multi-joint movements and rich dynamics.

Q learning can be use to make neural networks act as function approximators for action-value pairs. But it is not possible to straightforwardly apply Q-learning to continuous action spaces, because in continuous spaces finding the greedy policy requires an optimization 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 action-value function is used in many reinforcement learning algorithms. It describes the expected return after taking an action $a_t$ in state $s_t$ and thereafter following policy $\pi$ is given by,

$$Q^{\pi}\left(s_t, a_t\right) = \mathbb{E}_{r_{i \geq t}, s_{i > t} \sim E, a_{i > t} \sim \pi}\left[R_t | s_t, a_t\right]$$

The critic is learned using Bellman equation

$$Q^{\pi}\left(s_t, a_t\right) = \mathbb{E}_{r_t, s_{t+1} \sim E}\left[r\left(s_t, a_t\right) + \gamma \mathbb{E}_{a_{t+1} \sim \pi}\left[Q^{\pi}\left(s_{t+1}, a_{t+1}\right)\right]\right]$$

and for the deterministic case is given by,

$$Q^{\mu}\left(s_t, a_t\right) = \mathbb{E}_{r_t, s_{t+1} \sim E}\left[r\left(s_t, a_t\right) + \gamma Q^{\mu}\left(s_{t+1}, \mu\left(s_{t+1}\right)\right)\right]$$

Hence we assume a gradient based approach where the function $Q^*(s, a)$ is presumed to be differentiable. 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. This is given by,

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

A major challenge of learning in continuous action spaces is exploration. An advantage of off policy algorithms such as DDPG is that we can treat the problem of exploration independently from the learning algorithm. We constructed an exploration policy $\mu'$ by adding noise sampled from a noise process N to our actor policy which was given by,

$$\mu'\left(s_t\right) = \mu\left(s_t | \theta_t^{\mu}\right) + \mathcal{N}$$

**Algorithm 1** DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random 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 loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
      $$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$
    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
```

Figure 3: A snippet of the DDPG algorithm is given[2]

### 1.3 HER

Hindsight Experience Replay[3] is a technique that is used to greatly reduce the number of training episodes required for an agent to begin performing well in an environment [3]. This technique becomes especially crucial for sparse reward formulations on challenging environments, as in these scenarios, it can take numerous episodes before the RL agent has explored enough of the environment to reach the goal state and begin learning from positive rewards.

The key idea behind Hindsight Experience Replay is the notion of learning from failed attempts, as they too contain information that can help the agent learn. If we assume that the state we reached was the goal, then we now know how to get to that state.

In the case of standard experience replay, we would add experiences along this trajectory, given by $(s_t, a_t, r_t, s_{t+1})$. With Hindsight Experience Replay, the idea is to consider our last achieved state $S_N$ to be the goal state $S_G$. This means that the trajectory can now be considered to have been successful in achieving its goal, and the agent can receive a positive reward at the end. Thus, we would set $S_G = S_N$ and have a new set of experiences given by $(s'_t, a_t, r'_t, s'_{t+1})$ where the states and rewards are altered to reflect the new goal. This new tuple is then added to the replay buffer.

This technique works well for some environments, but the performance has been shown to improve with the use of a different variant known as *future*. In this variant, we iterate over every state-action pair in our trajectory and pick K state-action pairs that are encountered after this pair, to be used as the substitute goals for HER. This is as opposed to using only the last state-action pair in the trajectory as the substitute goal for all pairs. Thus, for each of the K substitute goals associated with a state-action pair, we store all the pairs leading up to the substitute goal into the replay buffer. This effectively gives us far more hindsight experiences (proportional to the value of K) each including state-action pairs that receive positive rewards, thereby allowing for much faster learning. After implementing and comparing the performance of both strategies, we decided to use the *future* strategy in our final implementation.

**Algorithm 1** Hindsight Experience Replay (HER)

```
Given:
  • an off-policy RL algorithm $\mathbb{A}$,                    ▷ e.g. DQN, DDPG, NAF, SDQN
  • a strategy $\mathbb{S}$ for sampling goals for replay,      ▷ e.g. $\mathbb{S}(s_0, \dots, s_T) = m(s_T)$
  • a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$.   ▷ e.g. $r(s, a, g) = -[f_g(s) = 0]$
Initialize $\mathbb{A}$                                          ▷ e.g. initialize neural networks
Initialize replay buffer $R$
for episode = $1, M$ do
    Sample a goal $g$ and an initial state $s_0$.
    for $t = 0, T-1$ do
        Sample an action $a_t$ using the behavioral policy from $\mathbb{A}$:
            $a_t \leftarrow \pi_b(s_t||g)$                       ▷ || denotes concatenation
        Execute the action $a_t$ and observe a new state $s_{t+1}$
    end for
    for $t = 0, T-1$ do
        $r_t := r(s_t, a_t, g)$
        Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in $R$     ▷ standard experience replay
        Sample a set of additional goals for replay $G := \mathbb{S}($current episode$)$
        for $g' \in G$ do
            $r' := r(s_t, a_t, g')$
            Store the transition $(s_t||g', a_t, r', s_{t+1}||g')$ in $R$    ▷ HER
        end for
    end for
    for $t = 1, N$ do
        Sample a minibatch $B$ from the replay buffer $R$
        Perform one step of optimization using $\mathbb{A}$ and minibatch $B$
    end for
end for
```

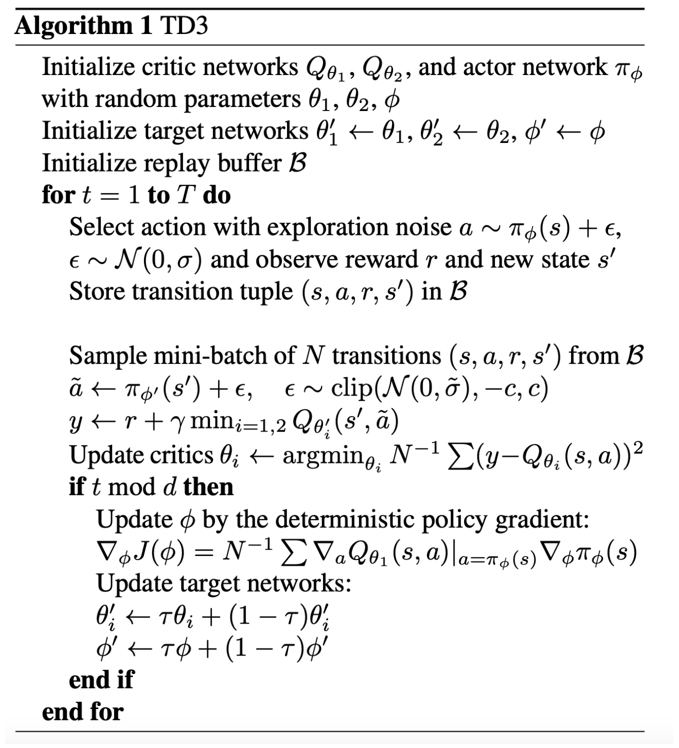
### 1.4 TD3

DDPG[2] has been shown to be unstable in many cases. Often, the critic network overestimates Q values. Policy updates on high-error states cause divergent behavior. To alleviate the instability of TD3 [4], three main changes were proposed by Fujimoto et al.:

- There are two critic networks. We take the Q value to be the min of the values produced by the two networks.
- We add noise to the target action (which is then used to calculate the target Q value). This prevents the actor from exploiting incorrect values.
- The actor networks are updated less frequently. The paper suggests updating half as often as the critic networks.

## 2 ExTD3-Extended TD3

Previous works have shown that TD3[4] is more stable than DDPG[2] because of the presence of delay as well as multiple critic networks. We will be trying a modification on top TD3 algorithm. Instead of 2 critic networks that are used to calculate 2 Q values, we will be introducing 3 critic networks. This will try to improve the stability of the



```
Algorithm 1 TD3
  Initialize critic networks $Q_{\theta_1}, Q_{\theta_2}$, and actor network $\pi_\phi$
  with random parameters $\theta_1, \theta_2, \phi$
  Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$
  Initialize replay buffer $\mathcal{B}$
  for $t = 1$ to $T$ do
    Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
    $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward $r$ and new state $s'$
    Store transition tuple $(s, a, r, s')$ in $\mathcal{B}$

    Sample mini-batch of $N$ transitions $(s, a, r, s')$ from $\mathcal{B}$
    $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
    $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
    Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
    if $t$ mod $d$ then
      Update $\phi$ by the deterministic policy gradient:
      $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
      Update target networks:
      $\theta'_i \leftarrow \tau \theta_i + (1 - \tau)\theta'_i$
      $\phi' \leftarrow \tau \phi + (1 - \tau)\phi'$
    end if
  end for
```

Figure 4: TD3 algorithm is given[4]

existing algorithm as we take the minimized value from the 3 critic networks. The majority of the ExTD3 algorithm follows the traditional TD3 approach.

```
Algorithm 1: Extended TD3
Initialize critic networks $Q_{\theta 1}, Q_{\theta 2}, Q_{\theta 3}$ and actor network with $\theta_1, \theta_2, \theta_3, \phi$;
Initialize Target networks $\theta'_1 = \theta_1, \theta'_2 = \theta_2, \theta'_3 = \theta_3, \phi' = \phi$
Initialize replay buffer
while t=1 to T do
    select action with exploration noise a and observe reward $r$ and new state $s'$;
    Store transition tuple $(s, a, r, s')$ in $B$;
    Sample mini batch from $B$;
    Perform actions similar to TD3 algorithm but considering the 3 critic networks;
end
```

## 3 Results and Experimental Evaluation

All the following experiments were run on Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz with accelerated computing using Nvidia GeForce GTX 1060. We began with a baseline DDPG implementation [5]. This was able to work on very basic environments, but we added normalization for the states in order to achieve any results with FetchReach.

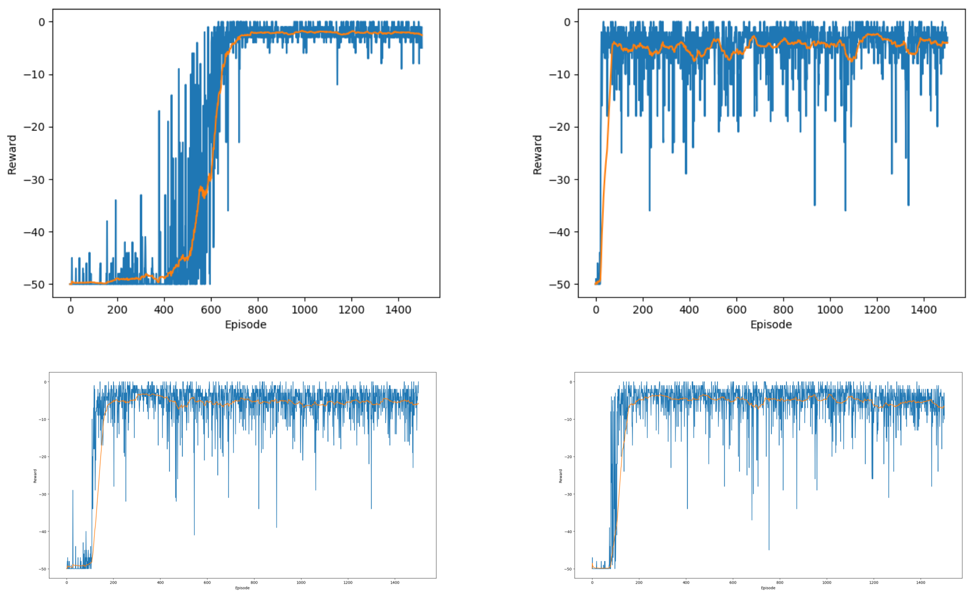


Figure 5: Top right: Vanilla DDPG, Top Left: DDPG w/ HER, Bottom Left: TD3 w/ HER, Bottom Right: Extended TD3 w/ HER

These results were obtained by tuning the network parameters as well the noise parameters for what we determined to be decent exploration/exploitation ratio for the agennt.The agent began to learn the task between episodes 600 to 800 and the performance saturated at after episode 850 after which the agent was able to perform the task quite successfully.

Following DDPG, we added Hindisght Experience Replay to our agent which enabled the agent to learn from failed experiences. The agent saw a significant boost in performance using HER and the following results were obtained. As is evident from the plot, the agent learned the task fairly quickly compared to just using DDPG. It began learning within 50 episodes and the performance more or less saturated close to 100 episodes. There was a lot of oscillations in performance and we believe it may be due the exploratory nature of the agent which may need to scheduled in a better way to favor exploitation as the episodes increase.

Following DDPG with HER, based on our literature review, we also decided to implement Twin-Delayed DDPG or TD3. The agent's performance was significantly better than DDPG but was similar to DDPG with HER. Following are the results for FetchReach using TD3 with HER. It can be noticed that even though the agent started learning slightly later than DDPG with HER (at around 150 episodes), the saturated performance of the agent, which it achieved at about episode 250 (again much quicker than vanilla DDPG) was much more stable than DDPG with HER. This could be attributed to the two critics in the algorithm which stabilise the performance.

As an addition to the TD3 algorithm, we decided to employ another critic in the algorithm to determine if addition of another critic network further smoothed stabilisation. The results show that the algorithm started learning a little quicker than TD3 with HER but there was not any visible stabilisation. The results have been summarized in the table below,

| Algorithm | Starts Learning (Episodes) | Performance Saturation (Episodes) | Approximate Final Reward (avg) |
|---|---|---|---|
| DDPG | 600 | 800 | -6.5 |
| DDPG w/ HER | 100 | 300 | -5.5 |
| TD3 w/ HER | 150 | 300 | -5.8 |
| ExTD3 w/ HER | 100 | 250 | -5.2 |

The difference in performance between DDPG w/ HER, TD3 w/ HER and Extended TD3 w/ HER is negligible and as such no definite conclusion can be drawn apart from the fact that Twin-Delayed DDPG style algorithm does seem to be more stable.

## 4 Conclusion

While DDPG is a powerful algorithm that allows us to work with continuous action spaces, we found that normalization as well as HER was required to achieve good performance with FetchReach.

We were unable to achieve results on more complicated robotic environments such as FetchPush. In this environment the robotic arm has the goal of pushing a box to a specific location. The reason our algorithm failed might be because we did not experience enough states where the robotic arm made contact with the box. This may have been due to insufficient exploration. Indeed, the HER authors noted that they initialized half of the states in the FetchPickAndPlace environment with the box already in the grasp of the gripper [3].

Finally, we would like to thank our mentor Bhuvan Agrawal for helping us out with the project. Also, we would like to thank our professor Bhiksha Raj for giving us the opportunity to complete the project.

## 5 Experiment details

In this section we provide more details on our experimental setup and hyperparameters used

### 5.1 State-goal distributions

The initial arm configuration, and thereby, position of the gripper is fixed. The goal, which is a 3D cartesian coordinate located within the reachable workspace of the arm is set randomly by the environment each episode.

### 5.2 Network architecture

Both actor and critic have three linear layers of size 256. Adam optimizer was used with learning rate 1e-3. The actor took in size (state dimension) and output size (action dimension). The critic took in size (state dimension + action dimension) and output size 1, which was the Q value.

### 5.3 Training procedure

We trained all scenarios for 1500 episodes. Each episode contains 50 timesteps. We update the network every timestep, as long as the buffer has the minimum number of elements required for update i.e. batch size we are sampling.

### 5.4 Normalizing

Normalizing is necessary based on our experiments with DDPG and fetch environment. Initially, the deep network did not learn or was very unstable without normalization. We normalized both the actions and the states that we get from the environment during training. We used inbuilt wrapper provided for OpenAIGym for action normalizing and wrote our own normalizer for states.

### 5.5 Exploration

In order for the agent to be able to explore its environment instead of using the same policy each episode, we add noise with the predicted action. This is a key step in the compromise between exploration and exploitation. We decided to implement Ornstein–Uhlenbeck Noise as it has been shown to work well for similar problems [2]. We varied parameters such as the noise standard deviation (beginning at $\sigma_{max} = 0.3$ and decaying down to $\sigma_{min} = 0.1$ over the 50 timesteps of the episode). Another parameter that is important to tune in OU noise is $\theta$, which varies the tendency of the noise to drift over time. Tuning these parameter turned out to be crucial to getting the model to perform well.

## References

[1] Deeplizard : Reinforcement learning - goal oriented intelligence. `https://deeplizard.com/learn/video/my207WNoeyA`.

[2] Alexander Pritzel Nicolas Heess Tom Erez Yuval Tassa David Silver Daan Wierstra Timothy P. Lillicrap, Jonathan J. Hunt. Continuous control with deep reinforcement learning. 2016.

[3] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay, 2017.

[4] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. `https://arxiv.org/abs/1802.09477`, 2018.

[5] Chris Yoon. Deep deterministic policy gradients explained. `https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b`, 2019.