Continuous Torque Control of an Anthropomorphic Arm with Deep Reinforcement Learning

Athanasios Raptakis
Raptakis@stud.uni-heidelberg.de
athraptakis@hotmail.com
Msc. Scientific Computing
Matrikel_Nr: 3512295

Abstract

In this project the task was to control a 3-link Anthropomorphic arm with Deep Reinforcement Learning. The method of Torque Control was chosen in order to control the movement of the robot. The goal is to learn a deterministic Policy which moves the robot from its fixed initial position in order to "reach" and "touch" an item randomly placed inside a predetermined cubic box. The first task was to develop in python a simple training environment which should include the robot and the rewards. For this purpose, the dynamic and kinematic equations of motion for the robot were derived based on Newtons laws and simulation of the robot was done at each training step with Runge Kutta numerical methods (ode45). Also rewards had to be shaped appropriety in order to train the robot for this specific task. Then Deep Deterministic Policy Gradient (DDPG) method was used to successfully teach the robot how to reach random targets in its workspace. The policy network read measurements about the robot and the target, and it provided Torque to the robot joints in order to reach the target successfully.

1 2

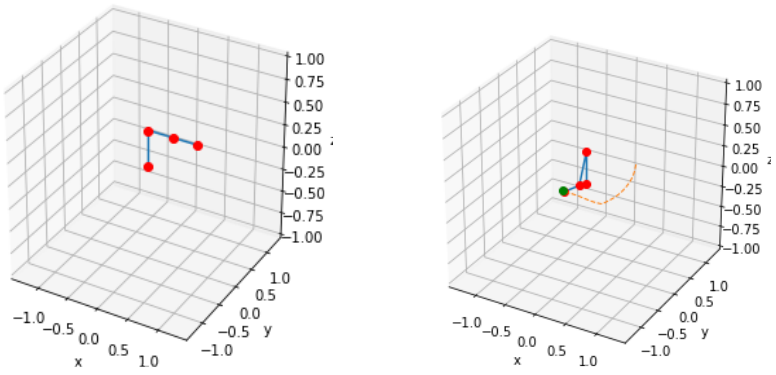
[1] You can see the trained policy at: Youtube Video of Trained Policy
[2] online Repository with the Code so far: GitHub Repository

Project for the class Human Reinforcement Learning: Algorithms and Hands-on Practice.

15 September 2019

1 Task Description and Control Scheme

1.1 Task Description: Random Target Reaching



(a) Robot in Initial configuration (b) Robot in Final configuration

Figure 1: Robot starts from initial position and reaches a random point (green) inside a cube with side = 0.2

In this project the task is to control a 3-link Anthropomorphic arm with the Deep Deterministic Policy gradient (DDPG) Method described in [1]. The goal is to train a policy network in order to solve the "Random Target Reaching" task described at [2].

- During "Random Target Reaching" a randomized target position is sampled from a cube of 0.2 m, providing diverse targets for our robotic arm to reach. More specifically, the robotic arm is starting always from the same "initial configuration" with angles $q = [0, 0, 0]$ and zero angular vellocity $\dot{q} = [0, 0, 0]$, and it tries to "turn" and "touch" the randomly generated target.

- The task is successfull if the arm can reach within 5 cm of the target.

- Success rate is computed from 100 random test episodes where an episode is successful if the arm can reach within 5 cm of the target.

1.2 Control Scheme

For each time step t our policy should compute a torque vector $u(t)$ of size [3x1]. The input to our Policy net is a concatenation of joint angles, angular vellocities, angular accellerations, the robot's End-Effector position and the position of the Target. Input = $[q(t), \dot{q}(t), \ddot{q}(t), (robot_x(t), robot_y(t), robot_z(t)), (x_{target}, y_{target}, z_{target})]$ of size [15x1]. Then the robot is moved by the Torque for time dt and produces the $q(t+dt), \dot{q}(t+dt), \ddot{q}(t+dt), (robot_x(t+dt), robot_y(t+dt), robot_z(t+dt))$. These new measurements are then feed back into the Policy network which produces $u(t+dt)$. In other words the system works like a closed feedback loop.

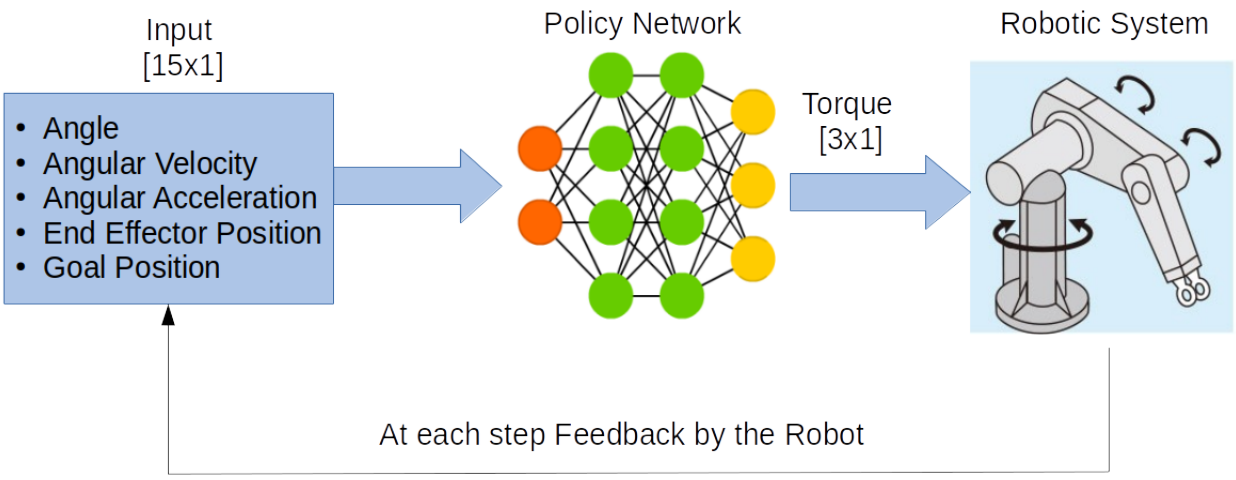


Figure 2: General Control Scheme: A policy network uses as input the joint Angle, angular velocity and angular acceleration, together with the End-Effector position and the Goal position in order to generate Torque for each link. For a robot with three links we have input for the Policy network of size [15x1] and output-Torque of size [3x1] at each time step.

In order to train the DDPG algorithm I chose the following structure for a single episode:

1. Initialise the robot at "Initial Configuration" with angles $q = [0, 0, 0]$ and zero angular vellocity $\dot{q} = [0, 0, 0]$.

2. Sample a new random target = $(x_{target}, y_{target}, z_{target})$ by a cube with side=0.2m.

3. Set $dt = 0.01$ the frequency of our control policy.

4. Set "maximum number of steps" for one episode (in my case 750 steps which means 7,5 sec in real time)

5. Do UNTIL (arm reached 5cm of the target) OR (maximum number of steps reached):

   - Policy Net Input = concatenate$[q, \dot{q}, \ddot{q}, (robot_x, robot_y, robot_z), (x_{target}, y_{target}, z_{target})]$ (a vector of size [15x1]).
   - The Policy Network calculates the Torque $u(t)$ for each robot link, a vector of real values between $[-1, 1]$ (size [3x1]).
   - Added Gaussian noise is added to the Torque u(t) in order to encourage exploration. I used mean=0 and std=5 and I was making the std smaller and smaller which I discuss later in the report.
   - The robot movement is calculated for a time duration of $dt = 0.01$ sec, by numerical integration by a Runge-Kutta method [4] (ode45). We solve an initial value problem by integrating the robot's equations of motion for dt. The initial value for the integration is the current state x_t and the integration will produce the "next state" x_{t+1} and then we calculate the kinematics to determine the robot's end effector position $(robot_x, robot_y, robot_z)$.
   - The environment gives a reward r_i based on the distance between the robot and the target.
   - The current state x_t, current action u_t, the reward r_t and the next state x_{t+1} are all saved in a Replay buffer [1] in order to be used for DDPG training.

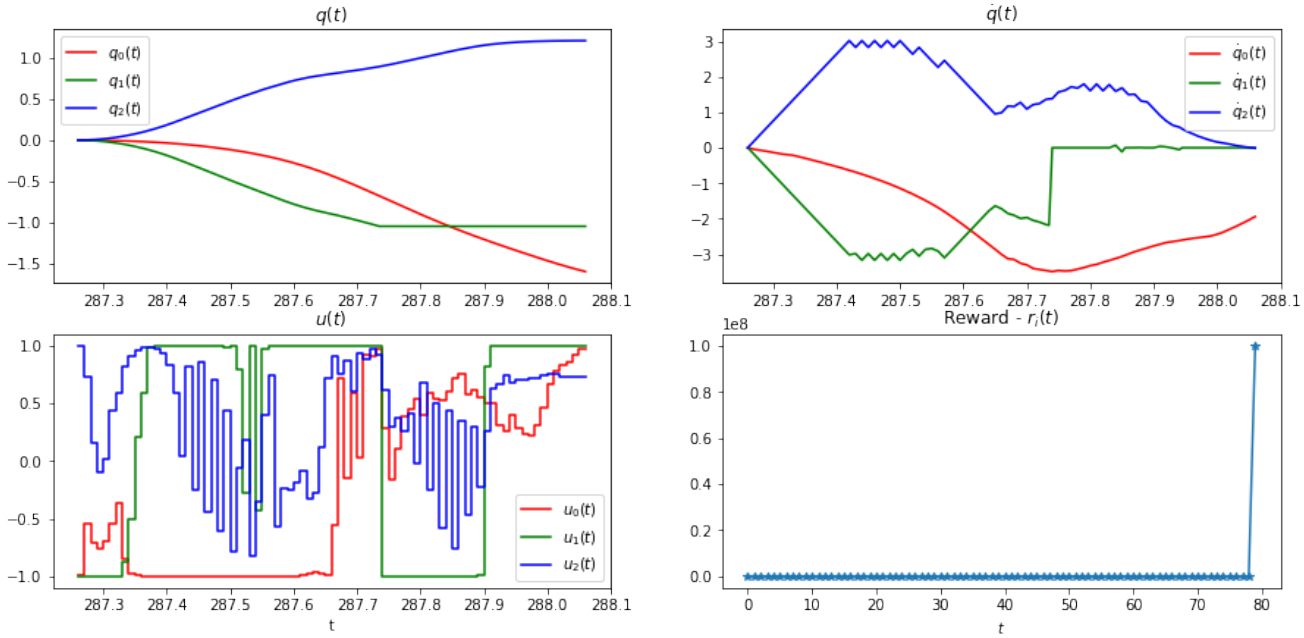


Figure 3: In this figure you can see what happens during one Episode. Torque $u(t)$ produced by the learned Policy Network is given as input to the Robot. Robot Joint Angles $q(t)$, angular Velocities $\dot{q}(t)$ are calculated by numerical integration (ode45) for intervals of dt = 0.01 sec. Each Episode is 7,5 seconds = 750 steps in total. Also you can see the Reward at each step.

1.2.1 Robot Equations of Motion - Constraints -Kinematics

During this step the dynamics of the robot are calculated by solving an initial value problem by using the ordinary differential equation of the form:

$$B(q) \cdot \ddot{q} + C(q, \dot{q}) \cdot \dot{q} + g(q) = Torque(t)$$

I calculated the Equations of motion using the "Lagrange Formulation" methodology which uses the Kinetic and Potential energy of the robot in order to derive the equations of motion. Vector q is the joint angles and $\dot{q}, \ddot{q}$ are the angular velocity and acceleration. I used chapter 7 "Dynamics" and chapter 2 "Kinematics" of the book [3]. The dynamics B, C, g matrices where calculated symbolicly at symbolic toolbox and then lamdified. The dynamics can be found at the file Dynamics.py . The kinematics are defined at the robot.py file.

I assumed that the robot has 3 identical links of mass m=0.5 kg and length=0.4 meters. I made the simplifying assumption that the links are just one dimentional rods with all their mass concentrated in the middle of the link.

During simulation I constrained q the link angles and $\dot{q}, \ddot{q}$ the angular velocity and acceleration in order to make the problem more realistic. Also I found out that this makes my Training to converge faster to meaningfull policies. I used the following constraints:

1. $-2\pi \leq q_0 \leq 2\pi, \ -\pi/3 \leq q_1 \leq 4(\pi/3), \ -5(\pi/6) \leq q_2 \leq 5(\pi/6)\ (rad)$
2. $-2\pi \leq \dot{q} \leq 2\pi\ (rad/sec)$
3. $-6\pi \leq \ddot{q} \leq 6\pi\ (rad/sec^2)$

1.3 Reward Shaping

In order to train the policy I had to use appropriate rewards during the training Process. Intuitively I chose a reward function that rewards the robot when it is close to the objective and gives no reward when the end effector is far away from the objective. A very big final reward is given when the End Effector is inside 5 cm from the Goal, which is considered to be the terminal state. The environment gives the following rewards:

1. Reward based to the distance of the robot's end-effector from the Goal position. The closer to the Goal the bigger the reward.
2. Negative reward which punishes generation of really big Torques.
3. Big negative reward and termination of an episode if the robot stops exploring (ex. when the joints are "locked" in min/maximum angle). In this way I avoid putting in my Replay buffer uninformative examples where the robot does nothing usefull. As a result, the Training process converges faster.
4. Intermediate big reward when the end effector is 0.1 meters close to the Goal. (Encourage exploration towards the objective and results in faster convergence during training).
5. Gigantic Final State reward. The final state happens when the End-Effector approaches inside 0.05 meters from the target Goal.

At each training Step:

#Calculate Eucledian distance between End Effector and Goal

$dist = \sqrt{\sum (End\_Effector\_Position - Goal\_Position)^2}$

$c_1 = 1000$
$c_2 = 100$

#First term rewards being close to the Goal
#Second term punishes big Torques

$reward = c_1 * min[1.0, (1.0 - dist/0.8)] - c_2 * L2\_norm(Torque)$

If the robot stops exploring (ex. is stuck at max/min angles)

if $velocity$ is very small:
    Give additional negative reward $r = -5000$
    Stop the Episode

#Intermediate reward / Remove it during Fine Tuning/ Don't make it too big

if $(dist < 0.1)$:
    $r = 5000$

#Give really big Final state reward

if $(dist < 0.05)$:
    $r = 100000000$
    $d = 1$ #d=1 means we are in Terminal state

return $reward + r, d$

2 Deep Deterministic Policy Gradient and Training

2.1 DDPG

In this section "Deep Deterministic Policy Gradient" (DDPG) method is very briefly presented based on [1]. DDPG is an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces. The motivation for DDPG is to provide a solution for problems with high-dimensional observation spaces and continuous (real valued) and high dimensional action spaces. Such an application is the problem of physical Control of systems like robots, cars, etc.. DDPG combines the actor-critic approach with insights from the Deep Q Network (DQN)[5]. DQN introduced two innovations in order to learn value functions robustly and stably: 1. the network is trained off-policy with samples from a replay buffer to minimize correlations between samples; 2. the network is trained with a target Q network (Q') to give consistent targets during temporal difference backups.

Algorithm 1 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 $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
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

The target networks Q', μ' are used to calculate y_i in order to make the training stable. The weights of these target networks slowly track the learned networks. This means that the target values are constrained to change slowly, greatly improving the stability of learning. Also it is very important to see that the policy here $\mu(s|\theta^\mu)$ is deterministic and produces real values. For our own application the policy produces the Torque $u = \mu(s|\theta^\mu)$. μ takes as state/input the real valued vector $[q(t), \dot{q}(t), \ddot{q}(t), (robot_x(t), robot_y(t), robot_z(t)), (x_{target}, y_{target}, z_{target})]$ of size [15x1] and calculates a vector of size [3x1] with real values of Torque $u \in [-1, 1]$Nm.

2.2 Actor - Critic Architecture and Training

- I used Adam for learning the neural network parameters with a learning rate of 10^{-4} and 10^{-3} for the actor and critic respectively. For Critic (Q) I included L2 weight decay of 10^{-2} and used a discount factor of $\gamma = 0.99$. For the soft target updates I used $\tau = 0.001$.
- The Policy was Trained for 20000 Episodes. Each Episode has 750 steps of duration $dt = 0.01$ sec each. This means that the Policy Network is giving input to the robot with a frequency of 100 Hz.
- I used Scheduling for the learning rates. The learning rates are updated every 5000 episodes by a simple formula $learning\ rate = 0.1 \cdot learning\ rate$.

I used Gaussian Noise as Exploration Noise with a mean = 0 and standard deviation that becomes smaller and smaller. The initial value for std=5 and it gets smaller together with the learning rates every 5000 episodes by a simple formula $noise\ std = 0.5 \cdot noise\ std$.

- The neural networks used Relu for all hidden layers. The final output layer of the actor was a tanh layer, to bound the actions between -1 and 1 Newton Meters. The Networks have 2 hidden layers with 400 and 300 units, respectively. Actions were not included until the 2nd hidden layer of Critic Q. The final layer weights and biases of both the actor and critic were initialized from a uniform distribution $[-3 * 10^{-3}, 3 * 10^{-3}]$. This was to ensure the initial outputs for the policy and value estimates were near zero. The other layers were initialized from uniform distributions $[-\sqrt{\frac{1}{f}}, \sqrt{\frac{1}{f}}]$ where f is the fan-in of the layer.
- I used a "WarmUp" period. I waited until 5000 examples were added to my Replay Buffer before starting the training of the Policy.
- I trained with minibatch size of 32.
- I used a replay buffer size of 10^6.

2.3 Fine Tuning the Policy

I found out that by giving an intermediate reward when the End Effector is 0.1 meters close to the objective speeds up the convergence during the training process. Unfortunately this led my policy network to learn how to accumulate multiple "intermediate rewads" instead of going directly for the final state. In order to tackle this problem I fine-tuned my policy after training. I just continued training the policy by turning off the intermediate reward and by using a clean new Replay buffer. Also I made my Final reward much bigger than the traing process to encurage the robot to move directly towards the Goal position. I used exactly the same method with training but this time for 50000 Episodes in total and updates in the learning rates and noise std every 10000 Episodes.

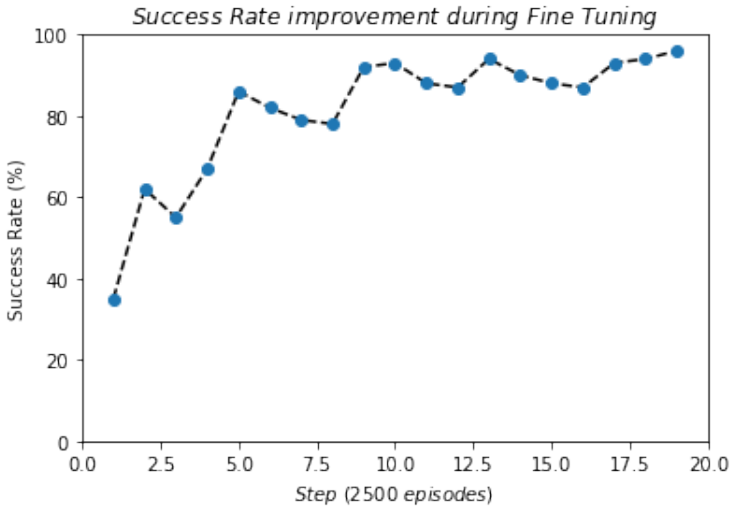


Figure 4: Success Rate improvement during the Fine Tuning stage.

We can see that after the fine tuning the robot has a success rate 96%.

3 Conclusion and Discussion

- During this Project I successfully trained a realistic 3 link robotic manipulator with DDPG method. After the training and fine-tuning procedures the robot has a success rate of 96%. I believe that the success rate would improve even more, if I had run the algorithm for more iterations.
- The Robot initially learned a "bad behaiviour" by spending all the 750 steps during an episode to collect small "intermediate" rewards. It would go for the final target (< 5 cm) only 40% of the episodes. 58% of the testing episodes the robot stayed exactly inside 10cm collecting "small" rewords. This happened because my Final reward was not big enough. In order to solve this problem I continued the training by "fine-tuning" without "intermediate" rewards and by making the final reward much much larger.
- In general the learned policies look intuitive and usefull. The arm learns to move smoothly and with relatively slow speed. One could say that the arm has learned to move in an "elbow down" way which is intuitive when we additionally want to minimise the Torque. Humans also have the hands with "elbow down" because the potential energy is minimised. The only "kink" we can observe in the trajectory is when the shoulder reaches the minimum angle. This happens because I assumed that when a joint reaches its max or min limit it hits a "hard stop" and the shoulder joint velocity becomes also zero. Sometimes the end-effector will "overshoot" the target but in the most cases iit will compensate and reach the target successfully.
- In general while one episode has 750 steps the robot reaches the target within 100 steps which is very fast. We can think that because doing each action costs a negative reward, the robot tries to complete the Episode as fast as posible.
- The above competes with the negative reward that punishes the robot for having big torques. Probably the robot would have a slower motion if I had increased the c_2 parameter in my reward shaping. But unfortunatelly the torque is not the only parameter affecting the speed. For example the robot free-falling under gravity with zero torque input would also move quickly. Thus punishing the high torque only affects speed of movement indirectly.
- I used Gaussian noise for exploration. I found that putting large noise std makes convergence faster in the first steps of the training, while small noise std benefits the fine tuning stage.
- Initially I tried to force the robot to learn good behaiviours with low speed and small angles indirectly. I was punishing dangerous movements of high speed or "crazilly swinging" the arm around. But it was impossible for the algorithm to converge in any meaningfull solution. Generally, it is not clear how to shape the rewards to teach the robot a reasonable behaviour while still accomplishing the target. Thus, I used only the few rewards that I have already explained in a previous chapter together with bounds in the robots state space.
- I found out that by constraining the robots state space $q, \dot{q}, \ddot{q}$ the algorithm converges much faster. Constraining how much and how fast a joint can turn is not only realistic but also means that the state space is bounded and not "infinitelly" big. The robot will explore sufficiently a bounded state space and it will have enough time to learn by usefull informative examples.

4 References

[1] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra, "Continuous control with deep reinforcement learning" (https://arxiv.org/abs/1509.02971)

[2] Shixiang Gu Ethan Holly, Timothy Lillicrap and Sergey Levine, "Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates" (https://arxiv.org/pdf/1610.00633.pdf)

[3] Siciliano, B., Sciavicco, L., Villani, L., Oriolo, G., "Robotics Modelling, Planning and Control"

[4] https:en.wikipedia.orgwikiRungeKutta_methods

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller "Playing Atari with Deep Reinforcement Learning" (https:arxiv.orgpdf1312.5602.pdf)