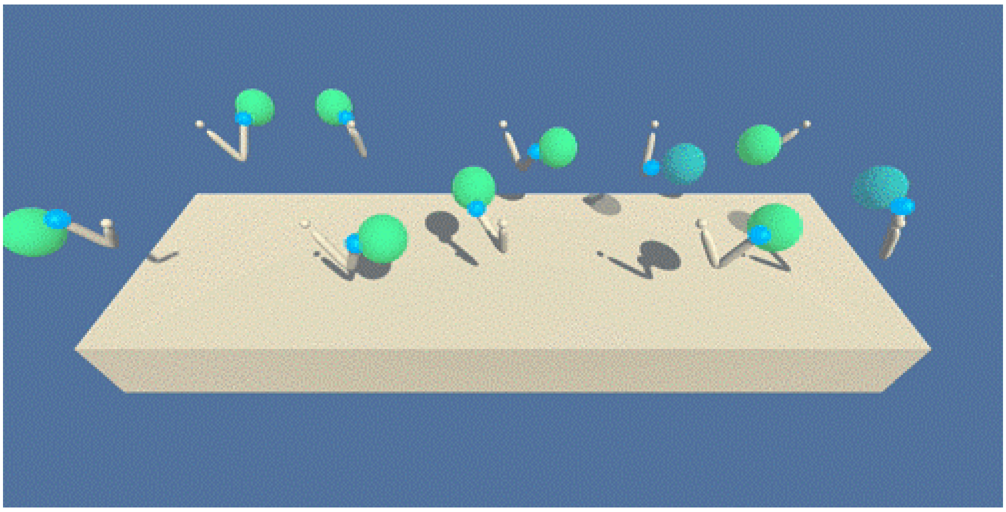
Project 2 - Continuous Control

Deep Reinforcement Learning Nanodegree

M. Lippl

March 22, 2021



1 Introduction

This project illustrates the so-called Policy methods of reinforcement learning. The setting is a scenario where one (alternatively 20 simultaneous) robot arms with 4 degrees of freedom corresponding to two joints ('shoulder' and 'elbow') need to be trained to follow a sphere which orbits around their shoulder mount. Each arm is rewarded with 0.1 per time step points as long as the abstract "hand" stays in the target region. The following image illustrates this concept:

As indicated before, one may solve this challenge via two different paths: The first one involves training only one single arm, whereas the second one utilizes 20 independent arms simultaneously.

At the end of each episode, the score of each individual arm is calculated, and the final score of this episode is calculated as the average over these 20 scores (which me may call the 'episodic average'). The challenge is considered solved, if the average score of the episodic averages is greater than +30 over 100 consecutive episodes. In formulæ:

$$\text{avg}_N(i) = \frac{1}{N}\sum_{j=0}^{N} \overline{s}_{i-j} \overset{!}{>} 30$$

where $\overline{s}_i$ is the mean score over all arms in episode i. Of course, in the case of one single arm, the mean score over all arms is only the score itself, i.e. $\overline{s}_i = s_i$.

2 Learning Strategy

The challenge's desciption by Udacity suggests three different learning strategies. These are:

1. Proximal Policy Optimization (PPO) [1]
2. Asynchronous Advantage Actor-Critic (A3C) [2]
3. Distributed Distributional Deep Deterministic Policy Gradient (D4PG) [3]

We will make a short digression and outline different concepts of policy gradient methods in the following section.

2.1 Policy Gradient

Recall that the utility function is defined as the expectation value:

$$U_\theta(\tau) = \mathbb{E}_\theta\left[\sum_{t=0}^{T} r(s_t, a_t)\right]$$

where $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$ is the path the agent has taken. It can be shown that

$$U_\theta = \mathbb{E}\left[\mathbb{E}\left[Q_{P_\theta}(s_0, a_0)\right]\right],$$

which means that, assuming that all initial states are of equal probability, maximizing the action-value function Q is equal to maximization of the utility function U.

Taking the sum over all possible paths τ, and incorporating the probability $P_\theta(\tau)$:

$$U_\theta = \sum_\tau \left(P_\theta(\tau)\left(\sum_{t=0}^{T} r(s_t, a_t)\right)\right)$$

Using the Reinforce-log-trick, we arrive at:

$$\nabla_\theta U_\theta = \mathbb{E}\left[\sum_{t=0}^{T}\left(\nabla_\theta \log(P_\theta(a_t \mid s_t))\left(\sum_{t=0}^{T} r(s_t, a_t)\right)\right)\right]$$

The REINFORCE-trick is a simple substitution which is well-known from the field of statistical thermodynamics.

$$p(\tau; \theta, \phi) = \prod_{t=1}^{T} p(r_t, s_{t+1} \mid s_t, a_t; \phi)\pi(a_t \mid s_t; \theta)$$

Utility function:

$$\mathbb{E}_{p(\tau|\theta,\phi)}[R(\tau)]$$

We'd like to perform a gradient ascent into the direction of the utility function's gradient:

$$\hat{g} = \nabla_\theta \mathbb{E}_{p(\tau|\theta,\phi)}[R(\tau)]$$

Formally, this leads us to the update equation for the weights,

$$\theta \longleftarrow \theta + \alpha\hat{g}.$$

The objective is to maximize this function which can be done by using gradient ascent. In order to evaluate this equation, we are taking a closer look at the gradient. First of all,

$$\hat{g} = \nabla_\theta \mathbb{E}_{p(\tau|\theta,\phi)}[R(\tau)] \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta[R(\tau_i)]$$

Using the identity $\nabla_x \log x = 1/x$, we can write

$$\nabla_\theta p(z;\theta) = p(z;\theta)\nabla_\theta \log p(z;\theta).$$

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta,\phi)}[R(\tau)] \approx \frac{1}{N}\sum_{i=1}^{N}[R(\tau_i)\nabla_\theta \log p(\tau_i;\theta,\phi]$$

Assuming a single sample trajectory:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta,\phi)}[R(\tau)] \approx R(\tau)\sum_{t=1}^{T} \nabla_\theta \log p(a_t \mid s_t;\theta)$$

Very high variance, so a baseline is needed.

2.2 The Actor-Critic Algorithm

Consider the gradient of the utility function again:

$$\nabla_\theta U_\theta = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T}\Big(\nabla_\theta \log(P_\theta(a_t \mid s_t))\Big(\sum_{t=0}^{T} r(s_t,a_t)\Big)\Big)$$

Theoretically, we have:

$$Q_{P_\theta}(s,a) = r(s,a) + \mathbb{E}\Big[\sum_{t}^{T} r(s_t,a_t)\Big]$$

Now using the identity

$$Q_{P_\theta} = Q_{P_\theta} = \sum_{s=t}^{T} r(s_s,a_s)$$

we may substitute:

$$\nabla_\theta U_\theta = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T}\big[\nabla_\theta \log(P_\theta(a_t \mid s_t)) \cdot Q_{P_\theta}(s_t,a_t)\big]$$

Here, $\nabla_\theta \log(P_\theta(a_t \mid s_t))$ is called the *actor*, while $Q_{P_\theta}(s_t,a_t)$ is called the *critic.*

Given a loss function $\mathcal{L}(\theta)$, the weights are updated as follows:

$$\theta \longleftarrow \theta - \beta\nabla_\theta\mathcal{L}(\theta)$$

Using Q-Learning (SARSA-max), out TD-target is given by:

$$Q_\theta(s_t, a_t) \approx r(s_t, a_t) + \max_a \hat{Q}(s_{t+1}, a, \theta)$$

Hence, the gradient of the loss function is:

$$\nabla_\theta L(\theta) = -\Big(r(s_t, a_t) + \max \hat{Q}(s_{t+1}, a, \theta) - \hat{Q}(s_t, a_t, \theta) \Big) \nabla_w \hat{Q}(s_t, a_t, \theta)$$

The actor network is updated using the update equation, while the critic network is updated using the loss function.

2.3 Advantage-Actor-Critic (A2C)

A second approach to minimizing the variance of the gain is to subtract a baseline b which is independent of both the action a and the parameter θ. If we consider the gradient of the utility function,

$$\nabla_\theta U(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \left(\sum_{t=0}^{T} \Big[\nabla_\theta \log(P_\theta(a_t \mid s_t)) \cdot \big(\hat{Q}_{P_\theta}(s_t, a_t, \theta) - b \big) \Big] \right)_i ,$$

then we can easily see, that by decreasing the expression in the right pair of brackets, we lower the contribution of the gradient of the log-probability, $\nabla_\theta \log P_\theta$. A natural candidate for this baseline is the well-known state-value function V, i.e.

$$b_t = V(s_t).$$

This leads us to the definition of the Advantage function A,

$$A(s, a) = Q(s, a) - V(s).$$

In other words, we are taking the (state-)value-function $V(s)$ as the baseline. Substituting the bias in the above gradient equation yields Advantage-Actor-Critic:

$$\nabla_\theta U(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=0}^{m} \left(\sum_{t=0}^{T} \Big(\nabla \log(P) \hat{A}(s_t, a_t, \theta) \Big) \right)_i$$

Again, update is given by:

$$\theta \longleftarrow -\beta \nabla_\theta L(\theta)$$

Main downside of the REINFORCE algorithm is that the gradient estimator is a Monte-Carlo estimator

$$b = \hat{V}(s, \theta) = \mathbb{E}\Big[\hat{Q}(s, a, \theta) \Big]$$

$$\hat{A}(s, a, \theta) = \hat{Q}(s, a, \theta) - \hat{V}(s, \theta)$$

Approximate the state function via an ANN:

$$\hat{V}(s, \theta) \approx \hat{V}(s, \theta)$$

Advantage function can then be written as:

$$\hat{A}(s, a, \theta) \approx \hat{A}(s, a, \theta, \theta_v) = \hat{Q}(s, a, \theta) - \hat{V}(s, \theta_v)$$

The loss function's gradient is

$$\nabla_\theta \mathcal{L}(\theta) = -\hat{A}_{P_\theta}(s, a, \theta)\nabla_\theta \hat{V}(s, \theta)$$

and therefore, the update equation is given by:

$$\theta \longleftarrow \theta - \beta \nabla_\theta \mathcal{L}(\theta)$$

2.4 Deterministic Policy Gradient Algorithm (DPG)

The Deterministic Policy Gradient method (DPG) is the natural continuation of DPG using ANNs as function approximators. It uses two different networks network-pairs, each pair consisting of a local and target-network, as familiar from DQL: One network-pair to learn the policy (the "Actor") and one pair to learn the value-function (the "Critic").

Let the target policy be denoted by μ_θ. This policy is considered deterministic, i.e. $\mu_\theta(s) = a$. Define the behavior policy by adding noise:

$$b_\theta(s) = \mu_\theta + \mathcal{P}_t$$

where $\mathcal{P}$ is a stochastic process of our choice. In our case, this can be either a Wiener process a Ornstein-Uhlenbeck-process which is described below.

Let μ_θ be the target policy. Then the actor network is updated as follows:

$$\theta \longleftarrow \theta + \alpha_\theta \nabla_\theta \hat{Q}_\theta(s_t, \mu_\theta(s_t), \theta)$$

Assuming a continuous action, and therefore being able to apply the chain rule yields:

$$\theta \longleftarrow \theta + \alpha_\theta \nabla_a \hat{Q}_\theta(s_t, a, \theta)\nabla_\theta \mu_\theta(s_t)$$

On the other hand, the critic network is updated as: The TD-error is given by:

$$\delta_{\text{TD}} = r(s, a) + \hat{Q}_\theta(s', \mu_\theta(s'), \theta) - \hat{Q}_\theta(s, a \sim b(s), \theta) - \hat{Q}_\theta(s, a \sim b(s), \theta)$$

This error results in the weight update:

$$\theta \longleftarrow \theta + \alpha_w \delta_{\text{TD}} \nabla_\theta \hat{Q}_\theta(s, a \sim b(s), \theta)$$

2.5 Deep Deterministic Policy Gradient Algorithm (DDPG)

The Deep Deterministic Policy Gradient (DDPG) algorithm is a version of the DPG which has improved adaptation to ANNs. The main two improvements which may not come surprising to those already familar with project 1 ("Navigation") are the use of a replay buffer and the addition of target networks for both the actor and the critic. The motivations for these features are the same as with DQL and DDQL, namely the elimination of training instabilities and reusage of prior experiences.

2.6 Wiener Process

While the formal definition of stochastic processes are rather formal and The Wiener process is defined as follows

1. $W_0 = 0$
2. $W_{t+s} - W_t \geq 0$ for all $t, a \geq 0$
3. $W_{t+s} - W_t \sim \mathcal{N}(0, s)$, where $\mathcal{N}$ is a normal distribution
4. W_t is continuous in t

2.7 Ornstein-Uhlenbeck Process

The Ornstein-Uhlenbeck process is given by

$$dx_t = -\theta x_t dt + \sigma d W_t$$

where $\theta, \sigma > 0$, and W_t denotes the Wiener process.

3 Implementation and Results

3.1 Structure of the Project

As the previous projects before, our project has the simple structure:

```
udacity-rl-p2/
├── checkpoints_torch/
├── Continuous_Control.ipynb
├── agent_torch.py
├── networks_torch.py
├── train.py
└── play.py
```

The Jupyter-notebook `Continuous_Control.ipynb` is self-contained and contains the results of the last experiment. The file `agent_torch.py` contains the definition of the MultiAgent- and Agent-classes. `networks_torch.py` - Definition of the actor- and critic-networks. `train.py` - Script containing setup and the main training loop `play.py` - Script containing the replay loop. [1]

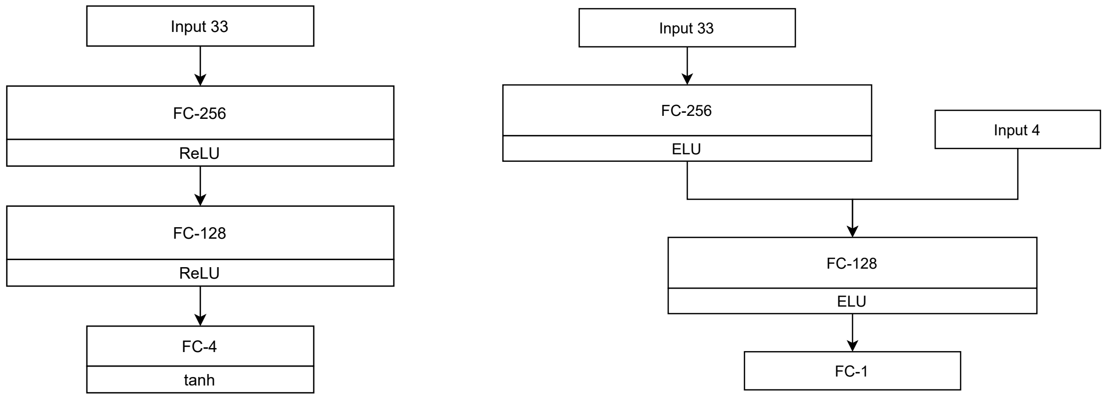
As environment, we have chosen the Reacher20 environment, i.e. started with

3.2 Architecture

The network definition is given in file `networks.py`: The output layer consists of four neurons with a 'tanh'-activation layer.

The architecture of the actor- and critic-networks is very simple. The actor (fig. 1a) basically only consists of two fully connected neural network layers, one with 256 neurons, and one with 128. In the critic (fig. 1b) however, the additional action inputs of the agent are concatenated

[1] Note that, as with project 1, the suffix `_torch` stems from the fact that there have been additional efforts to implement the algorithms using TensorFlow 2.x instead of PyTorch.



(a) Actor (b) Critic

Figure 1: Network architectures used in the implementation

with the state input which were already forwarded through the first fully connected layer of the network.

3.3 Hyperparameters

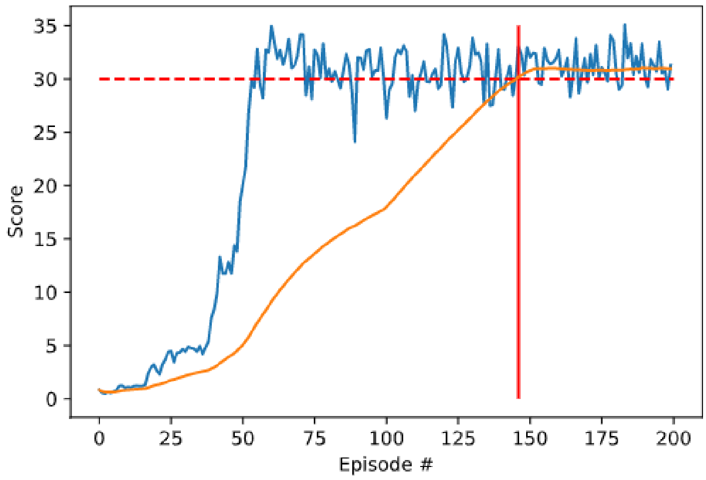
The hyperparameters were as follows:

Hyperparameter	Value	Description/Comment
`BUFFER_SIZE`	10,000	Total size of replay buffer
`BATCH_SIZE`	64	Size of training batches
`GAMMA`	0.98	Discounting factor
`TAU`	0.001	Soft updates
`learn_rate`	$5 \cdot 10^{-4}$	Learning rate of Adam-optimizer
`seed`	5	Manual seed for neural networks
`epsilon`	0.05	Final value during training

3.4 Results

The results of our experiments are illustrated in the shown in `https://github.com/asiopueo/udacity-rl-p2/Continuous_Control.ipynb`.

As one can see in the following graph and has been logged in the Jupyter-notebook, the challenge was solved after 146 episodes:



One fact that immediately sets this graph apart from the other two projects of this course is the apparent stability of the model. The model has reached its full long-term performance after about 60 episodes and does not seem to need any further training afterwards, nor does more training result in catastrophic failure as it can be the case in the multi-agent challenge in project 3.

3.5 Further Research Directions

Investing more time into proper hyperparameter optimization as the margin of safety (i.e. the differnce between the threshold and the trailing average) is relatively thin. According to fellow students, an average score of 35 to 40 points should be possible using our current methodology.

Another interesting path to follow isthe application of the Distributed Distributional Deep Deterministic Policy Gradient (D4PG) method [3] [4].

From a technical point of view, we have started implementing the algorithm using TensorFlow instead of PyTorch as framework. This project has not been fully pursued due to time constraints but will be continued in the future.

References

[1] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017.

[2] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” 2016.

[3] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. Lillicrap, “Distributed distributional deterministic policy gradients,” 2018.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” 2019.