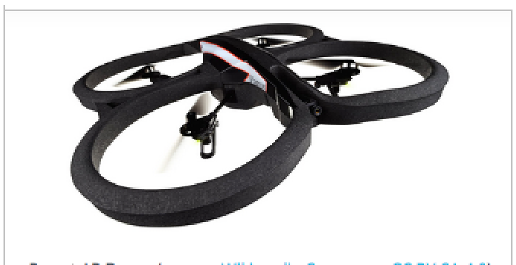
Machine Learning Project : Teach a Quadcopter How to Fly


Parrot AR Drone (source: Wikimedia Commons; CC BY-SA 4.0)

Content

Introduction

Regarding the project title, this Udacity machine learning quadcopter project is a domain example of complex tasks of sensory input. This means, we are working with continuous and high-dimensional action spaces and having a regression problem that shall be solved with a reinforcement learning method. For technical domain details see the Task chapter at the end of this documentation.

In general, the reinforcement learning framework includes an agent that interacts with an environment in discrete time steps.

For the project domain, the quadcopter shall autonomously achieve the control about one task – take-off or landing – as a simulation, not for a real device. To reach this simulation goal a specific reinforcement learning concept is implemented, called actor-critic method using deep Q-learning neural networks and lower-dimensional action spaces. This concept combines the advantages of Policy Gradient and DQNs. According research papers it has a good convergence result.

The following diagram introduces the general reinforcement learning architecture of this actor-critic method.

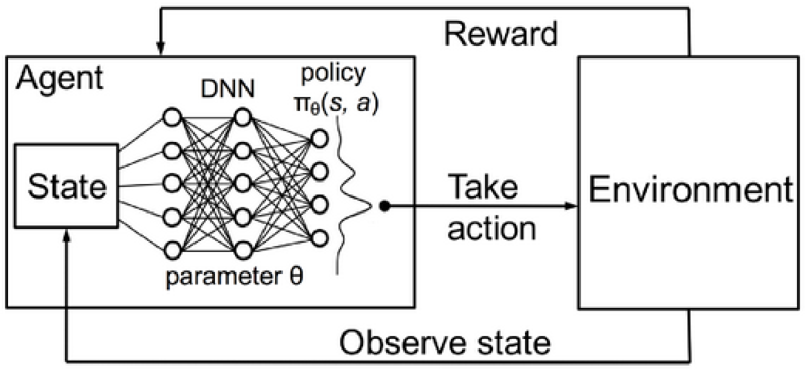


image source
http://people.csail.mit.edu/hongzi/content/publications/DeepRM-HotNets16.pdf

Actor-Critic Architecture

To solve the simulated physics task and to find a high performant policy, as a solution such model-free, off-policy actor-critic method using deep function approximators is choosen, because it can learn policies for such high-dimensional, continuous domain action spaces.

In general, Q-learning as part of this approach, is an off-policy algorithm. In other words, the policy it chooses to select actions, is different from the policy it is learning.

The actor-critic algorithm uses the 'Deep Deterministic Policy Gradient' mechanism, but instead of being tabular based, it is modified by having deep Q-networks as part of the agent component. Such agent component is called DDPG-Agent and includes 2 networks:

- The 'critic' network is a value based evaluation approach how good the choosen action is, in other words, the action is the input to produce the Q values from the DQN to judge about the associated policy
- The 'actor' counterpart predicts on a policy based approach what continuous space action shall be taken as output for its state input

The general schema of the **actor-critic method with advantage** as the chosen principle looks like

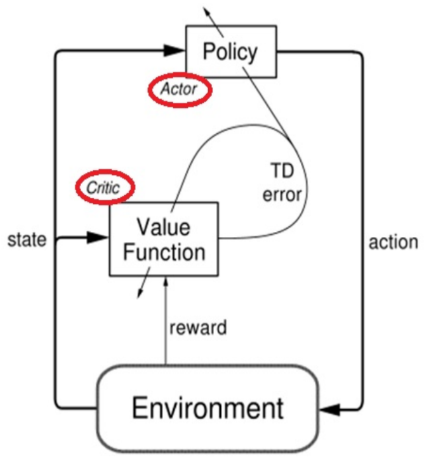


image source
https://www.groundai.com/project/a-brandom-ian-view-of-reinforcement-learning-towards-strong-ai/

The advantage function value stabilises the learning process and can be estimated by the TD error (TD – Temporal Differences). Associated equations for explanation see below in the following training chapter.

Training

The sources of this chapter are:

- https://medium.com/free-code-camp/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d
- the Udacity Advanced Machine Learning course (e.g. images with black background).

During separate training of both networks at each learning step, the actor parameters (with policy gradients and advantage value A(s,a)) and the critic parameters (with minimizing the mean squared error with the Bellman update equation) are improved in parallel.
Because doing an update at each learning step, the general reinforcement learning goal to find the total maximum future reward R(t) of given state-action pair is not possible anymore. The reason is, that this R(t) value is calculated by the reward function in policy gradient at the end of the episode only. Now, our new critic network component approximates such Q value function**.**
And according this concept, both, the actor and the critic can make more efficient use of the interactions with the environment.

In general, for the training of 2 networks, 2 weight sets are necessary:

- for the actor: θ (theta), characterises the policy π (pi), the probability of taken an action from a given state
- for the critic: w, encodes the value q hat of taking that action from that state

$$\text{Policy Update: } \Delta\theta = \alpha \, \nabla_\theta(log\ \pi_\theta(s,a))\underbrace{\hat{q}_w(s,a)}_{\substack{\text{q learning function approximation} \\ \text{(estimate action value)}}}$$

$$\text{Value update: } \Delta w = \underbrace{\beta}_{\substack{\text{Policy and value have} \\ \text{different learning rates}}} \underbrace{(R(s,a) + \gamma\hat{q}_w(s_{t+1},a_{t+1}) - \hat{q}_w(s_t,a_t))}_{\text{TD error}}\underbrace{\nabla_w\hat{q}_w(s_t,a_t)}_{\substack{\text{Gradient of our value} \\ \text{function}}}$$

During training, the critic component computes the value by taking the given action at that state and the actor component updates its policy parameters (weights) using this q value.

By changing the score function to an advantage function A(s,a), the learning will be stabilised:

$$A(s,a) = \underbrace{Q(s,a)}_{\text{q value for action a in state s}} - \underbrace{V(s)}_{\text{average value of that state}}$$

As visible, such function delivers the additional reward compared to the average value of the given state s and associated action a. It can be estimated by the TD error, so, it is not necessary to calculate 2 value functions.

$$A(s,a) = \underbrace{\boxed{Q(s,a)}}_{r+\gamma V(s')} - V(s)$$

$$A(s,a) = \underbrace{r+\gamma V(s') - V(s)}_{\text{TD Error}}$$

How does it work?

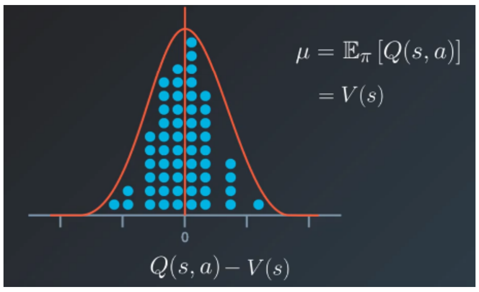
Regarding the policy gradient methods for the policy and value update rules with

$$\triangle\theta = \alpha \, \nabla_\theta \, J(\theta)$$

$$\text{gradient} \quad \nabla_\theta J(\theta) = \underbrace{\mathbb{E}_\pi}_{\text{expectation}} [\nabla_\theta (\log\pi\,(s,a,\theta))\,\underbrace{\hat{q}(s,a,\mathbf{w})}_{\text{variance}}]$$

$$\text{policy update} \quad \triangle\theta = \alpha\,\nabla_\theta\,(\log\pi\,(S_t,A_t,\theta))\,\hat{q}\,(S_t,A_t,\mathbf{w})$$

the expected value is mapped to an associated variance. And by estimation of this expected value, the variance shall be small to have a stable process. As visible, the variance is influenced by the value score function q hat. So, the actor is updated in a gradient direction based on the critic information. Assume its value distribution is a Gaussian one, then its mean is the expected value of Q and we get our new score function called advantage function:



The new score, advantage function value informs about how much more reward is available beyond the expected value of the state.

Finally, using the TD error as estimation of the advantage function, we have as policy gradient:

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t,a_t)$$

Algorithms

General actor-critic algorithm principles are e.g. explained in:

- R. Sutton and A. Barto - *Reinforcement Learning: An Introduction. Second Edition,* MIT Press, Cambridge, MA (II Approximate Solution Methods)
- V. R. Konda and J. N. Tsitsiklis - *Actor-Critic Algorithms,* Laboratory for Information and Decision Systems, MIT, Cambridge, MA, 02139
- T. P. Lillicrap and J. J. Hunt et al. - *Continuous Control With Deep Reinforcement Learning*, ICLR 2016 paper, Google DeepMind, London UK (https://arxiv.org/abs/1509.02971; chapter 3. Algorithm); this paper is mainly used for this Udacity quadcopter project

Implementation

So, the given project files and their concepts are explained in the mentioned ICLR 2016 paper *Continuous Control With Deep Reinforcement Learning* from Timothy P. Lillicrap, Jonathan J. Hunt et al.

The implementation of the concept uses i.a. the Keras library and includes the following files:

- Environment (mainly jupyter notebook file with quadcop kernel and Task)
- Agent factory class with its interfaces to the concrete class
    - DDPG-Agent class
- specific agent Model factory class (don't mix it up with Keras Model class) with its interfaces to the concrete classes
    - Actor
    - Critic

  For the training of the models the Keras backend mechanism is used for implementation.
- OUNoise class: Calculation of Ornstein-Uhlenbeck Process for getting noise values

  OU Process = $dxt = \theta(\mu - xt)dt + \sigma\ dWt$

  Where:

  xt = the particle's current position

  θ = a mean reversion constant

  μ = the mean particle position

  σ = a constant volatility

  dWt = a Wiener process (Brownian motion)
- ReplayBuffer class: fixed-size buffer to store experience tuples
- Task class: defines the goal and provides feedback to the agent
- PhysicsSim class: implements the physical basics (parameters and functions)

As mentioned, the factory design pattern is used to create specific class instances of the abstract superclass type. Advantages of the factory design pattern are amongst others:

- outsourcing of complex object creations and
- it is not necessary for its client to know the concrete object type

So, adding e.g. future new concrete model types or exchanging them with the existing ones is easier and other software architecture parts don't have to be changed because of that new software feature.

(note: because of project time pressure no further UML diagrams are added)

Task

General Domain Information

It is clear that not knowing anything about the quadcopter domain, it is difficult to create a detailed specification of a general labelled task like 'take-off' or 'landing'. Specific detailed knowledge about i.a. geography landscape and weather aspects, mechanical and electrical or other technical properties, user behaviour, device maintenance and the dependencies from and with all such topics have to be taken into account. Not having in mind all the needed mathematical, physical and chemical knowledge for writing proper and useful documentation about it.

What does this mean for the quadcopter software implementation?

Let us assume knowing a lot of things of this domain. Anyway, it would be a huge and time consuming requirement engineering process to create such software task specifications for all use cases, as it is generally necessary for software architecture design, implementation and testing.

With reinforcement learning there is no need to do that in details, because it is an inherent major property of reinforcement learning „to train an agent by telling it *what* the desired goal is instead of *how* to reach it“ (see paper *Autonomous Quadrotor Control with Reinforcement Learning* from Michael C. Koval et al., http://mkoval.org/projects/quadrotor/files/quadrotor-rl.pdf). Such abstraction clearly simplified this projects software realisation, but it is still not easy.

Nevertheless, to get a feeling how it is to fly a drone, even knowing it is not such a stable device, I tried to fly a Revel control 'Helicopter Cardinal'. This was much harder than expected, because the device didn't start, fly and land constantly. Already the take-off was difficult. Many tries were necessary to get such tasks under control having smooth processes for the helicopter. Very often, the weather and my inexperience led the drift getting the major control about the device. But I have got the finding that in the reinforcement learning algorithm, this drift part should be added as penalty property. Furthermore, for the take-off the geographical start position and its landscape features were important as well. At some ground positions it was easier compared to others. So, the start position and desired target position where the device shall start to fly around (device condition: take-off process ends) have been insights too.

Chosen Task: take-off

For this project, because of this real experience, the 'take-off' task is chosen and we start now with getting an overview of some technical domain aspects.

Some features are:

Construction

As mentioned in that *Quadrotor Control* paper and as visible on the drone image at the beginning of this document, a quadcopter is supported by four horizontaly-oriented blades surrounding a central chassis that houses the device electronics.

States

Important for moving and flying around an unknown three-dimensional environment space is the drone capability of relative speed changes of its rotors.

„A quadrotor’s state is a function of its position, velocity, and acceleration: continuous variables [...].“

Therefore a reinforcement learning actor-critic concept is choosen that can deal with such continuous conditions. The „*modelfree* algorithms directly learn the value of each state-action pair without ever learning an explicit model of the environment.“

Whereat

- position is expressed in cartesian coordinates

$$\begin{aligned} x &= r \cdot \sin\theta \cdot \cos\varphi \\ y &= r \cdot \sin\theta \cdot \sin\varphi \\ z &= r \cdot \cos\theta \end{aligned}$$

- distance is expressed in meters
- velocity in m/s
- acceleration in m/s^2
- Euclidean rotations as quaternions

  (https://en.wikipedia.org/wiki/Quaternion)

Actions

The device is controlled by its velocities. Its physical parameters are defined in the PhysicsSim class.

- (linear) velocity
- angular velocity

So at each time step, the quadcopter’s action is a vector of desired linear and angular velocities.

Positions for the take-off process

- start position: device on the ground; in the Task class `runtime`, `init_pose`, `init_velocities` and `init_angle_velocities` are used as starting condition
- target position: one of the ending conditions of the take-off process

End conditions of the take-off process

- target position: end point of the take-off and with process transition starting point of the process flying around

- trundle: drift downwards, tailspin; not reaching the target take-off position and being in danger to crash the device
- crash: device may be defect

Reward

- drift: penalty condition because of unstable device behaviour and not reaching the desired target position
- crash: higher penalty compared to drift, device may be defect
- smoothly flying from start position to desired target position partly with discontinuity: positive reward
- smoothly flying from start position to desired target position without discontinuity: some more positive reward
- reaching the target position: highest positive reward

To explain the drone behaviour more detailed expert knowledge would be necessary which I don't have, so, it is best to cite the *Quadrotor Control* paper again:

„ [...]

To remain in a stationary hover the net force and torque on the quadrotor must both be zero.

This is achieved by driving the pairs of opposite motors in the same direction, or equivalently, driving adjacent motors with equal speeds in opposite directions. Any deviation from this stable configuration will cause translation or rotation to occur in the plane of the quadrotor.
Slowing two adjacent rotors by equal amounts, as one would expect, causes the net force on the robot to increase and the torque to remain zero (Figure 3a). This configuration allows the drone to make translational motion with no rotation about the drone's center. Conversely, decreasing the speed of two opposing motors causes net force on the drone to remain zero while perturbing the net torque. This, as expected, causes the drone to rotate about its center with zero translational velocity (Figure 3b)
[...]
Because reinforcement learning algorithms are only as good as the accuracy and timeliness their knowledge of the world, it is first necessary to obtain accurate measurements of the drone's position, velocity, and acceleration in three-dimensional space.
[...]
This measured pose will then be combined with the drone's on-board sensors to produce a single estimate of the quadrotor's position, orientation, velocity and acceleration with an enhanced Kalman filter [9].
This vector of data (or, perhaps, a subset of it) will then be used as state information for a reinforcement learning algorithm. At each instant in time, the quadrotor's action is a vector of desired linear and angular velocities which are interpreted with the AR.Drone's built-in control software. Given the appropriate reward function and an ample amount of data, a reinforcement learning algorithm designed for continuous state-action spaces should be capable of relatively quickly learning high-level actions [...]"

Final note,
because of project time pressure it has not been possible to implement all insights, in other words, an open toDo list remains.

Reference List

Course

- Udacity Advanced Machine Learning course

Book

- R. Sutton and A. Barto - *Reinforcement Learning: An Introduction. Second Edition,* MIT Press, Cambridge, MA (II Approximate Solution Methods)

Papers:

- http://people.csail.mit.edu/hongzi/content/publications/DeepRM-HotNets16.pdf
- V. R. Konda and J. N. Tsitsiklis - *Actor-Critic Algorithms,* Laboratory for Information and Decision Systems, MIT, Cambridge, MA, 02139
- Michael C. Koval et al. - *Autonomous Quadrotor Control with Reinforcement Learning*, http://mkoval.org/projects/quadrotor/files/quadrotor-rl.pdf
- T. P. Lillicrap and J. J. Hunt et al. - *Continuous Control With Deep Reinforcement Learning*, ICLR 2016 paper, Google DeepMind, London UK (https://arxiv.org/abs/1509.02971; chapter 3. Algorithm)

Links:

- https://www.groundai.com/project/a-brandom-ian-view-of-reinforcement-learning-towards-strong-ai/
- https://medium.com/free-code-camp/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d
- https://en.wikipedia.org/wiki/Quaternion