Deep Deterministic Policy Gradient for controlling a double inverted pendulum

introduction

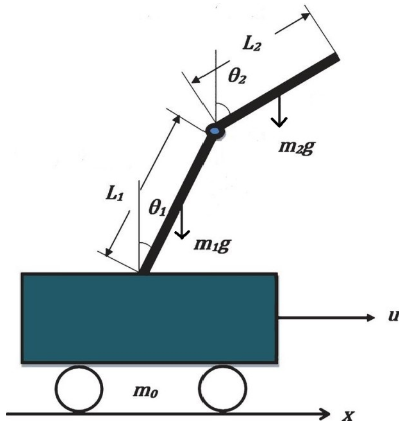
*The **DDPG** algotithm is a reinforcement learning algorithm developed by Google Deepmind (**Timothy P. Lilllicrap et al 2016**) which aims to apply the benefits of deep learning to approximate the policy of an agent in a continuous action domain. The method follows the model of the Actor-Critic agent which chooses an action according to the policy obtained by applying policy gradient (the actor) first, and then evaluates the action chosen through the Bellmann equation (the critic).*

The aim of this work is to train a pendulum placed on a cart, with two rotational joints, to stay up by applying an horizontal force on the cart, which is also constrained to move on a limited rail.
*The problem is formalized as a deterministic **Markov decision process**, which is defined entirely by reward, actions and state observation.*
*The **DDPG** algorithm can treat both low dimensional environments and pixel dimensional environments. When dealing with the low dimensional state problem, the state observation coincide with the state, so the action is chosen based only on the current state, while, in the high dimensional case, things get a bit more challenging.*

State of the art

The proposed algorithm merges the benefits of the **DQN** algorithm *(Mnih 2015)*, which could only treat environment with low dimensional action space, and the **NFQCA** algorithm *(Hafner & Riedmiller, 2011)*, where batch learning makes hard dealing with large networks. In addition, the use of batch normalization *(Ioffe & Szegedy) in* DDPG ensures scalability to many different environments.
A remarkable reinforcement learning for continuous policies is **Gprop** *(Ghifary 2015)* . The method is based two innovations: the first is a temporal-difference based method for learning the gradient of the value function and the second is a deviator-actor-critic model, which takes advantage of three neural networks to estimate the value function, the gradient and the actor's policy respectively. This algorithm achieves the best performance, up to date, on the octopus arm problem.
Direct policy search methods are often employed in high-dimensional applications such as robotics, since they can deal with high dimensional spaces and they guarantee convergence *(Peters & Schaaal 2008)*. However it is often necessary to choose a specialized policy class to learn the policy in a reasonable number of iterations without falling into local optima.
Guided Policy Search *(Levine 2015)* is a recently developed algorithm that aims to learn policies with very general and flexible representation. To do so, this approach learns a set of trajectories for the desired motion skill by using iteratively refitted time-varying linear models and then unifies these trajectories into a single control policy that can generalize to new situations. This method has proved its success in many tasks such as putting together a toy airplane, stacking tight-fitting lego blocks and so forth. Trajectory optimization can guide the policy search away from local optima, the algorithm uses differential dynamic programming to generate guiding samples, which assist the policy search by exploring high-reward regions
An interesting application which enhances Guided Policy Search regard controlling an unmanned aircraft vehicle *(Zhang 2015)*. This is done by combining **MPC** (a method of process control) with the aforementioned algorithm.

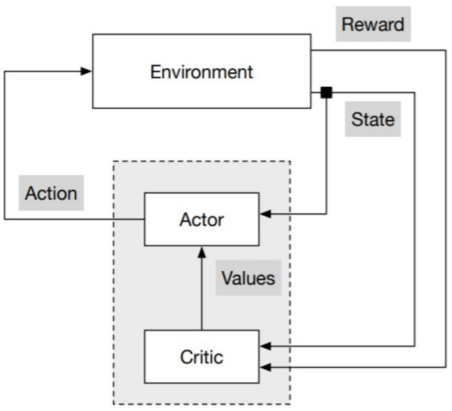
The environment



The **inverted double pendulum** environment is entirely defined by the *Mujoco* library, however it's worth describing the dynamic of the model to have some insights on the complexity of the task.
The model is two dimension and it is entirely represented by the image on the right, with the reference frame placed in the middle of the cart.

The state of this environment returns the coordinates: x, θ_1 and θ_2 (actually it returns cosinus and sinus of both the joint angles) and the velocities: dx/dt $d\theta_1/dt$ and $d\theta_2/dt$. In addition there are three more values representing the constraint on the force due to the position and the joints angle respectively. The action space which allows to try and control the environment is the force value (*u* on the figure above) and it is constrained among the interval [−10, +10] Newton. The problem presents 3 degree of freedom and only one parameters in which is possible to act to preserve equilibrium. The equations of motion can be obtained with the principle of Lagrangian mechanics, that provides a system of ordinary differential equations, solved by the library using the *Runge-Kutta* algorithm for each time step. The standard environment computes the reward in accordance to the distance from the origin, the height of the tip of the second bar and the joint velocities, which all give a negative reward. The only positive reward is given by an alive bonus.
The formula is the following: $r = \text{alive} - 0.01*x^2 - (y-2)^2 + \; - 10^{-3} * (d\theta_1/dt)^2 + 5*10^{-3} * (d\theta_2/dt)^2$; where alive is 10 and an episode is considered concluded when the y (tip position) drops below 1.

algorithm description



The algorithm follows an actor-free, off policy actor-critic model.
The scheme is the following: there are four functions approximated by four neural networks which are the actor's policy $\mu(s|\theta^\mu)$, the critic's value function $Q(s,a|\theta^Q)$, the target actor's policy $\mu'(s|\theta^{\mu'})$ and the target critic's value function $Q'(s,a|\theta^{Q'})$; the actor chooses an action according to its policy function, the critic evaluates it and the target actor and the target critic allow a smooth and regularized convergence by providing a target value to confront with the current value function.

The critic and the actor, and the target networks are initialized with some weights; the target functions have the same weights of their corresponding function initially. At each iteration (for each time step of each episode) an action is chosen according to the current policy, with some noise (which allows exploration), and it is executed. Observation and reward are collected. The noise used is the Ornstein-Uhlenbeck process *(1930)* which generates temporally correlated exploration, which is very efficient in physical control problems with inertia. The tuple containing the transition is stored in the replay buffer, and the training does not start until in the buffer there are at least minibatch size tuples. The training of the network is done by sampling a number equal to the minibatch size of tuples from the replay buffer and using states and actions collections sampled to feed the networks.

The value function is optimized with mean squares, with respect to the target y_i, computed using Bellman's equation with the value function that results from the target critic network using as action the output of the target actor network.
The actor is updated applying the chain rule to the expected return from the start distribution J with respect to the actor parameters :

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

The target networks, instead, update their targets with a simple linear law:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

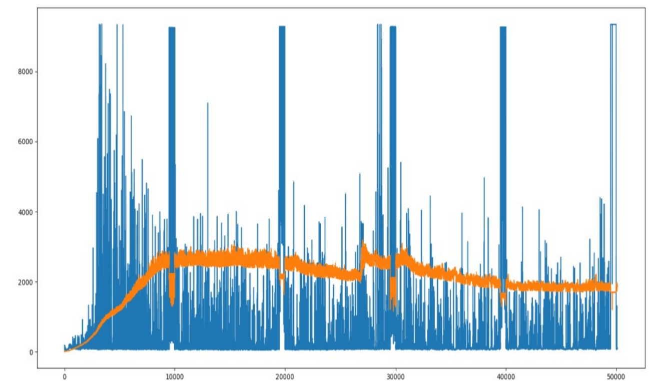
Implementation insights

The implementation of the algorithm is done by using *tensorflow* and *tflearn* libraries, in particular, tensorflow allows an easy computation of the gradient through the command "tensorflow.gradient" which is useful for the actor's training to compute the gradient of the policy with respect to its parameters and the gradient of the value function with respect to the actions. The optimization of the networks exploits the *Adam* optimizer, an adaptive learning rate method, that uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network. The structure of the algorithm is the same when acquiring low dimensional features instead of raw pixels from the environment, even though, for the latter, three convolutional layers are added at the beginning of the network, to acquire the features from the data. As a matter of fact, there is no way to elaborate the feature "speed" of the pendulum and the cart, from a single frame. To solve this issue, the paper suggests to elaborate an action chosen for a state, two more times, making the observation of a state composed by three frames, so its size is "n. pixel x 9". The extra steps should be done on a support environment, so that the one on which the work is going on, is not modified. This could not be done since the environment used doesn't allow to create a new one and start it in a desired state, in addition, making a copy of the object environment (even with the dependecies, by using the command) does not work , due to the many dependecies inside the library.
As an attempt to take into consideration the lack of the dynamic, the algorithm executes 2 times an action on the running environment at each step, so the observation's dimension is as big as "n. pixel x 6". Thanks to this method the feature velocity its supposed to be forecast to the neural network which can try and acquire these features.
Another issue in implementing the algorithm is that the Mujoco library does no provide any render of its environment, so for this case, the Pendulum environment has been chosen to test the algorithm.
The neural network has been developed in accordance to the paper. The structure used, in the low dimensional case, for the four networks is the same: two hidden layers with 400 and 300 units respectively, which use ReLu as activation function while the hidden layer uses tanh. The weights are initialized with uniform values belonging in the range +- (1/sqrt(f)) for all layers but the hidden one, which has as outer valued (–0.003, + 0.003) for the low_dimensional case and (–0.0003, +0.003) for the pixel case. Batch normalization is used between all layers.
For the actor the input is just the observation and its output lies in the action space (if not the output is scaled to fit in it), while the critic gets both actions and states, which are combined (simply appended) in the second fully connected layers where the actions starts to be taken into consideration.
The structure of the pixel it's slightly different from the one described: there are three convolutional layers preceding the other mentioned. These gets as input two rgb frames of 64x64 pixels organized as a 64x64x6 array, first layer with 64 filters, second with 32 filters, and third with 32 filters.

Results obtained

1) standard results

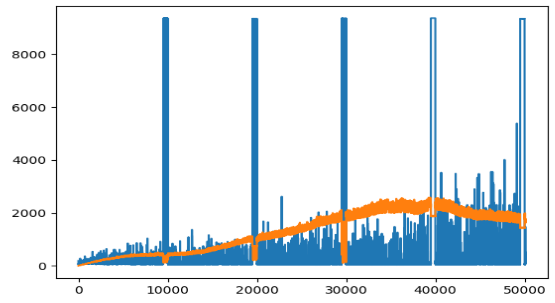
The following chart represent the algorithm running in low dimensional case and in the standard environment with the values
tau = 0.001, actor learning rate = 0.001,
critic learning rate = 0.001,
gamma = 0.99, minibatch size = 64,
buffer size = 50 000, number of episodes = 50 000.



The results show an increase in the value function approximation (the orange curve), until a certain point, after which it decreases.
The evaluation of the policy is done every 10 000 episodes for 500 episodes in which there is no training and only the best actions are chosen (only exploitation, so no noise), which are the areas where the cumulative reward (the blue one) reaches a peak, as expected.

2) lower learning rate (10 times lower)

Using the same configuration of case 1) but with lower learning rates (10 times lower for both critic and actor) grants a smoother convergence but it provides the same results in terms of cumulative reward.

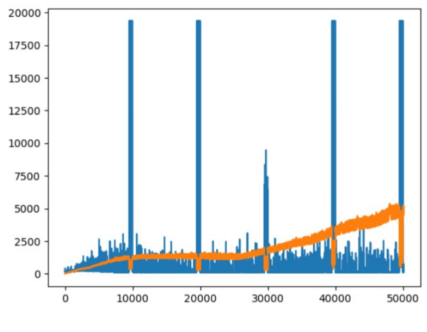


3) network modified

Some attempt to modify the structure of the network are the follwing: 2 hidden, fully connected layers are added (with 200 and 100 units respectively) before the output layer, the learning rates have been increased. The results obtained are poor and converge immediately to a local optima.
Another attempt with the use of sigmoid activation for the output layer provides bad results as well, with convergence stuck after few iterations.

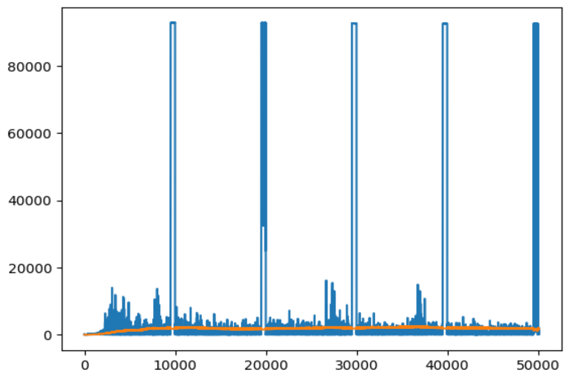
4) modified environment

In this training the environment has been modified to have softer constraints on the pendulum, allowing for a wider space of actions and state. The penalties for staying distant from the origin and for the velocities are reduced, the results though, don't stress any considerable difference from case 1.



5) increasing the storage of the buffer and the episode length

This training requires more computational resources, using a buffer of 10^6 tuples and the episode maximum length
is set to 10^4, by modifying the
register of the environment inside the Gym library.



Conclusions

The first attempt *(1)* shows an very high cumulative reward reached in the evaluation interval. This, among with the growth of the value function, means that the training is working properly, even if after 10 000 the agent is not learning anything new, since the reward does not get much higher than the value 9000. This could be caused by a too high learning rate, which can't reach the optima because its steps are too big, or by a low flexibility of the environment. Another reason could be that the network is not able to acquire the function for the policy better than that.

In experiment number *(2)*, the algorithm reaches the same results in term of reward, so the problem algorithm converges to a global optima. Some other attempts to use a different structure of the network *(3)* did not provide good results at all, they, instead, made clear that the layers structure provided by the paper outperforms other layouts.

Modifying the environment *(4)* could give more flexibility to the behaviour of the agent since it gets less penalties for staying in the middle and for having joints velocities. The goal of this attempt is to allow some brute changes of forces to reach equilibrium.
The results obtained are tricky since rewards are computed in a different way, which make difficult to make any comparisons with the standard *(1)* case. It's interesting the fact that its chart shows that the value function keeps growing over time, differently from the base case.

In all the attempts made, the maximum length of any episodes was 999, even though the number of maximum iterations allowed are more. The problem is due to a limit on the number of steps intrinsic in the Gym libraries, set to 1000 by default. By modifying the register the limit is set to 10 000.
This allows for more iterations, so along with this modification, the buffer size is increased from 50 000 to 10^6 to allow better performances, even if more computation is required. The results obtained provide cumulative rewards 10 times bigger than the base case *(1)* which could be increased even more, since the maximum length of 10 000 is reached. It would be interesting to see how far the algorithm could go by increasing more an more the number of time steps allowed, among with the buffer size (and also the minibatch size). Doing this, though would require a lot of computational effort.

References:

https://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html
https://spinningup.openai.com/en/latest/algorithms/ddpg.html
https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287
https://arxiv.org/abs/1509.02971
https://www.mitpressjournals.org/doi/abs/10.1162/0899766003 00015961
https://link.springer.com/article/10.1007/s10994-011-5235-x
https://arxiv.org/abs/1509.03005
http://www.jmlr.org/proceedings/papers/v28/levine13.pdf
http://proceedings.mlr.press/v28/levine13.pdf
https://www3.math.tu-berlin.de/Vorlesungen/SS12/Kontrolltheorie/matlab/inverted_pendulum.pdf
https://github.com/openai/gym/pull/740/commits/4d243e8143e1fc488046e98989790a4535e17c4a
https://github.com/openai/gym/issues/374
https://github.com/openai/baselines/tree/master/baselines/ddpg