Project 1: Continuous Control

Edoardo Bacci

Deep Reinforcement Learning Nanodegree

Udacity

13/08/2019

Keywords: DDPG, Continuous Control, Deep Reinforcement Learning

1 INTRODUCTION

For this project we were required to train an agent in a continuous action space. The testing benchmark used in this project was a simulated environment developed using Unity where the agent is supposed to control a two-jointed robot arm to follow a moving goal within some time boundary.

2 ENVIRONMENT

For this project we used the Reacher environment in ML-Agents from Unity. In this scenario we control 20 two-jointed robot arms with a single brain (agent).

The aim of the agent is to reach the moving goal represented with a green sphere and keep the tip of the arm within the goal for as long as possible. A small positive reward is provided for each time step that the agent's hand is in the goal location. These 20 agents run in parallel and have no interaction with each other.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector of 4 real numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

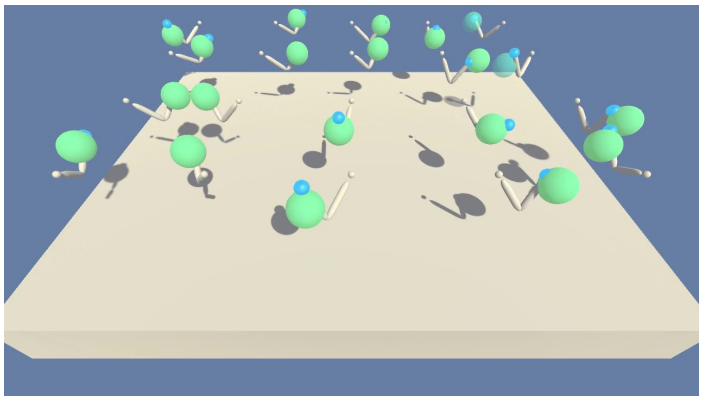

Figure 1. A representation of the Reacher environment

3 THEORY

With Deep Deterministic Policy Gradient[3] we use two networks to control the agent, the actor and the critic.

The actor learns the policy to be executed by the agent: the output of the network are the actions to be executed. These actions can be perturbed with some noise to encourage exploration.

The critic learn the value of state-action pairs. These values will help the agent predict how good the return will be in the future given that we execute some action a in state s.

In a discrete Q-learning scenario we would use the formula:

$$Q(s_t, a_t, \theta_Q) \leftarrow Q(s_t, a_t, \theta_Q) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a, \theta_Q) - Q(s_t, a_t, \theta_Q)) \quad (1)$$

but $\max_a Q(s_{t+1}, a)$ is impossible to compute given that we are in a continuous domain. For this reason we use the critic that will act as a surrogate of the max operator.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma(s_{t+1}, \mu(s_{t+1}) - Q(s_t, a_t)) \quad (2)$$

where $\mu(s)$ is the output of the actor network.

In order to train the critic neural network we use the loss function

$$L(\theta_Q) = \mathbb{E}_{(s,a,r,s')}\left[\left((r + \gamma \max_a Q(s', a, \theta_Q) - Q(s, a, \theta_Q))\right)^2\right] \quad (3)$$

For the critic, we will train the network to pick the actions with the highest values given by the critic. For this reason we will use the loss function

$$L(\theta_\mu) = -\mathbb{E}_{(s_t \sim \rho^\beta)}(Q(s_t, \mu(s_t, \theta_\mu), \theta_Q)) \quad (4)$$

that in short means "maximise the action that will return the highest Q-value". Notice the negative sign at the beginning for performing gradient ascent.

3.1 Double DQN

In the above method the maximum value of each state-action pair is reliable only when the weights are properly trained. When the network is still at the beginning of the training the initial estimates will probably be random. The normal DQN methods suffers from instabilities due to the fact that θ_Q and θ_μ are both updated and used as a target at the same time.

In order to counteract this behaviour we use a new method for selecting actions called Double DQN[2]. In Double DQN we select action using using one network but we estimate their value using another.

One way of doing this is selecting the values using the local critic network and estimating their value using its copy, the target critic network. To update the target we perform a "soft update", where every time we train part of the local network is merged with the target network according to the weighted sum regulated by τ:

$$\theta' \leftarrow (1 - \tau)\theta + \tau\theta' \quad (5)$$

The same operation is carried on also on the actor network which is split again in local and target networks.

The updated equation for the critic loss is :

$$L(\theta_Q) = \mathbb{E}_{(s,a,r,s')}\left[\left(r+\gamma Q(s', \mu(s', \theta'_\mu), \theta'_Q) - Q(s, a, \theta_Q)\right)^2\right] \quad (6)$$

Where θ_Q is the local critic network, θ'_Q is the target critic network and θ'_μ is the target actor network.

3.2 Prioritised Experience Replay

Prioritised Experience Replay (PER) is an extension for off-policy algorithms.

Normally, off-policy algorithms save the transition encountered into a memory buffer. In this way they can access past experiences and learn from them. However not all experiences are made equal: some of them are more difficult to learn for the network than others.

PER works by sorting the transitions by the TD error. The memories at the beginning of the experience replay buffer will be chosen more often in a probabilistic fashion. In this way, the agent will train more often on transitions that will contribute more to its learning optimising the time spent training.

4 METHODS

The algorithm used for this project is a DDPG as described in the paper (using target networks for both actor and critic) with the addition of Prioritised Experience techniques described above (as done in [4])and N-step TD estimate (as suggested in [11]). The full pseudocode can be found in Algorithm 34. The full code can be found at https://github.com/phate09/drl_continuous_control.

4.1 Architecture

The architecture that was used consisted of two simple feed-forward neural networks with 2 fully connected layers, one for the actor and the other for the critic. The actor terminates with a tanh layer to cap the actions within the [-1;1] interval. The critic receive a combination of state+action as an input and terminates with a single value as output that represents the Q value of the state-action pair.

The final architecture is shown in Figure 2.

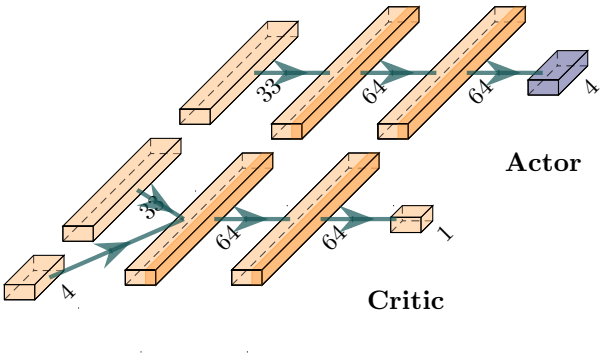

Figure 2. Architecture for the agent model (yellow layers are linear, red ones are RELU, purple is tanh activation function)

Algorithm 1: DDPG

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, episodes budget T, n-step TD, training episodes Q, target update τ.
Initialise θ_μ randomly and copy to θ'_μ
Initialise θ_Q randomly and copy to θ'_Q
Initialise replay memory $\mathcal{H} = \emptyset$
for $i \leftarrow 0$ **to** T **do**
  $s \leftarrow$ reset environment
  while *not done* **do**
    choose $a \sim \pi(s|\theta_\mu)$
    s', r, *done* = environment step (s,a)
    Compute TD-error δ = $r + \gamma Q(s', \mu(s', \theta'_\mu), \theta'_Q) - Q(s, a, \theta_Q)$
    store (s,a,r,s', *done*) in experience replay prioritised by δ
    $s \leftarrow s'$
    for $q \leftarrow 0$ **to** Q **do**
      if $t \mod K = 0$ **then**
        for $j \leftarrow 0$ **to** k **do**
          Sample transition s_j, a_j, s'_j, r_j from $\mathcal{H}$
          Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
          Compute TD-error δ_j = $r_j + \gamma Q(s'_j, \mu(s'_j, \theta'_\mu), \theta'_Q) - Q(s_j, a_j, \theta_Q)$
          Update transition priority $p_j \leftarrow |\delta_j|$ in $\mathcal{H}$
          Accumulate weight-change:
          $\Delta_Q \leftarrow \Delta_Q + w_j \cdot \delta_j \cdot \nabla_\theta Q(s_j, a_j)$
          $\Delta_\mu \leftarrow \Delta_\mu + \nabla_a Q(s, a, \theta_Q)|_{a=\mu(s)} \nabla_{\theta_\mu} \mu(s)$
        end
        Update weights:
        $\theta_Q \leftarrow \theta_Q + \eta \cdot \Delta_Q$
        $\theta_\mu \leftarrow \theta_\mu + \eta \cdot \Delta_\mu$
        Update target weights:
        $\theta'_Q \leftarrow (1-\tau) \cdot \theta'_Q + \tau \cdot \theta_Q$
        $\theta'_\mu \leftarrow (1-\tau) \cdot \theta'_\mu + \tau \cdot \theta_\mu$
        Reset $\Delta = 0$
      end
    end
  end
end

4.2 Hyperameters

To reach good results, the algorithm has some hyperparameters that needs to be tuned to the problem. For this project the parameters that have been used are:

- Optimiser (actor & critic) = Adam
- Learning rate (actor & critic) $\eta = 1e-4$
- Batch size $k = 64$
- Episodes budget $T = 2000$
- Action noise $\gamma = 0.2$
- Experience Replay Size $k = 1e6$
- Replay period $K = 120$
- Training episodes $Q = 4$
- N-steps TD $n = 1$
- Discount factor $\gamma = 0.99$
- Target update coefficient $\tau = 1e-3$

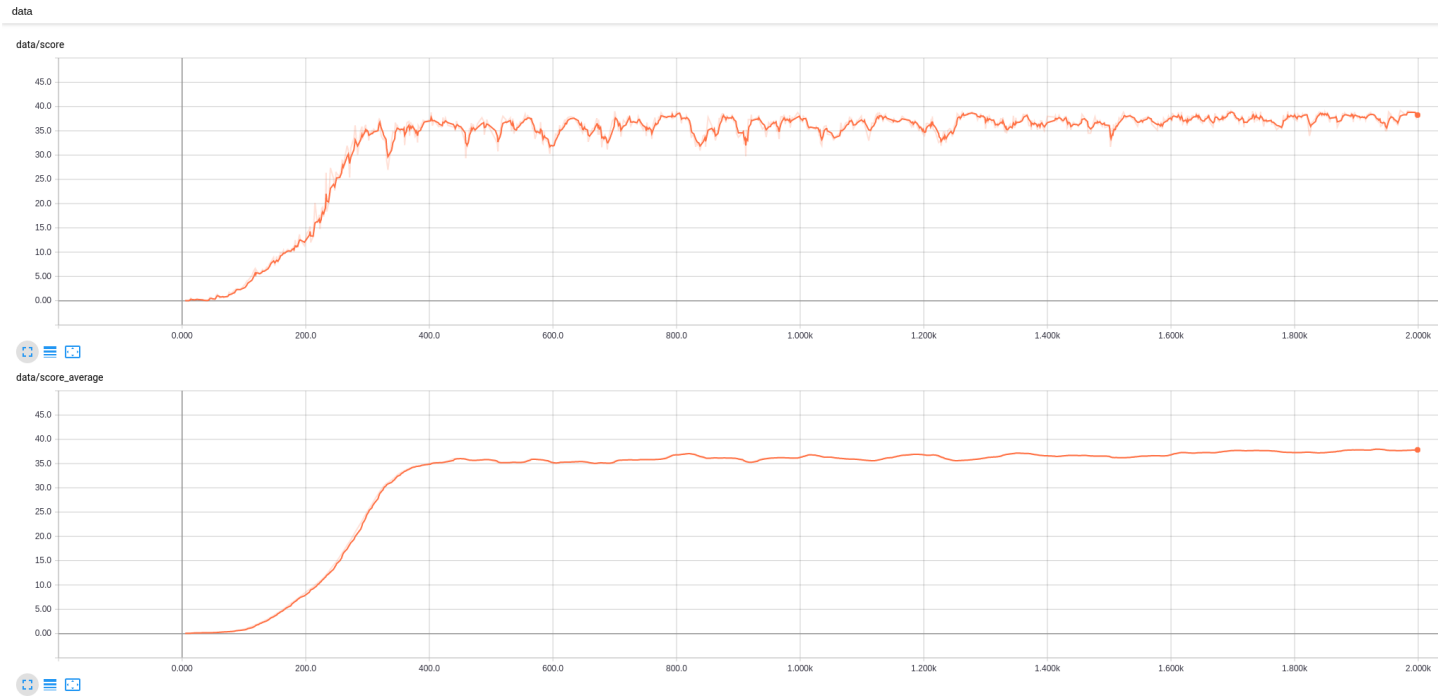

Figure 3. Up: Plot of the score of a single agent during training. Down: Plot of the average of the last 100 episodes. From the plot it can be seen that the algorithm reaches an average of 30+ around iteration 325

- Prioritised experience exponents $\alpha = 0.4, \beta = 0.5 \rightarrow 1.0$ annealed during entire training

5 RESULTS

The results are reported in Figure 3. The algorithm improves steadily its performance until iteration 325 where it passes its required mean score of 30. The agent was left to train more episodes to check its progress and the performance stabilise around iteration 400. From there it remains steady with a score of 35 and slowly continue to increase it for the remaining of the training. A video of the agent in action can be found at https://youtu.be/2kSd00ENSbI. A graph that plots the score of a single run can be found at Figure 3.

6 FUTURE IMPROVEMENTS

As seen before, combining different techniques to counteract stability issues helped to make the agent more stable during training and converge faster to a near optimal solution. Possible improvements to the algorithm in here described could be the use of "Noisy Networks"[11] to aid exploration by introducing noise, "Generalised Advantage Estimation" an implementation of the concept of TD(λ) to even out instabilities, "Distributed" experience replay"[5] to leverage parallel actors and speed up learning, "Distributional values"[6] to keep track of the state value distribution rather than just the mean. The latter two seems particularly promising as demonstrated in [4]. These improvements will aid the training of the algorithm and make it converge even faster.

7 CONCLUSION

This project was my first attempt to a more difficult problem such as continous control.

It was an exciting experience where I learnt a lot. The hardest part has been the hyperparameter tuning because if the parameters are not chosen accurately even a correct algorithm with no bugs will not converge. Luckily, I was able to find appropriate values from the research papers and through the help of the community.

REFERENCES

[1]"Dueling Network Architectures for Deep Reinforcement Learning", Ziyu Wang and Nando de Freitas and Marc Lanctot, 2015
[2]"Deep Reinforcement Learning with Double Q-learning", Hado van Hasselt and Arthur Guez and David Silver, 2015
[3]"Continuous control with deep reinforcement learning", Lilicrap et al. , 2015
[4]"Distributed Distributional Deterministic Policy Gradients"c Barth-Maron et al. , 2018
[5]"Distributed Prioritized Experience Replay"c Horgan et al. , 2018
[6]"A Distributional Perspective on Reinforcement Learning" Bellemare et al. , 2017
[7]"Reinforcement Learning", Sutton and Barto 2018
[8]"High-dimensional continuous control using Generalised Advantage Estimation"c Schulman et al. , 2016
[9]"Human-level control through deep reinforcement learning", Volodymyr Mnih et al. , 2015
[10]"Prioritized Experience Replay", Tom Schaul, John Quan, Ioannis Antonoglou, David Silver, 2015
[11]"Rainbow: Combining Improvements in Deep Reinforcement Learning", Matteo Hessel et al., 2017