Project Report: Continuous Control

Rui Ding

March 2020

1 Introduction

In this project, I applied a DDPG agent which is structured so as twenty DDPG agents working in parallel sharing the same networks and experience replay buffer. The twenty agents are trained to complete the task of continuous control by maintaining its position in the target location for as long as possible. The environment is built through the Unity Reacher environment for twenty agents in parallel.

2 Description of Environment

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of the agent is to maintain its position at the target location for as many time steps as possible.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1. The environment is episodic and an average score is computed every 100 episodes. The environment is considered solved is the average score over the last 100 episodes exceeds 30.

3 Algorithm

In this project, I applied a modified DDPG agent that takes parallel actions over twenty agents. A single agent object is instantiated which keeps an actor network, a critic network, and a replay buffer with capacity 1000000 which are shared across the twenty parallel agents.

The DDPG algorithm works as follows. The actor network takes in a state as input and outputs a deterministic action vector, in this case of dimension 4. Hence denote the deterministic policy by:

$$\mu : S -> A$$

where the policy $\mu(s|\theta^\mu)$ is parametrized by θ^μ. The Q values corresponding to such a policy is:

$$Q^\mu(s_t, a_t) = E_{r_t, s_{t+1}}[r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1}))]$$

The Q network is parametrized by θ^Q and is learned by the critic using Q-learning with the following loss function:

$$L(\theta^Q) = E_{s_t, a_t, r_t}[(r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q) - Q(s_t, a_t|\theta^Q))^2]$$

The actor is updated using the following policy gradient which is obtained by applying chain rule:

$$\nabla_{\theta^\mu} J = \nabla_{\theta^\mu} E_{s_t}[Q(s, a|\theta^Q)|s = s_t, a = \mu(s_t|\theta^\mu)] = E_{s_t}[\nabla_a Q(s_t, \mu(s_t)|\theta^Q) \nabla_{\theta^\mu} \mu(s_t|\theta^\mu)]$$

Target networks are used for both the actor(μ') and the critic(Q') for stable training. The target networks are updated following a soft update rule with hyperparameter τ such that in every learning step:

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

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

Another key component of the algorithm is the noise process, which is defined in the original paper by an Ornstein-Uhlenbeck process that progresses through the entire episode. The exploration policy is set to be $\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \epsilon N_t$.

In our implementation, for each action step the twenty agents will take such an exploratory action in parallel and the twenty experiences tuple will be stored in the replay buffer. Then we define an hyperparameter UPDATE_EVERY such that after that many steps, a learning step will be performed and the network weights will be updated. The learn step samples a minibatch of experiences from the replay buffer with a fixed batch size. Empirical gradients are computed based on these samples to update the networks weights.

When any of the twenty agents experiences a terminal state, the current episode ends. The noise process is refreshed for every new episode with a decay in the scaling factor ϵ which is initiated to be 1.0.

4 Training Results

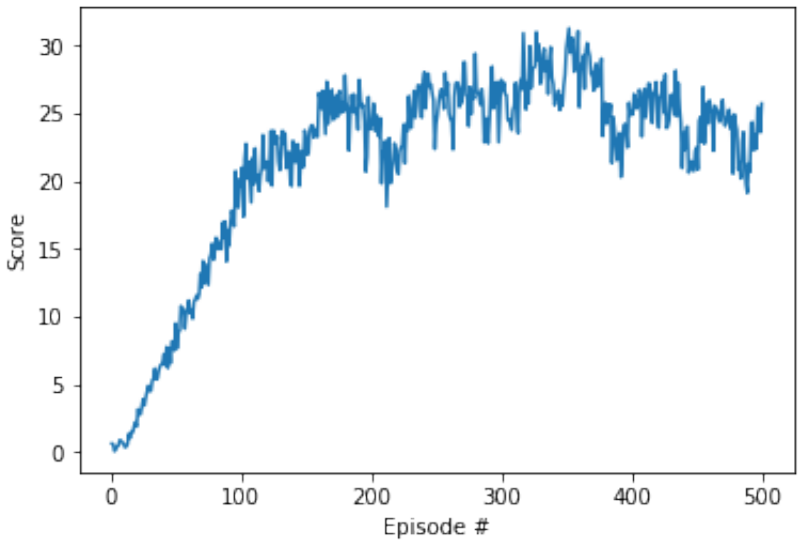
For this project I tried several sets of hyperparameters and ended up with drastically different results. It is clear that without a good choice of learning rates, batch size, exploration noise, and network structures, the agent might not learn at all or will learn up to some threshold and decay backwards.

I selected to implement both the actor and critic networks (local and target) with neural networks composed of two hidden layers, both having 256 cells. This is different from the original implementation but works well for this task, which has input dimension 33. The actor networks has output dimension 4

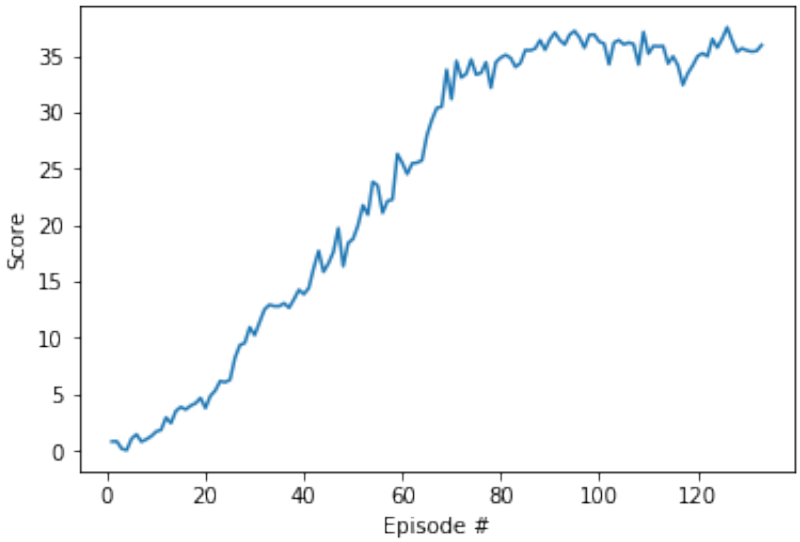
corresponding to an action. The critic network takes in the action vector in the second layer and outputs a scalar value. ReLu activation is used for all activation functions.

For the DDPG agent, the batch size is selected to be 128, $\gamma = 0.99$, $\tau = 0.001$, and UPDATE_EVERY = 1. The learning rate for actor and critic networks are both 0.0003.

In a first attempt, the epsilon scaling factor is set to have no decay so the exploratory noise remains the same scale for every episode. The agent makes some quick learning in the first 200 episodes and reached an average of around 25 in 100 episodes. But it stopped learning and starts oscillating for the next 300 episodes around this threshold. I realized that this may be caused by the fact that the exploratory noise is too large such that after learning an approximately good policy the agent is still exploring too much hence restraining it to learn better by exploiting the currently learnt policy. The score plot is below.



In the second attempt, I choose an epsilon decay factor of 0.003. The agent makes quick and stable training in the first 100 episodes and without making large drawdowns in scores, it is able to solve the environment in 33 episodes and reach an average score of 30.04 over the 100 episodes from episode 34 to 133. The score plot is below.



The weight of the local actor and critic networks in the final solution is saved as the files "checkpoint_actor.pth" and "checkpoint_critic.pth" respectively.

5 Future Extensions

One particular extension is the D4PG algorithm: Distributed Distributional Deterministic Policy Gradients. This particularly suits the parallel agent setting which we have and is shown to be more stable with better performance.

Other directions of work include comparing with proximal policy optimization agents.

6 References

1. Course Materials and Codes

2. Continuous Control with Deep Reinforcement Learning. T. P. Lillicrap, J. J. Hunt, Aa. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra (2015).

3. Benchmarking Deep Reinforcement Learning for Continuous Control. Y. Duan, X. Chen, R. Houthooft, J. Schulman, and Pieter Abbeel (2016).

4. Distributed Distributional Deterministic Policy Gradients. G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. Lillicrap (2018).