DDPG Algorithm - Reacher Continuous Control

Overview

Using the Unity agent/environment "Reacher", this deep reinforcement learning task trains an AI agent to keep its double jointed arm on a ball for as long as possible. The task is considered solved when the agent, who receives various rewards for maintaining its position, receives an average reward of 30 over 100 consecutive episodes.

The Unity environment has two options - using a single agent or 20 agents in parallel. In the latter case, the experiences and rewards of all 20 agents running at the same time are combined during evaluation and training. This repo uses the 20 agent version.

In either environment, each agent receives feedback in the form of a reward (+0.1) or no reward after taking each action. The environment conveys to the agent the state of the agent in the form of 33 different values related 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.

At first the agent randomly takes actions and records the feedback, but eventually begins to take those experiences and learn from them using a Deep Deterministic Policy Gradient (DDPG) algorithm.

The attached code written in Python, using PyTorch and presented in a Jupyter Notebook, demonstrates how the agent learns and eventually achieves an average score of 30 over 100 consecutive episodes.

Model Architecture

The Udacity provided DDPG code in PyTorch was used and adapted for this 20 agent (version 2) environment.

I used two deep neural networks (actor-critic), each with two hidden layers of 256-128 nodes, with eLU activation functions on the hidden layers and tanh on the output layers. The changes I made vs the original DDPG research paper - https://arxiv.org/pdf/1509.02971.pdf - was reducing network complexity, after experimenting I found it converges faster this way (from 400-300 to 256-128).
Also, in the original source code from Udacity the learning rates where different for the two networks, I found it better to use same for both 1e-4.

DDPG algorithm description

Policy-Gradient (PG) algorithms optimize a policy end-to-end by computing noisy estimates of the gradient of the expected reward of the policy and then updating the policy in the gradient direction.

DDPG is a policy gradient algorithm that uses a stochastic behavior policy for good exploration but estimates a deterministic target policy, which is much easier to learn. Policy gradient algorithms utilize a form of policy iteration: they evaluate the policy, and then follow the policy gradient to maximize performance.
The Actor-Critic learning algorithm is used to represent the policy function independently of the value function. The policy function structure is known as the actor, and the value function structure is referred to as the critic. The actor produces an action given the current state of the environment, and the critic produces a TD

(Temporal-Difference) error signal given the state and resultant reward. If the critic is estimating the action-value function, it will also need the output of the actor. The output of the critic drives learning in both the actor and the critic. In Deep Reinforcement Learning, neural networks can be used to represent the actor and critic structures.
In practice, we let the agent explore the environment according to initial network output and we store the results in a replay buffer (experiences). To deal with the exploration/exploitation dilemma, we add some random noise in the training, and also use the epsilon greedy approach. After enough experiences have been accumulated in the replay buffer, we sample a minibatch-sized number of experiences, and instruct the agent to learn from them. Learning process is governed by the following equations:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q)^2)$$

Here, we sample a minibatch of size N from the replay buffer, with the i index referring to the i'th sample. The target for the temporal difference error computation, yi, is computed from the sum of the immediate reward and the outputs of the target actor and critic networks, having weights θμ′ and θQ′ respectively. Then, the critic loss can be computed with respect to the output Q(si,ai|θQ) of the critic network for the i'th sample. See [1] and [2] for more details.

Hyperparameters

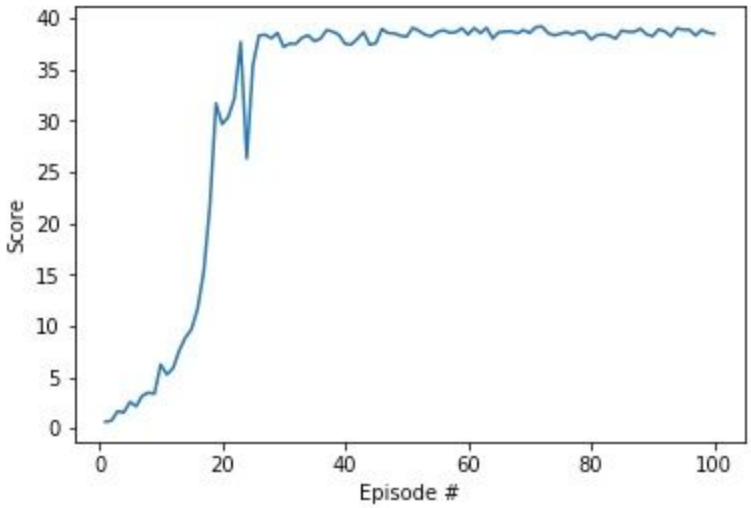
A learning rate of 1e-4 on each DNN and batch size of 128 were used along with replay buffer size of 1e5, gamma .99 and Tau of 1e-3. There was no change to the default Ornstein-Uhlenbeck noise parameters (0.15 theta and 0.2 sigma.)

Training

I spilt the training into two parts: at first, since we are using 20 agents that share experiences, thus generating a lot of experiences, I train the agents 3 times per episode, until it reaches an acceptable value over the averages (33 chosen). After this level is achieved, I train each 10 episodes.
In addition to Ornstein-Uhlenbeck noise, I found helpful to add an epsilon greedy logic to the agent.

Results and Future Work

The model was able to achieve the 30 average reward goal around episode 20, and it maintained performance afterwards, thus achieving the goal in the first 100 episodes.



In the future, I plan experimenting with other architectures, such as TRPO and PPO. Why ? The main issue of DDPG is that you need to pick the step size that falls into the right range. If it is too small, the training progress will be extremely slow. If it is too large,

conversely, it tends to be overwhelmed by the noise, leading to tragic performance. Took me almost three weeks of parameter and network architecture tuning in order to solve the environment.

Trust Region Policy Optimization (TRPO) improves the performance of DDPG as it introduces a surrogate objective function and a KL divergence constraint, guaranteeing non-decreasing long-term reward. Proximal policy optimization (PPO) further optimizes TRPO by modifying the surrogate objective function, which improves the performance as well as decreasing the complexity of implementation and computation. I hope to see this in action.

Bibliography

1. Silver, et al. Deterministic Policy Gradients
2. Mnih, et al. Human-level control through deep reinforcement learning
3. Patrick Emami Deep deterministic policy gradients in Tensorflow