Report : Project #2 Continuous Control

5/4/2019

Soyoung Park

Jupyter notebook can be viewed here:
https://nbviewer.jupyter.org/github/parksoy/Soyoung_Udacity_ND_DeepReinforcementLearning/blob/master/p2_continuous-control/Continuous_Control.ipynb

This project is built with the multi-agent Reacher environment (Unity Machine Learning Agents, open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents). 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. The goal of your 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 version chosen for this project contains 20 identical agents, each with its own copy of the environment. The barrier for solving the multi-agent version of the environment is slightly different, to take into account the presence of many agents. In particular, your agents must get an average score of +30 (over 100 consecutive episodes, and over all agents). Specifically, after **each episode,** we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 20 (potentially different) scores. We then take the average of these 20 scores. This yields an **average score** for each episode (where the average is over all 20 agents). The environment is considered solved, when the average (over 100 episodes) of those **average scores** is at least +30.

1. Learning Algorithm

- **Learning algorithm**

Deep Deterministic Policy Gradients (DDPG) algorithm (DDPG paper) was used and its template code from the Actor-Critic Methods lesson was modified to adapt to run Reacher multi-arm version. The various hyperparameters and settings were tweaked.

- **The chosen hyperparameters**

Attempt	max_steps	batch_size	actor_learn_rate	critic_learn_rate	update_every	num_updates	tau	gamma	noise_sigma	noise_factor_decay	layer_sizes
24	1000	1024	0.001	0.001	20	10	0.005	0.99	0.2	1.00E+06	[128,128]

- max_steps
  : In one episode, how many maximum number of steps to be explored to move onto the next step

- num_episodes
  : How many times to iterate of training of max_steps

- batch_size
  : How many Replay buffer memory to be sequentially to learn through the neural networks in a batch processing manner

- buffer_size
  : Replay buffer size that consists of namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])

- actor_learn_rate
  : How fast Actor neural network learns

- critic_learn_rate
  : How critic Actor neural network learns

- update_every
  : Update occurs only when certain number of steps is reached to learn more stably.

- num_updates
  : For every certain step that is defined by update_every, only limited number of update is made.

- tau
  : Control soft update. Only portion of local network is updated, while the majority of update comes from the stable target.

- gamma
  : Discount factor. Future reward is less valuable than immediate reward.

- noise_sigma
  : Amount of noise user wants to introduce.

- layer_sizes
  : Unit size of the first and second hidden layer

- noise_factor_decay
  : Controls how fast the noise decays over episodes as agents learn the process.

- **Model architectures for neural networks.**

Actor network

```
fc1 = nn.Linear(state_size, fc1_units)
fc2 = nn.Linear(fc1_units, fc2_units)
fc3 = nn.Linear(fc2_units, action_size)
```

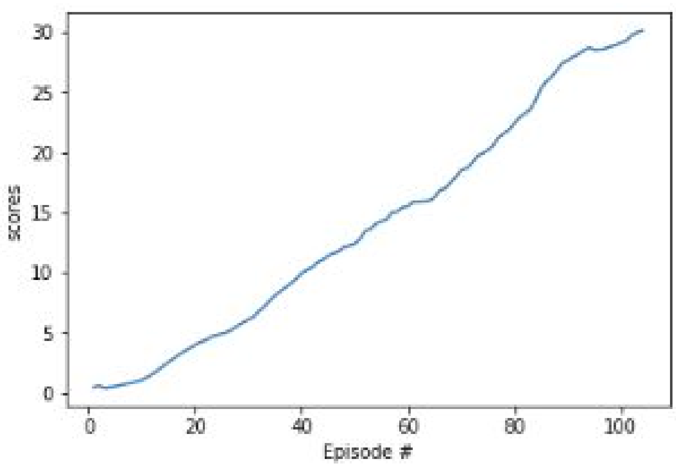
Critic network

```
fcs1 = nn.Linear(state_size, fcs1_units)
 fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
 fc3 = nn.Linear(fc2_units, 1)
```

where `state_size=33, action_size=4, fc1_units=400, fc2_units=300`

2. Plot of Rewards: Optimize the hyperparameters

Many attempts were made with different hyperparameters, different tricks to sample/learn to solve the environment as shown in the following table. The final attempt #24 as shown in the following figure demonstrated that 104 episodes were needed to solve the environment (i.e. to receive an average reward 30 points) over all 20 agents.



Attempt	max_steps	batch_size	actor_learn_rate	critic_learn_rate	update_every	num_updates	tau	gamma	noise_sigma	noise_factor_decay	layer_sizes	score @75 episode	what's changed
1	300	256	0.0005	0.001	10	10	0.0005	0.99	0.2	0.000001	[128, 128]	0.25	POR
2	300	256	0.0005	0.001	20	10	0.0005	0.99	0.2	0.000001	[128, 128]	0.2	increased buffer size from 3e5 to 3e6 update_every from 10 to 20
3	300	256	0.0005	0.001	20	10	0.0005	0.99	0.2	0.000001	[128, 128]	0.12	torch.nn.utils.clip_grad_norm(self.critic_local.parameters(), 1)
4	1000	256	0.001	0.001	20	10	0.005	0.99	0.2	0.000001	[128, 128]	0.2	to learn faster; tau or soft update of target parameters= 0.0005 to 0.005 actor_learn_rate=0.0005 to 0.001 max_steps=300 to 1000
5	1000	256	0.001	0.001	20	10	0.005	0.99	0.2	0.000001	[128,	0.2	decay noise: actions += self.epsilon *

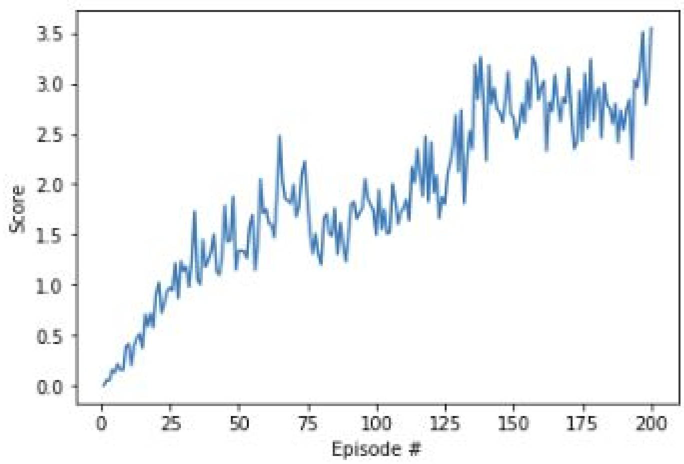
											128]		self.noise.sample() #decay noise
6	1000	1024	0.001	0.001	20	10	0.005	0.99	0.2	0.000001	[128, 128]	0.2	batch_size=256 to 1024
7	1000	1024	0.001	0.001	10	10	0.005	0.99	0.2	0.000001	[128, 128]	0.15	update_every=20 to 10
8	1000	1024	0.001	0.001	10	10	0.005	0.99	0.2	0.000001	[128, 128]	1.5	for _ in range(num_updates): self.learn(experiences, GAMMA)
9	1000	1024	0.001	0.001	20	10	0.005	0.99	0.2	0.000001	[128, 128]	10	UPDATE_EVERY=10 to 20 num_updates=10
10	1000	1024	0.001	0.001	20	10	0.005	0.99	0.2	0.000001	[128, 128]	2.5	avescore_allagents_oneepisode = np.mean(scores) scores_aveofallagents_5episode_deque.append(avescore_allagents_oneepisode) avescore_allagents_5episode = np.mean(scores_aveofallagents_5episode_deque) scores_allagent_allepisode_list.append(avescore_allagents_5episode) avescore_allagents_allepisodes_list.append(avescore_allagents_5episode)
11	1000	512	0.001	0.005	20	10	0.005	0.99	0.2	0.000001	[128, 128]	1.21	max_steps=300 -->max_steps=1000 actor_learn_rate=0.001 critic_learn_rate=0.001-> critic_learn_rate=0.005 batch_size=1024 ->batch_size=512
12	300	128	0.001	0.005	20	20	0.005	0.99	0.2	0.000001	[128, 128]	0.23	max_steps=1000 max_steps=300 update_every=20 num_updates=10->20 batch_size=512->batch_size=128
13	300	512	0.001	0.005	20	10	0.005	0.99	0.2	1.00E+06	[128, 128]	0.32	num_updates=20 to10 batch_size=128 -> batch_size=512
14	300	1024	0.001	0.005	20	10	0.005	0.99	0.2	1.00E+06	[128, 128]	0.2	batch_size=512 -> batch_size=1024 num_updates=20 ->10
15	300	128	0.001	0.005	20	10	0.005	0.99	0.5	1.00E+06	[128, 128]	0.11	NOISE_SIGMA=0.2-> 0.5
16	300	128	0.001	0.005	20	10	0.005	0.99	0.2	1.00E+06	[128, 128]	0.22	noise_sigma=0.5->0.2 turned off batchnorm
17	300	1024	0.001	0.005	20	10	0.005	0.99	0.2	1.00E+06	[128, 128]	0.15	batch_size=128 ->1024
18	300	128	0.001	0.001	20	10	0.005	0.99	0.2	1.00E+06	[128, 128]	0.13	critic_learn_rate=0.005->0.001
19	300	128	0.001	0.001	20	10	0.001	0.99	0.2	1.00E+06	[128, 128]	0.15	tau=0.005 ->tau=0.001
20	300	128	0.001	0.005	20	10	0.005	0.99	0.2	1.00E+06	[400, 300]	0.16	layer_sizes=[128,128] ->[400,300]
21	300	128	0.001	0.005	20	10	0.005	0.99	0.2	1.00E+06	[128, 128]	0.18	removed for _ in range(num_updates):
22	300	128	0.001	0.005	20	10	0.005	0.99	0.2	1.00E+06	[128, 128]	0.2	fixed scores to score
23	300	128	0.001	0.005	20	10	0.005	0.99	0.2	1.00E+06	[128, 128]	0.16	fixed scores to score
24	1000	1024	0.001	0.001	20	10	0.005	0.99	0.2	1.00E+06	[128, 128]	18	Final: repeat attempt # 9 hyperparameters

The key breakthrough occurred at Attempt #8, #9. In agent_reacher.py, two major points were helpful to start learning.

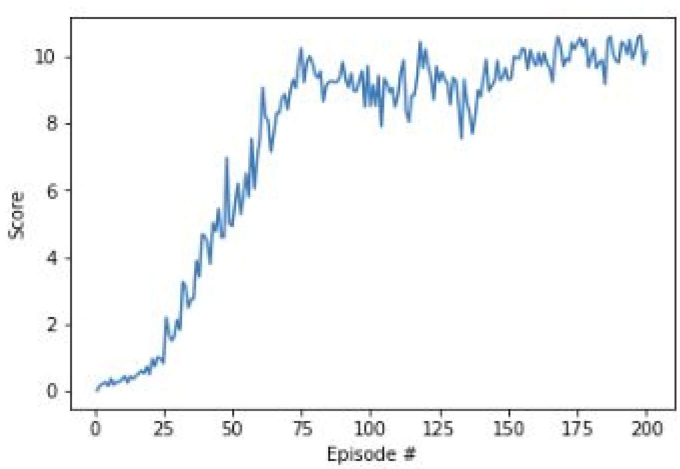
```
def step(self, states, actions, rewards, next_states, dones, t):
    for state, action, reward, next_state, done in zip(states, actions, rewards, next_states, dones):
        self.memory.add(state, action, reward, next_state, done)

    if len(self.memory) > BATCH_SIZE and t % UPDATE_EVERY == 0:
        experiences = self.memory.sample()
        for _ in range(num_updates):
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)
```

In Attempt #8, `for _ in range(num_updates)`: was added so learning takes only less than `num_updates` times (hyperparameter, set as 10) every `UPDATE_EVERY` (set as 10) step.



In Attempt #9, `UPDATE_EVERY` was updated from 10 to 20 to learn more stably. Agents step and memory is filled and memory is sampled every `UPDATE_EVERY` (increased from 10 to 20) but learning occurs only `num_updates`(10) times.



3. Ideas for Future Work

For improving the agent's performance, the various deep RL algorithms on continuous control tasks, such as, REINFORCE, TNPG, RWR, REPS, TRPO, CEM, CMA-ES and DDPG, may be pursuited based on suggestion in the review paper.