

Report: Continuous Control Project

Environment details:

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.

1. The goal of your agent:

To maintain its position at the target location for as many time steps as possible.

2. Project Details:

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 Unity environment used contains 20 identical agents, each with its own copy of the environment.

3. Environment solved criteria:

Agents must get an average score of +30 (over 100 consecutive episodes, and over all agents).

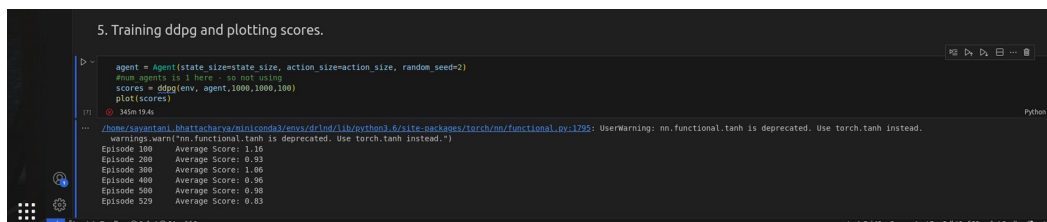
Specifically,

- 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.

Learning Algorithm:

Deep Deterministic policy gradient method is used along with experienced replay. The optimization of network is done using Adam optimizer. Four networks are being trained, actor-local, actor-target, critic-local and critic-target. Ornstein-Uhlenbeck process of adding noise is used to increase exploratory observations. All the 20 agents store the state in the same replay buffer, which is used to train the model. Thus the network being trained is same for all of them. This is possible to do because they have the same environment and individual step does not depend on the other agent's steps. Each of the network consists of normalization layer, three fully connected layers with leaky Rectified Linear Unit (ReLU) activation function between them.



```
5. Training ddpg and plotting scores.

In [ ]: agent = Agent(state_size=state_size, action_size=action_size, random_seed=2)
        #num agents is 1 here - so not using
        scores = ddpg(env, agent, 1000, 1000, 100)
        plot(scores)

Out [ ]: 345m 19.4s

/home/sayantani.bhattacharya/miniconda3/envs/drind/lib/python3.6/site-packages/torch/nm/functional.py:1793: UserWarning: nm.functional.tanh is deprecated. Use torch.tanh instead.
  warnings.warn("nm.functional.tanh is deprecated. Use torch.tanh instead.")

Episode 100 Average Score: 1.16
Episode 200 Average Score: 0.93
Episode 300 Average Score: 1.96
Episode 400 Average Score: 0.96
Episode 500 Average Score: 0.98
Episode 600 Average Score: 0.83
```

Initially I was getting pretty low rewards, making the following changes helped me a lot: 1. to simultaneously train 20 agents (second env over the first one provided). 2. Adding a normalisation layer to all the networks. And 3. to use leaky ReLU over ReLU.

Layers:

1. Actor (local and target):

Batch normalization is applied to the state first, followed by the application of leaky ReLU activation functions to the output of the first and second hidden layers. Finally, the output is passed through a tan h activation function in the last layer to ensure that the action values are within the range [-1, 1].

n1: batch normalisation layer, that is used to stabilize training and speed up convergence by normalizing the input to the neural network. Here the input and output dimension is the state_size.

fc1: fully connected layer (linear layer) taking input from the normalised state to fc1_units.

fc2: fully connected layer mapping from fc1_units to fc2_units.

fc3: fully connected layer mapping from fc2_units to action_size

2. Critic (local and target):

Batch normalization is applied to the state first, followed by the application of leaky ReLU activation functions to the output of the first and second hidden layers. Finally, the output is passed through a tan h activation function in the last layer to ensure that the action values are within the range [-1, 1].

n1: batch normalisation layer, that is used to stabilize training and speed up convergence by normalizing the input to the neural network. Here the input and output dimension is the state_size.

fcs1: fully connected layer (linear layer) taking input from the normalised state to fcs1_units.

fc2: fully connected layer mapping from fc1_units+action to fc2_units.

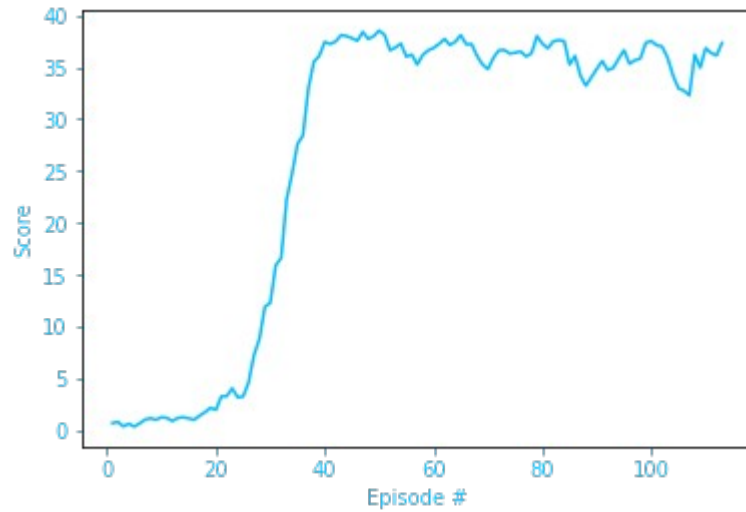
fc3: fully connected layer mapping from fc2_units to 1.

Hyper-parameters:

Parameters	Value	Description
BUFFER_SIZE	1e5	Replay buffer size
BATCH_SIZE	128	Batch size
GAMMA	0.99	Discount factor for expected rewards
TAU	1e-3	soft update for the target network weights
LR_ACTOR	1e-4	Learning rate of the actor network
LR_CRITIC	1e-3	Learning rate of the critic network
WEIGHT_DECAY	0	L2 weight decay
n_episodes	1000	Maximum number of training episodes
max_t	1000	Maximum number of timesteps per episode
print_every	100	Number of episodes for calculating the average score value
fc1_units	400	Hidden layer 1 unit for both models
fc2_units	300	Hidden layer 2 unit for both models

Plot of scores:

Environment solved in 113 episodes! Average Score: 30.06



Ideas of Future Work:

1. I plan to integrate **prioritized experience** replay to this model. The bigger the error, the more we expect to learn from it. So, the error value is reflective of priority and is stored along with each corresponding tuple in the replay buffer. And while selecting samples from the replay buffer we use this value for guided selection.
2. I would also like to try out Distributed distributional policy gradient later some time.