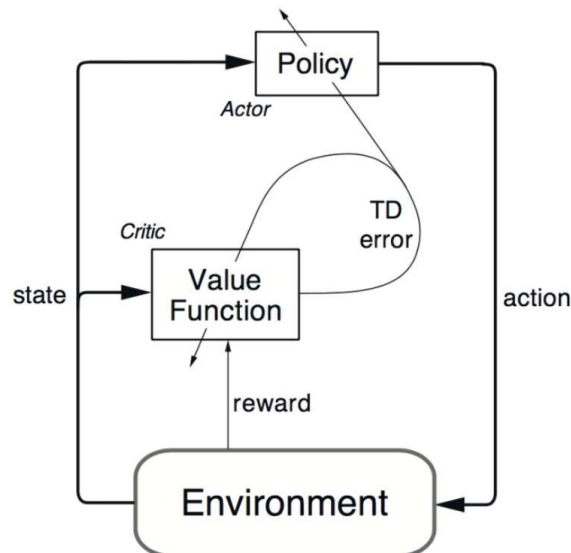


Report

The solution is based on DDPG (Deep Deterministic Policy Gradient) architecture.

The DDPG approach is an actor-critic policy gradient algorithm. In this case two neural networks are used. The first network is called the 'actor' and approximates the optimal policy. The optimal policy is the policy that maximizes the reward of an agent when it is followed. In this case the agent performs the action in any given state, which provide the highest reward for this state.

The second neural network is called the 'critic', it evaluates the policy by estimating the reward following that approximately optimal policy given by the actor. To improve, actor and critic network update through their loss functions. Actor and critic perform in a loop to improve by comparing the estimated reward to the actual reward and the actor there for gets closer to the true optimal policy. The following image describes the architecture (source: <https://arxiv.org/abs/1509.02971>)



One of the challenges of the DDPG approach, is that it is not really performing exploration for the action. An approach to overcome this is to add noise to the parameter or action space. In this case the Ornstein-Uhlenbeck Random process is used to introduce noise to select a specific action in a timestep.

So what happens, when the training starts?

In the beginning the networks for actor and critic are initialized with random values. The actor then gets the state of the environment and the actor returns an action based on the policy. Noise (Ornstein-Uhlenbeck) is added and the action on the step is determined. The action taken leads to a new state of the environment and a reward is given to the actor. This happens for every time step t in an episode. Information on state, taken action, reward, next state is stored in the replay buffer. This holds an amount of these tuples and randomly gives a sample to the critic network. The critic network approximates the expected Q value. Based on the mean squared value between the expected and the actual Q value

e, both networks are updated. The actor network is updated towards a better policy using gradient ascent.

Below is a description of the algorithm (source: <https://arxiv.org/abs/1509.02971>)

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ .
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
 Initialize a random process \mathcal{N} for action exploration
 Receive initial observation state s_1
 for t = 1, T **do**
 Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
 Execute action a_t and observe reward r_t and observe new state s_{t+1}
 Store transition (s_t, a_t, r_t, s_{t+1}) in R
 Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
 Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
 Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
 Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

Update the target networks:

$$\begin{aligned}\theta^{Q'} &\leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}\end{aligned}$$

end for
end for

For more details see the following paper on DDPG(<https://arxiv.org/abs/1509.02971>)

For the implementation, the foundation for the solution was provided in the Udacity repository for the Deep Reinforcement Learning Nanodegree (<https://github.com/udacity/deep-reinforcement-learning>)

While the training approach is based on the agent.py in the repository folder ddpq-pendulum, the model.py file is based on the model definition provided in the ddpq-bipedal folder of the same repository. Agent.py has been modified to accommodate for multiple agents in the environment, while model.py has not been changed from its original version (except of the fc_unit settings).

1. Environment Description

```
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
    Number of Brains: 1
    Number of External Brains : 1
    Lesson number : 0
```

```

Reset Parameters :
    goal_speed -> 1.0
    goal_size -> 5.0
Unity brain name: ReacherBrain
    Number of Visual Observations (per agent): 0
    Vector Observation space type: continuous
    Vector Observation space size (per agent): 33
    Number of stacked Vector Observation: 1
    Vector Action space type: continuous
    Vector Action space size (per agent): 4
    Vector Action descriptions: , , ,

```

2. State and action spaces

```

Number of agents: 20
Size of each action: 4
There are 20 agents. Each observes a state with length: 33
The state for the first agent looks like: [ 0.00000000e+00 -
4.00000000e+00  0.00000000e+00  1.00000000e+00
-0.00000000e+00 -0.00000000e+00 -4.37113883e-08  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00 -1.00000000e+01  0.00000000e+00
 1.00000000e+00 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  5.75471878e+00 -1.00000000e+00
 5.55726624e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00
-1.68164849e-01]

```

3. Model Description (actor_target, critic_target):

```

Actor(
  (fc1): Linear(in_features=33, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=4, bias=True)
)
Critic(
  (fcs1): Linear(in_features=33, out_features=256, bias=True)
  (fc2): Linear(in_features=260, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=1, bias=True)
)

```

The summary above provides the description of the underlying deep neural networks used in the approach as part of this solution. The solution uses two networks, one for actor and one for the critic.

4. Hyperparameter:

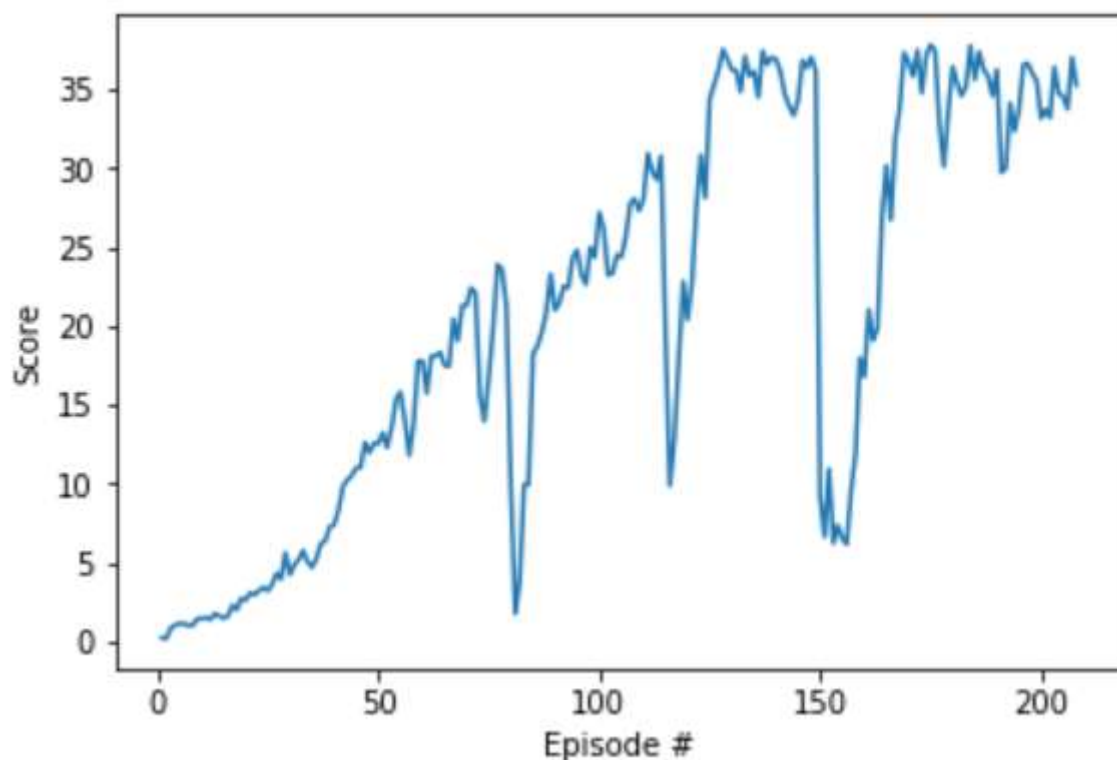
Parameter	Value
n_episodes (int): maximum number of training episodes	2000
max_t (int): maximum number of time steps per episode	1000
num_agents : amount of agents in the environment	20
DDPG Agent HyperParameters	
BUFFER_SIZE: replay buffer size	int(1e5)
BATCH_SIZE: minibatch size	64
GAMMA: discount factor	0.99

TAU: for soft update of target parameters	1e-3
LR ACTOR: learning rate for actor	1e-4
LR_CRITIC: learning rate for critic	1e-4
WEIGHT_DECAY: L2 weight decay	0

5. Training Summary:

```
....
Episode 202    Mean score (last 100 episodes): 29.49
Episode 203    Mean score (last 100 episodes): 29.62
Episode 204    Mean score (last 100 episodes): 29.72
Episode 205    Mean score (last 100 episodes): 29.82
Episode 206    Mean score (last 100 episodes): 29.91
Episode 207    Mean score (last 100 episodes): 30.00
Episode 208    Mean score (last 100 episodes): 30.07
```

Environment solved in 208 episodes! Mean score (last 100 episodes): 30.07



The plot above shows the agents score achieved during the training episodes. the agent can receive an average reward (over 100 episodes, and over all 20 agents) of >30 after 208 episodes.

6. Ideas for optimizing the solution:

- **Optimizing hyperparameters:** During the development of the exercise I experimented with the hyperparameters (especially with the

fc_units, learning rates, tau and batch size) I strongly believe that there is plenty of room of further optimization here with automating the experimentation efforts. Creating a script that changes parameters based on statistical methods and combining it with an automated AWS script to spin up multiple environments, can give a good solution to scale experimentation efforts.

- **Changing the Algorithm:** As an option Proximal Policy Optimization (PPO) and Distributed Distributional Deterministic Policy Gradients (D4PG) methods could be explored.