Project: Continuous Control

The goal of this project is to train an Agent to control a double-jointed arm to continiously reach a moving target.
A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Therefore, the goal of an 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 is a number between -1 and 1.

The task is episodic, and in order to solve the environment, an agent must get an average score of +30 over 100 consecutive episodes.

The project environment is running inside the Unity simulation engine and trained Agent can interact with it using the UnityAgents API. The variant of Unity environment chosen for this project contains 20 identical agents, each with its own copy of the environment. Therefore to achieve the goal of the project all agents of the environment must get an average score of +30 (over 100 consecutive episodes, and over all agents). Specifically, after each episode, rewards that each agent received are added up and the average of these 20 scores is taken as the resulting episode score.

Learning Algorithm

Because the action space of project environment is not discrete (action values variate in a range (-1,+1)), it is not possible to use such proven method as Q-learning (or similar) for training an agent, because in continuous spaces finding the greedy policy requires an optimization of at at every timestep. Such optimization would be too slow to learn the Agent in large state and action spaces online.
Therefore to achieve the goal of this project I decided to use the Deep Deterministic Policy Gradient (DDPG) method, which combines the actor-critic approach based on the DPG algorithm with elements of Deep Q Network (DQN) method for training the Agent.
The DDPG algorithm maintains a parameterized actor function $\mu(s|\theta\ \mu)$ which specifies the current policy by deterministically mapping states to a specific action. The critic Q(s, a) is learned using standard the Bellman equation similarly to the Q-learning method.
DDPG algorithm also includes the following two improvements initially introduced as part DQN algorithm to minimize the known weak points of standard Q-Learning method

- **Fixed Q-targets** The TD target is dependent on the network parameters that we're trying to learn. This can lead to divergence. To remove this effect, DDPG algorithm uses separate networks both for actor and critic with identical architecture. The target networks are being updated with weights from local networks after N amount of Agent's actions, and not by simple copying the weights,but during a so-called soft update with hyperparameter TAU. And local networks get updated regularly after each Agent's action.
- **Experience Replay** When the agent interacts with the environment, the sequence of experience tuples can be highly correlated. Agent that learns from each of these experience tuples in sequential order runs the risk of getting biased by the effects of this correlation. By sampling from

stored experience records at random, the action values can be prevented from oscillating or unwanted diverging.Transitions are sampled from the environment according to the exploration policy and the tuples (St, At, Rt, St+1) are stored in the replay buffer. When the replay buffer gets full the oldest samples are discarded. At each timestep the actor and critic are updated by sampling a minibatch uniformly from the buffer. Because DDPG is an off-policy algorithm, the replay buffer can be large, allowing the algorithm to benefit from learning across a set of uncorrelated transitions.

The exploration policy of actor μ(s|θ μ) is constructed by adding noise sampled from Ornstein-Uhlenbeck noise process, which is generally used to generate temporally correlated exploration for exploration efficiency in physical control problems with inertia.

Implemented algorithm for training an Agent consist of following steps:

- Initialize replay memory buffer.
- Initialize the actor's and critic's networks with random weights.
- Clone the actor's and critic's networks, and call them the *target network*.
- *For each episode:*
    - *Initialize a random process for action exploration*
    - Initialize the starting state.
    - *For each time step:*
        - Select an action *Via exploration or exploitation*
        - Execute selected action in an emulator and observe reward and next state.
        - Store experience in replay memory.
        - Sample random batch from replay memory.
        - Preprocess states from batch.
        - Calculate loss between output Q-values and target Q-values.
        - Update critic by minimizing the loss using Gradient descent
        - Update the actor policy using the sampled policy gradient
        - After X time steps, weights in the target networks for actor and critic are updated to the weights of the corresponding policy networks.

The Actor's Neural Network has following architecture :

- Input (33 nodes)
- Fully Connected Hidden Layer (256 nodes, Relu activation)
- Fully Connected HIdden Layer (128 nodes, Relu activation)
- Output (4 nodes)

Optimizer type: Adam optimizer with a learning rate LR=1e-3. The final output layer of the actor network is a tanh layer, to bound the resulting action values in a range (-1; +1).

The Critic's Neural Network has following architecture :

- Input (33 nodes)
- Fully Connected Hidden Layer (256 nodes, Relu activation)
- Fully Connected HIdden Layer (128 nodes, Relu activation)
- Output (1 nodes)

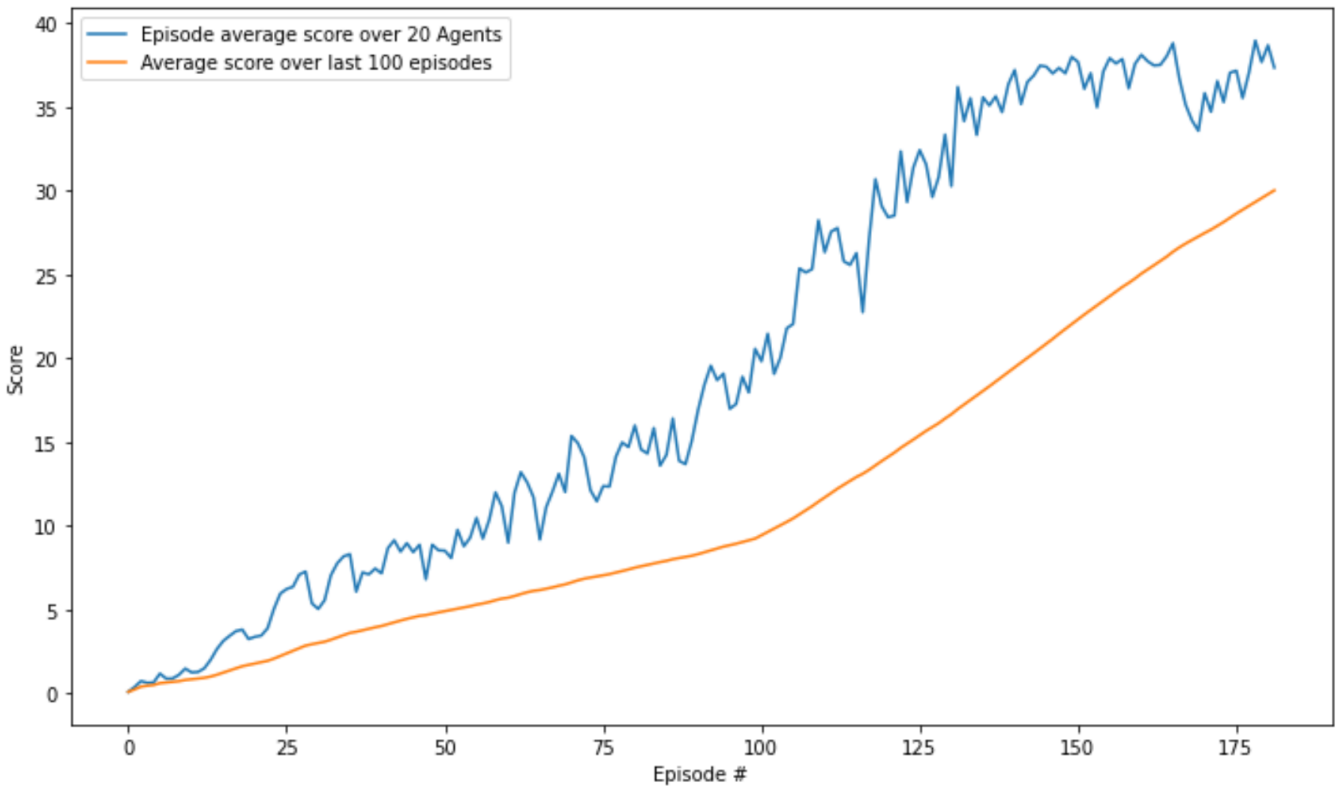
Optimizer type: Adam optimizer with a learning rate LR=1e-3.

Agent training hyperparameters:

- Max episode length: 1000 time steps
- Experience replay memory buffer size: 100000
- Batch size: 128
- discount factor for prioritizing future rewards (gamma): 0.99
- Fusion ratio hyperparameter for soft update of weights of target network (Tau): 0.001
- Frequency of updates for the weights of target networks: every 20 actions of Agent

Plot of Rewards

Following graph demonstrates the progression of rewards score per episode during the Agents' training phase. Additionally, the progression of mean score over the 100-episodes sliding window is displayed in orange color. The graph shows that the group of Agents successfully learned to reach an average reward of at least 30 over all Agents of environment and over 100 consecutive episodes. The Agents managed to achieve this goal after 182 episodes of learning.



Conclusions and ideas for Future Work

For solving the goal of this project I have used the Deep Deterministic Policy Gradient method, which is quite straightforward to implement. The main challenge in making this implementation successful in solving the project goal was to find a proper combination of hyperparameters for all components of the learned Agent via multiple sessions of learning. Initial try to achieve the target goal of the project using a single agent Unity environment was unfortunately not successful. After multiple experiments with parameters the maximum average score over 100 episodes could not exceed 23. Therefore it was decided to switch to multi-agent project environment, that allowed to reach target score

Following are my main observations from fine tuning of hyperparameters for implemented solution:

- Changing the number of nodes for the first hidden layer of actor and critic network from 256 to 128 did not allow to achieve a maximum average score 30 over the last 100 episodes. The maximum score achieved was 8.5 at 110th episode of training, and further training did not bring any improvements.
- Batch sizes of 128 and 64 seems to show the best learning results
- Using more than 1000 timesteps per episode did not show any benefit in training results. I suspect that an even smaller amount of time steps (e.g. 600-800) could do the job, but then some extra play with parameter values of Noise Process (used for policy exploration) and learning rates of actor and critic networks would be needed.

One of the ways to improve the current solution could be introducing a batch normalization for the first layer of both actor and critic networks., since different components of the observation space have different physical units (for example, positions versus velocities). This leads to an increase of the learning time for the networks and makes it more difficult to find proper hyper-parameters. Therefore introducing his technique into the algorithm would help to normalize each dimension across the samples in a minibatch to have unit mean and variance.
Another future improvement can be introduction of noise factor (epsilon) decay over network learning, to decrease the space for exploration the more 'experienced' the agent gets.

Also alternative ways to solve the goal of this project could be via implementing one of following approaches:

- Proximal Policy Optimization
- Asynchronous advantage actor critic (A3C)
- D4PG

which are more complex in implementation, but show quit promising results according declared results in above mentioned research papers