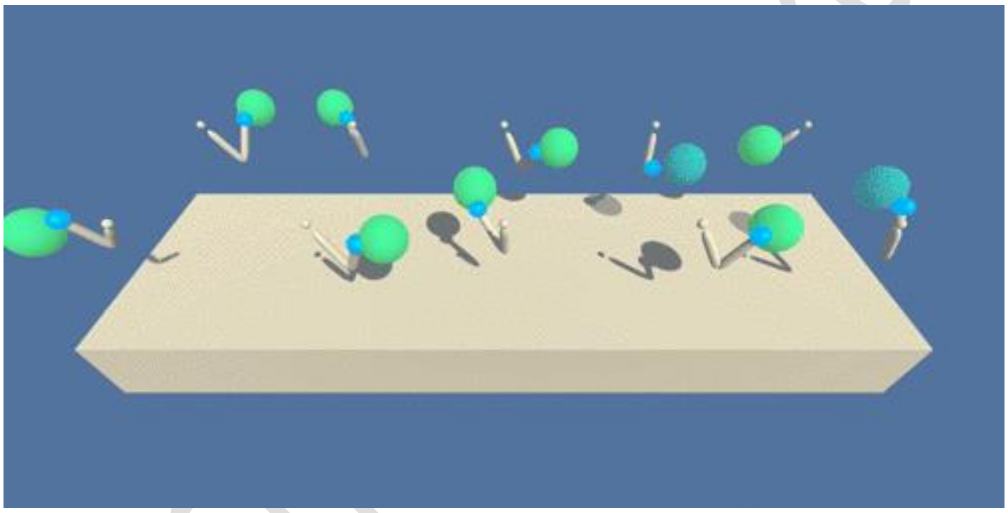
A simple project to implement Deep Deterministic Policy Gradient (DDPG) network agent using Python, PyTorch and Unity ML-Agent's environment

Understanding Environment

This project uses a modified version of Unity ML-Agents Reacher example environment. The environment includes a double-joined arm that can move to target locations. The goal of the agent is to maintain its position at the target location for as many time steps as possible. For the version of the Reacher environment being used in this project, it is considered that the agent has solved the environment when the average score over the last 100 episodes >= +30.0



Example view of agents (20) of the Reacher environment.
(The task is to maintain the position of double-joined arm at the target location for as many time steps as possible)

State Space

The agent is trained from the vector input data (not the raw pixels as input data). The state space consists of 33 variables corresponding to position, rotation, velocity and angular velocities of the arm.

Action Space

Since we are training an agent to take actions (that are continuous values), the action here is vector of 4 numbers, corresponding to the torque applicable to two joints. And, every entry in the action vector (consisting of 4 entries) should be a number between -1 and +1.

Reward

A reward of +0.1 is provided for each step that the agent's hand is in the goal location.

Training Environment – Versions

For this project, two separate version of the Unity Environment are provided:

- The first version contains a single agent.
- The second version contains 20 identical agents, each with its own copy of the environment.
    a. The second version is useful for algorithms like, PPO, A3C, D4PG that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

Solving the environment

Option 1: Solve the environment with single agent

- The task is episodic, and in order to solve the environment, the agent must get an average score of +30 over 100 consecutive episodes.
- You may refer to train_agent.py module. 1 agent is assigned with 1 brain, which encapsulates the decision-making process.

Option 2: Solve the environment with 20 identical agents

- The barrier for solving the second version of the environment is slightly different, to take into account the presence of many agents. In particular, the 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 those 20 scores.
- This yields an average score for each episode (where the average is over all 20 agents).
- You may refer to train_agent_dist.py module. Due to hardware limitations and training time constraint, I have used the strategy to assign 1 brain to 20 agents.

Learning Algorithm

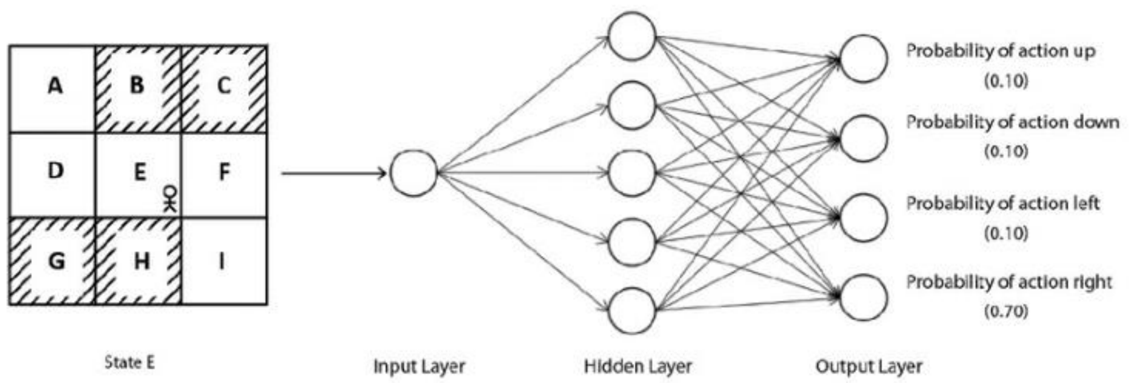
One of the disadvantages of the value-based method is it is suitable only for discrete environments (environments with discrete action space), and we cannot apply value-based methods in continuous environments (environments with a continuous action space). Most of the real-world problems have continuous action space, say, a self-driving car, or a robot learning to walk and more.

Policy gradient method – REINFORCE Intuition

Policy gradient is one of the most popular algorithms in deep reinforcement learning. As we have learned, policy gradient is a policy-based method by which we can find optimal policy without computing the Q function. It finds the optimal policy by directly parameterizing the policy using some parameter θ

The policy gradient method uses a stochastic policy, we select an action based on the probability distribution over the action spaces. Say we have a stochastic policy π, then it gives the probability of taking an action a given the state "s". It can be denoted by π (a/s). In the policy gradient method, we use a parameterized policy, so we can denote our policy π_θ, where theta indicates that our policy is parameterized, where θ is the parameter of the neural network.

Say we have a neural network with a parameter θ. First, we feed the state of the environment as an input of the network and it will output the probability of all the actions that can be performed in the state. That is, it outputs a probability distribution over an action space. We have learned that the policy gradient, we use a stochastic policy. So, the stochastic policy selects an action based on the probability distribution given by the neural network. In this way, we can directly compute the policy without the Q function.


A
B
C
D
E
F
G
H
I
State E
Input Layer
Hidden Layer
Output Layer
Probability of action up (0.10)
Probability of action down (0.10)
Probability of action left (0.10)
Probability of action right (0.70)

Policy network

Objective Function: $$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta log\pi_\theta(a_t/s_t)R(\tau)]$$

Learning/updating our network parameter θ : $\theta = \theta + \alpha\nabla_\theta J(\theta)$

More precisely,

$$\theta = \theta + \alpha \underbrace{\frac{1}{N}\sum_{i=1}^{N}}_{\boxed{\text{Sum over N trajectories}}} \underbrace{\left[\sum_{t=0}^{T-1} \nabla_\theta log\pi_\theta(a_t/s_t)R(\tau)\right]}_{\boxed{\text{For each step in the trajectory}}}$$

Problem with REINFORCE

- Monty Carlo type, and hence the task has to be episodic.
- Output or predicted action is not deterministic rather a stochastic one.

Deterministic Policy Gradient

Traditionally, policy gradient algorithms are being used with stochastic policy function. That means policy function is represented as a distribution over actions $\pi(./s)$. For a given state, there will be probability distribution for each action in the action space. In DPG, instead of stochastic policy, π, deterministic policy $\mu(./s)$is followed. For a given state, s, there will be deterministic decision: a = μ(s) instead of distribution over actions.

The grad objection function of the stochastic Policy gradient algorithm can be written as below:

$$\nabla_\theta J(\theta) = E[\nabla_\theta log\pi_\theta(a/s)Q^w(s,a)]$$

We can rewrite the above equation for deterministic policy by replacing pi with mu. Till 2014, deterministic policy to a policy gradient algorithm was not possible. In DPGA paper, David Silver conceived the idea of DPG and provided the proof.

$$\nabla_\theta J(\theta) = E[\nabla_\theta \mu_\theta(s)\nabla_a Q^\mu(s,a)]$$

Q-learning

Q Learning is a value-based off-policy temporal difference (TD) reinforcement learning. Off-policy means as agent follows a behaviour policy for choosing the action to reach the next state,

s_t+1 from state ***s_t***. From ***s_t+1***, it uses a policy pi, that is different from behaviour policy. In Q-learning, we take absolute greedy action as policy pi, from the next state ***s_t+1***.

$$\boldsymbol{Q(s,a) = r(s,a) + \gamma max_a Q(s',a)}$$

As we discussed in the action-value function, the above equation indicates how we compute the Q-value for an action a starting from state s in Q learning. It is the sum of immediate reward using a behaviour policy (eps-soft, etc). From state s_t+1, it takes an absolute greedy action (choose the action that has the maximum Q value over the other actions).

Actor-critic

In simple term, Actor-Critic is a Temporal Difference (TD) version of Policy gradient. It has two networks: Actor and Critic. The actor decides which action should be taken and critic inform the actor how good was the action and how it should adjust it. The learning of the actor is based on policy gradient approach. In comparison, critics evaluate the action produced by the actor by computing the value function.

Difference between on-policy and off-policy

On-Policy: The agent behaves using one policy and also tried to improve the same policy. That it, in the on-policy method, we generate the episodes using one policy and also improve the same policy iteratively to find the optimal policy. For instance, the on-policy MC control method, we generate the episodes using the policy p1, and we also try to improve the same policy p1 on every iteration to compute the optimal policy

Off-policy: The agent behaves using one policy ***b*** (behaviour policy) and tries to improve a different policy ***p*** (target policy). That is, in the off-policy method, we generate episodes using one policy and we try to improve the different policy iteratively to find the optimal policy. Q-learning algorithm is one such example

The target network are time-delayed copies of the original networks that slow track the learned networks. Using these target value networks greatly improves the stability in learning. Here’s why: In methods that do not use target networks, the update equations of the network are interdependent on the values calculated by the network itself, which makes it prone to the divergence.

For example:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a) + \gamma max \overbrace{Q(s',a')}^{\text{This depends Q function itself (at the moment it is being optimized)}} - Q(s,a)]$$

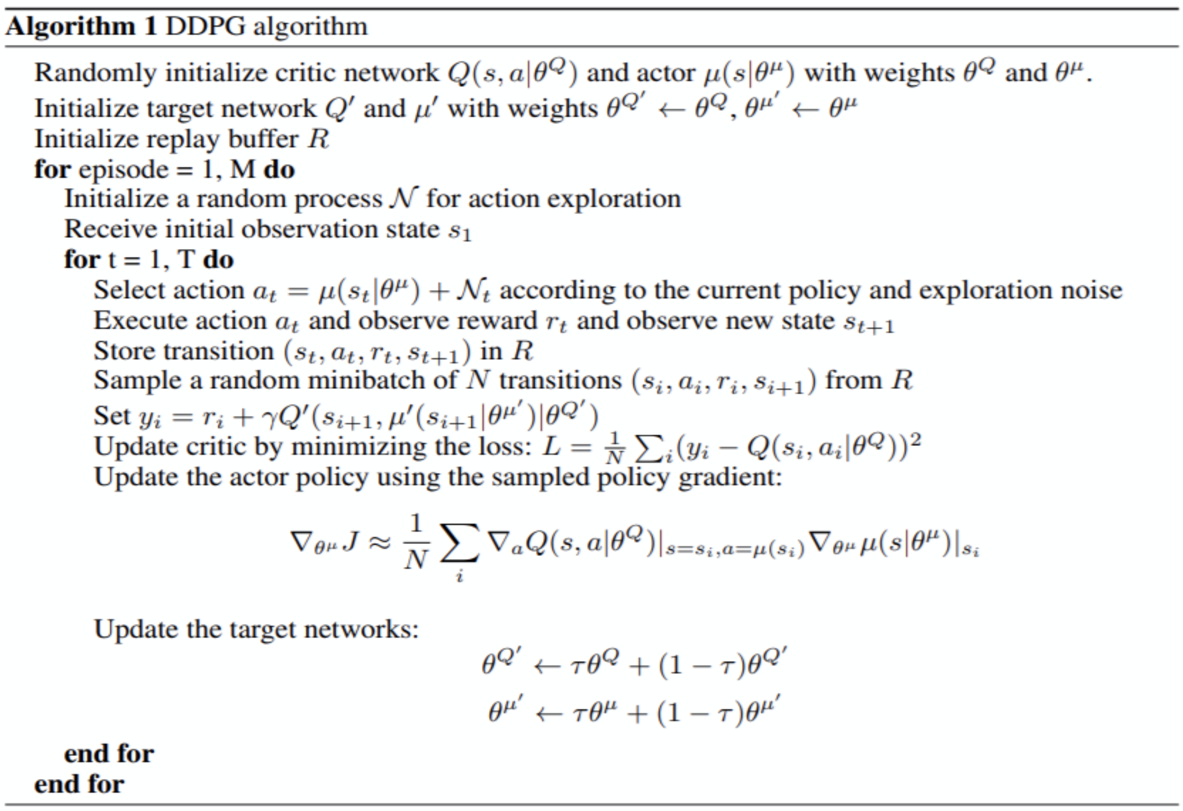
Deep Deterministic Policy Gradient (DDPG)

DDPG is a model-free off-policy actor-critic algorithm that combines Deep Q learning (DQN) and DPG. Original DQN works in a discrete action space, DPG extends it to the continuous action space while learning a deterministic policy.

As it is an off-policy algorithm, it uses two separate policies for the exploration and updates. It uses a stochastic behaviour policy for the exploration and deterministic policy for the target update.

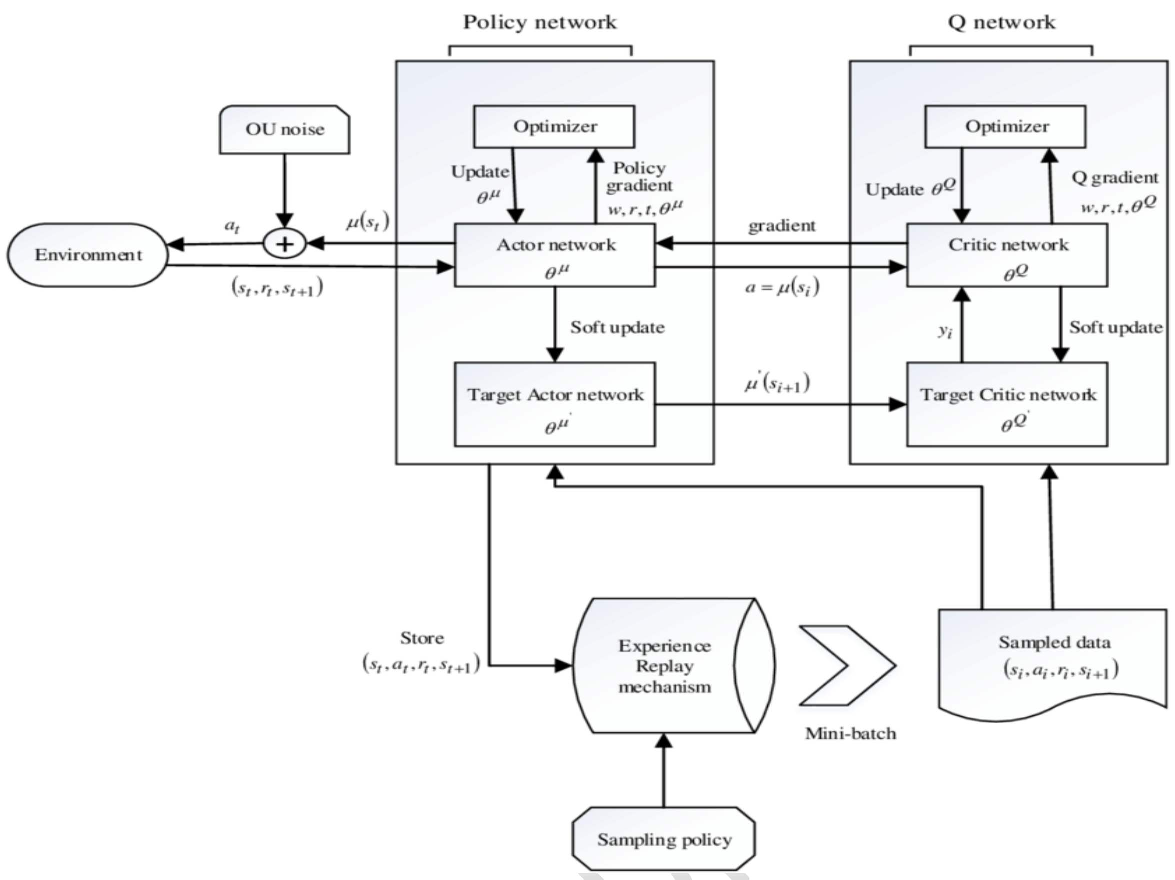
DDPG is an actor-critic algorithm; it has two networks: actor and the critic. Technically, the actor produces the action to explore. During the update process of the actor, TD error from a critic is used. The critic network gets updated based on the TD error similar to Q-learning update rule.

We did learn the fact that the instability issue that can raise in Q-Learning with the deep neural network as the function approximator. To solve this, experience replay and target networks are being used.



```
Algorithm 1 DDPG algorithm
  Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
  Initialize target network $Q'$ and $\mu'$ 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:
          $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
          $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
    end for
  end for
```

Pseudocode for DDPG algorithm (Lillicrap et al, 2015)


Policy network
Q network
OU noise
Optimizer
Update θ^{μ}
Policy gradient w,r,t,θ^{μ}
Optimizer
Update θ^{Q}
Q gradient w,r,t,θ^{Q}
Environment
a_t
$\mu(s_t)$
(s_t, r_t, s_{t+1})
Actor network θ^{μ}
gradient
$a=\mu(s_i)$
Critic network θ^{Q}
Soft update
y_i
Soft update
Target Actor network $\theta^{\mu'}$
$\mu'(s_{i+1})$
Target Critic network $\theta^{Q'}$
Store (s_t, a_t, r_t, s_{t+1})
Experience Replay mechanism
Mini-batch
Sampled data (s_i, a_i, r_i, s_{i+1})
Sampling policy

The DDPG algorithm structure framework

GitHub Code

DDPG_Continuous_Control is the GitHub repository that contains the code, environment and additional scripts that can be cloned to local repository and used to train the agent. However, one can use Continuous_Control.ipynb to quickly experiment with DDPG agent. The repository also includes the links to download the Unity Environment (Reacher) for testing. This Unity application and testing environment was developed using ML-Agents Beta v0.4. The version of this Reacher environment employed for this project was developed by the Udacity Deep Reinforcement Learning Nanodegree Team. For more information about his course visit:

Installation and get started

The repository contains ***/python/*** directory, which contains the ML-Agents files and dependencies required to run the Reacher Environment. In addition to that, ***requirements.txt*** and ***setup.py*** script can be used to install required virtual environment to train the agent.

Deep Deterministic Policy Gradient (Actor-Critic Network)

- The “Critic” estimates the value function. This could be the action-value (the Q value) or state-value (the V value)
- The “Actor” updates the policy distribution in the direction suggested by the Critic (such as with policy gradients).

And both the Critic and the Actor functions are parameterized with neural networks.

In the off-policy method, we use two policies called the behaviour policy and the target policy (both. As the name suggests, we behave (generate episodes) using the behaviour policy and we try to improve the other policy called the target policy.

As mentioned, the input/vector data is used to learn the policy by the agent. And the action is a continuous value. we will be using behaviour and target networks of same configuration.

Naming convention of behaviour and target networks in the scripts:

- Behaviour network: “local_actor” and “local_critic” names are being used to refer actor and critic networks respectively
- Target network: “target_actor” and “target_critic” names are being used to refer actor and critic target network respectively.

Actor Network

Input:

The input tensor/vector consists of 33 values (corresponding to position, rotation, velocity and angular velocity of the arm). Hence, the input layer of the Actor Network (both behaviour and target network) consists of 33 neurons.

Output:

The output tensor/vector consists of 4 values (corresponding to the torque applicable to two joints). And every entry in the action vector should be a number between -1 and +1. Hence, the output layer of the Actor Network (both behaviour and target network) consists of 4 neurons, and tanh activation function would be used to limit the output values to -1 and +1.

Neural Network Architecture:

- 1 Input Layer with dimensions of (33, 128)
- 2 Hidden Layers with dimensions of (128, 128)
- 1 Batch normalization Layer with dimension of (128), and is placed after 1st hidden layer.
- 1 Output Tanh Layer with dimension of (128, 4)

Critic Network

Input:

The critic networks take both the current state and the action (predicted by the actor network) as inputs and concatenates them. However, the concatenation operation is performed after the batch normalization operation and before second hidden layer. Since the input tensor/vector of state consists of 33 entries, the input layer of the critic network has 33 neurons.

Output:

The Critic network estimates the Q value of the current state and the action(predicted by the actor network) and hence the output layer of critic network has only 1 neuron.

Neural Network Architecture:

- 1 Input layer with dimensions of (33, 128)
- 2 Hidden layers with dimensions of (128, 128)
- 1 Batch normalization layer with dimensions of (128), and is placed after 1st hidden layer.
- Concatenation operation is done before 2nd hidden layer. Concatenates the output of batch normalization layer and the action (predicted by the actor network) along the dim=1 (along the column)
- 1 output layer with no activation and of dimension (128, 1)

Modules to solve the Reacher environment (in the Continuous Control GitHub Repository):

- ***models.py:*** Source code that contains the DDPG (Actor-Critic) network.
- ***memory.py:*** Source code for Replay buffer to store the experiences
- ***ddpg_agent.py:*** DDPG Agent source code.
- ***train_agent.py:*** To train the DDPG agent.
- ***test_agent.py:*** To test the DDPG Agent (this script loads the trained DDPG weights from saved_models directory)
- ***noise.py:*** The base class of OUNoise process, object of this class (noise) will be added to the actor's output to facilitate the exploration.
- ***utils.py:*** This script has few functions to get the device (CPU/GPU) to train the model, plot the graphs etc.,
- ***ddpg_agentV2.py:*** This script has the Agents definition that trains 20 agents that are linked to the same brain.
- ***train_agentV2.py:*** This is the script that user can interact to train 20 agents linked to the same brain.

DDPG Agent

Parameters:

- ***state_size:*** Total number of dimensions of each state. (33, this would be the input shape of the tensor input to DDPG Actor and Critic networks).
- ***action_size:*** Total number of dimensions of each action (4, this is the output dimension of the DDPG Actor network).

DDPG Agent Hyperparameters:

- ***Buffer_size:*** Size of the buffer to store the agent's experience tuple (state, action, reward, next_state, done)
- ***Batch_size:*** Size of the memory batch used for model updates.
- ***Gamma:*** Parameter for setting the discount value for future rewards.
- ***Tau:*** Parameter for the soft update of the target network weights and other parameters.
- ***LR_actor:*** Learning rate of the actor network
- ***LR_critic:*** Learning rate of the critic network
- ***Weight decay:*** Weight decay of the critic network optimizer.
- ***Episodes:*** total number of episodes that the DDPG agent is trained
- Steps: Total number of steps (range of trajectory) in the episode.

Below are the values of the Hyperparameters used in for training the DDPG agent:

- **Buffer_size: 1e5**
- **Batch_size: 128**
- **gamma: 0.99**
- **Tau: 1e-3**
- **LR_actor: 2e-4**
- **LR_critic: 2e-4**
- **Weight_decay: 0**
- **Episodes: 1000**
- **Steps: 10000**

DDPG Agent methods

- ***Hard_copy_weights(self, target_model, local_model):*** This method is to make sure that both the local and target (actor and critic models) have the same initial parameter configuration (weight matrices and biases)
- ***Step(self, state, action, reward, next_state, done):*** This method takes the current (state, action), reward earned by the agent, next_state information and done (if agent has reached the terminal stage). This method basically stores the experiences of the agent. However, if there is enough memory (as defined by hyperparameter Batch_size), then the agent calls learn() method.
- ***Learn(self, experiences, gamma):*** The agent randomly samples experiences from the replay buffer. Using the hyperparameter gamma the values of target q_value, critic_loss

and actor_loss are computed. Critic loss is similar to DQN i.e., mean square error. Adam optimizer is used for gradient ascent.

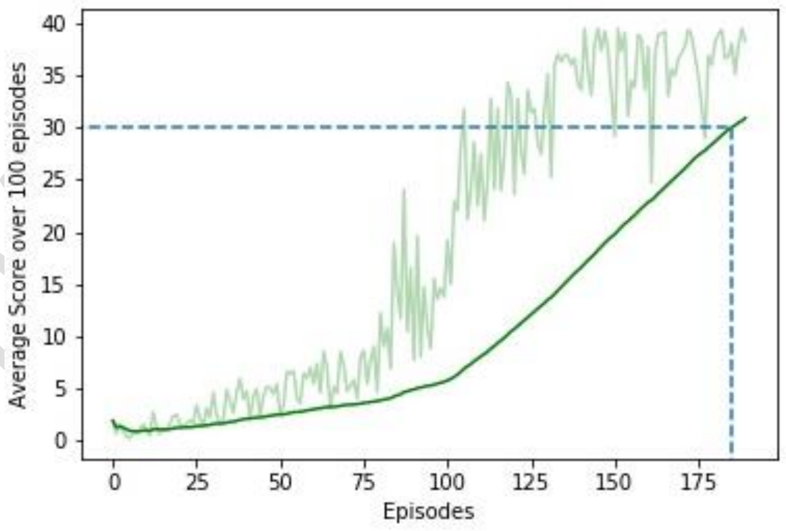
- ***Act(self, state, add_noise=True):*** This method is used by the agent to get possible (continuous values) from the actor_local network for the current state. However, noise is added to the action predicted by actor_local network, and this facilitates the agent to explore the environment.
- ***Soft_update(self, local_model, target_model, tau):*** This method is used by the agent to update the parameters (weight matrices, biases) of the target_model. The parameter "tau" is the interpolation parameter.

Target_model_parameters = τ*local_model_paramters + (1 - τ)*target_model_parameters

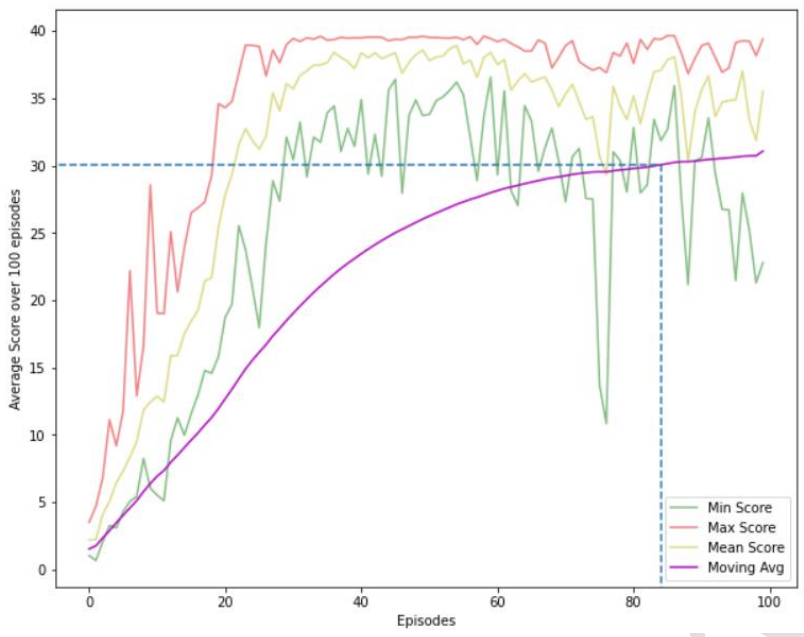
Training Phase

The DDPG agent with the above hyperparameters setting has been trained in 2 versions.

- ***Version1:*** Training single agent: the DDPG agent has been able to solve the Unity ML Reacher environment (i.e., to be able to score of +30.0 over 100 episodes) in less than 200 episodes. However, the least number of episodes required to solve the environment was **178** episodes.
- ***Version2:*** Training 20 agents: 20 DDPG agents have been trained, but are linked to single brain. So, essentially a single brain will be updated. This version of training took around **85** episodes to solve the environment (score of +30.0 over 100 episodes).



DDPG Agent training performance for solving the Reacher environment (+30.0 rewards for over 100 episodes). The graph tells us the average score per episode.



20 DDPG agents (linked to the same brain of Unity ML Agent) training performance for solving the Reacher environment (+30.0 rewards for over 100 episodes). The graph tells us the minimum, maximum, mean (average) and moving average (over 100 episodes) score per episode

Future Work

- Train 20 agents linking to 20 respective brains (using A3C or D4PG).
- D4PG tries to improve the accuracy of DDPG with the help of distributional approach. A SoftMax function is used to prioritize the experiences and provide them to the actor.

References

- https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b
- https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14
- https://julien-vitay.net/deeprl/DPG.html
- https://medium.com/intro-to-artificial-intelligence/deep-deterministic-policy-gradient-ddpg-an-off-policy-reinforcement-learning-algorithm-38ca8698131b
- DPGA David Silver
- Ravichandiran, Sudharsan. Deep Reinforcement Learning with Python: Master classic RL, deep RL, distributional RL, inverse RL, and more with OpenAI Gym and TensorFlow, 2nd Edition . Packt Publishing. Kindle Edition.