Multi-Agent DPDG RL Agent

Project Reacher

As part of Udacity Deep Reinforcement Learning Nanodegree

Sourav Babi Seal (sourav.seal@gmail.com)
June 2025

Summary: Train multiple agents using Deep Deterministic Policy Gradient Reinforcement Learning to train a double jointed arm to move to target locations. The environment provided by Unity is provided by Unity Machine Learning agents.

Environment

State Space

The state space has **33 continuous variables** that include the corresponding position, rotation, velocity and angular velocities of the arm.

Action Space

Each action is a vector with **four continuous numbers** corresponding to torque applied to the two joints. Every entry in the action vector should be a number between -1 and 1.

Rewards

- +0.1 for each step that the agent's hand is in the goal location
- To solve this environment the agent must get an average of +30 over 100 consecutive episodes.

Methods

Actor and Critic Model Architecture

Note that agents learn from vector data, not observed image data. Hence we employ two fully connected layers for function estimation.

Employs Deep Neural Network to approximate the Actor Policy function and the Critic Q-value function. optimal action-value function and map the input dimension of 33 variables to an output dimension corresponding to each of the 4 actions in a single forward pass.

It has 3 fully connected (FC) layers

- FC 1 dimensions: input state size 33, and the output 256
- FC 2 dimensions: input state size 256, and the output 128
- FC 3 dimensions: input state size 128, and the output 4

FC 1 and FC2 have ReLU activation functions. Since the outputs are continuous values a tanh function is used to keep the values between -1 and 1.

Training algorithm for DDPG

Since we have continuous valued action spaces, we cannot straightforwardly apply DQN since that relies on finding the discrete action that maximizes the action-value function. Discretization also is not a good option since it needlessly throws away information about the structure of the action domain.

Instead we rely on a model-free, off-policy actor-critic algorithm using deep function approximators that learn policies in high-dimensional, continuous action spaces - Deep DPG (Deterministic Policy Gradient) as outlined in Continuous Control with Deep Reinforcement Learning by Timothy P. Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra accepted at ICLR 2016.

The algorithm for a single agent as documented in the paper :

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:

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

  end for
end for

We leverage the **`dpdg_agent.py`** implementation provided in Udacity exercise implementation in ddp-pendulum (github).

We modified the above implementation and incorporated support for MultiAgent implementations of DDPG thus:

1. **Agent Class Instantiation:**
    - Instead of `agent = Agent(state_size, action_size, random_seed)`, we created one instance of the Agent class that will manage the learning for all parallel environment agents.
      `agent = Agent(state_size, action_size, random_seed, num_agents)`
    - The `num_agents` parameter is added to Agent.
2. **Shared Replay Buffer:**
    - The `ReplayBuffer` is still a single instance (self.memory within the Agent). All parallel agents contribute their experiences to this *one* shared buffer, and the learning process samples from it. This allows the agent to learn from a more diverse set of experiences.

3. **Noise Application:** The `self.noise` is now a list of `OUNoise` objects, one for each parallel agent. Noise is applied to each agent's action independently in the act method's loop.The `Agent` now initializes `num_agents OUNoise` instances, each with a unique seed (e.g., random_seed + i). This ensures independent noise processes for each parallel agent. Note that when applying sampling we use np.random.randn for standard normal distribution as typically done in OUNoise

4. **Update to `step` Method:**

    - **Loop for memory.add:** It iterates through these parallel experiences (e.g., `for i in range(self.num_agents):`) and calls `self.memory.add()` for each *individual* agent's experience. This ensures the buffer stores individual (s,a,r,s',d) tuples, not batches of tuples.

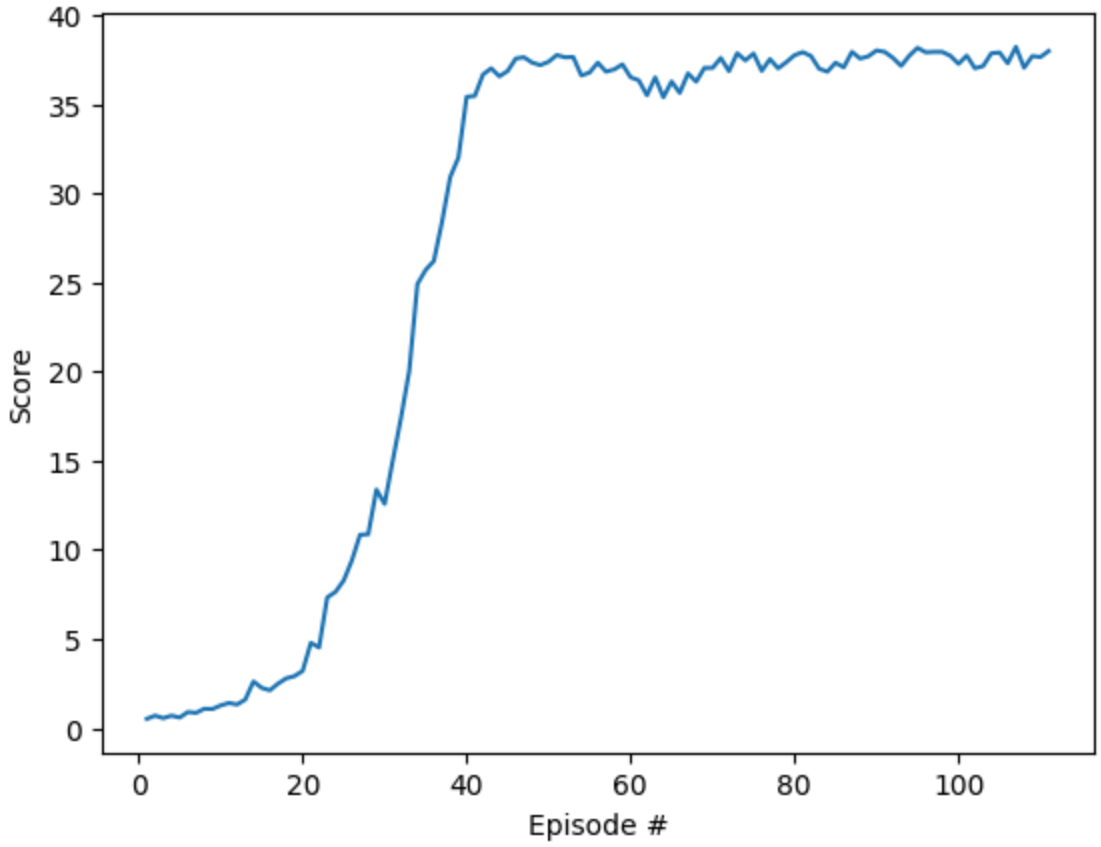
Hyperparameters

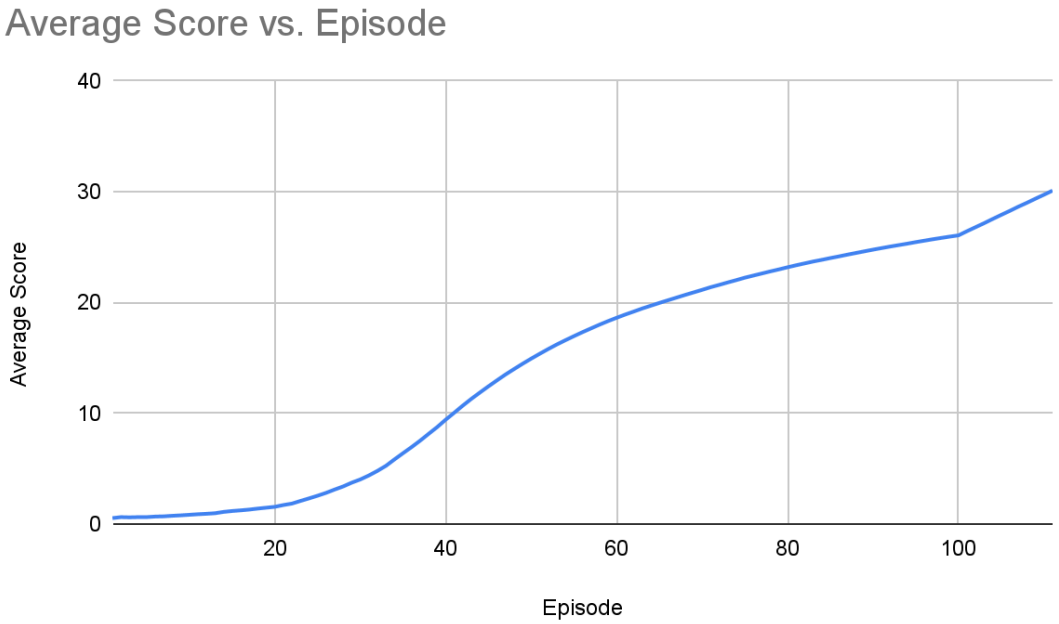
Hyperparameter	Value
Batch Size	128
Discount Factor (GAMMA)	0.99
Learning Rate Actor (LR_ACTOR)	1e-4
Learning Rate Critic (LR_CRITIC)	1e-4
Weight Decay	0

Training Parameters	Value
Episodes	700
Max time step of an episode	1000

Results

We solved the environment in 111 episodes, sustained an average of 30+ score over the past 100 episodes.


Score
Episode #
0
5
10
15
20
25
30
35
40
0
20
40
60
80
100


Average Score vs. Episode
Average Score
Episode
0
10
20
30
40
20
40
60
80
100

Future Directions

There were multiple optimizations and hyperparameters that were recommended in the Udacity course. We would like to do a hyper-parameter sweep to see if we can get better performance in training than currently achieved.

We would like to compare the performance of **PPO**, **A3C** and **D4PG** that uses multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

References

Continuous Control with Deep Reinforcement Learning by Timothy P. Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra accepted at ICLR 2016.