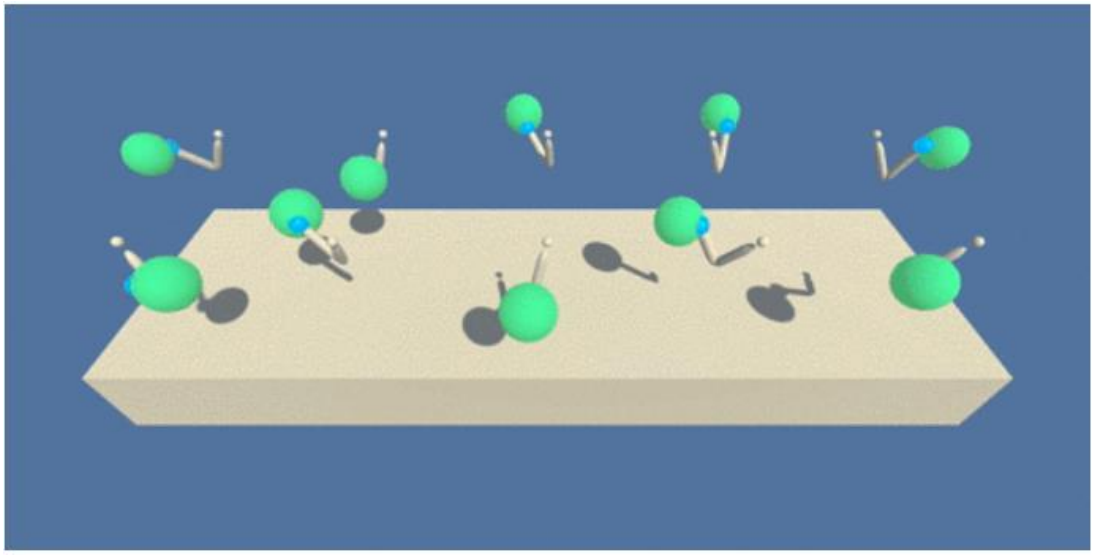
Udacity
Deep Reinforcement Learning
Nanodegree Program

Project: Continuous Control

Developer: Koh Thiam Chye

1. Background



Unity ML-Agents Reacher Environment

For this project, we used the Unity ML-Agents Reacher simulated environment to simultaneously train 20 agents with double-jointed arm to move to target locations. A reward of +0.1 is provided for each step that the agents' hand are in the goal location. The state 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 environment is considered solved if the 20 agents can receive an average reward (over 100 episodes) of at least +30.

2. Training Environment Setup

a) Operating Environment: Ubuntu 18.04 LTS

b) Imported Libraries:

i. UnityAgents (ver. 0.4.0)
ii. Numpy
iii. Random
iv. Sys
v. Torch
vi. Matplotlib
vii. Collections

3. Implementation

a) Deep Deterministic Policy Gradients (DDPG)

Hyperparameters:

- n_episodes (int): 1000 # max no. of episode
- max_t (int): 1,000,000 # no. of time-step in each episode
- BUFFER_SIZE = int(1e6) # max size for replay buffer
- BATCH_SIZE = 128 # size of experiences to be sampled
- GAMMA = 0.99 # discount factor
- TAU = 1e-3 # for soft update of target parameters
- LR_ACTOR = 3e-4 # Actor network learning rate
- LR_CRITIC = 3e-4 # Critic network learning rate

Description of Model Architecture:

20 agents, each with its own Actor and Critic network models (with local and target instances) were created with random seed values (between 1 and 99).

For a maximum of 1000 episodes, at each time-step, the 20 agents interacted with the environment to get an independent action. For this, the state values were channeled into the local instance of Actor model to compute corresponding action vector, which was clipped to range [-1,1] and augmented with random noise.

The agents then took a step in the environment with the actions generated, which in turn returned the next-state, reward and done status. The agents update a shared memory replay buffer with the information. Using a delayed intensive training approach for greater model stability, the agents are trained at an interval of 20 time-steps for 10 rounds by sampling past experiences (state, action, reward, next_state, done).

To implement reinforcement learning using the Actor-Critic model, the next action was derived by feeding next-state to the target instance of the actor model. The Q-values for next-

state is then generated from the target instance of the Critic model by feeding it with both next-state and next action. Following which, the Q-values for next-state is then used, together with other parameters including reward, gamma discount factor and done status, to compute the targeted Q-values for current state. In addition, the expected Q-values for current state is separately computed using the local instance of the Critic model using current state and action.

Next, the mean-square-loss between the targeted Q-values and expected Q-values are used in the backpropagation of the local instance of the Critic model to adjust its weights. Gradient clipping was applied when training the Critic model.

The loss of the Actor model is computed by negating the average of output from the local instance of the Critic model using current state and action predicted by local instance of the actor model.

The weights for the local instances and target instances are then combined based on an interpolation parameter (i.e. tau) and copied to the target instances of the Actor and Critic models for the subsequent rounds of reinforcement learning.

When the environment for any of the 20 agents is done, the episode ends, and the cumulative scores are averaged for the 20 agents to check whether target reward (in this case +30) has been achieved. The training cycle continues with the next episode until the target reward is achieved or maximum episode is reached.

Please refer to Continuous_Control.ipynb and ddpg_agent.py for more details.

Neural Network Model:

The model consists of two fully connected Actor and Critic neural networks. The state space of 33 values was embedded onto two input and hidden layers with 256 nodes each. The output layer consists of 4 nodes for the action vector.

ReLu activation and batch normalization were implemented for the input and hidden layers of the Actor network, while the output layer was activated using tanh activation.

For the Critic network, reLu activation was implemented for the input and hidden layers while batch normalization was only applied to the input layer.

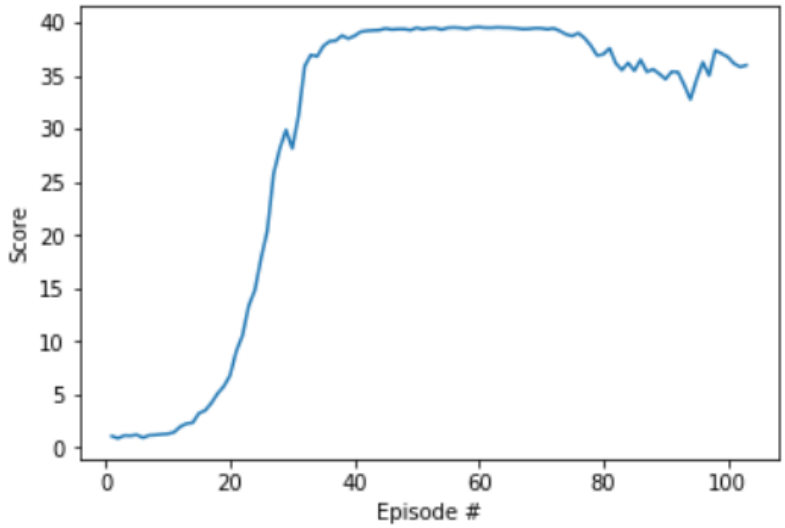
The Actor and Critic networks were also initialized with randomized weights, limited to the inverse square-root of the number of nodes in the respective layers.

Please refer to model.py for more details.

Training Outcome:

For a target score of at least +30, the environment was solved using the above mentioned DDPG model in 103 episodes.

```
Episode 100     Average Score: 28.98
Episode 103     Average Score: 30.03
Environment solved in 3 episodes!       Average Score: 30.03
```



4. Ideas for Future Work:

The project could be re-implemented using the following methods,

- Distributed Distributional Deterministic Policy Gradients (D4PG)

  Involves modifications of the off-policy actor-critic DDPG to include a distributional critic update, the use of distributed parallel actors, N-step returns for estimating TD errors, and prioritization of the experience replay.

- Asynchronous Advantage Actor-Critic (A3C)

  Overcome the short comings of using experience replay which are memory and computationally intensive per interaction by asynchronously executing multiple agents in parallel across multiple instances of the environment, such that at different time-steps, the agents will be experiencing a variety of different states.

- Proximal Policy Optimization (PPO)

  Focus on alternating between sampling data through interaction with the environment and optimizing a “surrogate” objective function using stochastic gradient ascent. The key difference is while standard policy gradient methods perform one gradient update per data sample, PPO seeks to enable multiple epochs of minibatch updates to achieve stability in training.

Other methods include Trust Region Policy Optimization (TRPO) and Truncated Natural Policy Gradient (TNPG).