Deep Reinforcement Learning – Continuous Control

AIM

The environment is considered solved, when the average (over 100 episodes) of those average scores is at least +30.

The environment provided is that of a Unity ML Agents scenario.

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your 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 should be a number between -1 and 1.

Two versions of the environment are provided here.

- The first version contains a single agent.
- The second version contains 20 identical agents, each with its own copy of the environment.

IMPLEMENTATION

Algorithm

DDPG(Deep Deterministic Policy Gradient) Algorithm.

The base code of the project is derived from the solution provided in Udacity Deep Learning Nanodegree Github repo for solving the pendulum scenario from OpenAI Gym.
https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum

The same solution was modified and updated for the Unity ML Agents environment provided.

The notebooks

- **Continuous_Control_Single.ipynb** : Contains the implementation for training and visualising the untrained agent initially for the Single Agent.
- **Continuous_Control_Multi.ipynb :** Contains the implementation for training and visualising the untrained agent initially for 20 Agents respectively. Then the training code is implemented.
- **ddpg_agent.py :** Contains the code to understand and determine how the agent interacts with the environment and learns to optimize the reward.
- **model.py :** Contains the architecture of the deep learning model used in this implementation.

Algorithm

Deep Deterministic Policy Gradient (DDPG)

The algorithm is outlined in this paper, Continuous Control with Deep Reinforcement Learning. In this paper, the authors present "a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces." They highlight that DDPG can be viewed as an extension of Deep Q-learning to continuous tasks.

Actor-Critic Method

Actor-critic methods leverage the strengths of both policy-based and value-based methods.

Using a policy-based approach, the agent (actor) learns how to act by directly estimating the optimal policy and maximizing reward through gradient ascent. Meanwhile, employing a value-

based approach, the agent (critic) learns how to estimate the value (i.e., the future cumulative reward) of different state-action pairs. Actor-critic methods combine these two approaches in order to accelerate the learning process. Actor-critic agents are also more stable than value-based agents, while requiring fewer training samples than policy-based agents.

It also utilizes the replay buffer to learn from past experiences in the way DQN Algorithm does.

Model Architecture

There are two model architectures used here defined in the file model.py :

Actor

- Hidden layers: 2
- 1st hidden layer nodes (FC): 400
- 2nd hidden layer nodes (FC): 300
- Input parameters (states): 33
- Activation function: ReLU
- Applied Batch Normalisation

Critic

- Hidden layers: 2
- 1st hidden layer nodes (FC): 400
- 2nd hidden layer nodes (FC): 300+action_size
- Output layer nodes (Q-value): 1 (outputs the expected return value)
- Input parameters (states): 33
- Activation function: ReLU
- Applied Batch normalisation
- torch.cat operation : denote the mapping between the states and actions as defined in the algorithm

Agent

Deep Q-Learning agent which interacts with the environment. It references the local and target network from the model defined in model.py.

Step: Experiences in the replay memory. After a certain set of predefined intervals, it also causes the network to learn from the replay buffer a certain number of times.

Act: Returns the action determined by the local Actor Network. The output is of size 4 corresponding to each actions but in the range of -1 to 1 as expected by the network. Furthermore, noise is added via the Ornstein-Uhlenbeck process to encourage exploration.

Learn: Learns

Critic network learning : We randomly sample a batch from the experience buffer in the form of (states, actions, rewards, next_states, dones) and pass on the next_states to the actor target network to determine the next set of actions which in return is passed on to the critic target network.

The return from the critic target network is actually the Q values determined by the target network.

The expected Q values are computed from these next states and compute the Mean Squared Error loss between Q_targets and Q_expected and update the critic target network accordingly.

ACTOR NETWORK LEARNING

Here we get the predicted actions from the local actor network based on the current states. The loss is computed as the mean of the Q values corresponding to the different state action pairs. We use the negative sign here because we want to maximise the gradients and hence using gradient ascent.

Then we run the soft_update function to update the target network with the local network parameters

Soft_update: Here we update the target networks with the local networks parameters using the formula

$$\theta\_target = \tau*\theta\_local + (1 - \tau)*\theta\_target$$

Hyperparameters

The hyperparameters used to train the agent are:

- BUFFER_SIZE = int(1e6) # replay buffer size
- BATCH_SIZE = 128 # minibatch size
- GAMMA = 0.99 # discount factor
- TAU = 1e-3 # for soft update of target parameters
- LR_ACTOR = 2e-4 # learning rate of the actor
- LR_CRITIC = 2e-4 # learning rate of the critic
- WEIGHT_DECAY = 0 # L2 weight decay
- UPDATE_EVERY=20 # Update interval
- NUM_UPDATES=10 # Number of learning steps after every predefined interval in UPDATE_EVERY

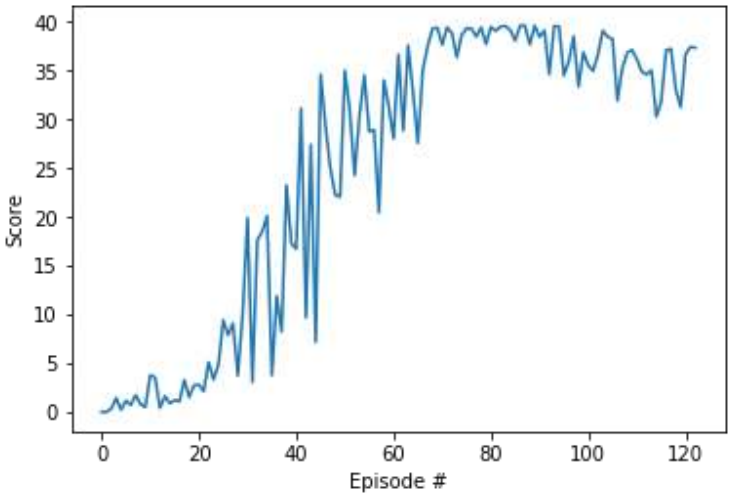
Plot of Rewards

The plot of average rewards for every 100 episodes for **Single Agent** are listed below.

Episode 100 Average Score: 22.44 Score: 36.90

Episode 123 Average Score: 30.24 Score: 37.36

Environment solved in 123 episodes! Average Score: 30.24



Ideas for Future Work

The Reinforcement Learning agent was trained using Deep Deterministic Policy Gradients

- Reducing learning rates LR_ACTOR and LR_CRITIC
- Increasing number of neurons in Actor and Critic networks
- Other algorithms like TRPO, PPO, A3C might have worked in this case as well and might give different or even better performance.
- The 20 agents scenario reached the target score much before 100. Might need to run the process for longer periods of time to monitor how much the stability and score is maintained over a longer period of time