DeepRL-PRJ-Continuous Control

This project repository contains my work for the Udacity's Deep Reinforcement Learning Nanodegree Project 2: Continuous Control.

1. Project Details

Project Description

For this project, we will train an agent (robot with double-jointed arm) to move (it's arm) to target locations. The goal of the agent is to maintain its position at the target location for as many time steps as possible.

Reward

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

Observation (State) Space

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm.

Action

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 action space is *continuous*, which allows each agent to execute more complex and precise movements. There's an unlimited range of possible action values to control the robotic arm, whereas the agent with discrete action space was limited to four *discrete* actions: left, right, forward, backward.

Project Goal

The goal of the agent is to maintain its position at the target location for as many time steps as possible. The task is episodic, and in order to solve the environment, your agent must get an average score of +30 over 100 consecutive episodes.

2. The Environment

The details are taken from the Udacity's Deep Reinforcement Learning Nanodegree program. The environment is based on Unity ML-agents. Please read more about ML-Agents by perusing the GitHub repository.

The project environment provided by Udacity is similar to the Reacher environment on the Unity ML-Agents GitHub page.

The environment contains one brain. I have chosen to solve the first version of the environment with the single agent option using the off-policy DDPG algorithm. The task is episodic, and in order to solve the environment, the agent must get an average score of +30 over 100 consecutive episodes.

3. Agent Implementation

This continuous control project implements an off-policy method called DDPG, Deep Deterministic Gradient Policy, which is a different kind of actor-critic method as described in the paper Continuous control with deep reinforcement learning.

We present an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces. Using the same learning algorithm, network architecture and hyper-parameters, our algorithm robustly solves more than 20 simulated physics tasks, including classic problems such as cartpole swing-up, dexterous manipulation, legged locomotion and car driving. Our algorithm is able to find policies whose performance is competitive with those found by a planning algorithm with full access to the dynamics of the domain and its derivatives. We further demonstrate that for many of the tasks the algorithm can learn policies “end-to-end”: directly from raw pixel inputs.

Actor-critic method
The policy-based methods tend to have high variance and low bias and use Monte-Carlo estimate whereas the value-based methods have low variance but high bias as they use TD estimates. The agent can learn the policy directly from the states using Policy-based methods. Now, the actor-critic methods were introduced to solve the bias-variance problem by combining the two methods. In Actor-Critic, the actor is a neural network which updates the policy and the critic is another neural network which evaluates the policy being learned which is, in turn, used to train the actor. In vanilla policy gradients, the rewards accumulated over the episode is used to compute the average reward and then, calculate the gradient to perform

gradient ascent. Now, instead of the reward given by the environment, the actor uses the value provided by the critic to make the new policy update.

Deep Deterministic Policy Gradient (DDPG)
Below details and the algorithm screenshot are taken from the DDPG algorithm from the Spinning Up website

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function and uses the Q-function to learn the policy.

DDPG is an off-policy algorithm. DDPG can only be used for environments with continuous action spaces. DDPG can be thought of as being deep Q-learning for continuous action spaces.

Deep Deterministic Policy Gradient (DDPG) leverages the class of Actor-Critic Methods. The actor produces a deterministic policy instead of the usual stochastic policy and the critic evaluates the deterministic policy. The critic is updated using the TD-error and the actor is trained using the deterministic policy gradient algorithm.

Pseudocode

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

4. Code Implementation

High Level Approach:

Here are the high-level steps followed in building the agent that solves this environment.

1. Evaluate the state and action space.
2. Establish performance baseline using a random action policy.

3. Select an appropriate algorithm and begin implementing it.
4. Run experiments, make revisions, and retrain the agent until the performance threshold is reached.

Github link: https://github.com/kishoregajjala/DeepRL-PRJ-ContinuousControl

Readme.md provides more details of the project and environment.

The implementation of this project consists these files:

1. **model.py**: This implements the **Actor** and the **Critic** classes which each implements a Target and a Local Neural Networks used for the training.

2. **ddpg_agent.py**: This implements the DDPG agent, OUNoise and replay buffer memory which is used by the agent.
   - The Actor's Local and Target neural networks, and the Critic's Local and Target neural networks are instantiated by the Agent's constructor

   - **Network Architecture** The network architecture is comprised of two fully connected hidden layers of 128 units each with ReLU activations. To help speed up learning and avoid getting stuck in local minimum, batch normalization after activation was introduced to each hidden layer. The hyperbolic tan activation was used on the output layer for the actor network as it ensures that every entry in the action vector is a number between -1 and 1.
     Adam was used as an optimizer for both actor and critic networks with a learning rate of 2e-4 and batch size of 128.

   - The **step()** method saves experience in replay memory, and use random sample from buffer to learn

   - The **act()** method returns actions for given state as per current policy

   - The **learn()** method updates the policy and value parameters using given batch of experience tuples.

   - The **soft_update()** method:
     In DDPG, target networks are updated using a soft update strategy. In DDPG, you have two copies of your network weights for each network - regular for the actor, regular for the critic, and a target for the actor, and a target for the critic. A soft update strategy consists of slowly blending in your regular network weights with your target network weights. So, every timestep you make your target network be

99.99 percent of your target network weights and only 0.01 percent of your regular network weights. You are slowly mixing your regular network weights into your target network weights. The regular network is the most up to date network because it is the one, we are training while the target network is the one we use for prediction to stabilize strain. This way you will get faster convergence by using the update strategy.

- **Implements Ornstein-Uhlenbeck (OU) Noise** class to add noise to the action. For discrete action spaces, exploration is done via probabilistically selecting a random action such as epsilon-greedy or Boltzmann exploration. For continuous action spaces, exploration is done via adding noise to the action itself.

- **Implements ReplayBuffer class**:
  In reinforcement learning, the agent interacts with the environment and learns from sequence of state(S), actions(A) and rewards(R) and next state(S') which is stored in the form of an experienced tuple. These sequence of experience tuples can be highly correlated which can lead to oscillation or divergence in the action values. This can be prevented by having a replay buffer, from which experience replay tuples are used to randomly sample from the buffer. The replay buffer contains a collection of experience tuples (S, A, R, S´). The tuples are gradually added to the buffer as we are interacting with the environment. The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

3. **Continous_Control.ipynb**

   This Jupyter notebooks allows to instantiate and train the agent. More in details it allows to:

   - Prepare the Unity environment and Import the necessary packages
   - Check the Unity environment
   - Define a helper function to instantiate and train a DDPG agent
   - Train and evaluate an agent using DDPG and various Hyperparameters
   - Plot the score results

4. **DDPG Parameters**

Few notes:

- Increasing the number of steps per episode helped the agent to start learning, but it impacts the training time
- Reducing the sigma values used in Ornstein-Uhlenbeck (OU) Noise class also helped the agent in learning
- Adding the batch normalization layer after activation layer helped to improve the model.
- Adjusted the learning rate to improve the learning

- Hyperparameters and their values:

```
BUFFER_SIZE = int(1e5)  # 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
mu=0.                   # Ornstein-Uhlenbeck noise parameter
theta=0.15              # Ornstein-Uhlenbeck noise parameter
sigma=0.1               # Ornstein-Uhlenbeck noise parameter
```

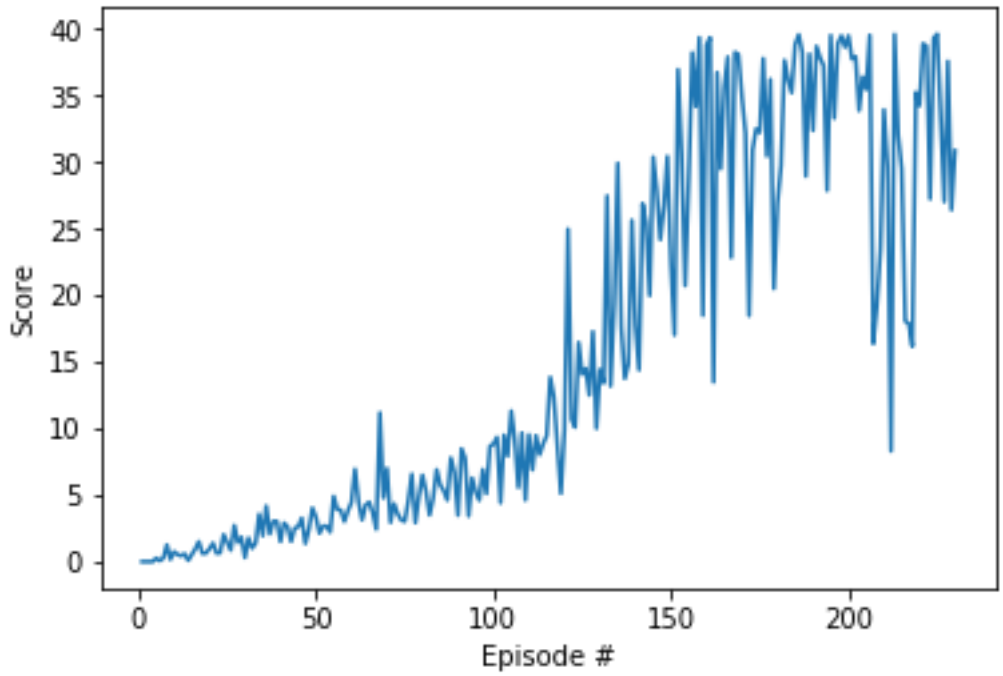
5. Results and plot of rewards

The problem was solved using DDPG algorithm where the average reward of +30 over at least 100 episodes was achieved in 230 episodes.

The result depends significantly on the fine tuning of the hyperparameters. For example, if the number of time steps are too low, learning rate or the seed is too large or small, the system can fall into a local minima where the score may start to decrease after a certain number of episodes. Batch normalization layer was added to the network architecture to help improve training.

The plot of the rewards across episodes is shown below:

```
Episode 100    Average Score: 3.21
Episode 200    Average Score: 24.14
Episode 230    Average Score: 30.09
Environment solved in 230 episodes!  Average Score: 30.09
```



6. Future Improvements

1. I have only solved the single agent problem. Future work should including solving the multi agent continuous control problem with DDPG. We will need to amend the DDPG code to work for multiple agents, to solve version 2 of the environment:

   To make the code work with 20 agents, we will need to modify the code so that after each step:
   - each agent adds its experience to a replay buffer that is shared by all agents, and
   - the (local) actor and critic networks are updated 20 times in a row (one for each agent), using 20 different samples from the replay buffer.
   - 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 these 20 scores.
   - This yields an average score for each episode (where the average is over all 20 agents). The environment is considered solved, when the average (over 100 episodes) of those average scores is at least +30.

2. As discussed in this paper, Trust Region Policy Optimization (TRPO), Truncated Natural Policy Gradient (TNPG) and implementation of Proximal Policy Optimization (PPO) should achieve better performance and yield improved stability

3. I also like to explore the recent Distributed Distributional Deterministic Policy Gradients (D4PG) algorithm as another method for adapting DDPG for continuous control.

4. We used replay buffer in DDPG. I would like to try implementing it with prioritized replay which may yield better results.