Project Continuous Control Report

Environment Background

The environment used for this project is a version of the Unity ML Agents Reacher environment. The agent in this environment is a double jointed robotic arm that is tasked with moving itself into a specific position as if it were reaching for that position. The agent receives a reward of +0.1 for each timestep that the arm is in the target position. The goal of the agent is to maintain itself in this position for as many timesteps as possible.

The state space is the position, rotation, velocity, and angular velocities of the arm, totaling 33 variables. The action space is the torque vectors for the two joints, totaling an action size of 4, on the arm each variable with a range from -1 to 1. There are two versions of this environment. In the first version, there is one robotic arm that solves the environment when it averages a score of 30 or greater over 100 episodes. In the second version, there are 20 different agents each with a copy of the environment. This environment involves distributed training and works best for specific algorithms. The version used for this project was the single agent version that solves the environment when the average score exceeds 30 over a 100 episodes.

DDPG Learning Algorithm

Theory

The DDPG algorithm stands for the Deep Deterministic Policy Gradient Algorithm. It was first presented in a paper that applied Deep Q-learning to a continuous action space. In the original paper, the authors used DDPG to solve different physics tasks. The algorithm as presented in the original paper is an actor-critic model free algorithm and is recreated as it was presented in the original paper in the figure below.

Figure 1: DDPG Algorithm

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

Implementation

The DDPG Algorithm solution presented in this project involves defining an Actor, a Critic, a DDPG agent, a noise process, a replay buffer, and the algorithm used to train the agent. Each components implementation is described below.

Actor: The Actor is a neural network that consists of three layers. The first layer has inputs of the environment's state size and outputs of size 330. The second layer has inputs of size 330 and outputs of size 110. The third layer has inputs of size 110 and outputs of the agents action size. The Actor has three methods: __init__, reset_parameters, and forward.

- The __init__ method creates the neural network with the architecture described above.
- The reset_parameters method resets the weight values for the neural network using the hidden_init function, which takes a layer as input and returns a range of values to fill that layers weights with.
- The forward method takes the state as an input. It propagates the input state through the network with Rectified Linear Units(ReLU) functions between each layer. The hyperbolic tangent function is applied to the output of layer 3. The output of the tangent function is returned from the method.

Critic: The Critic is a neural network that consists of three layers. The first layer has inputs of the environment's state size and outputs of size 330. The second layer has inputs equal to the sum of the agent's action size and 330. This layer has and output of size 110. The third layer has inputs of size 110 and output of size 1. The Critic has three methods: __init__, reset_parameters, and forward.

- The __init__ method creates the Critic neural network with the architecture described above.
- The reset_parameters method resets the weight values for the neural network using the hidden_init() function.
- The forward method takes a state and action as an input. The state is propagated through the first layer. The output of the firs t layer and the action input are concatenated using torch.cat into the input of the second layer. The output of the third layer is then returned from the method.

DDPG Agent: The DDPG Agent is created by the DDPGAgent class. The class is defined with the following methods: init, step, act, reset, learn, and soft update.

- The __init__ method takes the action_size, state_size, and random_seed as parameters. It creates the agent's actor and critic networks, the noise process used for exploration, and the memory buffer. Each agent has two actor networks, a local and a target, and two critic networks, a local and a target. To allow for updating network weights at intervals, it creates a timestep counter, t_step
- The step method takes an experience tuple(state, action, reward, next_state, done) components as individual parameters. The memory buffer is updated with that tuple. If the timestep count is a multiple of the UPDATE_EVERY hyperparameter, the agent samples a batch from the buffer and learns from that batch.
- The act method takes the state of the environment and a boolean addNoise value. The agent uses the state to decide on and return an action to take in the environment. If the

addNoise boolean is True, then noise is added to the decided action value to encourage exploration of the environment.

- The reset method is used to reset the noise process.
- The learn method takes an experience batch and a gamma hyperparameter as parameters. The agent uses the experience batch to update the Critic and Actor networks.
- Soft update takes a local network, a target network, and tau as parameters. The method updates the target network.

The Agent has the following hyperparameters

- BUFFER_SIZE : the size of the memory buffer
- BATCH_SIZE: the size each batch sampled from the memory buffer
- GAMMA: the gamma value for reward discounting
- TAU: the interpolation factor used to update the target network
- LR_ACTOR: the learning rate for the actor local and target network
- LR_CRITIC: the learning rate for the critic local and target network
- WEIGHT_DECAY: the L2 weight decay used by the critic network
- UPDATE_EVERY: the number of timesteps after which to update the actor and critic networks.

OUNoise: The OUNoise is an Ornstein-Uhlenbeck noise process. This noise is used to encourage exploring early in the learning stages for the agent. The class is defined with the following methods: init, reset, and sample.

- The init method initializes the noise process by using parameters of size and random seed along with values for theta, mu, and sigma. Theta, mu, and sigma are components of the noise process and can be manipulated to alter the magnitude of the resulting noise.
- The reset method resets the noise process
- The sample method calculates and returns a new noise value that is later added to the action in the DDPG act method.

Replay Buffer: The ReplayBuffer class defines a memory buffer that stores experience tuples. The buffer is defined with the following methods: __init__, add, sample, and __len__.

- The __init__ method takes parameters of action_size, buffer_size, batch_size, and random seed value. The method initializes a ReplayBuffer of size, BUFFER_SIZE, with a namedtuple with field names of state, action, reward, next_states, and dones for the experience tuples. The random seed value is used to define a random seed.
- The add method takes in a state, action, reward, next_state, and done as input parameters. It adds this tuple as an experience tuple to the memory buffer
- The sample method extracts a set of experience tuples with a size of batch_size. The method separates them by states, actions, rewards, next_states, and dones and returns each set differently.
- The __len__ method returns the length of the memory buffer.

DDPG Algorithm: The algorithm is defined in the function ddpg(). The function takes input parameters of n_episodes, max_t, and print_every.

- n_episodes is the number of episodes
- max_t is the maximum number of timesteps per episode
- print_every is the number at which to print the average scores. This is used to observe and track the progress of the agent as it learns.

The function proceeds to iterate through the episode numbers as the agent acts and receives a reward value after each timestep. If the episode number is below 300, the agent's act method is called with addNoise=True. If the episode number is above 300, the agent's act method is called with addNoise=False. This structure was used to encourage exploration for the first 300 episodes after which the agent was found to learn a potentially successful policy. Exploration is essential to enable the agent to identify and learn a policy that enables the agent to solve the environment.

The rewards at each timestep are summed to a score per episode. The scores are stored in the variable scoreList, a list of length print_every. If the episode number is a multiple of print_every, then the average score per episode which is the average of the scores in scoreList. If the score is equal to or exceeds +30, the network weights are saved using PyTorch's inbuilt save function.

Hyperparameters of a Successful Solution

After testing and experimenting with different hyperparameter values, the following values produced a successful solution. The random seed value played a critical role as this environment was found to be heavily dependent on the random seed value.

DDPG Agent and Replay Buffer Hyperparameters

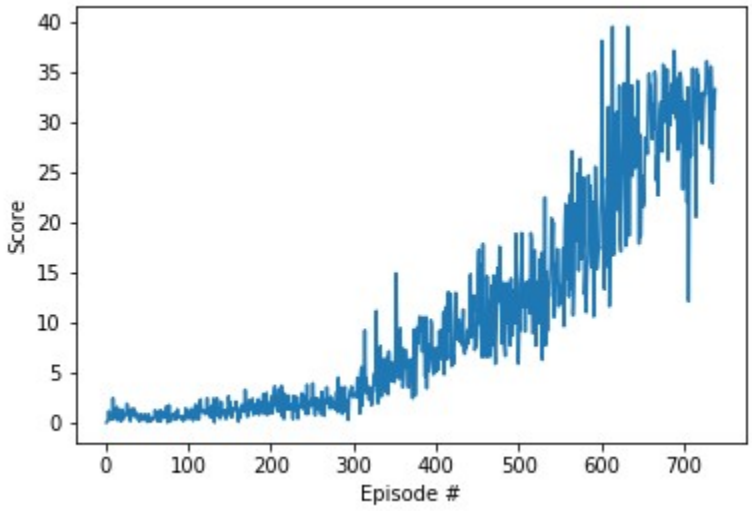
BUFFER_SIZE = 1e5
BATCH_SIZE = 512
GAMMA = 0.99
TAU = 1e-3
LR_ACTOR = 1e-3
LR_CRITIC = 1e-3
WEIGHT_DECAY = 0
UPDATE_EVERY = 20
random_seed = 2

ddpg Function Parameters

n_episodes = 1500
max_t = 1200

The agent solves the environment with an average score of 30.02 in 738 episodes. This means the average score of 30.02 was achieved from episodes 638 to 738, fitting the requirement for achieving a greater than or equal to 30 average score over 100 episodes A plot of the score per episode is included in Figure 2.

Figure 2



Future Directions

The future improvements that can be made this solution are to use distributed training and an algorithm such as D4PG on version 2 of the environment with 22 agents. This version and algorithm would learn faster by taking advantage of the information from multiple agents.

References

DDPG Paper: https://arxiv.org/pdf/1509.02971.pdf