DDPG for Unity Reacher

Environment

1. Introduction

Reinforcement learning is rooted in psychological and neurological perspective of animal behavior. In many cases the environment can be too complex that the number of states is not enumerable to a finite set. Animals and humans learn the complex environment around them through a harmonious combination of reinforcement learning and hierarchical sensory processing systems. DQN was one of the ground-breaking use of Deep neural networks in reinforcement learning to learn Atari games. The Deep Q-Network agent surpassed all previous algorithms and achieved levels comparable to professional human players. But DQN was not suitable for continuous action spaces. This is because most of the interesting problems in robotic control, fall in this category.

This implementation to solve the Unit Reacher Environment, is based on the DDPG algorithm. The software agent (DDPG) interacts with the environment thru a sequence of observations (33) actions (4) and rewards. The goal of the agent is to select an action that maximizes the cumulative future reward. This implementation uses an MLP (Multi-layer perceptron) network to approximate the optimal policy function fig 1.

2. Policy Gradients & Actor-Critic methods

While DQN solves problems with high-dimensional observation spaces, with low-dimensional action spaces. Many tasks of interest, have continuous (real valued) and high dimensional action spaces. It is not straightforward to apply DQN to continuous domains, since it relies on a finding the action that maximizes the action-value function, which in the continuous valued case requires an iterative optimization process at every step.

An obvious approach to adapting deep reinforcement learning methods such as DQN to continuous domains is to discretize the action space. However, this has many limitations, most notably the curse of dimensionality: the number of actions increases exponentially with the number of degrees of freedom.

2.1 Policy Gradient methods

Policy Gradient methods learns a parametrized policy that can select actions without consulting a value function. A value function may be used to learn the policy parameter, but not required for action selection. Let $\theta \in R^{d'}$ denote the policy's parameter vector then

$$\pi(a|s,\theta) = P_r\{A_t = a \,|S_t = s, \theta_t = \theta\} \text{ --------> (1)}$$

$\pi(a|s,\theta)$, denotes the probability that action a is taken at time t when in states. Policy gradients uses optimization based on the gradients of scalar performance measure $J(\theta)$, with respect to the policy parameter. They use gradient ascend in J as in Eqn. 1.

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \text{ ----------> (2)}$$

Ideally a lot of good training examples of high rewards for good actions and low rewards for bad one's will help the learning process. Just as credit assignment problem is hard in discrete action spaces, it becomes even more difficult in continuous spaces.

2.2 Actor-Critic Methods:

Methods that learn approximations to both policy and value function are often called actor-critic methods where 'actor' is a reference to the learned policy and 'critic' refers to the learned value function, usually a state-value function fig. 1 (2) . The actor produces an action based on the state of the environment and critic produces a TD error signal based on state and the reward. If the action value function is estimated by the critic, it would need the output of the actor too. Both actor and critic can be represented by Neural Networks.

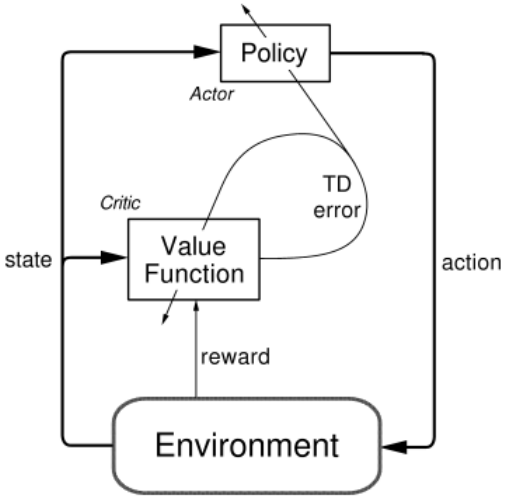

Fig. 1

2.3 On-Policy Vs Off-Policy

Reinforcement learning uses both on-policy and off-policy algorithms for learning. On-policy algorithms use the policy that is being estimated to sample trajectories during training, while off-policy algorithms use another (behavior) policy that is independent of the policy being improved to sample trajectories during training. For example, Q-learning is an off-policy algorithm, as it updates the Q-values without considering the actual policy being followed. The advantage of the separation is that behavior policy can sample all actions whereas estimation policy can be greedy.

2.4 Model free Algorithms

Model free algorithms does not make any assumptions about how the agent interacts with the environment. They directly learn the optimal policy or value functions thru iterative methods rather than exploring the whole state-actions space.

3. DDPG Algorithm:

DDPG is a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces. It is based on the DPG algorithm (3). It also uses

some of the learning tricks from DQN including replay buffer and local and target networks, one for each actor and critic.

Applying Q-learning to continuous state is not straight forward as finding the greedy policy requires optimization of a_t at every timestep and is too time consuming. Instead DPG algorithm maintains a parametrized actor function $\mu(s|\theta^{\mu})$ which specifies the current policy by deterministically mapping states to a specific action. The critic $Q(s,a)$ is learned using the Bellman equation as in Q-learning. DPG algorithm by Silver et al (5) showed that the following is the policy gradient, the gradient of the policy's performance. (1)

$$\nabla_{\theta^{\mu}} J \approx E_{s_t \sim \rho^{\beta}} [\nabla_a Q(s,a|\theta^Q)|_{s=s_t\, a=\mu(s_t)} \nabla_{\theta_{\mu}} \mu(s|\theta^{\mu})|_{s=s_t} \quad \text{-------> (3)}$$

It also uses ideas from DQN to make the learning algorithm more stable.

1) ***Replay Buffer***
   Like, DQN, DDPG uses experience replay to make the learning algorithm more stable. The agent's experience at each time step t is stored as a tuple e_i = (s, a, s', r) and these tuples are added to a buffer B = { $e_1, e_2, \ldots \ldots \ldots e_n$} . While learning (in the inner loop) data is sampled in mini batches from this buffer to train the networks. The samples are picked at random. Correlation between consecutive samples affects the stability of the learning process. Randomizing reduces the correlation between consecutive samples thus reducing the variance of updates.
2) ***Two Networks***
   Like DQN, DDPG also uses a target network and local network to improve the stability of the algorithm. This is done as follows. The network is trained with at target Q network to give consistent targets during temporal difference backups. In DDPG instead of directly copying weights from target, a soft target update is done for the actor-critic.
   A copy of the actor-critic networks, $Q'(s,a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$ respectively, that are used for calculating target values. The weights of these target networks are then updated as below.

   $$\theta' = \tau\theta + (1-\tau)\theta' \; with \; \tau < 1 \quad \text{----> (4)}$$

   This makes the algorithm more stable compared to online Q-learning as the target values are constrained to change slowly. There is both a target actor: Q' and critic: μ' to train the critic without divergence. This slows the learning process while make the algorithm stable.

3) **Batch Normalization**

   When learning from low dimensional feature vector observations, the different components of the observation may have different physical units (for example, positions, velocities etc.) and the ranges may wary across environments. This makes it difficult to identify hyper-parameters the learn effectively across environments with different scales and state values. To address this issues DDPG uses batch normalization.

4) Exploration

To facilitate exploration in a continuous action space, noise sampled from a noise process N is added to the exploration policy μ'

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + N \qquad \text{--------> (5)}$$

Orinstein-Uhlenbeck process is used to generate the noise.

3.2. The Algorithm

The Algorithm from the original DDPG paper is as below.

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

4. Architecture

The Architecture of the DQN implementation for Unity Environment: Reacher has five main components

1) Agent 2) The Actor-Critic Networks 3) Training 4) Saved Model

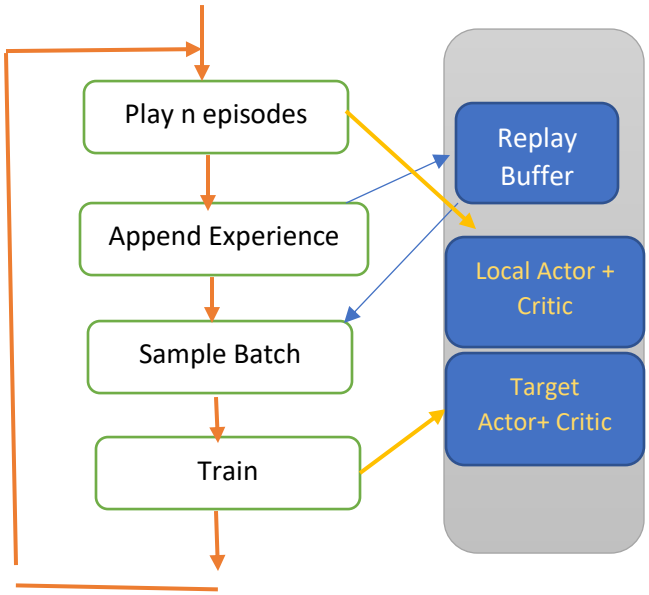

Fig. 2

4.1 Agent

The Agent has five components, four networks: local actor and critic networks to train and the other, target actor and critic networks to verify and a Replay buffer to save episodes for training. Data from the replay buffer is sampled to train the actor network while the target network is updated slowly.

4.2. The Actor-Critic Networks

The Unity Reacher environment has 33 observations for each state and 4 possible actions (see Environment below). The 33 observations form the input of a MLP network with 2 hidden layers and 4 outputs. The hidden layers have 128 neurons each. See Fig 2. All four networks have the same topology. The critic network concatenates the actions of the actor to the second hidden layer.

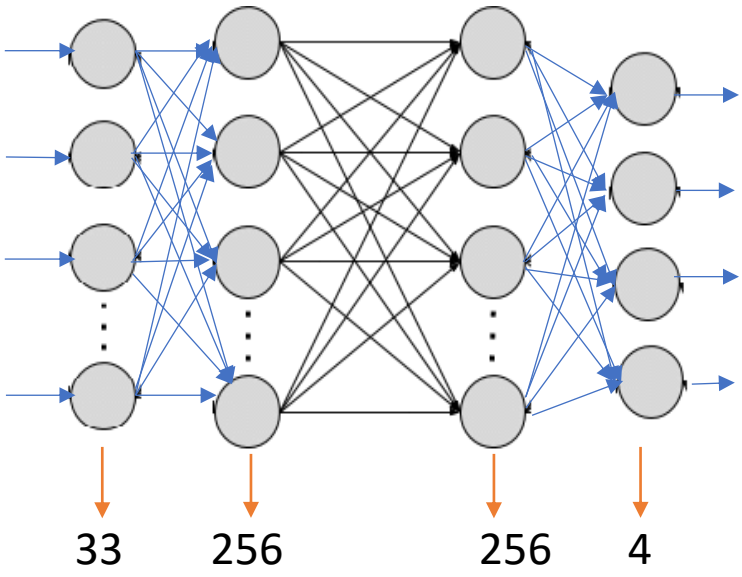

Fig 2

4.3 Training & Testing

The Agent is trained for 700 episodes. With this network an average score of 30+ over 100 episodes was achieved within 500 episodes consistently. The training can be run by executing

- python Reacher.py.

The parameters used for the model are

Episode count : Maximum : 700

Learning Rate & TAU :

```
Learning rate Actor : 2e-04 to 1e-04
Learning rate Critic : 2e-04 to 1e-04

TAU for soft updates : 1e-3

BATCH SIZE : 256
```

4.4 Saved Model execution.

The model is saved once a score of 30+ is reached. This may not consistently perform and show a score of 30+ each time. But will gain a reasonable score in most runs and 30+ in few cases. To run saved model

- python run_model.py

The code runs the game 3 times using saved model

5. Environment

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. The goal of the 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.

In this implementation a single agent version of the Reacher environment is solved. The task is episodic, and to solve the environment, the agent must get an average score of +30 score over 100 consecutive episodes.

6. Code

There are 4 files

1. Reacher.py

    This file has the code to train the networks. To run training execute this script.

2. ddpg_agent.py

   The Agent and replay buffer is defined in this file. The Agent class instantiates four networks: two actors and two critics. Replay buffer is used to store episodes of game play. Data is randomly retrieved from the replay buffer to train the networks.

3. model.py

   Model.py has the MLP definition: both forward and backward pass of actor and critic networks. The critic networks add the actions of the actor to the second layer of the MLP during forward pass.

4. run_model.py

   run_model.py if executed run the saved model. Make sure the Banana application path is valid before running this.

7. References

1. Continuous Control with Deep Reinforcement Learning.

2. Reinforcement Learning An Introduction Richard S. Sutton & Andrew G. Barto

3. Deep reinforcement Learning hand-On. Maxim Lapan.

4. Deep Deterministic Policy Gradients in Tensorflow. Patrick Emami

5. Deterministic Policy Gradient Algorithms. Silver et. al