Reinforcement Learning
DDPG Implementation based on Tensor-flow2

Emanuele Giacomini 1743995

February 2019

Contents

1 Deep Deterministic Policy Gradient

The Algorithm to be implemented is called **DDPG** and it can be used in problems with continuous action spaces.
The idea behind the DDPG is to implement an algorithm which concurrently learns the **Q-function** and a deterministic policy μ_θ.
This is achieved through the application of the **Bellman Equation** (1) for estimating the Q-function, which will later be used to estimate the policy μ.

$$V(s) = \max_a (R(s,a) + \gamma V(s')) \tag{1}$$

The motivation behind this rules is that by knowing the optimal Q-function Q^*, we can decide the best action based on the equation (2)

$$a^*(s) = \max_a Q^*(s,a) \tag{2}$$

However this approach does not work in continuous action space due to the high computational cost and the non triviality of exploring exhaustively the whole action space.
By assuming that $Q^*(s,a)$ is differentiable w.r.t. the action input space, it is possible to structure a gradient-based optimization method for $\mu(s)$ such that:

$$Q(s, \mu(s)) \approx \max_a Q(s,a) \tag{3}$$

1.1 Q-learning

As mentioned before, the Q-function should be able to work like an estimator for the bellman equation. The Q^* function may be expressed in the Bellman form:

$$Q^*(s,a) = E_{s' \sim P}\big[r(s,a) + \gamma \max_{a'} Q^*(s',a')\big] \tag{4}$$

The first task of the algorithm is to produce an approximator for Q^*, in this case called $Q_\phi(s,a)$, with parameters ϕ.
By taking a set of experiences $< s, a, r, s', d > \in \mathcal{D}$, the authors designed the following MSBE (Mean Squared Bellman Error) Loss:

$$\mathcal{L}(\phi, \mathcal{D}) = E_{(s,a,r,s',d) \in \mathcal{D}}\left[\left(Q_\phi(s,a) - (r + \gamma(1-d)\max_{a'} Q_\phi(s',a')\right)^2\right] \tag{5}$$

Where

- s : Current state
- a : Action
- r : Reward $r(s,a)$
- s' : Next state following a in s

- d : Terminal flag

Notice how $\mathcal{D}$ contains experiences previously obtained, possibly with outdated policies or additive noises. The important factor is that the approximator Q should not consider these factors, as the Bellman equation should be applied to every possible transition [1].
A tricky part of the equation is the **target** t term:

$$r + \gamma(1-d)\max_{a'} Q_\phi(s', a') \tag{6}$$

Notice that t does depend on ϕ which are the parameters that MSBE Loss is trying to optimize, hence making the optimization procedure *unstable*.
The solution is to use a new set of parameters ϕ_{targ}, similar to ϕ but with a time delay. This is achieved by building a symmetric Q network $Q'_{\phi_{\text{targ}}}$ which is initialized with $\phi_{\text{targ}} = \phi$ and then updated at each interaction by *polyak* averaging:

$$\phi_{\text{targ}} \longleftarrow \tau\phi_{\text{targ}} + (1-\tau)\phi \tag{7}$$

with $\tau \in [0, 1]$ is the polyak update factor.

1.2 Computing the best action

In order to compute actions that maximizes $Q_{\phi_{\text{targ}}}$ a **target policy network** $\mu_{\theta_{\text{targ}}}$ is established, and updated with the same rule for the target Q-function of polyak averaging from the policy network μ_θ. The final form of the MSBE loss to minimize is the following:

$$\mathcal{L}(\phi, \mathcal{D}) = E_{(s,a,r,s',d)\in\mathcal{D}}\left[\left(Q_\phi(s,a) - (r + \gamma(1-d)Q'_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))\right)^2\right] \tag{8}$$

1.3 Policy Gradient

Optimizing the policy is quite straightforward. The deterministic policy $\mu_\theta(s)$ should learn which action a optimizes which state s. This can be done through a gradient ascent on the Q-function:

$$\max_\theta E_{s\sim\mathcal{D}}\left[Q_\phi(s, \mu_\theta(s))\right] \tag{9}$$

[1] This is the reason why DDPG is defined as **off-policy** algorithm

1.4 Exploration

Begin a deterministic policy, each state is associate with only one action. This means that the agent will likely be stuck with a small subset of explored state space. To overcome this, noise is added to the actions performed by μ_θ in training phases. As the paper suggests, a good noise function is the **Ornstein–Uhlenbeck** process, which act as a correlated noise function.

1.5 Algorithm

Here the algorithm is shown in order to clarify the steps on the Implementation section later to be seen.

Algorithm 1: DDPG

begin
  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 $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{\text{low}}, a_{\text{high}})$ where $\epsilon \sim N$
    Observe next state s', reward r and done signal d to indicate whether s' is terminal
    Store (s, a, r, s', d) in the replay buffer $\mathcal{D}$
    if *d is terminal* **then**
      Reset the environment state
    end
    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

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

    Update policy by one step of gradient **ascent** using

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

    Update target networks with

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

  until *convergence*;
end

2 Implementation

The algorithm was implemented using **python3.7** and **tensor-flow 2.0.0**.
The structure of the program was designed in an hierarchical structure, with bottom-up approach, starting from basic self independent structures for the Actor/Critic networks, up to an interface layer that hides the training and evaluation of the algorithm.

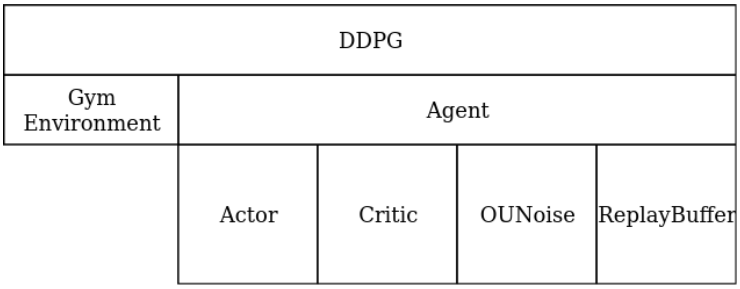

Figure 1: Abstract block scheme of the algorithm implementation.

The following sections will describe at higher level how each class is built in order to accomplish its tasks, in the following order:

- ReplayBuffer
- Actor
- Critic
- Agent
- DDPG

2.1 Replay Buffer

As mentioned in section 2.1, the DDPG algorithm is defined as an off-policy algorithm, hence it requires a set of previous experiences of the form (s, a, r, s', s) that must be randomly sampled at each update iteration.
The proposed method to store the transitions is through the use of a **deque** structure implemented by python itself. The deque is preferred over other structures for its interface functions that are optimal for this task:

- **append** (left and right) $\in \mathcal{O}(1)$
- **pop** (left and right) $\in \mathcal{O}(1)$
- Random access $\in \mathcal{O}(1)$

Notice how every operation has a very low computational cost.
The following code snippets show how the script handles the new interface functions implemented for the algorithm.

```
class ReplayBuffer:
    """
    A ReplayBuffer object stores a set of experiences under the
    form of:
    (state, action, reward, is_done, next_state)
    """

    def __init__(self, size):
        self.size = size
        self.count = 0
        self.buffer = deque()

    def store(self, s, a, r, done, ns):
        elem = (s, a, r, done, ns)
        if self.count < self.size:
            self.buffer.append(elem)
            self.count += 1
        else:
            self.buffer.pop()
            self.buffer.appendleft(elem)

    def sample(self, batch_size):
        _batch = random.sample(self.buffer, batch_size)
        s_batch = [_[0] for _ in _batch]
        a_batch = [_[1] for _ in _batch]
        r_batch = [_[2] for _ in _batch]
        d_batch = [_[3] for _ in _batch]
        ns_batch = [_[4] for _ in _batch]
        return s_batch, a_batch, r_batch, d_batch, ns_batch

    ...
```

The two highlighted functions implemented are the I/O operations allowed on the buffer:

- store(s, a, r, s', d) for storing new experiences in the buffer
- sample(batch_size) for sampling batch_size experiences

2.2 Actor

The Actor class implements the policy μ_θ described in the paper. Begin a function approximator, the decision was to make this class inherit proprieties of a **Keras Model** class, as it represents neural network models, and comes with its set of pre-made interfaces used for weights manipulations.
Regarding model hyper-parameters, according to the paper, these are the decisions:

- State input (for the MountainCarContinuous case, 2 features)
- Two hidden layers with respective 300 and 400 units
- Re-LU activations for hidden layers
- Hyperbolic tangent as output activation (for the MountainCarContinuous case, 1 real value)

As mentioned, μ_θ should learn to assign the optimal action for each possible state s given as input. Notice that learning procedure is not directly linked to the actor library as it is an operation that involves the Q-function (or Critic), hence it is handled at higher levels.
Here are some snippets of the code involved:

```
class Actor(tf.keras.Model):
    def __init__(self, state_size, action_size, max_action,
                 name='Actor'):
        super().__init__(name=name)
        self.state_size = state_size
        self.action_size = action_size

        self.max_action = max_action

        self.l1 = Dense(300, name='L1')
        self.l2 = Dense(400, name='L2')
        self.l3 = Dense(action_size, name='L3')

        with tf.device("/cpu:0"):
            self(tf.constant(
                np.zeros(shape=(1,)+state_size,
                    dtype=np.float64)))

    def call(self, inputs):
        features = tf.nn.relu(self.l1(inputs))
        features = tf.nn.relu(self.l2(features))
        features = self.l3(features)
        action = self.max_action * tf.nn.tanh(features)
        return action
```

The decision of using so many units for a simple problem like the one requested, is the fact that such structure may be able to perform well on harder proposed problems.

2.3 Critic

The Critic class implements the Q-function described before. Such approximator should take as inputs a tuple (s, a) and evaluate how good a is, knowing that the current state is s.
The structure itself of the approximator is quite similar to the Actor one, except for the input state which differs, as it became a concatenation of state shape and action shape. As for the Actor, since weight manipulation is an operation that requires the target networks, no optimization steps are added to the class.

```
class Critic(tf.keras.Model):
    def __init__(self, state_shape, action_dim, name='Critic'):
        super().__init__(name=name)

        self.l1 = Dense(300, name='L1')
        self.l2 = Dense(400, name='L2')
        self.l3 = Dense(1, name='L3')

        dummy_state = tf.constant(
                            np.zeros(shape=(1,)+state_shape,
                                    dtype=np.float64))
        dummy_action = tf.constant(
                            np.zeros(shape=[1, action_dim],
                            dtype=np.float64))
        with tf.device("/cpu:0"):
            self([dummy_state, dummy_action])

    def call(self, inputs):
        states, actions = inputs
        features = tf.concat([states, actions], axis=1)
        features = tf.nn.relu(self.l1(features))
        features = tf.nn.relu(self.l2(features))
        features = self.l3(features)
        return features
```

The size of the network is the same as for the Actor, except that output activation is *linear* as no squashing is required in the paper.

2.4 Agent

The agent represent the core part of the library as it inherits both Actor, Critic, Replay buffer and OU noise generator.
The target networks are also initialized inside the agent, as deep copies of both μ_θ and Q.
Finally the optimizers for the networks have been chosen to be **Adam**.

```
class Agent(object):
    def __init__(self, state_space, action_space, max_action,
                 device):
        self.state_size = state_space.shape[0]
        self.action_size = action_space.shape[0]
        self.max_action = max_action
        self.device = device
        self.actor_local = Actor(state_space.shape,
                                 action_space.high.size,
                                 max_action)
        self.actor_target = Actor(state_space.shape,
                                 action_space.high.size,
                                 max_action)
        self.actor_optimizer = optimizers.Adam(LR_ACTOR)
        # let target be equal to local
        self.actor_target.set_weights(
                self.actor_local.get_weights())

        self.critic_local = Critic(state_space.shape,
                                 action_space.high.size)
        self.critic_target = Critic(state_space.shape,
                                 action_space.high.size)
        self.critic_optimizer = optimizers.Adam(LR_CRITIC)
        # let target be equal to local
        self.critic_target.set_weights(
                self.critic_local.get_weights())

        self.noise = OUNoise(self.action_size)
        self.memory = ReplayBuffer(BUFFER_SIZE)
```

The agent should perform mainly two relevant operations:

- **Act** : given one or a set of states s output the respective optimal action a for evaluation and a noisy action a_n for training.
- **Perceive** : Store the current experience (s, a, r, s', d) and eventually perform a training step.

Act function does take as input a batch of actions (np.ndarray) and returns a tuple containing $[a_p, a_n]$ where a_p represent the pure prediction $\mu_\theta(s)$ while $a_n = \mu_\theta(s) + \epsilon$ with $\epsilon \sim \mathcal{N}$. On DDPG layer, one of the two actions will be discarded, based on the current episode mode (training or evaluation).

```
def act(self, state, add_noise=True) -> (float, float):
    state = np.array(state).reshape(1, self.state_size)
    pure_action = self.actor_local.predict(state)[0]
    action = self.noise.get_action(pure_action)
    return action, pure_action
```

The **Perceive** operation is quite bigger as it contains multiple blocks, relative to experience storage, critic and actor training and finally target updating.
Experimentally it is possible to notice that before training, it's convenient to *warm-up* the replay buffer by storing a relatively large number of experiences before starting the training.
Another experiment tested was to execute update steps every *tot* steps. However it was proven that updating once per step was stable enough.

```
def step(self, state, action, reward, done, next_state,
        train=True) -> None:
    self.memory.store(state, action, reward, done, next_state)
    if train and self.memory.count > BATCH_SIZE and
            self.memory.count > MIN_MEM_SIZE:
        if self.current_steps % UPDATE_STEPS == 0:
            experiences = self.memory.sample(BATCH_SIZE)
            self.learn(experiences, GAMMA)
        self.current_steps += 1
def learn(self, experiences, gamma) -> None:
    states, actions, rewards, dones, next_states = experiences
    ...
    self.critic_train(states, actions, rewards, dones, next_states)
    self.actor_train(states)
    self.update_local()
    return
```

The two following functions *critic_train* and *actor_train* should be considered computing intensive operations, therefore were decorated with the **@tf_function** decorator, which forces the tensor-flow graph operation construction in the program memory. This ensures both extremely faster operations and robustness during multi-network gradient computation.

```
@tf.function
def critic_train(self, states, actions, rewards, dones, next_states):
    with tf.device(self.device):
        # Compute yi
        u_t = self.actor_target(next_states)
        q_t = self.critic_target([next_states, u_t])
        yi = rewards + GAMMA*(1-dones)*q_t
        # Compute MSBE
        with tf.GradientTape() as tape:
            q_l = tf.cast(self.critic_local([states, actions]),
                          dtype=tf.float64)
            loss = (q_l - yi) * (q_l - yi)
            loss = tf.reduce_mean(loss)
        # Update critic by minimizing loss
        dloss_dql = tape.gradient(loss,
                    self.critic_local.trainable_weights)
        self.critic_optimizer.apply_gradients(
            zip(dloss_dql, self.critic_local.trainable_weights))
    return

@tf.function
def actor_train(self, states):
    with tf.device(self.device):
        with tf.GradientTape(watch_accessed_variables=False) as tape:
            tape.watch(self.actor_local.trainable_variables)
            u_l = self.actor_local(states)
            q_l = -tf.reduce_mean(self.critic_local([states, u_l]))
        j = tape.gradient(q_l, self.actor_local.trainable_variables)
        self.actor_optimizer.apply_gradients(
            zip(j, self.actor_local.trainable_variables))
    return
```

Notice how during the actor's training, the variable q_l which represent the loss, is multiplied by -1. This ensures a maximization of the loss instead of a minimization during the optimizer steps.

Finally the target networks are update by polyak averaging:

```
def update_local(self):
    def soft_updates(local_model: tf.keras.Model,
            target_model: tf.keras.Model) -> np.ndarray:
        local_weights = np.array(local_model.get_weights())
        target_weights = np.array(target_model.get_weights())
        assert len(local_weights) == len(target_weights)
        new_weights = TAU * local_weights +
                (1 - TAU) * target_weights
        return new_weights

    self.actor_target.set_weights(soft_updates(self.actor_local,
                                               self.actor_target))
    self.critic_target.set_weights(soft_updates(self.critic_local,
                                                self.critic_target))
```

2.5 DDPG layer

The last and more abstract layer of the presented script contains the basic interfaces for users. The constructor takes as input the environment on which the algorithm take place, and the device on which to run the gradient computation (usually /GPU:0).

```
class DDPG(object):
    def __init__(self, env, device):
        self.env = env
        self.agent = Agent(self.env.observation_space,
                           self.env.action_space,
                           env.action_space.high,
                           device)
        ...
```

The main function used by the DDPG layer is the *run*. It takes as parameters the number of episodes to execute and the maximum steps per episode. During the run, infos are stored in memory regarding action statististics and episode reward per episode.

```
def run(self, max_episodes: int, max_iterations: int, render: bool):
    rewards = []
    for ep in range(max_episodes):
        train_r, train_d, train_mean, train_std, train_s =
            self.run_epoch(max_iterations,
                           render=render)
        test_r, test_d, test_mean, test_std, test_s =
            self.run_epoch(max_iterations,
                           render=render,
                           training=False)
        print_episode(ep, train_r, train_d, train_mean, train_std,
                      train_s, test_r, test_d, test_mean, test_std,
                      test_s)
        rewards.append(test_r)
        if ep % 2 == 0 and ep > 0:
            plot_ddpg_decisions(ep,
                                self.agent.actor_local,
                                self.agent.critic_local,
                                self.env)
            plot_reward(ep, list(zip(range(0, ep+2), rewards)))
            self.agent.store_weights(ep)
    self.env.close()
```

In the presented form, the run function stores the Agent actor's weights for backup and plot both the total reward history and decisions every 2 episodes.

3 Experiments

For the presented experiment, the following hyper-parameters were used:

- ReplayBuffer size : 10^4
- Batch size : 128
- ReplayBuffer warmup: 5000
- $\gamma = 0.99$
- $\tau = 0.001$
- Actor's lr = 0.002
- Critic's lr = 0.003

By launching the *mountain_car_continuous_ddpg.py* script, the following results have obtained:
It is also possible to visualize the actor's decisions with an heatmap. Notice that this is possible in this case since the state is composed by two features, and the action space is $\in \mathcal{R}$
From Figure 2, notice how the agent converges to an almost optimal solution after around 40 episodes, While from Figure 3 notice how the actor (PI(s)) assign positive (yellow) velocities on a specific set of states.
The Q(s, PI(s)) graph shows the Q-value for each state, given the actor's decision. Notice that the actor is almost able to optimize the action for the whole state space.

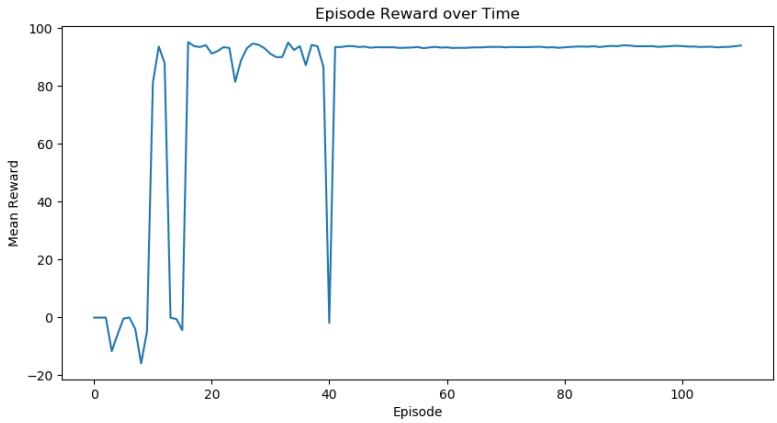

Figure 2: Reward History.

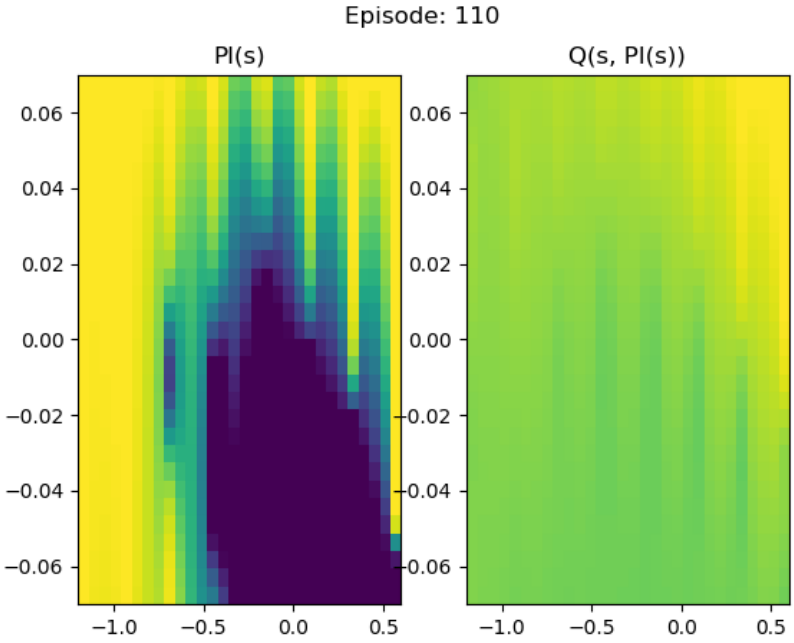

Figure 3: Decision Heatmap