AVD871 Project Report

Deep Deterministic Policy Gradient Method (DDPG)

Edera Venkata Naga Sai Bharadwaja
SC20M104
Machine Learning and Computing
Indian Institute of Space Sciene and Technology

Table of Contents

List of Figures

Title of the Paper

CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING

Authors : Timothy P. Lillicrap , Jonathan J. Hunt , Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver Daan Wierstra

venue of publication: Google Deepmind London, UK

Abstract

Source of Paper : https://arxiv.org/pdf/1509.02971.pdf

Underlying idea is to use Deep Q-Learning to the continuous action domain.To operate over continuous action spaces , the paper presents model-free algorithm based on the deterministic policy gradient know as actor-critic Algorithm. By using the DQN and Actor Critic algorithm, network architecture and hyper-parameters,algorithm robustly solves more than 20 simulated physics tasks, including classic problems such as cartpole swing-up, dexterous manipulation, legged locomotion and car driving . This 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

In the Reinforcement Learning section of our course **AVD871** we've a topic called Policy Gradient Methods. So this paper deals with **Modern Policy Gradient Algorithm** called **Deep Deterministic Policy Gradient algorithm**. So in my point of view this algorithm will be an extension of **Actor Critic Methods** and it is **Modern Policy Gradient Algorithm to Deal with continuous Space**.

1. Introduction to Policy Gradient Theorem

Policy Gradient methods learn the policy directly with a parameterized function respect to θ, π(a/s;θ).The reward function (opposite of loss function) as the expected return and train the algorithm with the goal to maximize the reward function.

In discrete space:

$$\mathcal{J}(\theta) = V_{\pi_\theta}(S_1) = \mathbb{E}_{\pi_\theta}[V_1]$$

where S_1 is the initial starting state.

Or in continuous space:

$$\mathcal{J}(\theta) = \sum_{s \in \mathcal{S}} d_{\pi_\theta}(s) V_{\pi_\theta}(s) = \sum_{s \in \mathcal{S}} \Big(d_{\pi_\theta}(s) \sum_{a \in \mathcal{A}} \pi(a|s,\theta) Q_\pi(s,a) \Big)$$

$$\begin{aligned}
\mathcal{J}(\theta) &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi(a|s;\theta) Q_\pi(s,a) \\
\nabla \mathcal{J}(\theta) &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \nabla \pi(a|s;\theta) Q_\pi(s,a) \\
&= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi(a|s;\theta) \frac{\nabla \pi(a|s;\theta)}{\pi(a|s;\theta)} Q_\pi(s,a) \\
&= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi(a|s;\theta) \nabla \ln \pi(a|s;\theta) Q_\pi(s,a) \\
&= \mathbb{E}_{\pi_\theta}[\nabla \ln \pi(a|s;\theta) Q_\pi(s,a)]
\end{aligned}$$

$$\nabla \mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta}[\nabla \ln \pi(a|s,\theta) Q_\pi(s,a)]$$

The policy function π(. / s) is always modeled as a probability distribution over actions $\mathbb{A}$ given the current state and thus it is stochastic.In DPG or DDPG ,instead models the policy as a deterministic decision: a = μ(s).

- $\rho_0(s)$: The initial distribution over states
- $\rho^\mu(s \to s', k)$: Starting from state s, the visitation probability density at state s' after moving k steps by policy μ.
- $\rho^\mu(s')$: Discounted state distribution, defined as $\rho^\mu(s') = \int_\mathcal{S} \sum_{k=1}^\infty \gamma^{k-1} \rho_0(s) \rho^\mu(s \to s', k) ds$.

The objective function to optimize for is listed as follows:

$$J(\theta) = \int_\mathcal{S} \rho^\mu(s) Q(s, \mu_\theta(s)) ds$$

2. Deterministic policy gradient theorem

According to the chain rule, we first take the gradient of Q w.r.t. the action a and then take the gradient of the deterministic policy function μ w.r.t to θ

$$\begin{aligned}\nabla_\theta J(\theta) &= \int_S \rho^\mu(s) \nabla_a Q^\mu(s,a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)} ds \\ &= \mathbb{E}_{s\sim\rho^\mu}[\nabla_a Q^\mu(s,a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}]\end{aligned}$$

3. Deep Q-Network (DQN)

Theoretically, we can memorize Q(.) for all state-action pairs in Q-learning, like in a gigantic table. However, it quickly becomes computationally infeasible when the state and action space are large. Thus people use functions (i.e. a machine learning model) to approximate Q values and this is called function approximation. For example, if we use a function with parameter θ to calculate Q values, we can label Q value function as Q(s,a;θ).

Unfortunately Q-learning may suffer from instability and divergence when combined with an nonlinear Q-value function approximation and bootstrapping.Deep Q-Network aims to greatly improve and stabilize the training procedure of Q-learning by two innovative mechanisms:

3.1 Experience Replay

All the episode steps $e_t = (S_t, A_t, R_t, S_{t+1})$ are stored in one replay memory $D_t = e_1, \ldots, e_t$. D_t has experience tuples over many episodes. During Q-learning updates, samples are drawn at random from the replay memory and thus one sample could be used multiple times. Experience replay improves data efficiency, removes correlations in the observation sequences, and smooths over changes in the data distribution.

3.2 Periodically Updated Target

Q is optimized towards target values that are only periodically updated. The Q network is cloned and kept frozen as the optimization target every C steps (C is a hyperparameter). This modification makes the training more stable as it overcomes the short-term oscillations.

The loss function looks like this:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\Big[\big(r + \gamma \max_{a'} Q(s', a'; \theta^{-}) - Q(s, a; \theta)\big)^2\Big]$$

where U(D) is a uniform distribution over the replay memory D; θ^{-} is the parameters of the frozen target Q-network.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Fig. 1. Algorithm for DQN

4. Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient, is a model-free off-policy actor-critic algorithm, combining DPG with DQN.As DQN (Deep Q-Network) stabilizes the learning of Q-function by experience replay and the frozen target network. The original DQN works in discrete space, and DDPG extends it to continuous space with the actor-critic framework while learning a deterministic policy.

In order to do better exploration, an exploration policy μ' is constructed by adding noise $\mathbb{N}$:

$$\mu'(s) = \mu_\theta(s) + \mathbb{N}$$

In addition, DDPG does soft updates ("conservative policy iteration") on the parameters of both actor and critic,
with $\tau << 1$:

$$\theta' \leftarrow \tau * \theta + (1-\tau) * \theta'$$

In this way, the target network values are constrained to change slowly, different from the design in DQN that the target network stays frozen for some period of time.

One detail in the paper that is particularly useful in robotics is on how to normalize the different physical units of low dimensional features. For example, a model is designed to learn a policy with the robot's positions and velocities as input; these physical statistics are different by nature and even statistics of the same type may vary a lot across multiple robots. Batch normalization is applied to fix it by normalizing every dimension across samples in one minibatch.

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

Fig. 2. DDPG algorithm

Actor is denoted by μ
Parameters of Actor are denoted by θ_μ
Critic is denoted by Q
Parameters of Critic are denoted by θ_Q

5. Implementation

5.1 Replay Buffer

```python
import numpy as np

class ReplayBuffer:
    def __init__(self, max_size, input_shape, n_actions):
        self.mem_size = max_size
        self.mem_cntr = 0
        self.state_memory = np.zeros((self.mem_size, *input_shape))
        self.new_state_memory = np.zeros((self.mem_size, *input_shape))
        self.action_memory = np.zeros((self.mem_size, n_actions))
        self.reward_memory = np.zeros(self.mem_size)
        self.terminal_memory = np.zeros(self.mem_size, dtype=np.bool)

    def store_transition(self, state, action, reward, state_, done):
        index = self.mem_cntr % self.mem_size

        self.state_memory[index] = state
        self.new_state_memory[index] = state_
        self.action_memory[index] = action
        self.reward_memory[index] = reward
        self.terminal_memory[index] = done

        self.mem_cntr += 1

    def sample_buffer(self, batch_size):
        max_mem = min(self.mem_cntr, self.mem_size)

        batch = np.random.choice(max_mem, batch_size, replace=False)

        states = self.state_memory[batch]
        states_ = self.new_state_memory[batch]
        actions = self.action_memory[batch]
        rewards = self.reward_memory[batch]
        dones = self.terminal_memory[batch]

        return states, actions, rewards, states_, dones
```

Fig. 3. Replay Buffer

5.2 Actor Network and Critic Network

```
import os
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.layers import Dense

class CriticNetwork(keras.Model):
    def __init__(self, fc1_dims=512, fc2_dims=512, n_actions=2,
            name='critic', chkpt_dir='tmp/ddpg'):
        super(CriticNetwork, self).__init__()
        self.fc1_dims = fc1_dims
        self.fc2_dims = fc2_dims
        self.n_actions = n_actions

        self.model_name = name
        self.checkpoint_dir = chkpt_dir
        self.checkpoint_file = os.path.join(self.checkpoint_dir,
                    self.model_name+'_ddpg.h5')

        self.fc1 = Dense(self.fc1_dims, activation='relu')
        self.fc2 = Dense(self.fc2_dims, activation='relu')
        self.q = Dense(1, activation=None)

    def call(self, state, action):
        action_value = self.fc1(tf.concat([state, action], axis=1))
        action_value = self.fc2(action_value)

        q = self.q(action_value)

        return q

class ActorNetwork(keras.Model):
    def __init__(self, fc1_dims=512, fc2_dims=512, n_actions=2, name='actor',
            chkpt_dir='tmp/ddpg'):
        super(ActorNetwork, self).__init__()
        self.fc1_dims = fc1_dims
        self.fc2_dims = fc2_dims
        self.n_actions = n_actions

        self.model_name = name
        self.checkpoint_dir = chkpt_dir
        self.checkpoint_file = os.path.join(self.checkpoint_dir,
                    self.model_name+'_ddpg.h5')

        self.fc1 = Dense(self.fc1_dims, activation='relu')
        self.fc2 = Dense(self.fc2_dims, activation='relu')
        self.mu = Dense(self.n_actions, activation='tanh')

    def call(self, state):
        prob = self.fc1(state)
        prob = self.fc2(prob)

        mu = self.mu(prob)

        return mu
```

Fig. 4. Actor Network and Critic Network

5.3 Agent

```
import numpy as np
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.optimizers import Adam
from buffer import ReplayBuffer
from networks import ActorNetwork, CriticNetwork

class Agent:
    def __init__(self, input_dims, alpha=0.001, beta=0.002, env=None,
            gamma=0.99, n_actions=2, max_size=1000000, tau=0.005,
            fc1=400, fc2=300, batch_size=64, noise=0.1):
        self.gamma = gamma
        self.tau = tau
        self.memory = ReplayBuffer(max_size, input_dims, n_actions)
        self.batch_size = batch_size
        self.n_actions = n_actions
        self.noise = noise
        self.max_action = env.action_space.high[0]
        self.min_action = env.action_space.low[0]

        self.actor = ActorNetwork(n_actions=n_actions, name='actor')
        self.critic = CriticNetwork(n_actions=n_actions, name='critic')
        self.target_actor = ActorNetwork(n_actions=n_actions, name='target_actor')
        self.target_critic = CriticNetwork(n_actions=n_actions, name='target_critic')

        self.actor.compile(optimizer=Adam(learning_rate=alpha))
        self.critic.compile(optimizer=Adam(learning_rate=beta))
        self.target_actor.compile(optimizer=Adam(learning_rate=alpha))
        self.target_critic.compile(optimizer=Adam(learning_rate=beta))

        self.update_network_parameters(tau=1)

    def update_network_parameters(self, tau=None):
        if tau is None:
            tau = self.tau

        weights = []
        targets = self.target_actor.weights
        for i, weight in enumerate(self.actor.weights):
            weights.append(weight * tau + targets[i]*(1-tau))
        self.target_actor.set_weights(weights)

        weights = []
        targets = self.target_critic.weights
        for i, weight in enumerate(self.critic.weights):
            weights.append(weight * tau + targets[i]*(1-tau))
        self.target_critic.set_weights(weights)

    def remember(self, state, action, reward, new_state, done):
        self.memory.store_transition(state, action, reward, new_state, done)
```

```python
def save_models(self):
    print('... saving models ...')
    self.actor.save_weights(self.actor.checkpoint_file)
    self.target_actor.save_weights(self.target_actor.checkpoint_file)
    self.critic.save_weights(self.critic.checkpoint_file)
    self.target_critic.save_weights(self.target_critic.checkpoint_file)
def load_models(self):
    print('... loading models ...')
    self.actor.load_weights(self.actor.checkpoint_file)
    self.target_actor.load_weights(self.target_actor.checkpoint_file)
    self.critic.load_weights(self.critic.checkpoint_file)
    self.target_critic.load_weights(self.target_critic.checkpoint_file)

def choose_action(self, observation, evaluate=False):
    state = tf.convert_to_tensor([observation], dtype=tf.float32)
    actions = self.actor(state)
    if not evaluate:
        actions += tf.random.normal(shape=[self.n_actions],
                mean=0.0, stddev=self.noise)
    # note that if the environment has an action > 1, we have to multiply by
    # max action at some point
    actions = tf.clip_by_value(actions, self.min_action, self.max_action)
    return actions[0]

def learn(self):
    if self.memory.mem_cntr < self.batch_size:
        return

    state, action, reward, new_state, done = \
            self.memory.sample_buffer(self.batch_size)

    states = tf.convert_to_tensor(state, dtype=tf.float32)
    states_ = tf.convert_to_tensor(new_state, dtype=tf.float32)
    rewards = tf.convert_to_tensor(reward, dtype=tf.float32)
    actions = tf.convert_to_tensor(action, dtype=tf.float32)

    with tf.GradientTape() as tape:
        target_actions = self.target_actor(states_)
        critic_value_ = tf.squeeze(self.target_critic(
                            states_, target_actions), 1)
        critic_value = tf.squeeze(self.critic(states, actions), 1)
        target = reward + self.gamma*critic_value_*(1-done)
        critic_loss = keras.losses.MSE(target, critic_value)

    critic_network_gradient = tape.gradient(critic_loss,
                                    self.critic.trainable_variables)
    self.critic.optimizer.apply_gradients(zip(
        critic_network_gradient, self.critic.trainable_variables))

    with tf.GradientTape() as tape:
        new_policy_actions = self.actor(states)
        actor_loss = -self.critic(states, new_policy_actions)
        actor_loss = tf.math.reduce_mean(actor_loss)

    actor_network_gradient = tape.gradient(actor_loss,
                                self.actor.trainable_variables)
    self.actor.optimizer.apply_gradients(zip(
        actor_network_gradient, self.actor.trainable_variables))

    self.update_network_parameters()
```

Fig. 5. Agent

5.4 Main

```python
import gym
import numpy as np
from ddpg_tf2 import Agent
from utils import plot_learning_curve
if __name__ == '__main__':
    env = gym.make('Pendulum-v0')
    agent = Agent(input_dims=env.observation_space.shape, env=env,n_actions=env.action_space.shape[0])
    n_games = 250

    figure_file = 'plots/pendulum.png'

    best_score = env.reward_range[0]
    score_history = []
    load_checkpoint = False

    if load_checkpoint:
        n_steps = 0
        while n_steps <= agent.batch_size:
            observation = env.reset()
            action = env.action_space.sample()
            observation_, reward, done, info = env.step(action)
            agent.remember(observation, action, reward, observation_, done)
            n_steps += 1
        agent.learn()
        agent.load_models()
        evaluate = True
    else:
        evaluate = False

    for i in range(n_games):
        observation = env.reset()
        done = False
        score = 0
        while not done:
            action = agent.choose_action(observation, evaluate)
            observation_, reward, done, info = env.step(action)
            score += reward
            agent.remember(observation, action, reward, observation_, done)
            if not load_checkpoint:
                agent.learn()
            observation = observation_

        score_history.append(score)
        avg_score = np.mean(score_history[-100:])

        if avg_score > best_score:
            best_score = avg_score
            if not load_checkpoint:
                agent.save_models()

        print('episode ', i, 'score %.1f' % score, 'avg score %.1f' % avg_score)

    if not load_checkpoint:
        x = [i+1 for i in range(n_games)]
        plot_learning_curve(x, score_history, figure_file)
```

Fig. 6. Main

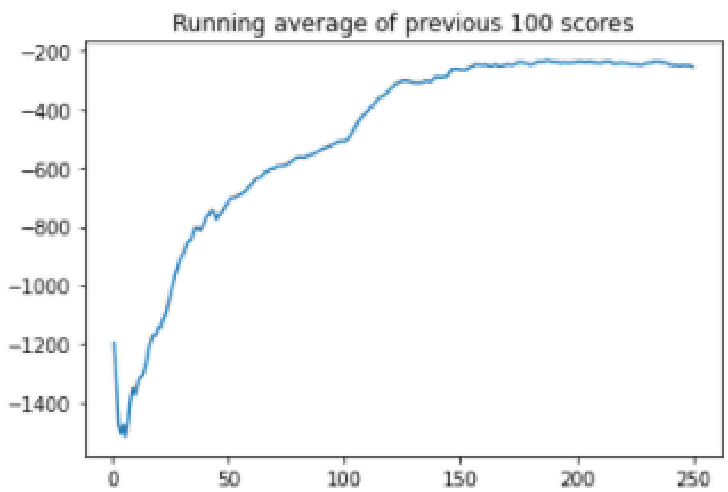
5.5 Output

```
[2021-04-17 19:46:38,445] Making new env: Pend
C:\Users\Bharadwaj\.conda\envs\gputest\lib\sit
ters to load are deprecated.  Call .resolve an
  result = entry_point.load(False)
```

```
... saving models ...
episode  0 score -1197.8 avg score -1197.8
episode  1 score -1481.4 avg score -1339.6
episode  2 score -1754.1 avg score -1477.8
episode  3 score -1600.0 avg score -1508.3
episode  4 score -1344.2 avg score -1475.5
episode  5 score -1731.5 avg score -1518.2
episode  6 score -1167.6 avg score -1468.1
episode  7 score -883.2 avg score -1395.0
episode  8 score -1006.3 avg score -1351.8
episode  9 score -1596.1 avg score -1376.2
episode  10 score -995.7 avg score -1341.6
episode  11 score -1054.5 avg score -1317.7
episode  12 score -1221.0 avg score -1310.3
```

After 248 Episodes

```
episode  248 score -508.4 avg score -251.8
episode  249 score -572.1 avg score -255.1
```



6. Code for the Project

https://github.com/BharadwajEdera/Bharadwaj-Machine-Learning-and-computing/tree/main/DDPG%20mini%20project%20RL

References

- **Source of Paper :** https://arxiv.org/pdf/1509.02971.pdf
- https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#dpg
- https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html#key-concepts
- https://www.udemy.com/course/deep-q-learning-from-paper-to-code/?couponCode=DQN-APR-2021