**Computational Robotic 2023 - SC. Department**

Presented to: Dr. Mohammed Marey

|  |  |  |
| --- | --- | --- |
| **Evaluation points** | | |
| **1** | **reading** |  |
| **2** | **examples** |  |
| **3** | **practical (source code example)** |  |
| **4** | **system architecture & robot platform** |  |
| **5** | **sensors and its data** |  |
| **6** | **idea (approaches and methods)** |  |
| **7** | **understanding ( approaches and methods)** |  |
| **8** | **Applications** |  |
| **9** | **presenting the work** |  |
| **10** | **References** |  |

|  |  |  |
| --- | --- | --- |
| Published year | Paper title | No. of citation |
| 2020 | Robotic Arm Control and Task Training through Deep Reinforcement Learning | 24 |

|  |  |  |
| --- | --- | --- |
| **Presented by: Team Members** | | |
| **ID** | **Name** | **Tasks (Contributions)** |
| 20191700091 | اسامة عنتر محمد عفيفي | Algorithms |
| 20191700059 | احمد محمد إبراهيم محمد | Introduction, idea, & goals |
| 20191700086 | ادهم محمد توفيق محمد عبد ربه | Algorithms |
| 20191700068 | احمد محمد علي عبدالرحمن السيد | System architecture, sensors, data, & applications |
| 20191700322 | طارق أشرف محمود حسين | Experiment & results |

|  |
| --- |
| The code |
| Paper related |
| import gym  import numpy as np  import random  from collections import deque  import tensorflow as tf  from tensorflow.keras.layers import Dense, Input, Lambda  from tensorflow.keras.models import Model  from tensorflow.keras.optimizers import Adam  class DQN\_NAF:  def \_\_init\_\_(self, state\_dim, action\_dim, action\_bounds):  self.state\_dim = state\_dim  self.action\_dim = action\_dim  self.action\_bounds = action\_bounds  self.memory = deque(maxlen=100000)  self.gamma = 0.99  self.tau = 0.001  self.q\_learning\_rate = 0.0001  self.actor\_learning\_rate = 0.0001  self.critic\_learning\_rate = 0.001  self.q\_network = self.\_build\_q\_network()  self.target\_network = self.\_build\_q\_network()  self.actor\_network = self.\_build\_actor\_network()  self.critic\_network = self.\_build\_critic\_network()  def \_build\_q\_network(self):  input\_state = Input(shape=(self.state\_dim,))  input\_action = Input(shape=(self.action\_dim,))  h1 = Dense(64, activation='relu')(input\_state)  h2 = Dense(64, activation='relu')(h1)  h3 = Dense(64, activation='relu')(h2)  q\_values = Dense(1, activation='linear')(h3)  mu = Dense(self.action\_dim, activation='tanh')(h3)  sigma = Dense(self.action\_dim, activation='softplus')(h3)  norm\_dist = Lambda(lambda x: x \* self.action\_bounds)(mu)  model = Model(inputs=[input\_state, input\_action], outputs=[q\_values, norm\_dist, sigma])  model.compile(optimizer=Adam(lr=self.q\_learning\_rate), loss='mse')  return model  def \_build\_actor\_network(self):  input\_state = Input(shape=(self.state\_dim,))  h1 = Dense(64, activation='relu')(input\_state)  h2 = Dense(64, activation='relu')(h1)  h3 = Dense(64, activation='relu')(h2)  mu = Dense(self.action\_dim, activation='tanh')(h3)  sigma = Dense(self.action\_dim, activation='softplus')(h3)  norm\_dist = Lambda(lambda x: x \* self.action\_bounds)(mu)  model = Model(inputs=input\_state, outputs=[norm\_dist, sigma])  model.compile(optimizer=Adam(lr=self.actor\_learning\_rate), loss='mse')  return model  def \_build\_critic\_network(self):  input\_state = Input(shape=(self.state\_dim,))  input\_action = Input(shape=(self.action\_dim,))  h1 = Dense(64, activation='relu')(input\_state)  h2 = Dense(64, activation='relu')(h1)  h3 = Dense(64, activation='relu')(h2)  h4 = Dense(64, activation='relu')(input\_action)  h5 = Dense(64, activation='relu')(h3)  h6 = Dense(64, activation='relu')(h4)  h7 = tf.keras.layers.Concatenate()([h5, h6])  state\_value = Dense(1, activation='linear')(h7)  model = Model(inputs=[input\_state, input\_action], outputs=state\_value)  model.compile(optimizer=Adam(lr=self.critic\_learning\_rate), loss='mse')  return model  def remember(self, state, action, reward, next\_state, done):  self.memory.append((state, action, reward, next\_state, done))  def act(self, state):  action, \_ = self.actor\_network.predict(np.expand\_dims(state, axis=0))  return action[0]  def replay(self):  if len(self.memory) < self.batch\_size:  return  batch = random.sample(self.memory, self.batch\_size)  states, actions, rewards, next\_states, dones = zip(\*batch)  states = np.asarray(states)  actions = np.asarray(actions)  rewards = np.asarray(rewards)  next\_states = np.asarray(next\_states)  dones = np.asarray(dones)  # Compute target Q values  \_, next\_actions, next\_sigmas = self.actor\_network.predict(next\_states)  target\_q\_values = self.target\_network.predict([next\_states, next\_actions])[0]  next\_values = target\_q\_values - next\_sigmas\*\*2 / 2  target\_q\_values = rewards + (1 - dones) \* self.gamma \* next\_values  # Train Q network  self.q\_network.train\_on\_batch([states, actions], [target\_q\_values, actions, actions])  # Train actor and critic networks  with tf.GradientTape() as tape:  mu, sigma = self.actor\_network(states)  values = self.critic\_network([states, mu])  actor\_loss = -tf.reduce\_mean(values)  actor\_gradients = tape.gradient(actor\_loss, self.actor\_network.trainable\_variables)  self.actor\_network.optimizer.apply\_gradients(zip(actor\_gradients, self.actor\_network.trainable\_variables))  with tf.GradientTape() as tape:  target\_actions, target\_sigmas = self.actor\_network(next\_states)  target\_values = self.target\_network([next\_states, target\_actions])  critic\_target = rewards + (1 - dones) \* self.gamma \* target\_values  critic\_loss = tf.reduce\_mean((critic\_target - values)\*\*2)  critic\_gradients = tape.gradient(critic\_loss, self.critic\_network.trainable\_variables)  self.container\_network.optimizer.apply\_gradients(zip(critic\_gradients, self.critic\_network.trainable\_variables))  def update\_target\_network(self):  q\_weights = self.q\_network.get\_weights()  target\_weights = self.target\_network.get\_weights()  for i in range(len(q\_weights)):  target\_weights[i] = self.tau \* q\_weights[i] + (1 - self.tau) \* target\_weights[i]  self.target\_network.set\_weights(target\_weights)  def train(self, episodes):  for episode in range(episodes):  state = env.reset()  total\_reward = 0  done = False  while not done:  action = self.act(state)  next\_state, reward, done, \_ = env.step(action)  self.remember(state, action, reward, next\_state, done)  state = next\_state  total\_reward += reward  self.replay()  self.update\_target\_network()  print("Episode:", episode, "Total reward:", total\_reward)  # Set hyperparameters  gamma = 0.99  tau = 0.001  q\_learning\_rate = 0.0001  actor\_learning\_rate = 0.0001  critic\_learning\_rate = 0.001  buffer\_size = 100000  batch\_size = 64  action\_bounds = 1  # Set up the environment  env = gym.make('RoboticArm-v0')  state\_dim = env.observation\_space.shape[0]  action\_dim= env.action\_space.shape[0]  action\_bounds = env.action\_space.high  # Create DQN\_NAF agent and train  agent = DQN\_NAF(state\_dim, action\_dim, action\_bounds)  agent.train(episodes=1000) |
|
|
|

|  |
| --- |
| Paper main idea |
| 1. Introduction 2. Idea 3. Goal 4. Algorithms 5. System architecture 6. Sensors and its data 7. Applications 8. Experiment 9. Result |