Tampere University

Ronal Bejarano

# OPTIMIZATION OF GRASPING METHODS FOR AN UNDER-ACTUATED GRIPPER BY REINFORCEMENT LEARNING

ASE-9517 Special Assignment in Factory Automation

Faculty of Engineering and Natural Sciences
August-2019

# CONTENTS

# 1. INTRODUCTION

Under-actuation defines the property of a robotic component to have less actuators than degrees of freedom. For robotic grasping systems, this concept aims to simplify the control systems and optimize the sources of mechanical power by reducing the amount of actuators by distributing the movement through multiple sections fluid or mechanically coupled [1].

Nowadays, researchers have develop many mechanical and hydraulic under-actuated grasping systems such as AMADEUS, BarrettHand BH8, Belgrade/USC, Karlsruhe RC, MARS, RTR Hand 2, among others; mostly inspired on human hands. Replicating accurately such system might require 20 degrees of freedom and over 17.000 sensors, which makes it complicated to manufacture, control and maintain. However, principles such as tendon – phalanx mechanical transmission system inspired most of the models currently on the market, making them effective but also expensive, hard to customize and compatible with few robots. Yale Open Hand Project (YOH) from the Yale GRAB Lab at Yale University, provided different open source models for research in hardware and software systems [2] following the principle mentioned. The YOH T42 model available on FAST-Lab, implemented on the project “Design and implementation of an under-actuated gripper for picking coins” by L. Amezua [3], demonstrated the ability of picking special objects with simple force control methods.

Even the control systems becomes simpler and the mechanical system adapts to the object grasped, tendon – phalanx configurations might require specific execution of commands to achieve proper grasping of target objects.

In this document, the author approaches the need of discover a specific sequence of commands for the YOH T42 under-actuated gripper, from a machine learning point of view, formulating states and applying a Q-learning algorithm, to discover the shortest series of commands for picking up effectively a coin from a flat surface.

# 2. OBJECTIVES AND SCOPE

## 2.1 General objective

Formulate, implement and analyze a system to optimize the grasping methods for an under-actuated gripper by reinforcement learning.

## 2.2 Specific objectives

- Improve the installation and control of the YOH T42 available on robot cell A at FAST-Lab
- Formulate all the agents and requirements to optimize the grasping methods for the YOH T42 by reinforcement learning
- Install and start up all interconnected devices required to provide all the data necessary for a Q-learning algorithm
- Test and analyze the benefits from the solution implemented

## 2.3 Scope

Section 2.1 describes the scope of this project, following the proposal presented and accepted by Professor Jose Martinez on 31/05/2019.

The application includes the implementation using Python 3 programing language, executed form a Raspberry pi 3 B board, commanding an ABB IRB-140 Robot 1 on cell A at FAST-Lab, which contains a control script implemented on RAPID language. The script also commands an NI 1774C smart camera installed and programmed using NI Vision Builder 2015. The formulation of the Q-Learning algorithm uses as target object a white coin of 30 mm diameter for demonstrative purposes. Any other component different from the aforementioned, such as user interface, or different machine learning algorithm, will not be included on this work since is not part of the objectives.

# 3. SOLUTION

The solution goes through four topics based on the specific objectives.

## 3.1 Improvement of Yale Open Hand T42

This section presents the activities towards the improvement of the gripper matter of optimization in this project.

The existent YOH T42 model implemented on FAST-Lab was provided by the project “Design and implementation of an uder-actuated gripper for picking coins” by L. Amezua. Its installation included a controller Arbotix-M with built in interface dedicated to each servomotor separately.

The system receive improvements on its mechanical components by removing the Arbotix-M controller from the gripper structure and replacing the fastening method for bolts. A Raspberry Pi 3 B installed on the second arm of the robotic manipulator replaced the former Arbotix-M controller. The new controller installation took place beside the IO box, which moved 5 centimeters to give enough room for the new box. The connection of the servomotors changed to daisy-chain mode.

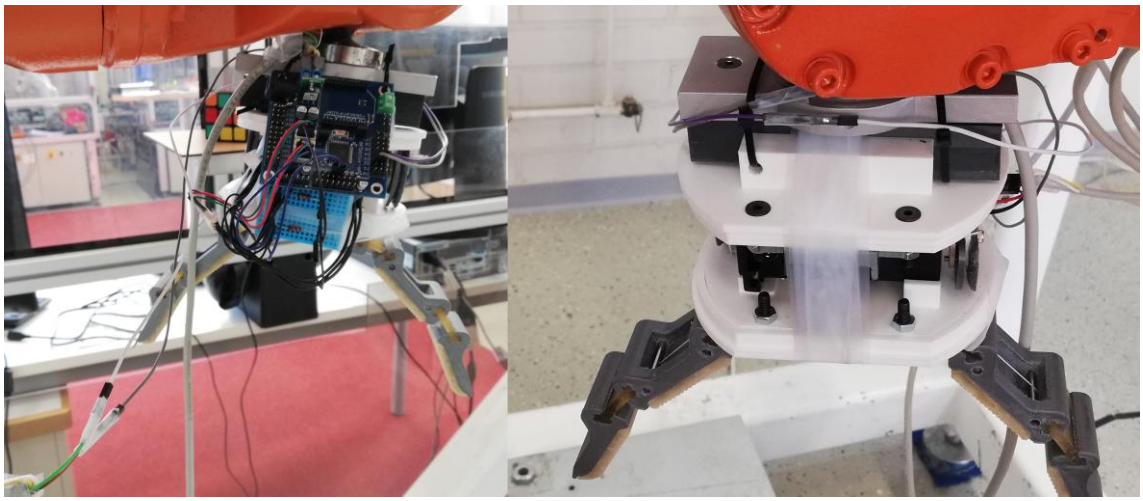

***Figure 1*** *YOH T42 former installation*

The power supply remained as derivation from 24Vdc pins on the existing IO box, but a 12Vdc regulator LM7812 Integrated Circuit (IC) equipped with heatsink replaced the adjustable power supply installed out of the IO box. Separately, a new adjustable buck converter module based on LM2596M IC supplies 5Vdc with more than 2A capacity for the Raspberry Pi 3 B controller.

Two Dynamixel MX-28 are the servomotors available on the YOH T42, both with TTL half-duplex control interface. The Raspberry Pi 3 B as new controller has full duplex serial port, thus it required a serial conversion. A buffer driver (74LS241 IC) carried out the conversion, controlled by a direction flag indicator (digital signal) to let pass through data in one direction at the time and address it to the correct port on the controller. The circuit requires an additional pull-up resistor as well. The implementation of the connections used the GPIO port (including serial TX-RX) from the controller. The controller box had enough room to contain the buffer driver circuit too [4].

The manual from Robotis (manufacturer) contains all the functions and commands required to transmit by serial port. However, for this implementation a the author used a simplified approach, proposed on the library published on GitHub by T. Hersan and R. Ajna on 2014, for the artistic project *memememe* designed for Dynamixel AX-12 servo-motors, which uses the same communication protocol than MX-28 model (Dynamixel 1.0). The API provided is intuitive and easy to use, but does not include all the functions, such adjusting the gains for PID control. For that reason, was necessary to improve the API including functions to update control parameters on the motors, required for this implementation [5].

The new control system operated properly on the initial tests, but the position control presented often error or steady state. By default, the servos use P control with gain $K_p$ of four. Then, it was necessary to apply the Ziegler-Nichols method for classic PID tuning following the Equations 1 to 3, where $K_p$ represents the proportional gain as 0.6 of the gain necessary to make the system oscillate ($K_u$). Then $K_i$ as the integral gain includes $P_u$ on the calculation, which is the period of oscillation at $K_u$. Finally, the differential gain ($K_d$) can be calculated using $K_u$ and $P_u$.

$$K_p = 0.6K_u \tag{1}$$

$$K_i = 2\,K_p/P_u \tag{2}$$

$$K_d = K_p P_u/8 \tag{3}$$

Experimentally, the values found for $K_u$ and $P_u$ were 128 and 1 second respectively. Equations 5, 7 and 9 present the values of all gains and the conversion to controller units on the range 0 to 254. For $K_d$ the value was adjusted to 0.8 instead of 2 to fit the value range predefined for the manufacturer (0-254).

$$K_p = 0.6 * 27 = 16 \tag{4}$$

$$P_{gain} = 16 * 8 = 128 \quad (5)$$

$$K_i = 2 * 16/1 = 32 \quad (6)$$

$$I_{gain} = 32 * 2048/1000 = 66 \quad (7)$$

$$K_d = 16/8 = 2 \quad (8)$$

$$D_{gain} = 2 * 1000/4 = 500 \rightarrow 200 \sim K_d = 0.8 \quad (9)$$

Since the API on the Ax12 library on python does not include the functions to modify PID gains, which are stored on each servo's RAM, it was necessary to create 3 new functions called *setPgain(id, value)*, *setIgain(id, value)* and *setDgain(id, value)*. By updating the gain values, the gripper responded fast and precisely enough to reach any position on the grasping range with ±1 position count tolerance.


***Figure 2*** *New installation for YOH T42*

To detect the grasping range, it was necessary to review the poses achieved by the control method used previously and validate the load applied by each motor to grasp a coin. Before, the load was monitored by reading the electrical current through a sensing circuit, but now, the *readLoad(id)* function will provide this information without using any additional circuit. The range found is presented on Table 1 for each finger, it was found a gap of 1200 position counts between poses at safe load values under 40% of the maximum value (1023 in load counts).

***Table 1.*** *Actuator parameters for preset poses*

| Pose | Finger 1 | | Finger 2 | |
|---|---|---|---|---|
| | **Position counts** | **Load counts** | **Position counts** | **Load counts** |
| Open | 2000 | 0 | 1850 | 0 |
| Close | 800 | 140 | 650 | 380 |

## 3.2 Agents and requirements for reinforcement learning

Reinforcement learning is an area of machine learning focused on decision-making, based in obtaining the best reward possible by executing a series of actions in a predefined environment. The correct operation of a reinforcement learning system depends mainly on convergence of the acting agent activities towards a high reward state. Besides, the effective operation of all other specific agents makes possible to evaluate the reward of random actions and memorize the best decisions.

The algorithm selected, memorize a “Q score” or quality score in a status versus actions array (Q table). Equation 10 provides the quality score for given state, action, reward and maximum Q score among actions of next state. The learning rate (α) defines how much the new information overrides the old data and the discount factor (γ) determines how much the future state impact on the current reward, based on the highest future Q score. The algorithm selects an action, by using the Equation 11, which generates a random number (x) and compare it to a constant threshold (ε). If x is greater than ε, it generates a random action value, otherwise, verifies the logged values and selects the action with maximum Q score among the actions of the current state.

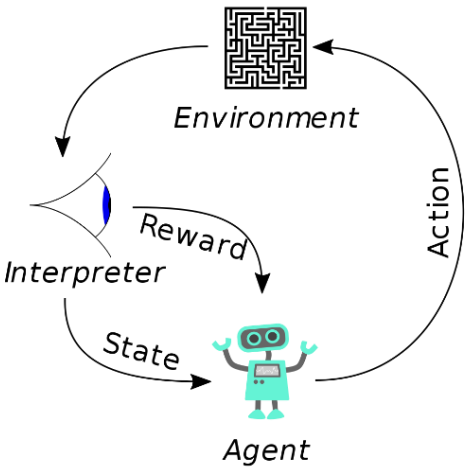


***Figure 3*** *Reinforcement learning sketch*[1]

[1] https://en.wikipedia.org/wiki/Reinforcement_learning#/media/File:Reinforcement_learning_diagram.svg

$$Q(state, action) \leftarrow (1-\alpha) * Q(state, action) + \alpha\left(reward + \gamma * \max_{Q\ score} Q(next\ state)\right) \quad (10)$$

$$next\ action \leftarrow \begin{Bmatrix} random & for\ x < \varepsilon \\ argmax\left(\max_{Q\ score} Q(state)\right) & for\ x \geq \varepsilon \end{Bmatrix} \quad (11)$$

This algorithm, also known as Q-Learning, is a concept of artificial intelligence, different from supervised and unsupervised learning; it does not require a physical/behavioral model for the acting agent or environment. This one focus essentially on selecting an optimal action for any given Finite Markov Decision Process (random decision). The action selection and computing might cease if the system achieves the goal state or if it reaches the maximum number of trials, then an episode ends and the environment should reset to a new initial state for a new episode. The iteration of episodes will continue until the system reaches the maximum number of episodes, accumulating all the learning in the Q table. Finally, the iterative algorithm obtains a list of status versus best action, serving as guide for the agent to take effective decisions, based on the current state.

Figure 4 illustrates the agents required for this case of study.

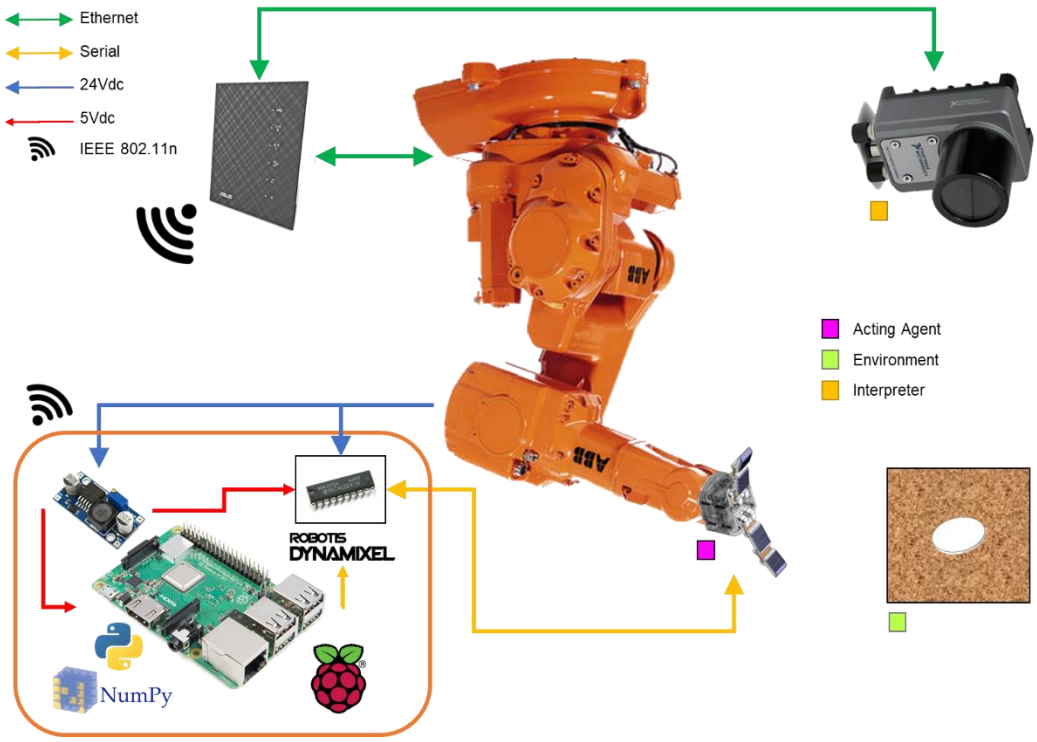


***Figure 4*** *Solution structure*

- Acting agent: Is the agent in charge of executing actions. In this case is the Yale Open Hand T41. The acting agent should have predefined discrete states. Since the grasping range detected is equal to 1200 position counts for each finger, it was decided by the author to make discrete decrements of 50 position counts every action, then it was possible to identify 3 possible actions, distributing the decrements as presented on Table 2.

***Table 2*** *Changes on actuator parameters due actions*

| **Action code** | **Decrements** | |
|---|---|---|
| | **Finger 1 (Position counts)** | **Finger 2 (Position counts)** |
| 0 | 50 | 0 |
| 1 | 0 | 50 |
| 2 | 50 | 50 |

- Environment: The environment is a flat smooth surface of 140 x 120 mm, holding a white coin of 30 mm diameter, 3 mm height. The gripper will operate in open loop mode, and the position states will be logged and executed virtually since its fingers nor the coin have any sensor able to confirm effective grasping, in addition, the position control suddenly can have a slight steady state error. In other words, the position sensor on the servos will not provide the position, but the position will take ideal values stored internally on the control code. An ABB IRB140 positions the acting agent around the environment to execute the actions.
- Interpreter: The interpreter is present in two agents, a NI1774C smart camera and a python function. The camera has two functions; it can detect the presence of the coin in the environment and provide information about it, such as location coordinate on the plain XY and its diameter. Then, the python function called *step* encodes the position counts of the acting agent into a discrete state code. As explained on the acting agent description, 25 possible states for each finger (multiples of 50 up to 1200, including 0 as state), make a product of 625 possible state codes (25x25). The same function can analyze the current state and provide the status code of the next state, by applying one action. Table 3 presents a list of reward rules for the coin and fingers of acting agent, considered together as the environment.

***Table 3*** *Reward table rules*

| Finger 1 or 2 | Coin | Reward | Description |
|---|---|---|---|
| x | X | -1 | Every action returns a reward of -1 |
| >1200 | X | -10 | Actions exceeding the limit have penalty of -10 and the position remains on 1200 for the finger intending to exceed 1200 |
| 1200 | Yes | -30 | Fingers closed and coin not extracted, gives penalty of 30 |
| 1200 | No | 20 | Fingers closed and coin correctly extracted gives award of 20 |
| <=1200 | No | 25 | Coin extracted without need to close completely, gives award of 25 |

## 3.3 Installation and start up devices for running the Q-learning algorithm

As presented on Figure 4, the green lines represent wired Ethernet links, while the wave symbol represent wireless IEEE 802.11n links. An access point is in charge of bridging the wired with wireless Ethernet connections and provide TCP/IP connectivity to the robot controller, the smart camera and the Raspberry Pi 3 B. Table 4 presents the list of IP addresses for all the devices.

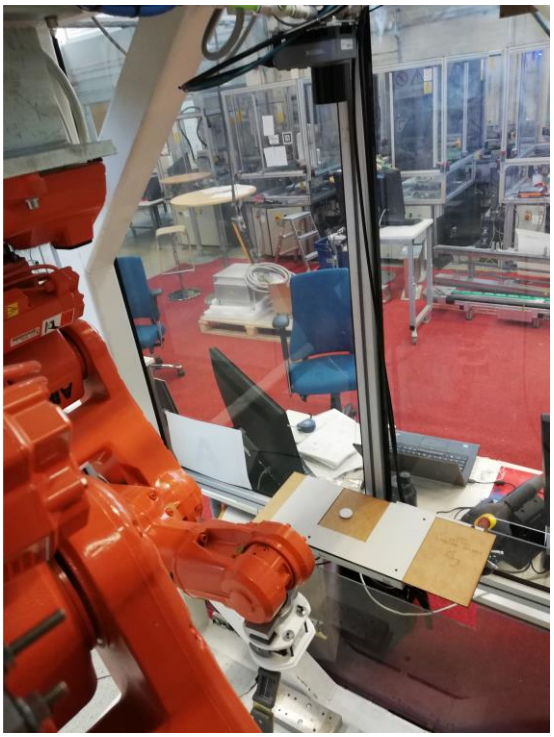

***Figure 5*** *System implementation on FAST-Lab robot cell A*

***Table 4*** *IP address list*

| Device | IP address | Connection method |
|---|---|---|
| NI 1774C Smart Camera | 130.230.141.10 | Wired |
| Raspberry Pi 3 B | 130.230.141.244 | Wireless |
| ABB Robot controller | 130.230.141.123 | Wired |

Figure 7 presents the sequence of activities on the training algorithm. The smart camera behavior corresponds to an UDP streaming, instead of TCP common requests (callback). Due several synchronization problems during laboratory tests of triggering visual inspections by a TCP/IP messages, using the socket library on python, UDP periodic streaming (2 seconds) was the simpler option to keep the system running. All communications with the YOH T42 use the serial port built in on the Raspberry Pi 3 B, then it was not needed any additional Ethernet connection.

The routine running on the smart camera has three states (*Read*, *Ok* and *Nothing*). *Read*, executes the visual inspection using the parameters on ***Table 5***, calibrated from pixels to millimeters and the same reference plane as engraved on the real workspace. Then it performs geometric matching, to find a white circle with more than 500 $mm^2$ surface area. If there is at least one circle, the inspection is considered as passed, activating the transition to status *Ok* which drops an UDP telegram to the controller containing the amount of coins detected, X coordinate, Y coordinate and diameter of the coin. If the

amount of white circles detected is equal to zero, the inspection is considered failed, and the routine proceeds to send an UDP telegram to the controller with 4 fields in zero. After *Ok* or *Nothing* states, the inspection restarts in 2 seconds, to avoid communication issues, due port timeouts.

***Table 5*** *Smart camera parameters*

| Step | Parameter | Value |
|---|---|---|
| Acquire image | Exposure time | 10-13 ms |
| | Gain | 100 |
| Geometric Matching | Threshold (lower value) | 200 |
| | Minimum object size | 10 $mm^2$ |
| | Area filter | 500 – 800 $mm^2$ |
| | Minimum number of objects to pass | 1 |

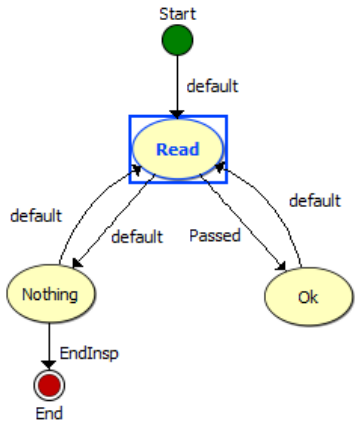


***Figure 6*** *Smart camera routine – State diagram*

Every time the ABB robot receive an action order by Web Sockets it contains offset coordinates in millimeters (X, Y, Z) and an action code, which defines if the robot is going to the workspace or if it is going away. The robot program reaching the workspace locates the joints attaching the fingers to the base, at 65 mm above the workspace surface, in perpendicular orientation (α=0°, h=65mm) as suggested on [6].

Finally, the script saves the training results on a Numpy file (results.npy). An additional python script can retrieve and execute the sequence of movements obtained from training. For demonstrative purposes, the execution of the learning file includes coin detection and robot movement sequence.

## 3.4 Testing

The application was set to train using the setup described on Table 6. Training process lasted 3 hours and 10 minutes approximately. The training algorithm required the robot to pick a coin 500 times in less than 50 movements.

***Table 6*** *Hyperparamenters for testing*

| Parameter | Value |
|---|---|
| Epochs | 50 |
| Episodes | 500 |
| Learning rate (α) | 0.3 |
| Discount factor (γ) | 0.7 |
| Exploration rate (ε) | 0.3 |

After completing the test execution, the training script stores a list of actions for every state, ordered by state code (0-624) in a numpy file (‘results.npy’), by extracting the argument of the maximum Q score for every row in the Q table. Another Python script can interpret this file by iteratively reading the current gripper position and applying the action on the list extracted from the results numpy file, until the gripper reaches close position.

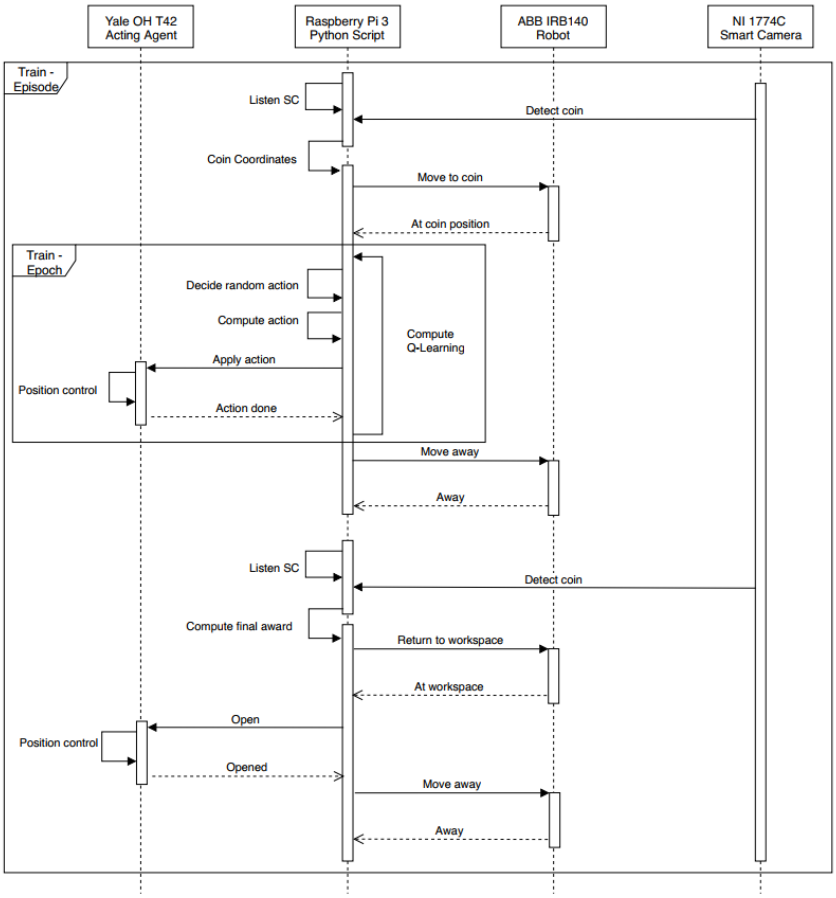


***Figure 7*** *Sequence diagram for training*

# 4. RESULTS

Executing the results from the Q learning demonstrated 100% accuracy on 10 attempts of picking the coin from different places of the workspace. The grasping time changed from 24 to 9 seconds and the amount of movements from 2400 to 42 as presented in Figure 8, from the former force monitoring to the proposed AI Q-learning method.

Time reduction of 266.6% verifies the optimization in operative time, based on longer displacement steps and simultaneous movements of both fingers at certain stages of the process, learned automatically by the algorithm. Besides, a notorious reduction of movement requests of 5714.2% due the reward policy and the selection of optimal simultaneous movements ensures optimal use of the control communication channel (serial half duplex).

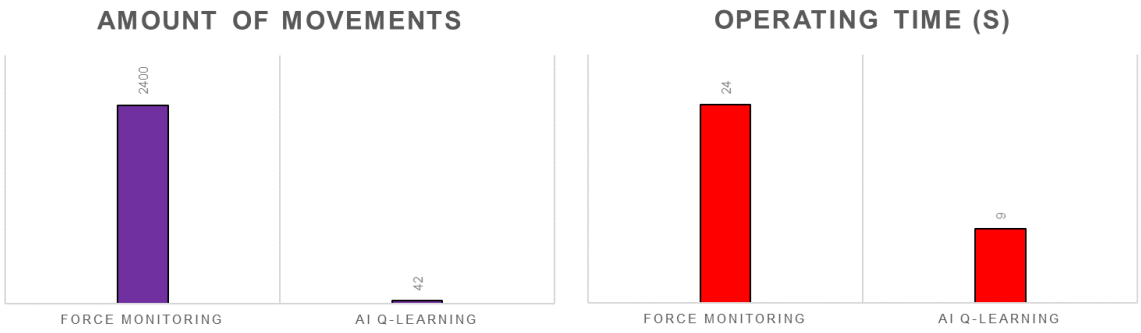


***Figure 8*** *Time and movements compared between control methods*

Figure 9 presents the behavior of the cumulative reward over the number of episode. This plot does not have a clear trend, since the exploration factor is 0.3, higher than the usual 0.1 value, besides, the training tested ran only for 500 episodes. For other examples of Q-learning for software agents, trends might appear after 4000 or 5000 episodes [7], [8], which can be executed in few seconds, in contrast with real scenarios using robots, where repeating 5000 episodes precisely, requires high and reliable resources (power, lighting, etc.) and long journeys of supervision.

Since the action selection on every epoch is completely random, tuning the hyper parameters (α, γ and ε) or reformulate the reward policies might have an unexpected impact on the reward evolution plot that might reveal trends (convergence) only after thousands of trials. However, even there is not trend on the graph, the learning results tested, presented acceptable results, by allowing the gripper to pick the coin repeatedly.

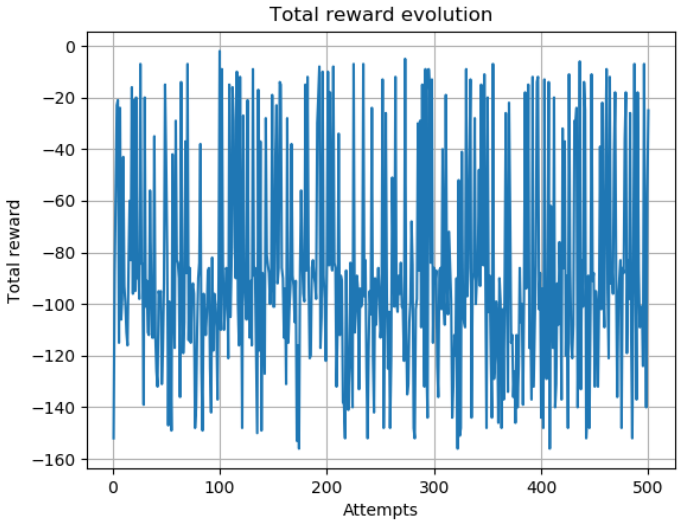


***Figure 9*** *Reward evolution on training*

# 5. CONCLUSIONS

This document presented the formulation, implementation and analysis of a system to optimize the grasping methods for an under-actuated gripper YOH T42 by using artificial intelligence through the Q-learning algorithm for reinforcement learning.

Implementing a Q-Learning system might help to simplify analysis and optimization of control methods for under-actuated grippers with higher complexity, since this method is based on rewards form real scenario tests, instead of physical models.

Even Q-learning is considered as model-free method, several mechanical constrains might impact on its efficacy, since external variables can influence unexpected perturbations on the system, that might create unknown states or mistakes on the reward evaluation. This algorithm depends critically on precision of the environment and proper state analysis.

Future works such as improve the control of environment conditions (light, vibrations, workspace material), include an auxiliary robot in charge of supply coins for training and enhance the training including the coin location on the environment, might lead to deeper research into enhancement of Q-learning variables and methods.

# 6. ANNEXES

## 6.1 Training script

```
  #*****************************************************
  #Tampere University
  #Futre Automation Systems and Technologies Laboratory
  #FAST-LAB
  #
  #Q-Learning applied to underactuated gripper control
  #
  #This script orchestrate an ABB IRB140 robot, a NI1774C smart camera
  #aiming to find a sequence of movements to pick a coin using a
  #2 finger underactuated gripper (Openhand T42 - Yale)
  #
  #The sequence is found by discretizing incremental movements of each
finger
  #and implement Q-Learning
  #
  #Results are stored as results.npy (numpy file) and can be read by
the
  #actuating script
  #
  #V0:21.08.2019  - RB
  #*****************************************************
  #Import libraries
  import random
  import socket
  import numpy as np
  import matplotlib
  import matplotlib.pyplot as plt
  from itertools import product
  from ax12 import Ax12
  import time


  # Hyperparameters
  alpha = 0.3
  gamma = 0.7
  epsilon = 0.3
  epomax = 50
  epimax = 500

  #Environment
  q_table = np.zeros([625, 3])
  #625 states. 1200 position units range in increments of 50 = 25. 25
positions for each finger: 25*25=625 states
  #3 possible actions, move -50 on each single finger or both
  Stateval = np.arange(0,Ran+50,50)
  s_table = list(product(Stateval, repeat=2))
```

```
  #Table of states with all possible states of both fingers
  histrew=np.array([0])
  #Total reward variable
  totreward=0
  # For controlling the gripper, set P, I, D Gains and positions
  M=Ax12()
  M.setPgain(1,128)
  time.sleep(0.2)
  M.setPgain(2,128)
  time.sleep(0.2)
  M.setIgain(1,66)
  time.sleep(0.2)
  M.setIgain(2,66)
  time.sleep(0.2)
  M.setDgain(1,200)
  time.sleep(0.2)
  M.setDgain(2,200)
  time.sleep(0.2)
  M1O=2000
  M2O=1850
  M1C=800
  M2C=600
  Ran=M1O-M1C

  #Define functions to operate robot, camera and gripper
  #Script to capture image using NI1774C and a TCP/IP socket server.
  #Cam streams as UDP messages to RPi with ammount of coins, coordinates
and
  #diameter of coin. Function returns data as list
  def acqimg():
    Rport = 12345
    data = ''
    for a in range(10):
      try:
        d = socket.socket()
        d.bind(('0.0.0.0', Rport))
        d.listen(5)
        x, addr = d.accept()
        while True:
          data = x.recv(1024)
          if data !='':
            #format of data: int list [ammount, x, y, diameter]
            data = data.split(',')
            data = list(map(int, data))
            break
        d.close()
      except socket.error, e:
          time.sleep(1)
          continue
      else:
          return data
          break

  #Function to call the IRB140 ABB robot using TCP/IP sockets
```

```
  #coinL are coordinates [X,Y,Z] of the target. process is 1 when the
robot
  #aproaches the coin and 2 when moves away of it
  def callrob(coinL, process):
    coindata                                                    =
'['+str(coinL[0])+','+str(coinL[1])+','+str(coinL[2])+']'
    ABBhost = '130.230.141.123'
    ABBport = 1025
    c = socket.socket()
    c.connect((ABBhost, ABBport))
    c.sendall(bytes(process))
    ans = c.recv(1024)
    c.sendall(bytes(coindata))
    ans = c.recv(1024)
    if process == 1:
      print ('Going to: '+repr(ans))
    if process == 2:
      print ('Moving Robot away')
    while True:
      ans = c.recv(1024)
      if ans == str('Done'):
        c.close()
        print ('Robot in place')
        break
      else:
        c.sendall('Ok')
  #Function to move the gripper to positions M1 and M2
  def MoveG(M1,M2):
      try:
          M = Ax12()
          mov1 = 0
          mov2 = 0
          M.move(1, M1)
          M.move(2, M2)
          mov1 = M.readMovingStatus(1)
          time.sleep(0.05)
          mov2 = M.readMovingStatus(2)
          time.sleep(0.05)
          while mov1==1 or mov2==1:
              mov1 = M.readMovingStatus(1)
              time.sleep(0.05)
              mov2 = M.readMovingStatus(2)
              time.sleep(0.05)
      except:
          mov1=1
          mov2=1
  #Function to return the coin to the work area
  def retCoin():
      Dropc=[50,30,0]
      callrob(Dropc,1)
      MoveG(M1O,M2O)
      callrob(Dest,2)

  #Define functions for QLearning
```

```
#Function to estimulate and action, requires Current state (Cstate),
#action (action) and a Table of all possible states (Stable)
def step(Cstate, action, Stable):
  #Reads the current state
  st=[Stable[Cstate][0],Stable[Cstate][1]]
  #Applies the action requested
  if action == 0:
    st = [st[0],st[1]+50]
  if action == 1:
    st = [st[0]+50,st[1]]
  if action == 2:
    st = [st[0]+50,st[1]+50]
  if action not in range(0,3):
    st=[0,0]
  #Every action gives reward of -1
  reward=-1
  #If the closing point is reached, process is finnished
  if st[0]>=Ran and st[1]>=Ran:
    done = True
    NState=Stable.index((Ran,Ran))
    return NState, reward, st, done
  else:
    done = False
  if not st[0] in range(0,1201) or not st[1] in range(0,1201):
        reward =-5
  #Evaluates if it reached the limit positions of the servomotors
  if st[0]>Ran:
    st[0]=Ran
  if st[1]>Ran:
    st[1]=Ran
  if st[0]<0:
    st[0]=0
  if st[1]<0:
    st[1]=0
  NState=Stable.index((st[0],st[1]))
  return NState, reward, st, done


#Training loop
for i in range(1, epimax):
  #Detect position of the coin and store it in coinL
  Dest = [0,0,0]
  coin = acqimg()
  coinL= coin[1:3]
  coinL.append(0)
  totreward=0
  #If any coin is found send the robot to it and start episode
  if coin[0] > 0:
    #Initialize episode. Initialize variables, open Gripper
    epok = False
    MoveG(M1O,M2O)
    CState=0
    callrob(coinL,1)
```

```
      epochs, penalties, reward, = 0, 0, 0
      done = False
      epok = False
      #While is not on closed position or has not reached max amount of
epochs
      while not done and not epok:
        #randomly select action
        if random.uniform(0, 1) < epsilon:
          action = np.random.randint(0,3) # out of 3 possible actions
        else:
          action = np.argmax(q_table[CState]) # Exploit learned values
        #Execute the action selected
        next_state, reward, counts, done = step(CState, action, s_ta-
ble)
        MoveG(M1O-counts[0],M2O-counts[1])
        #If the gripper is completely closed, check if it picked the
coin
        if done or epochs >= epomax:
          epok=True
          #Take the robot away
          callrob(Dest,2)
          #Look for coin
          coinp = acqimg()
          #If it picked the coin
          if coinp[0] == 0:
            #Gives best reward 20
            reward = 30
            #Drop the coin back to keep training
            retCoin()
          else:
              reward = -50
        #Get current probability value
        old_value = q_table[CState, action]
        #Find value of maximum probability for next state
        next_max = np.max(q_table[next_state])
        #Calculate new probability for current state, selected action
        new_value = ((1 - alpha) * old_value) + (alpha * (reward +
(gamma * next_max)))
        q_table[CState, action] = new_value
        #Go to next state, advance 1 epoch
        CState = next_state
        epochs += 1
        totreward=totreward+reward
        print('Epoch: '+str(epochs), ' Total reward: '+str(totreward))
        #If max epochs reached, remove the robot and check if it picked
the coin
      print("Episode: "+str(i))
      histrew = np.append(histrew,totreward)
    else:
      try:
          input("No coin detected, please insert coin and press Enter")
      except SyntaxError:
          i+=1
          pass
```

```
  #Save results
  FinalRes=np.argmax(q_table, axis=1)
  np.save('results.npy', FinalRes)
  np.save('Qres.npy', q_table)
  #Present reward vs attempts plot
  atmp= np.arange(0,epimax,1)
  histrew=np.delete(histrew,0)
  atmp=np.delete(atmp,0)
  fig, ax = plt.subplots()
  ax.plot(atmp,histrew)
  ax.set(xlabel='Attempts', ylabel='Total reward', title='Total reward
evolution')
  ax.grid()
  fig.savefig('REvsAt.png')
  print("Training finished.\n")
```

## 6.2 Script for reading results

This script includes visual detection of the coin and robot call.

```
  import numpy as np
  from ax12 import Ax12
  from itertools import product
  import time
  import socket
  #Load data from file
  a = np.load('results.npy')
  #Initialize variables and state table
  Dest = [0,0,0]
  st=[0,0]
  M1O=2000
  M2O=1850
  Stateval = np.arange(0,1250,50)
  s_table = list(product(Stateval, repeat=2))

  #Set PID gains on servos
  M=Ax12()
  M.setPgain(1,128)
  time.sleep(0.2)
  M.setPgain(2,128)
  time.sleep(0.2)
  M.setIgain(1,66)
  time.sleep(0.2)
  M.setIgain(2,66)
  time.sleep(0.2)
  M.setDgain(1,200)
  time.sleep(0.2)
  M.setDgain(2,200)
  time.sleep(0.2)
  #Coin detection
  def acqimg():
```

```
    Rport = 12345
    data = ''
    for a in range(10):
      try:
        d = socket.socket()
        d.bind(('0.0.0.0', Rport))
        d.listen(5)
        x, addr = d.accept()
        while True:
          data = x.recv(1024)
          if data !='':
            #format of data: int list [ammount, x, y, diameter]
            data = data.split(',')
            data = list(map(int, data))
            break
        d.close()
      except socket.error, e:
          time.sleep(1)
          continue
      else:
          return data
          break
  #Robot call
  def callrob(coinL, process):
    coindata                                                    =
'['+str(coinL[0])+','+str(coinL[1])+','+str(coinL[2])+']'
    ABBhost = '130.230.141.123'
    ABBport = 1025
    c = socket.socket()
    c.connect((ABBhost, ABBport))
    c.sendall(bytes(process))
    ans = c.recv(1024)
    c.sendall(bytes(coindata))
    ans = c.recv(1024)
    if process == 1:
      print ('Going to: '+repr(ans))
    if process == 2:
      print ('Moving Robot away')
    while True:
      ans = c.recv(1024)
      if ans == str('Done'):
        c.close()
        print ('Robot in place')
        break
      else:
        c.sendall('Ok')
  #Move the gripper
  def MoveG(M1,M2):
      try:
          M = Ax12()
          mov1 = 0
          mov2 = 0
          M.move(1, M1)
          M.move(2, M2)
```

```
        mov1 = M.readMovingStatus(1)
        time.sleep(0.05)
        mov2 = M.readMovingStatus(2)
        time.sleep(0.05)
        while mov1==1 or mov2==1:
            mov1 = M.readMovingStatus(1)
            time.sleep(0.05)
            mov2 = M.readMovingStatus(2)
            time.sleep(0.05)
    except:
        mov1=1
        mov2=1

#To read the sequence on file
def CloseGripper(st):
    NState=s_table.index((st[0],st[1]))
    while st[0]<1200 or st[1]<1200:
        if a[NState] == 0:
            st = [st[0],st[1]+50]
        if a[NState] == 1:
            st = [st[0]+50,st[1]]
        if a[NState] == 2:
            st = [st[0]+50,st[1]+50]
        if a[NState] not in range(0,3):
            st=[0,0]
        MoveG(M1O-st[0],M2O-st[1])
        #time.sleep(0.1)
        print(st)
        NState=s_table.index((st[0],st[1]))


#Open gripper
MoveG(M1O,M2O)
#Detect coin in environment
coin = acqimg()
coinL= coin[1:3]
coinL.append(0)
#If there is any coin
if coin[0] > 0:
    #Go to coin position
    callrob(coinL,1)
    #Close the gripper reading the file
    CloseGripper(st)
    #Go away
    callrob(Dest,2)
    #Drop the coin
    MoveG(M1O,M2O)
else:
    print('No coin detected')
```

# REFERENCES


[1] L. Birglen, T. Laliberté, and C. M. Gosselin, *Underactuated Robotic Hands*. Berlin Heidelberg: Springer-Verlag, 2008.

[2] “Yale OpenHand Project.” [Online]. Available: https://www.eng.yale.edu/grablab/openhand/#contact. [Accessed: 26-Aug-2019].

[3] L. Amezua, “Design and implementation of an under-actuated gripper for picking coins.” 2019.

[4] thiagohersanFollow, “How to Drive Dynamixel AX-12A Servos (with a RaspberryPi),” *Instructables*. [Online]. Available: https://www.instructables.com/id/How-to-drive-Dynamixel-AX-12A-servos-with-a-Raspbe/. [Accessed: 02-Sep-2019].

[5] “thiagohersan/memememe,” *GitHub*. [Online]. Available: https://github.com/thiagohersan/memememe. [Accessed: 02-Sep-2019].

[6] L. U. Odhner, R. R. Ma, and A. M. Dollar, “Open-Loop Precision Grasping With Underactuated Hands Inspired by a Human Manipulation Strategy,” *IEEE Trans. Autom. Sci. Eng.*, vol. 10, no. 3, pp. 625–633, Jul. 2013.

[7] “Reinforcement Q-Learning from Scratch in Python with OpenAI Gym.” [Online]. Available: http://learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/. [Accessed: 02-Sep-2019].

[8] G. Hayes, “Getting Started with Reinforcement Learning and Open AI Gym,” *Medium*, 03-Aug-2019. [Online]. Available: https://towardsdatascience.com/getting-started-with-reinforcement-learning-and-open-ai-gym-c289aca874f. [Accessed: 02-Sep-2019].