Αναφορά Project-2- Learning

ΤΕΧΝΟΛΟΓΙΕΣ ΕΥΦΥΩΝ ΣΥΣΤΗΜΑΤΩΝ ΚΑΙ ΡΟΜΠΟΤΙΚΗΣ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ 2021-2022

Ιωάννης Χαραλάμπους (1059685)

1. Περιγραφή Υλοποίησης

Στην παρούσα εργασία δημιουργήθηκαν τα αρχεία :

pendulum_td3.py:

Σε αυτό το αρχείο λύνεται το πρόβλημα του pendulum swing-up πρόβλημα με τον off-policy αλγόριθμο td3

pendulum_ppo.py: Σε αυτό το αρχείο λύνεται το πρόβλημα του pendulum swing-up πρόβλημα με τον on-policy αλγόριθμο td3

graphs_pendulum.py: Αυτο το αρχείο δημιουργεί τα γραφήματα για την λύση του swing-up προβλήματος με την χρήση td3 και ppo αλγορίθμων.

pendulum_td3.csv: Περιέχει τα δεδομένα για τα γραφήματα

pendulum_ppo.csv: Περιέχει τα δεδομένα για τα γραφήματα

Iiwa_td3.py: Σε αυτό το αρχείο το ρομπότ iiwa μαθαίνει να μεταφέρεται από μία αρχική θέση σε μία καινούργια με τον off-policy αλγόριθμο td3.

graphs_iiwa.py: Αυτο το αρχείο δημιουργεί τα γραφήματα για την λύση του iiwa προβλήματος με την χρήση td3 αλγορίθμου.

Iiwa_td3.csv: Περιέχει τα δεδομένα για τα γραφήματα

2. Κώδικες Υλοποίησης

2.1 pendulum_td3

Αρχικά δημιουργώ την κλάση όπου δημιουργείται το περιβάλλον του pendulum και τίθεται σε θέση κοιτώντας προς τα κάτω.

```
#class that creates pendulum enviroment resets its position and gives every step of every episode
class Env:
    def __init__(self, simu=rd.RobotDARTSimu(0.05), robot=rd.Robot("pendulum.urdf"), graphics=rd.gui.Graphics(rd.gui.GraphicsConfiguration(1024, 768))):
########## Create simulator object ##########
        # time that each command runs for the robot
        self.dt = 0.05
        self.simu = simu

########## Load our new robot ##########
        self.robot =  robot
        self.simu.add_robot(robot)
        self.robot.set_actuator_types("torque")

########## Create Graphics ##########
# create graphics object with configuration and a window of 1024x768 resolution/size
        self.graphics = graphics
        self.simu.set_graphics(graphics)
        self.graphics.look_at([0., 3., 2.], [0., 0., 0.])

########## Fix robot to world frame ##########
        self.robot.fix_to_world()

########## Initial positions to allow falling ##########
        self.robot.set_positions([np.pi])

#this  function resets the starting position of the pendulum at the start of each episode
```

Ο χρόνος στον οποίο εκτελούνται οι εντολές στο pendulum είναι 0.05s και χρησιμοποιούνται κινητήρες ροπής torque.

```python
#this  function resets the starting position of the pendulum at the start of each episode
    def reset(self):

        #initial position
        self.robot.set_positions([np.pi])

         #commands are reseted to zero
        self.robot.set_commands([None])
        self.simu.step_world()

        #starting angle position of pendulum normalized to [-pi,pi] with 0 being the upright position
        theta=((np.pi + np.pi) % (2 * np.pi)) - np.pi
        #staring angle velocity of pendulum
        thdot=self.robot.velocities().item()

        #starting state of pendulum
        state= np.array([np.cos(theta), np.sin(theta), thdot], dtype=np.float32)

        #state is returned for the first state of the episode
        return state
```

Μέσα στην κλάσση υπάρχει η συνάρτηση reset ,η οποία επαναφέρει σε κάθε επεισόδιο το pendulum στην αρχική του θέση και μηδενίζει τις εντολές που δέχεται.

Ενώ ταυτόχρονα υπολογίζει στην αρχική αυτή θέση την γωνία theta κανονικοποιημένη στο πεδίο [-πι,πι] με 0 τιμή στην θέση όπου το pendulum κοιτάει προς τα πάνω στη λύση δηλαδή.

Υπολογίζει τη γωνιακή ταχύτητα που έχει στην αρχική θέση και εν τέλη επιστρέφει το state στην αρχική αυτή θέση για να οριστεί ως η πρώτη κατάσταση στην αρχή κάθε επεισοδίου.

```python
#function that gives every command to the pendulum and returns the new state ,reward for the action and value that checks if the pendulum is upright
    def step(self, action):
        terminal=False

        #action for the pendulum turned from tensor to array
        action = np.array([action], dtype=np.float32)
        #command for the robot
        self.robot.set_commands(action)
        self.simu.step_world()

        #new angle of pendulum normalized to [-pi,pi] with 0 being the upright position
        theta=((self.robot.positions()[0] + np.pi) % (2 * np.pi)) - np.pi
        thdot=self.robot.velocities().item()

         #reward for the action using angle, velocity and action
        reward = theta ** 2 + 0.1 * thdot**2 + 0.001 * (action.item()*2)

        #new state of pendulum
        newstate = np.array([np.cos(theta), np.sin(theta), thdot], dtype=np.float32)

        return newstate, -reward, terminal, {}
```

Μέσα στην κλάση περιβάλλοντος υπάρχει η συνάστηση step. Αυτή η συνάρτηση δέχεται ως όρισμα το action για κάθε step που υπολογίζει ο αλγόριθμος μας .

Το action είναι και η εντολή που δέχεται σε κάθε step το pendulum.

Αφού δεχτεί την εντολή το pendulum υπολογίζεται η νέα γωνία του κανονικοποιημένη στο [-πι,πι] και η νέα γωνιακή ταχύτητά του.

Με αυτές τις τιμες theta , thdot και action υπολογίζει σε κάθε step το reward για την κίνηση που έκανε . Όταν το pendulum είναι σε θέση προς τα πάνω έχει μηδενική ταχύτητα και μηδενική ροπή επιβράβευση έχει την μεγαλύτερη τιμή της 0 . Στην χειρότερη περίπτωσή της με κακή θέση ,μεγάλη ταχύτητα και ροπή η επιβράβευση είναι περίπου στο -16.

Αφού υπολογιστεί και το reward υπολογίζεται και η νέα κατάσταση του pendulum με τις νέες γωνίες και ταχύτητες.

Τα state δημιουργούν τον χώρο παρατήρησης του αργορίθμου για όλα τα steps του και περιλαμβάνει τρεις τιμές cos(theta) , sin(theta) , thdot.

```
#inialize pendulum and the algorithms parameters
env = Env()
observe_dim = 3
action_dim = 1
action_range = 2.5
max_episodes = 2000
max_steps = 200
noise_param = (0, 0.2)
noise_mode = "normal"
solved_reward = -300
solved_repeat = 5
```

Στη συνέχεια εκτός της κλάσης env έχουμε την αρχικοποίηση του περιβάλοντος του pendulum και της παραμέτρους για τις μεθόδους actor-critic ,για τον θόρυβο του td3 και την τιμή που πρέπει να φτάσει το reward 5 φορές για να σταματήσει ο αλγόριθμος με επιτυχημένη λύση, καθώς και τα μέγιστα steps και επεισόδια του αλγορίθμου.

```python
# model definition
#actor with 3 layers
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, action_range):
        super().__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_dim)
        self.action_range = action_range

    def forward(self, state):
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        a = t.tanh(self.fc3(a)) * self.action_range
        return a
```

Εδώ ορίζεται η κλάση του μοντέλου Actor για τον υπολογισμό του policy των πιθανών action που θα ακολουθήσει το περιβάλλον και έχει 3 layers με διαστάσεις (3,16) , (16,16) , (16,1).

```python
#critic with 3 layers
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()

        self.fc1 = nn.Linear(state_dim + action_dim, 32)
        self.fc2 = nn.Linear(32, 16)
        self.fc3 = nn.Linear(16, 1)

    def forward(self, state, action):
        state_action = t.cat([state, action], 1)
        q = t.relu(self.fc1(state_action))
        q = t.relu(self.fc2(q))
        q = self.fc3(q)
        return q
```

Εδώ ορίζεται η κλάση του μοντέλου Critic για τον υπολογισμό του value function του policy του Actor .Έχει 3 layers με διαστάσεις (4,32) , (32,16) , (16,1).

```
#main
if __name__ == "__main__":
#initialize actor critic and td3 algorithms
    actor = Actor(observe_dim, action_dim, action_range)
    actor_t = Actor(observe_dim, action_dim, action_range)
    critic = Critic(observe_dim, action_dim)
    critic_t = Critic(observe_dim, action_dim)
    critic2 = Critic(observe_dim, action_dim)
    critic2_t = Critic(observe_dim, action_dim)

    td3 = TD3(
        actor,
        actor_t,
        critic,
        critic_t,
        critic2,
        critic2_t,
        t.optim.Adam,
        nn.MSELoss(reduction="sum"),
    )
```

Στη συνέχεια αρχικοποιείται ο αλγόριθμος Td3 για βελτιστοποίηση του policy και μεγιστοποίηση του reward σε βάθος χρόνου.

Ο συγκεκριμένος αλγόριθμος είναι off-policy δηλάδή βελτιστοποιεί το value χρησιμοποιώντας διαφορετικό policy .Δηλαδή χρησιμοποιεί observations από προηγούμενα policies για να μάθει.

```
    #counter for counting all the episodes
    n_iter = 1
    # array of every episode counter
    all_iter=[]
    # array for every episodes expected return
    all_exp_returns = []
    # array for every episodes runtime
    all_times = []
    episode, step, reward_fulfilled = 0, 0, 0
    smoothed_total_reward = 0

#every episode of the algorithm
    while episode < max_episodes:
        episode += 1
        total_reward = 0
        rewards = []
        terminal = False
        step = 0

        #starting state of episode and reseted pendulum
        state = t.tensor(env.reset(), dtype=t.float32).view(1, observe_dim)
        tmp_observations = []

        #timer
        start = timeit.default_timer()
    #every step of episode
```

Στη συνέχεια αρχικοποιούνται μερικές μεταβλητές και αρχίζουν να εκτελούνται τα επεισόδια . Στην αρχή κάθε επεισοδείου γίνεται επαναφορά του περιβάλλοντος env.reset() και υπολογίζεται η αρχική κατάσταση.

Επίσης αρχίζει να μετρά ένας timer για να μετρήσουμε πόσο χρόνο χρειάστηκε να εκτελεστεί κάθε επεισόδειο.

```
#every step of episode
    while step <= max_steps:
        step += 1
        with t.no_grad():
            #save previous state as old state
            old_state = state
            #action of every step from td3 algorithm with noise
            action = td3.act_with_noise(
                {"state": old_state}, noise_param=noise_param, mode=noise_mode
                )
            act = np.array([action.item()], dtype=np.float32)

            #new state, reward , terminal after using the action on pendulum
            state, reward, terminal, _ = env.step(act)
            state = t.tensor(state, dtype=t.float32).view(1, observe_dim)

            #save every reward of episode
            rewards.append(reward)
            #return of episode
            total_reward += reward

            #observation of episode so that they are used on td3
            tmp_observations.append(
                {
                    "state": {"state": old_state},
                    "action": {"action": action},
                    "next_state": {"state": state},
                    "reward": reward,
                    "terminal": terminal or step == max_steps,
                }
            )
```

Σε κάθε step του επισοδείου εκτελείται ο αλγόριθμος td3 με θόρυβο και υπολογίζει το action. Το action αυτό μεταφέρεται στην συνάρτηση env.step() για να γίνει η εντολή στο pendulum και επιστρέφονται η κατάσταση , η επιβράβευση και το terminal μια εντολή για υπολογισμό της επιτυχίας του αλγορίθμου.

Στη συνέχεια αποθηκέυονται το reward, το return (total_reward) και τα observation του κάθε επεισοδείου δηλαδή το old_state,action,state,reward τα οποία χρησιμοποιούνται στον αλγόριθμο td3.

```
#stop timer
stop = timeit.default_timer()
time = stop-start


#store observations of episode
td3.store_episode(tmp_observations)

 # show reward
rewards = np.array(rewards)
#array of every episode number
all_iter.append(n_iter)
#array of expected return of every episode
all_exp_returns.append(np.mean(rewards.sum()))
#array of run times of every episode
all_times.append(time)

#smoothed episode return from previous episodes so that we have a smooth stop on the algorithm and not a random one
smoothed_total_reward = smoothed_total_reward * 0.5 + total_reward * 0.5
logger.info(f"Episode {episode} smoothed_total_reward={smoothed_total_reward:.2f}")
print("Iteration: {:6d}\tRuntime: {:6.4f}\tExpected return: {:6.2f}".format(n_iter, time, np.mean(rewards.sum())))
n_iter += 1
```

Μετά από κάθε επεισόδειο σταματάει ο timer και αποθηκεύονται τα observations .Επίσης αποθηκεύονται τα rewards του επεισοδίου όλα σε ένα πίνακα.

Το ο αριθμός επεισοδείου ,το expected return και ο χρόνος εκτέλεσης επεισοδίου αποθηκεύονται και αυτά για χρήση στο γράφημα.

Τυπώνεται το smoothed_total_reward ,το ομαλοποιημένο return του επεισοδείου το οποίο χρησιμοποιείται και στον έλεγχο για τον αν είχαμε επιτυχημένη λύση προκειμένου να έχουμε ομαλό σταμάτημα του αλγορίθμου και όχι κάποιο σε τυχαίο χρόνο.

Τυπώνεται επίσης ο αριθμός επεισοδίου, ο χρόνος εκτελεσής του και το expected return.

```
        # after 20 episodes td3 is update so that we have enough observations
        if episode > 20:
            for _ in range(step):
                td3.update()

            #5 times smoothed return is more than -300 algorithm is solved and it it print the  graphs of expected return and runtime
            if smoothed_total_reward > solved_reward:
                reward_fulfilled += 1
                if reward_fulfilled >= solved_repeat:
                    logger.info("Environment solved!")
                    df = pd.DataFrame({"episode" : all_iter, "expected_return" : all_exp_returns, "runtime" : all_times})
                    df.to_csv("pendulum_td3.csv", index=False)
                    exit(0)
            else:
                reward_fulfilled = 0

        if (env.simu.step_world()):
            break

########## Run simulation ##########
```

Τέλος ο αλγόριθμος td3 κάνει update τα observations του μετά από 20 επεισόδειο ώστε να είναι αρκετά.

Επίσης μετά από 20 επεισόδεια ελέγχεται αν έχουμε 5 φορές το επιθυμητό ομαλοποιημένο reward προκείμενου να έχουμε λύσει το πρόβλημα .

Σε περίπτωση που λύσουμε το πρόβλημα αποθηκεύονται όλες οι τιμές του n_iter expected return και runtime σε ένα αρχείο pendulum_td3.csv.

2.2 pendulum_ppo

Αρχικά δημιουργώ την κλάση όπου δημιουργείται το περιβάλλον του pendulum και τίθεται σε θέση κοιτώντας προς τα κάτω.

```
#class that creates pendulum enviroment resets its position and gives every step of every episode
class Env:
    def __init__(self, simu=rd.RobotDARTSimu(0.05), robot=rd.Robot("pendulum.urdf"), graphics=rd.gui.Graphics(rd.gui.GraphicsConfiguration(1024, 768))):
########## Create simulator object ##########
        # time that each command runs for the robot
        self.dt = 0.05
        self.simu = simu

########## Load our new robot ##########
        self.robot =  robot
        self.simu.add_robot(robot)
        self.robot.set_actuator_types("torque")

########## Create Graphics ##########
# create graphics object with configuration and a window of 1024x768 resolution/size
        self.graphics = graphics
        self.simu.set_graphics(graphics)
        self.graphics.look_at([0., 3., 2.], [0., 0., 0.])

########## Fix robot to world frame ##########
        self.robot.fix_to_world()

########## Initial positions to allow falling ##########
        self.robot.set_positions([np.pi])

#this  function resets the starting position of the pendulum at the start of each episode
```

Ο χρόνος στον οποίο εκτελούνται οι εντολές στο pendulum είναι 0.05s και χρησιμοποιούνται κινητήρες ροπής torque.

```
#this  function resets the starting position of the pendulum at the start of each episode
    def reset(self):

        #initial position
        self.robot.set_positions([np.pi])

         #commands are reseted to zero
        self.robot.set_commands([None])
        self.simu.step_world()

        #starting angle position of pendulum normalized to [-pi,pi] with 0 being the upright position
        theta=((np.pi + np.pi) % (2 * np.pi)) - np.pi
        #staring angle velocity of pendulum
        thdot=self.robot.velocities().item()

        #starting state of pendulum
        state= np.array([np.cos(theta), np.sin(theta), thdot], dtype=np.float32)

        #state is returned for the first state of the episode
        return state
```

Μέσα στην κλάσση υπάρχει η συνάρτηση reset ,η οποία επαναφέρει σε κάθε επεισόδιο το pendulum στην αρχική του θέση και μηδενίζει τις εντολές που δέχεται.

Ενώ ταυτόχρονα υπολογίζει στην αρχική αυτή θέση την γωνία theta κανονικοποιημένη στο πεδίο [-πι,πι] με 0 τιμή στην θέση όπου το pendulum κοιτάει προς τα πάνω στη λύση δηλαδή.

Υπολογίζει τη γωνιακή ταχύτητα που έχει στην αρχική θέση και εν τέλη επιστρέφει το state στην αρχική αυτή θέση για να οριστεί ως η πρώτη κατάσταση στην αρχή κάθε επεισοδίου.

```python
#function that gives every command to the pendulum and returns the new state ,reward for the action and value that checks if the pendulum is upright
    def step(self, action):
        terminal=False

        #action for the pendulum turned from tensor to array
        action = np.array([action], dtype=np.float32)
        #command for the robot
        self.robot.set_commands(action)
        self.simu.step_world()

        #new angle of pendulum normalized to [-pi,pi] with 0 being the upright position
        theta=((self.robot.positions()[0] + np.pi) % (2 * np.pi)) - np.pi
        thdot=self.robot.velocities().item()

         #reward for the action using angle, velocity and action
        reward = theta ** 2 + 0.1 * thdot**2 + 0.001 * (action.item()*2)

        #new state of pendulum
        newstate = np.array([np.cos(theta), np.sin(theta), thdot], dtype=np.float32)

        return newstate, -reward, terminal, {}
```

Μέσα στην κλάση περιβάλλοντος υπάρχει η συνάστηση step. Αυτή η συνάρτηση δέχεται ως όρισμα το action για κάθε step που υπολογίζει ο αλγόριθμος μας .

Το action είναι και η εντολή που δέχεται σε κάθε step το pendulum.

Αφού δεχτεί την εντολή το pendulum υπολογίζεται η νέα γωνία του κανονικοποιημένη στο [-πι,πι] και η νέα γωνιακή ταχύτητά του.

Με αυτές τις τιμες theta , thdot και action υπολογίζει σε κάθε step το reward για την κίνηση που έκανε . Όταν το pendulum είναι σε θέση προς τα πάνω έχει μηδενική ταχύτητα και μηδενική ροπή επιβράβευση έχει την μεγαλύτερη τιμή της 0 . Στην χειρότερη περίπτωσή της με κακή θέση ,μεγάλη ταχύτητα και ροπή η επιβράβευση είναι περίπου στο -16.

Αφού υπολογιστεί και το reward υπολογίζεται και η νέα κατάσταση του pendulum με τις νέες γωνίες και ταχύτητες.

Τα state δημιουργούν τον χώρο παρατήρησης του αργορίθμου για όλα τα steps του και περιλαμβάνει τρεις τιμές cos(theta) , sin(theta) , thdot.

```
#inialize pendulum and the algorithms parameters
env = Env()
observe_dim = 3
action_dim = 1
max_episodes = 2000
max_steps = 200
solved_reward = -300
solved_repeat = 5
```

Στη συνέχεια εκτός της κλάσης env έχουμε την αρχικοποίηση του περιβάλοντος του pendulum και της παραμέτρους για τις μεθόδους actor-critic και του ppo και την τιμή που πρέπει να φτάσει το reward 5 φορές για να σταματήσει ο αλγόριθμος με επιτυχημένη λύση, καθώς και τα μέγιστα steps και επεισόδια του αλγορίθμου.

```python
# model definition
#actor with 3 layers
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.mu_head = nn.Linear(16, action_dim)
        self.sigma_head = nn.Linear(16, action_dim)

#normal distribution is used for calculating continuous act , act_log_prob, act_entropy
    def forward(self, state, action=None):
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        mu = self.mu_head(a)
        sigma = softplus(self.sigma_head(a))
        dist = Normal(mu, sigma)
        act = (
             action if action is not None else dist.sample()
        )
        act_entropy = dist.entropy()
        act_log_prob = dist.log_prob(act)


        return act, act_log_prob, act_entropy
```

Εδώ ορίζεται η κλάση του μοντέλου Actor για τον υπολογισμό του policy των πιθανών action που θα ακολουθήσει το περιβάλλον και έχει 3 layers με διαστάσεις (3,16) , (16,16) , (16,1). Ο Actor υπολογίζει πέρα από τα actions τις πιθανότητες των actions και την εντροπία τους.Αυτές οι τιμές είναι χρήσιμες για τον on-policy αλγόριθμος PPO.

```python
#critic with 3 layers
class Critic(nn.Module):
    def __init__(self, state_dim):
        super().__init__()

        self.fc1 = nn.Linear(state_dim, 32)
        self.fc2 = nn.Linear(32, 16)
        self.fc3 = nn.Linear(16, 1)

    def forward(self, state):
        v = t.relu(self.fc1(state))
        v = t.relu(self.fc2(v))
        v = self.fc3(v)
        return v
```

Εδώ ορίζεται η κλάση του μοντέλου Critic για τον υπολογισμό του value function του policy του Actor .Έχει 3 layers με διαστάσεις (4,32) , (32,16) , (16,1).

```
#main
if __name__ == "__main__":
    #initialize actor critic and ppo algorithms
    actor = Actor(observe_dim, action_dim)
    critic = Critic(observe_dim)

    ppo = PPO(actor, critic, t.optim.Adam, nn.MSELoss(reduction="sum"))

    #counter for counting all the episodes
    n_iter = 1
    # # array of every episode counter
    all_iter=[]
    # array for every episodes expected return
    all_exp_returns = []
    # array for every episodes runtime
    all_times = []
    episode, step, reward_fulfilled = 0, 0, 0
    smoothed_total_reward = 0
```

Στη συνέχεια αρχικοποιείται ο αλγόριθμος PPO για βελτιστοποίηση του policy και μεγιστοποίηση του reward σε βάθος χρόνου.

Ο συγκεκριμένος αλγόριθμος είναι on-policy δηλαδή βελτιστοποιεί το value με το policy που χρησιμοποιείται κάθε φορά ,χωρίς να χρησιμοποιεί τα observations από προηγούμενες policies.

Στη συνέχεια αρχικοποιούνται μερικές μεταβλητές.

```
while episode < max_episodes:
    episode += 1
    total_reward = 0
    rewards = []
    terminal = False
    step = 0

    #starting state of episode and reseted pendulum
    state = t.tensor(env.reset(), dtype=t.float32).view(1, observe_dim)
    tmp_observations = []

    #timer
    start = timeit.default_timer()
```

Αρχίζουν να εκτελούνται τα επεισόδια . Στην αρχή κάθε επεισοδείου γίνεται επαναφορά του περιβάλλοντος env.reset() και υπολογίζεται η αρχική κατάσταση.

Επίσης αρχίζει να μετρά ένας timer για να μετρήσουμε πόσο χρόνο χρειάστηκε να εκτελεστεί κάθε επεισόδειο.

```
while step <= max_steps:
    #env.render()  #shows graphics
    step += 1
    with t.no_grad():
        #save previous state as old state
        old_state = state

        #action of every step from ppo algorithm
        action = ppo.act({"state": old_state})[0]

        act = np.array([action.item()], dtype=np.float32)

        #new state, reward , terminal after using the action on pendulum
        state, reward, terminal, _ = env.step(act)
        state = t.tensor(state, dtype=t.float32).view(1, observe_dim)

        #save every reward of episode
        rewards.append(reward)
        #return of episode
        total_reward += reward

#observation of episode so that they are used on ppo
        tmp_observations.append(
            {
                "state": {"state": old_state},
                "action": {"action": action},
                "next_state": {"state": state},
                "reward": reward,
                "terminal": terminal or step == max_steps,
            }
        )
```

Σε κάθε step του επισοδείου εκτελείται ο αλγόριθμος ppo και υπολογίζει το action. Το action αυτό μεταφέρεται στην συνάρτηση env.step() για να γίνει η εντολή στο pendulum και επιστρέφονται η κατάσταση , η επιβράβευση και το terminal μια εντολή για υπολογισμό της επιτυχίας του αλγορίθμου.

Στη συνέχεια αποθηκέυονται το reward, το return (total_reward) και τα observation του κάθε επεισοδείου το old_state,action,state,reward τα οποία χρησιμοποιούνται στον αλγόριθμο td3.

```
    #stop timer
    stop = timeit.default_timer()
    time = stop-start


 #store observations of episode
    ppo.store_episode(tmp_observations)

#update algorithm
    ppo.update()


     # show reward
    rewards = np.array(rewards)
    #array of every episode number
    all_iter.append(n_iter)
#array of expected return of every episode
    all_exp_returns.append(np.mean(rewards.sum()))
 #array of run times of every episode
    all_times.append(time)

#smoothed episode return from previous episodes so that we have a smooth stop on the algorithm and not a random one
    smoothed_total_reward = smoothed_total_reward * 0.5 + total_reward * 0.5
    logger.info(f"Episode {episode} smoothed_total_reward={smoothed_total_reward:.2f}")
    print("Iteration: {:6d}\tRuntime: {:6.4f}\tExpected return: {:6.2f}".format(n_iter, time, np.mean(rewards.sum())))
    n_iter += 1
```

Μετά από κάθε επεισόδειο σταματάει ο timer και αποθηκεύονται τα observations .Επίσης ο αλγόριθμος PPO κάνει update τα observations του .

Μετά αποθηκεύονται τα rewards του επεισοδίου όλα σε ένα πίνακα.

Ο αριθμός επεισοδείου ,το expected return και ο χρόνος εκτέλεσης επεισοδίου αποθηκεύονται και αυτά για να αποθηκευτούν στο csv αρχείο.

Τυπώνεται το smoothed_total_reward ,το ομαλοποιημένο return του επεισοδείου το οποίο χρησιμοποιείται και στον έλεγχο για τον αν είχαμε επιτυχημένη λύση προκειμένου να έχουμε ομαλό σταμάτημα του αλγορίθμου και όχι κάποιο σε τυχαίο χρόνο.

Τυπώνεται επίσης ο αριθμός επεισοδίου, ο χρόνος εκτελεσής του και το expected return.

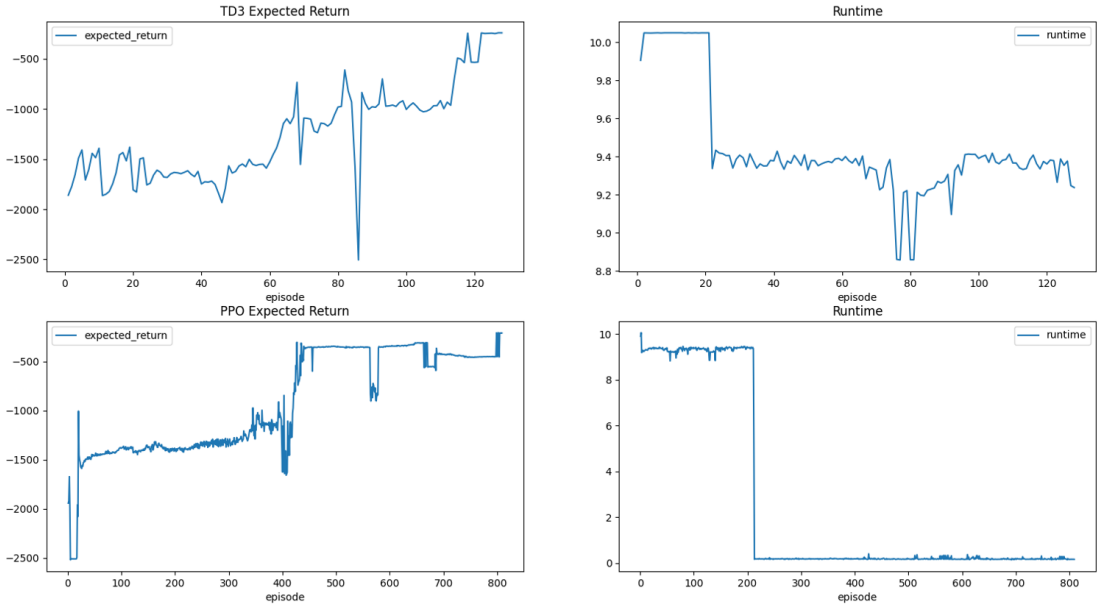
```
# start checking the returns after 2 episodes in case it starts with good reward
    if episode > 2:
    #5 times smoothed return is more than -300 algorithm is solved and it writes expected return and runtime on a csv file named pendulum.ppo.csv
        if smoothed_total_reward > solved_reward:
            reward_fulfilled += 1
            if reward_fulfilled >= solved_repeat:
                logger.info("Environment solved!")
                df = pd.DataFrame({"episode" : all_iter, "expected_return" : all_exp_returns, "runtime" : all_times})
                df.to_csv("pendulum_ppo.csv", index=False)
                exit(0)
        else:
            reward_fulfilled = 0

    if (env.simu.step_world()):
        break
```

Μετά από τα πρώτα 2 επεισόδεια ελέγχεται αν έχουμε 5 φορές το επιθυμητό ομαλοποιημένο reward προκείμενου να έχουμε λύσει το πρόβλημα .

Σε περίπτωση που λύσουμε το πρόβλημα αποθηκεύονται όλες οι τιμές του n_iter expected return και runtime σε ένα αρχείο pendulum_ppo.csv.

2.4 Σύγκριση αλγορίθμων

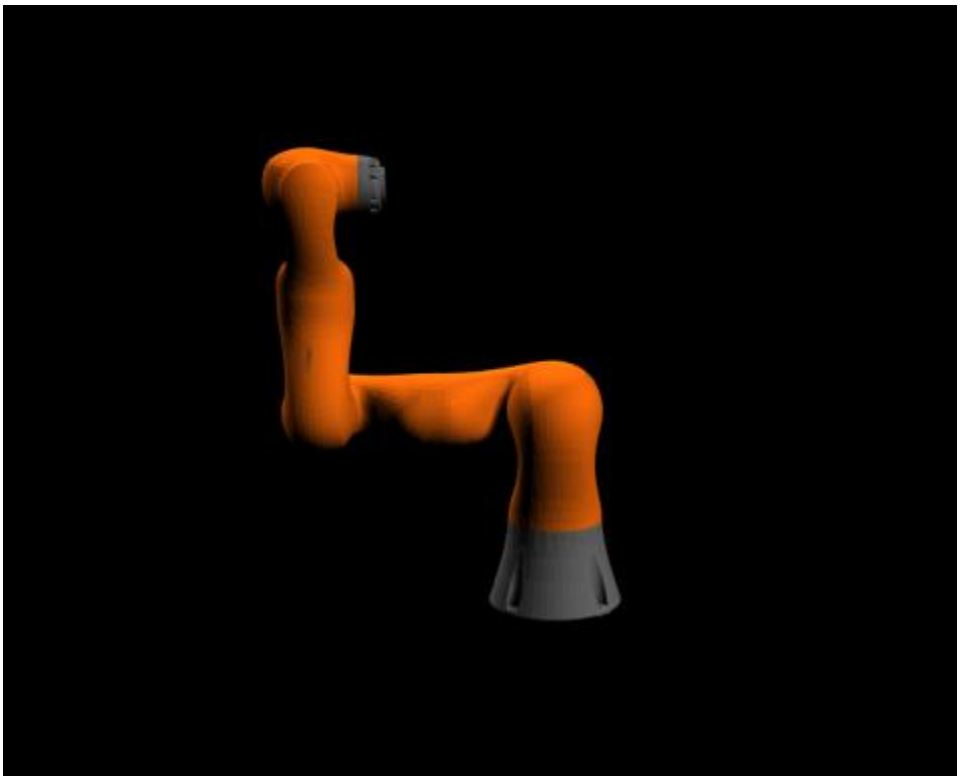
Το αρχείο graphs_pendulum.py τυπώνει τα γραφήματα από τα δεδομένα στα αρχεία pendulum_td3.csv και pendulum_ppo.csv. Αυτά τα αρχεία excel περιέχουν τα δεδομένα μία από τις καλύτερες λύσεις του κάθε αλγορίθμου που κατάφερα να πετύχω.



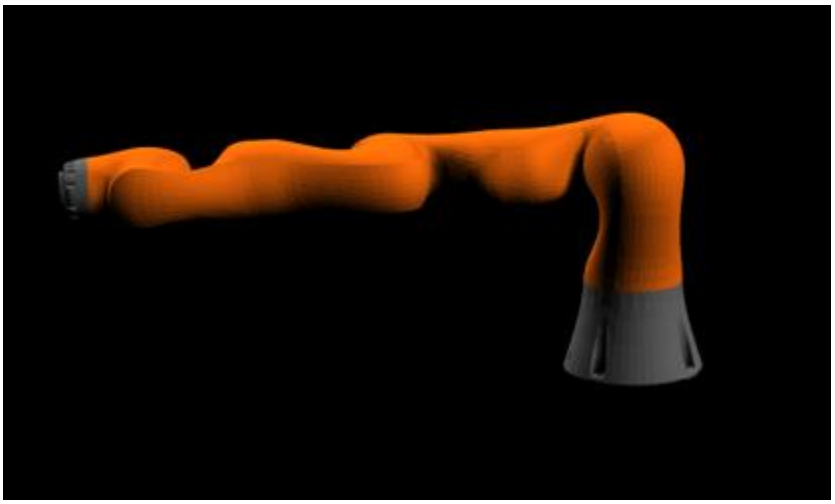
Παρατηρούμε από τα δύο γραφήματα ότι ο off-policy αλγόριθμος TD3 βρίσκει με πολύ λιγότερα επεισόδια λύση στο πρόβλημα έχοντας μεγάλες αυξομειώσεις στο reward του .Ενώ ο on-policy PPO θέλει περισσότερα επεισόδια για να λύσει το πρόβλημα παρόλαυτα αυτά έχει πιο ομαλές αλλαγές στο reward του.

2.4 iiwa_td3

Η αρχική θέση του iiwa είναι :



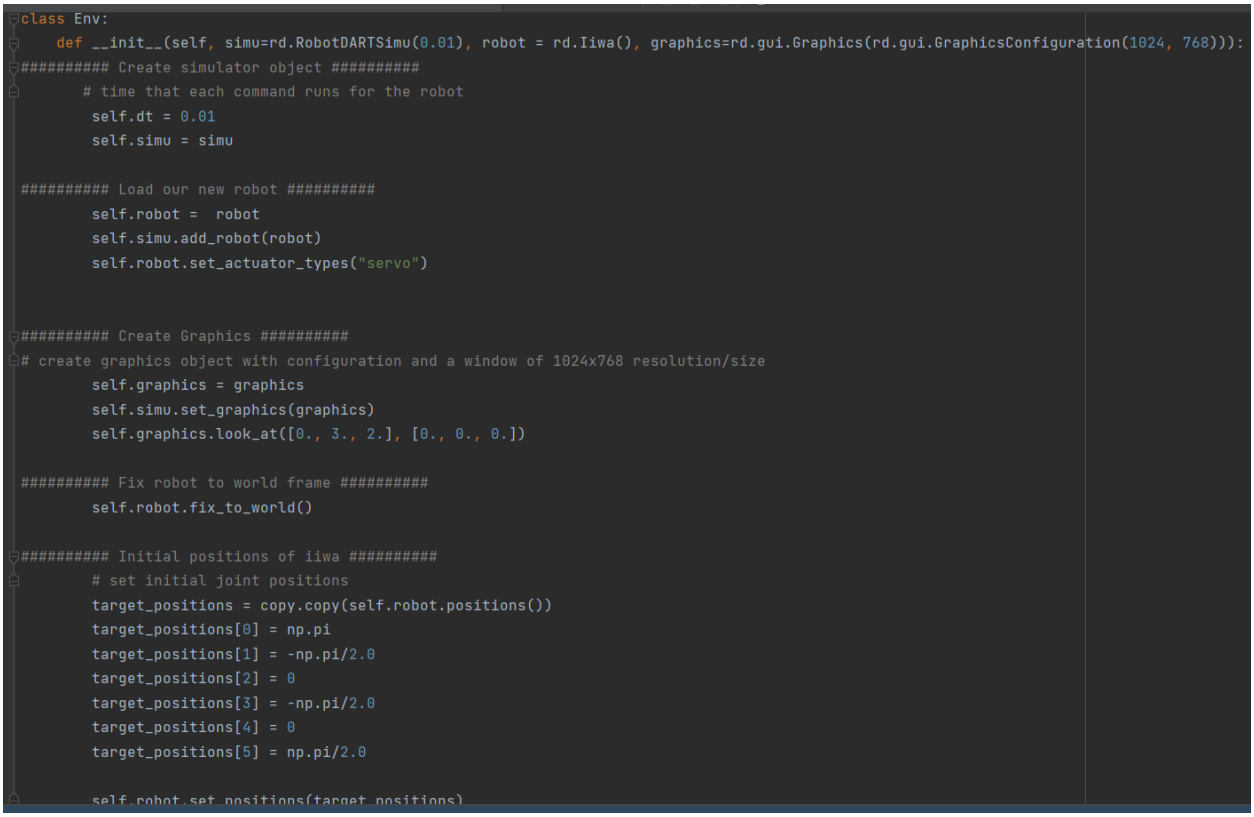
Η επιθυμητή θέση για λύση του αλγορίθμου είναι :



Στην ουσία η επιθυμητή λύση είναι η βάση του iiwa να έχει περιστροφή γωνίας pi . Το πρώτο σώμα να έχει γωνία pi/2 από τη βάση .Το δεύτερο σώμα να

έχει γωνία 0 από το πρώτο σώμα και το τρίτο σώμα να έχει γωνία 0 από το δεύτερο.

Αρχικά δημιουργώ την κλάση όπου δημιουργείται το περιβάλλον του iiwa και τίθεται στην αρχική του θέση.



```
class Env:
    def __init__(self, simu=rd.RobotDARTSimu(0.01), robot = rd.Iiwa(), graphics=rd.gui.Graphics(rd.gui.GraphicsConfiguration(1024, 768))):
########## Create simulator object ##########
       # time that each command runs for the robot
        self.dt = 0.01
        self.simu = simu

 ########## Load our new robot ##########
        self.robot =  robot
        self.simu.add_robot(robot)
        self.robot.set_actuator_types("servo")


########## Create Graphics ##########
# create graphics object with configuration and a window of 1024x768 resolution/size
        self.graphics = graphics
        self.simu.set_graphics(graphics)
        self.graphics.look_at([0., 3., 2.], [0., 0., 0.])

 ########## Fix robot to world frame ##########
        self.robot.fix_to_world()

########## Initial positions of iiwa ##########
        # set initial joint positions
        target_positions = copy.copy(self.robot.positions())
        target_positions[0] = np.pi
        target_positions[1] = -np.pi/2.0
        target_positions[2] = 0
        target_positions[3] = -np.pi/2.0
        target_positions[4] = 0
        target_positions[5] = np.pi/2.0

        self.robot.set_positions(target_positions)
```

Ο χρόνος στον οποίο εκτελούνται οι εντολές στο pendulum είναι 0.01s και χρησιμοποιούνται κινητήρες servo.

```
def reset(self):
    #initial position
    target_positions = copy.copy(self.robot.positions())
    target_positions[0] = np.pi
    target_positions[1] = -np.pi/2.0
    target_positions[2] = 0
    target_positions[3] = -np.pi/2.0
    target_positions[4] = 0
    target_positions[5] = np.pi/2.0

    self.robot.set_positions(target_positions)
    #commands are reseted to zero
    self.robot.set_commands([None,None,None,None,None,None,None])
    self.simu.step_world()

    theta=-np.pi+self.robot.positions()[0]
    #starting angle of iiwa`s lower body joint converted so that having a 90 degree angle from the ground gives value 0
    theta1=np.pi/2.0+self.robot.positions()[1]

    #starting angle of iiwa`s middle body joint giving 0 value when its paralell to the lower body
    theta2 = self.robot.positions()[3]

    #starting angle of iiwa`s higher body joint giving 0 value when its paralell to the middle body
    theta3 = self.robot.positions()[5]

    #angle velocity of every joint of iiwa
    thdot=self.robot.velocities()

     #starting state of iiwa
    state = np.array([np.cos(theta),np.cos(theta1),np.cos(theta2),np.cos(theta3) , np.sin(theta),np.sin(theta1), np.sin(theta2) ,np.sin(theta3),
    thdot[0] ,thdot[1] , thdot[2], thdot[3], thdot[4], thdot[5], thdot[6]], dtype=np.float32)
     #state is returned for the first state of the episode
    return state
```

Μέσα στην κλάση υπάρχει η συνάρτηση reset ,η οποία επαναφέρει σε κάθε επεισόδιο το iiwa στην αρχική του θέση και μηδενίζει τις εντολές που δέχεται.

Ενώ ταυτόχρονα υπολογίζει στην αρχική αυτή θέση την γωνία theta που έιναι η περιστροφή της βάσης του iiwa γύρω απο τον εαυτό τις με τιμές [-πι,πι] και με 0 τιμή στην θέση όπου η βάση του iiwa έχει περιστραφεί κατά πι γωνία.

Υπολογίζει στην αρχική αυτή θέση την γωνία theta1 που έιναι γωνία του πρώτου σώματος του iiwa σε σχέση με την βάση με τιμές [-πι,πι] και με 0 τιμή στην θέση όπου το πρώτο σώμα του iiwa έχει 0 γωνία.

Υπολογίζει στην αρχική αυτή θέση την γωνία theta2 που έιναι γωνία του δεύτερου σώματος του iiwa σε σχέση με το πρώτο σώμα με τιμές [-πι,πι] και με 0 τιμή στην θέση όπου το δεύτερο σώμα του iiwa έχει 0 γωνία.

Υπολογίζει στην αρχική αυτή θέση την γωνία theta3 που έιναι γωνία του τρίτου σώματος του iiwa σε σχέση με το δεύτερο σώμα με τιμές [-πι,πι] και με 0 τιμή στην θέση όπου το τρίτο σώμα του iiwa έχει 0 γωνία.

Επίσης υπολογίζονται και οι 7 γωνιακές ταχύτητες από όλες τις κινήσεις του iiwa

Οι 4 γωνίες και όλες οι γωνιακές αποτελούν την αρχική κατάσταση του επισοδείου.

```
   def step(self, action):
       terminal=False

       #immobilize iiwa`s rotation of the first body of iiwa from itself so that we make the learning process easier
       #ction[0]=0

       #commands for the robot
       self.robot.set_commands(action)
       self.simu.step_world()

       theta=-np.pi+self.robot.positions()[0]
#new angle of iiwa`s lower body joint converted so that having a 90 degree angle from the ground gives value 0
       theta1=np.pi/2.0+self.robot.positions()[1]

   #new angle of iiwa`s middle body joint giving 0 value when its paralell to the lower body
       theta2 = self.robot.positions()[3]

   #new angle of iiwa`s higher body joint giving 0 value when its paralell to the middle body
       theta3 = self.robot.positions()[5]

  #angle velocity of every joint of iiwa
       thdot=self.robot.velocities()


     #reward for the action using the 3 angles, all velocities and all actions
       reward = theta**2+theta1**2 + theta2**2 + theta3**2 + 0.1* thdot**2 + 0.001 * (action**2)

       newstate = np.array([np.cos(theta),np.cos(theta1),np.cos(theta2),np.cos(theta3) , np.sin(theta),np.sin(theta1), np.sin(theta2) ,np.sin(theta3),
       thdot[0] ,thdot[1] , thdot[2], thdot[3], thdot[4], thdot[5], thdot[6]], dtype=np.float32)

       return newstate, -reward.sum(), terminal, {}
```

Μέσα στην κλάση περιβάλλοντος υπάρχει η συνάστηση step. Αυτή η συνάρτηση δέχεται ως όρισμα τα 7 action για κάθε step που υπολογίζει ο αλγόριθμος μας .

Τα action είναι και οι εντολές που δέχεται σε κάθε step το iiwa.

Αφού δεχτεί τις εντολές το iiwa υπολογίζονται οι νέες γωνίες theta, theta1, theta2, theta3 του iiwa και οι νέες γωνιακές ταχύτητες.

Με αυτές τις τέσσερις γωνίες τις ταχύτητες και τα actions υπολογίζεται σε κάθε step το reward για την κίνηση που έκανε .Οι γωνίες προστίθεονται στους πινακες των γωνιακών ταχυτήτων και ο νεός πίνακας προστίθενται στον πίνακα των action .Το άθροισμα όλων των τιμών του τελικού πίνακα αποτελεί το τελικό reward ,το οποίο έχει την μικροτερη τιμή όταν οι γωνίες έχουν τιμή 0 δεν έχουμε ταχύτητες και commands στο iiwa

Αφού υπολογιστεί και το reward υπολογίζεται και η νέα κατάσταση του iiwaμε τις νέες γωνίες και ταχύτητες.

Τα state δημιουργούν τον χώρο παρατήρησης του αργορίθμου για όλα τα steps του και περιλαμβάνει τα cos και sin για τις τέσσερις γωνίες και όλες τις γωνιακές ταχύτητες.

```
#inialize iiwa enviroment and the algorithms parameters
env = Env()
observe_dim = 15
action_dim = 7
action_range = 2.5
max_episodes = 2000
max_steps = 500
noise_param = (0, 0.2)
noise_mode = "normal"
solved_reward = -2800
solved_repeat = 5
```

Στη συνέχεια εκτός της κλάσης env έχουμε την αρχικοποίηση του περιβάλοντος του iiwa και της παραμέτρους για τις μεθόδους actor-critic ,για τον θόρυβο του td3 και την τιμή που πρέπει να φτάσει το reward 5 φορές για να σταματήσει ο αλγόριθμος με επιτυχημένη λύση, καθώς και τα μέγιστα steps και επεισόδια του αλγορίθμου.

```python
# model definition
#actor with 3 layers
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, action_range):
        super().__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_dim)
        self.action_range = action_range

    def forward(self, state):
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        a = t.tanh(self.fc3(a)) * self.action_range
        return a
```

Εδώ ορίζεται η κλάση του μοντέλου Actor για τον υπολογισμό του policy των πιθανών action που θα ακολουθήσει το περιβάλλον και έχει 3 layers με διαστάσεις (15,16) , (16,16) , (16,7).

```python
#critic with 3 layers
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()

        self.fc1 = nn.Linear(state_dim + action_dim, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)

    def forward(self, state, action):
        state_action = t.cat([state, action], 1)
        q = t.relu(self.fc1(state_action))
        q = t.relu(self.fc2(q))
        q = self.fc3(q)
        return q
```

Εδώ ορίζεται η κλάση του μοντέλου Critic για τον υπολογισμό του value function του policy του Actor .Έχει 3 layers με διαστάσεις (22,64) , (64,32) , (32,1).

```
#main
if __name__ == "__main__":
#initialize actor critic and td3 algorithms
    actor = Actor(observe_dim, action_dim, action_range)
    actor_t = Actor(observe_dim, action_dim, action_range)
    critic = Critic(observe_dim, action_dim)
    critic_t = Critic(observe_dim, action_dim)
    critic2 = Critic(observe_dim, action_dim)
    critic2_t = Critic(observe_dim, action_dim)

    td3 = TD3(
        actor,
        actor_t,
        critic,
        critic_t,
        critic2,
        critic2_t,
        t.optim.Adam,
        nn.MSELoss(reduction="sum"),
    )
```

Στη συνέχεια αρχικοποιείται ο αλγόριθμος Td3 για βελτιστοποίηση του policy και μεγιστοποίηση του reward σε βάθος χρόνου.

Ο συγκεκριμένος αλγόριθμος είναι off-policy δηλάδή βελτιστοποιεί το value χρησιμοποιώντας διαφορετικό policy .Δηλαδή χρησιμοποιεί observations από προηγούμενα policies για να μάθει.

```
#counter for counting all the episodes
    n_iter = 1
     # array of every episode counter
    all_iter=[]
    # array for every episodes expected return
    all_exp_returns = []
    # array for every episodes runtime
    all_times = []
    episode, step, reward_fulfilled = 0, 0, 0
    smoothed_total_reward = 0

#every episode of the algorithm
    while episode < max_episodes:
        episode += 1
        total_reward = 0
        rewards = []
        terminal = False
        step = 0

    #starting state of episode and reseted iiwa
        state = t.tensor(env.reset(), dtype=t.float32).view(1, observe_dim)
        tmp_observations = []

        #timer
        start = timeit.default_timer()
    #every step of episode
```

Στη συνέχεια αρχικοποιούνται μερικές μεταβλητές και αρχίζουν να εκτελούνται τα επεισόδια . Στην αρχή κάθε επεισοδείου γίνεται επαναφορά του περιβάλλοντος env.reset() και υπολογίζεται η αρχική κατάσταση.

Επίσης αρχίζει να μετρά ένας timer για να μετρήσουμε πόσο χρόνο χρειάστηκε να εκτελεστεί κάθε επεισόδειο.

```
#every step of episode
    while step <= max_steps:
        step += 1
        with t.no_grad():
            #save previous state as old state
            old_state = state
            #action of every step from td3 algorithm with noise
            action = td3.act_with_noise(
                {"state": old_state}, noise_param=noise_param, mode=noise_mode
                )
            act = np.array([action.item()], dtype=np.float32)

            #new state, reward , terminal after using the action on pendulum
            state, reward, terminal, _ = env.step(act)
            state = t.tensor(state, dtype=t.float32).view(1, observe_dim)

            #save every reward of episode
            rewards.append(reward)
            #return of episode
            total_reward += reward

            #observation of episode so that they are used on td3
            tmp_observations.append(
                {
                    "state": {"state": old_state},
                    "action": {"action": action},
                    "next_state": {"state": state},
                    "reward": reward,
                    "terminal": terminal or step == max_steps,
                }
            )
```

Σε κάθε step του επισοδείου εκτελείται ο αλγόριθμος td3 με θόρυβο και υπολογίζει το action. Το action αυτό μεταφέρεται στην συνάρτηση env.step() για να γίνει η εντολή στο pendulum και επιστρέφονται η κατάσταση , η επιβράβευση και το terminal μια εντολή για υπολογισμό της επιτυχίας του αλγορίθμου.

Στη συνέχεια αποθηκέυονται το reward, το return (total_reward) και τα observation του κάθε επεισοδείου το old_state,action,state,reward τα οποία χρησιμοποιούνται στον αλγόριθμο td3.

```
        )
    #stop timer
    stop = timeit.default_timer()
    time = stop-start

    #store observations of episode
    td3.store_episode(tmp_observations)

     # show reward
    rewards = np.array(rewards)
    #array of every episode number
    all_iter.append(n_iter)
    #array of expected return of every episode
    all_exp_returns.append(np.mean(rewards.sum()))
    #array of run times of every episode
    all_times.append(time)

    #smoothed episode return from previous episodes so that we have a smooth stop on the algorithm and not a random one
    smoothed_total_reward = smoothed_total_reward * 0.5 + total_reward * 0.5
    logger.info(f"Episode {episode} smoothed_total_reward={smoothed_total_reward:.2f}")
    print("Iteration: {:6d}\tRuntime: {:6.4f}\tExpected return: {:6.2f}".format(n_iter, time, np.mean(rewards.sum())))
    n_iter += 1
```

Μετά από κάθε επεισόδειο στματάει ο timer και αποθηκεύονται τα observations .Επίσης αποθηκεύονται τα rewards του επεισοδίου όλα σε ένα πίνακα.

Ο αριθμός επεισοδείου ,το expected return και ο χρόνος εκτέλεσης επεισοδίου αποθηκεύονται και αυτά για χρήση στο γράφημα.

Τυπώνεται το smoothed_total_reward ,το ομαλοποιημένο return του επεισοδείου το οποίο χρησιμοποιείται και στον έλεγχο για τον αν είχαμε επιτυχημένη λύση προκειμένου να έχουμε ομαλό σταμάτημα του αλγορίθμου και όχι κάποιο σε τυχαίο χρόνο.

Τυπώνεται επίσης ο αριθμός επεισοδίου, ο χρόνος εκτελεσής του και το expected return.

```
# after 20 episodes td3 is updated so that we have enough observations
      if episode > 20:
          for _ in range(step):
              td3.update()

  #5 times smoothed return is more than -300 algorithm is solved and it writes expected return and runtime on a csv file named iiwa_td3.csv
          if smoothed_total_reward > solved_reward:
              reward_fulfilled += 1
              if reward_fulfilled >= solved_repeat:
                  logger.info("Environment solved!")
                  df = pd.DataFrame({"episode" : all_iter, "expected_return" : all_exp_returns, "runtime" : all_times})
                  df.to_csv("iiwa_td3.csv", index=False)

                  exit(0)
          else:
              reward_fulfilled = 0

      if (env.simu.step_world()):
          break
```

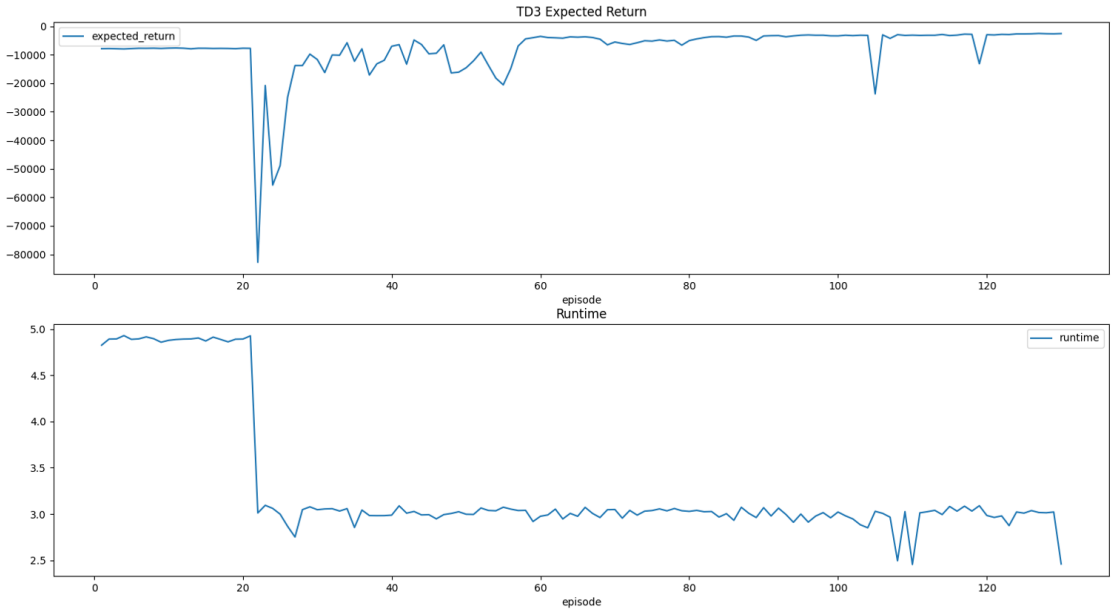
Τέλος ο αλγόριθμος td3 κάνει update τα observations του μετά από 20 επεισόδειο ώστε να είναι αρκετά.

Επίσης μετά από 20 επεισόδεια ελέγχεται αν έχουμε 5 φορές το επιθυμητό ομαλοποιημένο reward προκείμενου να έχουμε λύσει το πρόβλημα .

Σε περίπτωση που λύσουμε το πρόβλημα δημιουργούνται και τα γραφήματα του Expected return και Runtime.

2.5 Αποτέλεσμα λύσης του iiwa

Το αρχείο graphs_iiwa.py τυπώνει τα γραφήματα από τα δεδομένα στο αρχείο iiwa_td3.csv. Αυτό το αρχείο excel περιέχει τα δεδομένα μία από τις καλύτερες λύσεις του αλγορίθμου που κατάφερα να πετύχω.



Παρατηρούμαι ότι όταν έχουμε καλές τιμές στο reward στα πρώτα επεισόδεια πριν αρχίσει να ενημερώνεται ο td3 αλγόριθμος ,τότε το td3 φτάνει πολύ γρήγορα στη λύση.