LUT-yliopisto

LUT School of Energy Systems

LUT Kone

BK10A0402 Kandidaatintyö

REINFORCEMENT LEARNING IN MACHINERY CONTROL

VAHVISTUSOPPIMINEN KONEOHJAUKSESSA

Lappeenrannassa 25.8.2021

Samu Kallio

Tarkastaja	TkT Grzegorz Orzechowski
Ohjaaja	TkT Grzegorz Orzechowski

TIIVISTELMÄ

LUT-yliopisto
LUT School of Energy Systems
LUT Kone

Samu Kallio

Vahvistusoppiminen koneohjauksessa

Kandidaatintyö

2021

29 sivua, 15 kuvaa ja 3 liitettä

Tarkastaja: TkT Grzegorz Orzechowski

Ohjaaja: TkT Grzegorz Orzechowski

Hakusanat: Vahvistusoppiminen, PPO, koneohjaus, automaatio

Tässä kandidaatintyössä arvioidaan vahvistusoppimisen soveltuvuutta koneohjauksen automatisointiin. Tämä tehdään kouluttamalla kaksi simuloitua, tavanomaista koneohjauksen tehtävää edustavaa yksinkertaista robottia hyödyntämällä lupaavimpia vahvistusoppimisalgoritmeja. Oppimiskäyttäytymistä seurataan numeerisilla pisteytysarvoilla, joita algoritmi vastaanottaa toiminnastaan, sekä seuraamalla simulaation käyttäytymistä visuaalisesti.

Koulutuksen tuloksena molemmat robotit saatiin oppimaan vakaa käyttäytymismalli, joka vastasi haluttua pienin epätäydellisyyksin. Johtopäätöksenä tästä katsottiin, että vahvistusoppimisella on tietyin varauksin potentiaalia koneautomaation välineenä.

ABSTRACT

LUT University

LUT School of Energy Systems

LUT Mechanical Engineering

Samu Kallio

Reinforcement learning in machinery control

Bachelor’s thesis

2021

29 pages, 15 figures and 3 appendices

Examiner: D. Sc. (Tech.) Grzegorz Orzechowski

Supervisor: D. Sc. (Tech.) Grzegorz Orzechowski

Keywords: Reinforcement learning, PPO, machinery control, automation

This thesis focuses on assessing the suitability of reinforcement learning in machinery control. For this purpose, two simulated robots representing common machinery control tasks are trained by utilizing state-of-the-art reinforcement learning algorithms. The learning behavior is assessed with behavioral scores received from the robot’s environment, as well as visually observing the simulation behavior.

Both robots learned stable behavioral models that corresponded the desired behavior with small imperfections. This led to the conclusion that reinforcement learning has potential as a tool of machinery control automation with certain precautions.

TABLE OF CONTENTS

SYMBOLS AND ABBREVIATIONS

A/$\hat{A}$	advantage
c1	Value function coefficient
c2	Entropy coefficient
$\in$	Clipping parameter
r	Probability ratio
θ	Policy
ML	Machine learning
PPO	Proximal policy optimization
RL	Reinforcement learning
TRPO	Trust region policy optimization
URDF	Unified robot description format
VF	Value function

1 INTRODUCTION

Automation has an increasingly integral role in industrial applications, as advancing technology opens new possibilities. (Wollschlaeger, Sauter & Jasperneite 2017, p18.) With certain types of tasks, possibly intuitive for humans to comprehend, automation can be difficult to implement due to a high number of variables and their combined effect. The demand for increasingly complex automated tasks can push the limits of conventional automation and has encouraged different fields of industries towards utilizing machine learning (ML) in the processes to make it self-learning. (Rebala, Ravi & Churiwala 2019, p1-2.) Machine learning utilizes collected data to improve its behavior. This can simplify the automation process of otherwise difficult tasks, as the different variables and their combined effects are considered by the machine learning algorithm and not beforehand by the programmer. (Rebala, Ravi & Churiwala 2019, p2.) This thesis focuses on assessing the suitability of reinforcement learning (RL) in automation of common machinery control tasks. Reinforcement learning is an area of machine learning that learns tasks completely independently using rewards received from the environment to observe the consequences of its actions and improve its performance. (Nandy & Biswas 2017, p1.)

1.1 Research problem and research questions

This thesis focuses on automation of common machinery control tasks by applying a reinforcement learning scheme and reviewing the following research questions:

1. What are the current state-of-the-art solutions in reinforcement learning control?
2. How different hyperparameters affect reinforcement learning behavior?
3. Can reinforcement learning be effectively utilized in machinery control?

1.2 Research methods

The functionality of reinforcement learning in machinery control is observed by using a RL-algorithm to train two different simulated robots representing common machinery control tasks. One of the robots is available in the toolkit used, and the other is created for the specific purpose of this thesis. The behavior of the robots is observed qualitatively with simulation animation and quantitatively with numerical rewards.

1.3 Objective

The objective of the research is to assess the suitability of reinforcement learning in common machinery control tasks. To achieve this, a functioning simulated robot environment is developed and implemented, and a proximal policy optimization (PPO) algorithm able to control simulated robot models and learn desired behavior is used.

1.4 Scope

Out of many different RL algorithms the one used and comprehensibly reviewed is proximal policy optimization. The suitability to machinery control is assessed through simulation of two simple robots, using continuous action and observation spaces.

2 REINFORCEMENT LEARNING

Reinforcement learning is an area of machine learning that uses algorithms called “agents” to control a desired action. (Nandy & Biswas 2017, p1.) In addition to RL, there are two different main types of machine learning algorithms: supervised leaning and unsupervised learning. Supervised learning has a labeled set of inputs and outputs, meaning that the algorithms achieve their goal by performing actions in a pre-determined way. Unsupervised learning depends only on a set of inputs, meaning that the algorithms can only detect patterns in previously existing data. A key difference and benefit of RL compared to the other two main types of ML algorithms is that it defines the used approach completely independently, meaning that an RL algorithm can, under the right circumstances, learn the optimal way to perform tasks through a trial-and-error process, mimicking human-like learning. The agent takes actions and receives rewards from an environment based on the actions it takes. By observing rewards, the agent discovers ways to improve its performance and the behavior to meet the desired goal by learning the consequences of its actions, as behavior slowly becomes less random and more based on collected data. (Dangeti 2017, p358-359.) Figure 1 illustrates the basic principle of RL, where an agent performs an action in the environment, and then receives information about the state of the environment after performing the action as well as a reward representing the desirability of that action.

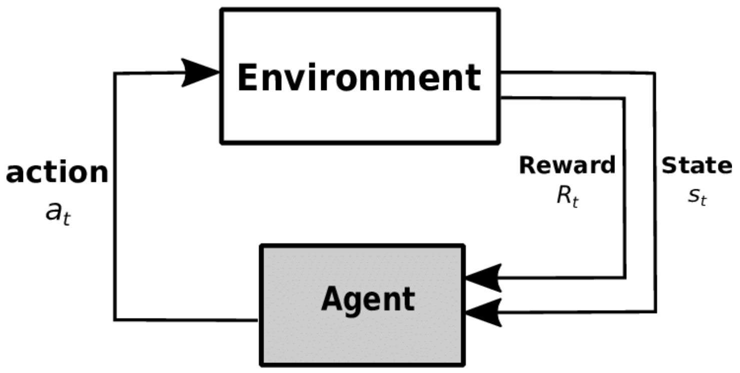


Figure 1. RL basic principle (Amiri, Mehrpouyan, Fridman, Mallik, Nallanathan, Matolak 2018, s3)

The key qualities of reinforcement learning also reflect to its key disadvantages. As RL algorithms, by definition, have the liberty to self-determine the approach they use to reach the desired goal, they must also explore and analyze countless different approaches before achieving desired behavior. This means that RL is sample-inefficient and depending on the available computing power and complexity of the scheme, can be time-consuming. Having no pre-defined approach also means that the algorithms do not know what the optimal behavior should be. Under some conditions this can lead to learning process converging and getting stuck too early to a local optimum, or the algorithm catastrophically diminishing already learned behavior due to an update step updating a policy too far from the old. (Ding & Dong, p249 – 257.)

2.1 General terminology

- **Agent** is the entity in RL, that interacts with the environment by taking actions, receiving rewards, and learning to improve its behavior (Dangeti 2017, p361.)
- **Environment** is where the agent operates. Environment defines what happens when an action is taken and how the agent is rewarded (Dangeti 2017, p361.)
- **Policy** defines the decisions made by the agent. The policy is updated with collected data, leading to better decision making and better actions. (Dangeti 2017, p362.)
- A **continuous** observation or action space has set limits for maximum and minimum values, in between of which the agent can use any continuous value. This means that the observation space provides a “measured” size for an observation dimension, and the agent can take variable size actions. Observation and action spaces are separate and do not dictate each other, meaning for example that an environment can have a continuous observation space and a discrete action space. (Delalleu, Peter, Alonso, Logut 2019, p2.)
- A **discrete** observation or action space has discrete values for each observation dimension or action, meaning that they cannot be measured or adjusted with continuous values (Delalleu, Peter, Alonso, Logut 2019, p2.)
- **Episode** is the sequence which the agent runs through before the environment is reset (Dangeti 2017, p362.)

2.2 RL algorithms

There are numerous different algorithms for reinforcement learning. These algorithms can be divided into two fundamentally different main categories: model-free algorithms and model-based algorithms. Model-based algorithms have a pre-defined model of the environment, based on which the agent can make predictions about the next state. Model-free algorithms do not have this feature and rely entirely on the rewards they receive from the environment. Zhang & Yu 2020, p127.) Model-based algorithms are not used or discussed further in this thesis.

On-policy (online) algorithms update the policy using the data the policy collected itself. Off-policy (offline) algorithms use a separate policy for learning and for making decisions, meaning that the acting policy accumulates experience data for updating. Some RL-algorithms can learn off-policy completely offline, meaning that the learning is completely based on previously collected data, and the agent does not interact with the environment or update the collected data during training. The basic principles of these algorithm types are illustrated in figure 2**.** In figure 2a, the agent acting upon current policy π_k is performing an action in the environment represented by a picture of a globe. The environment then returns state s and reward r to the agent acting upon the policy. When it is time for policy update, the information from the iteration is processed by the algorithm, and a new policy is generated accordingly. Learning is based on improving the acting policy over again. In figure 2b, a buffer stores all the previously accumulated experience for learning. The acting policy is separate from the learning policy and can be used for data collection only, the acting policy does not necessarily act according to any learned behavior. This means that off-policy algorithms can learn even if the acting policy is taking completely random actions. Figure 2c shows an offline reinforcement learning process, so all learning is based on previously collected data and the acting policy is not interacting with the environment at any point. (Levine, Kumar, Tucker, Fu 2020, p1-2.)

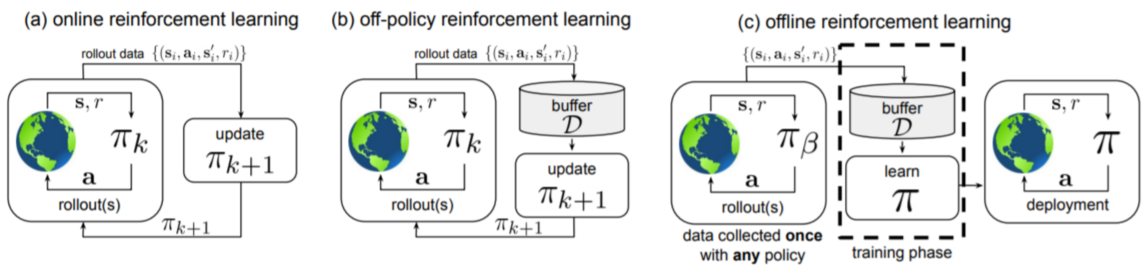

(a) online reinforcement learning
rollout data $\{(s_i, a_i, s'_i, r_i)\}$
s, r
π_k
a
rollout(s)
update
π_{k+1}
π_{k+1}
(b) off-policy reinforcement learning
rollout data $\{(s_i, a_i, s'_i, r_i)\}$
buffer
$\mathcal{D}$
update
π_{k+1}
(c) offline reinforcement learning
$\{(s_i, a_i, s'_i, r_i)\}$
π_β
learn
π
data collected **once** with **any** policy
training phase
deployment

Figure 2. On-policy, off-policy, and offline algorithm basic principles (Levine, Kumar, Tucker, Fu 2020, p2.)

2.2.1 Actor-critic methods

Actor-critic reinforcement learning, such as PPO, that have a separate neural network for approximating the policy (actor) and approximating the value function (critic). The actor network decides which actions to take, and the critic network evaluates those actions. As the critic network is updated over time, it becomes better at evaluating actions taken by the actor, thus becoming better at guiding the actor to a better policy. (Bhatnagar, Sutton, Ghavamzadeh, Lee 2009, p1.)

2.2.2 Proximal policy optimization

Proximal policy optimization (PPO) is class of policy-based algorithms introduced in the year 2017 that has proven to perform comparably or often even better than other state-of-the-art approaches, while being more stable and significantly less complicated in terms of code, computation, and ease of implementation (Schulman, Wolski, Dhariwal, Radfor, Klimov 2017, s1.) For this thesis, PPO is considered as the latest and most capable state-of-the-art approach, and for that reason chosen to be more comprehensively reviewed. The main objective function of PPO is (Schulman, Wolski, Dhariwal, Radfor, Klimov 2017, p3):

$$L^{CLIP}(\theta) = \widehat{E}_t[\min\left(r_t(\theta)\hat{A}_t,\ clip(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon\right)\hat{A}_t] \qquad (1)$$

PPO is based on how policy gradient methods generally work: by computing an expected return for a policy in a certain state, then increasing or decreasing probability of the action the agent took based on if the return was better or worse than expected. In equation 1, $L^{CLIP}(\theta)$ is the clipping function, where θ stands for the policy. Parameter $\widehat{E_t}$ denotes the expected value over a batch of samples (timesteps). The probability ratio between the new and old policy is represented by r_t. $\hat{A}_t$ stands for advantage estimate, where an estimated value for the return predicted by a value function neural network is subtracted from the actual return. The neural network is regularly updated and improved based on data collected from training and gives out noisy values as the estimation is not always accurate or correct. As the expected return is reduced from the real return, $\hat{A}_t$ is negative if the return as worse than expected, and positive if it was better than expected. Clipping parameter ϵ is the key feature in PPO that limits the size of policy updates. (Schulman, Wolski, Dhariwal, Radfor, Klimov 2017, p1-5.)

The clipping functionality is used to prevent a policy from updating too far from the old and causing un-learning. The same issue has also been addressed with Trust Region Policy Optimization (TRPO) algorithms, that add a constraint to the objective optimization to limit the size of policy updates. This approach uses second order optimization and being relatively complicated, overall adds to the complexity of computation and implementation. (Engstrom, Ilyas, Santurkar, Tsipras, Janoos, Rudolph, Madry 2019, s5.) The clipping parameter implemented directly to the PPO objective function enables it to limit the size of policy updates and still use first order optimization without the extra complexity that the constraint used by TRPO causes. For this reason, PPO is comparably computationally cost-effective, stable, and easy to implement. (Schulman, Wolski, Dhariwal, Radfor, Klimov 2017, p1.) The functionality of the clipping parameter ϵ is illustrated in figure 3. In figure 3, the red dot is located where $r = 1$, meaning that the old policy is the same size as the new policy. This is a starting point for the optimization. The left side of the figure represents cases where advantage $A > 0$ and the probability ratio is clipped at $1 + \epsilon$, and the right side represents cases where $A < 0$ and the probability ratio is clipped at $1 - \epsilon$. Thus, the probability of an action in the new policy cannot increase uncontrollably or drop down straight to zero based on a single estimate. The only occasions, where the unclipped function has a smaller value than the clipped version and is chosen by the min-operator, are when the algorithm has made a mistake of reducing the probability of an action even though it resulted in better-than-expected return or increasing the probability of an action even though it resulted in worse than expected return. This enables the algorithm to recover from such mistakes, as clipping would prevent moving "backwards" if the mistake was to exceed the clipping limit to the wrong direction. (Schulman, Wolski, Dhariwal, Radfor, Klimov 2017, p3.)

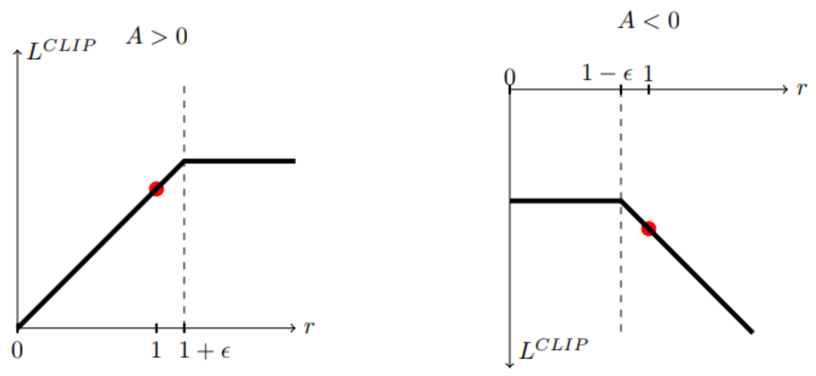


Figure 3. Clipping functionality in PPO (Schulman, Wolski, Dhariwal, Radfor, Klimov 2017, p3.)

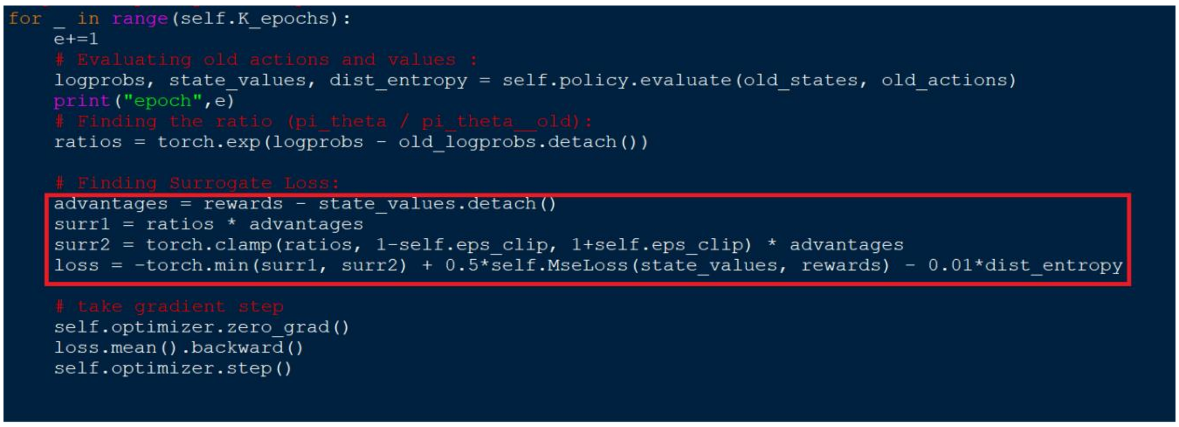
For actual training of a PPO-agent, two additional terms are introduced in addition to the objective function 1. The final objective function is formed as (Schulman, Wolski, Dhariwal, Radfor, Klimov 2017, p5.):

$$L_t{}^{CLIP+VF+S}(\theta) = \ \widehat{E_t}\ [L^{CLIP}(\theta) - c1\ L_t{}^{VF}(\theta) + c2\ S\ [\pi_\theta](s_t)] \tag{2}$$

In equation 2, the first additional term, $L_t{}^{VF}(\theta)$, is a mean square error of the value function. This is added, because using a neural network structure that shares parameters between the policy and value function means that the value function error term must be combined into the final training function. The second term, $S\ [\pi_\theta](s_t)$, adds an entropy bonus to the equation to encourage sufficient exploration. $c1$ and $c2$ are coefficients that value the magnitude of the additional terms (Schulman, Wolski, Dhariwal, Radfor, Klimov 2017, p5.)

3 RESEARCH METHODS

The performance and potential of PPO RL are evaluated through simulation. An actor-critic PPO agent is programmed with Python based on open-source software by (Barhate 2018.) The PPO-objective function is highlighted in figure 4 showing the part of the update function of the PPO-agent used for training. The agent is used to train two simulated robots, one pre-installed, and the other created as groundwork for this thesis for this specific purpose, representing basic tasks that could be encountered in industrial machinery control automation. Both robot environments feature continuous observation and control. The learning and performance of these robots is reviewed by visually inspecting the animated simulation and analyzing rewards received from the environment throughout the training process. The hyperparameters for satisfying performance are searched through trial and error by changing the hyperparameters and inspecting the impact on learning. In addition to their main purpose of being the data the agent uses for learning rewards are also useful in keeping track of the training process as they give a numeric value for the agent's performance.



```
for _ in range(self.K_epochs):
    e+=1
    # Evaluating old actions and values :
    logprobs, state_values, dist_entropy = self.policy.evaluate(old_states, old_actions)
    print("epoch",e)
    # Finding the ratio (pi_theta / pi_theta__old):
    ratios = torch.exp(logprobs - old_logprobs.detach())

    # Finding Surrogate Loss:
    advantages = rewards - state_values.detach()
    surr1 = ratios * advantages
    surr2 = torch.clamp(ratios, 1-self.eps_clip, 1+self.eps_clip) * advantages
    loss = -torch.min(surr1, surr2) + 0.5*self.MseLoss(state_values, rewards) - 0.01*dist_entropy

    # take gradient step
    self.optimizer.zero_grad()
    loss.mean().backward()
    self.optimizer.step()
```

Figure 4. Update function of PPO-agent used for training (Barhate 2018)

OpenAI Gym is a Python module that provides the basic interface and tools for RL development (OpenAI 2016.) PyBullet Gym is an implementation of OpenAI Gym that provides many additional useful tools for RL as well as PyBullet's open-source physics and robotics simulation interface (Coumans, Bai 2021, p2.) The environments and agents created for this thesis are built on, trained, simulated, and animated on the interface these modules provide in tandem. PyBullet Gym also includes a set of pre-installed robotic environments and pre-trained agents.

The PPO-agent uses an optimization algorithm called "Adam" for taking gradient steps and updating the policy. Adam computes adaptive learning rates individually for each learning aspect of an environment. The learning rate given as input for Adam only defines an upper limit as a starting point for learning rate. (Kingma, Ba 2015, p1)

3.1 Reacher robot

The first robot trained operates in a PyBullet Gym pre-installed environment called Reacher. Figure 5 shows the robot. It operates an arm by applying torque to two joints, one central joint and one elbow joint. The purpose of the robot is to manipulate the joints, so the end effector reaches a goal represented by the green sphere appearing in random locations at the beginning of every episode.

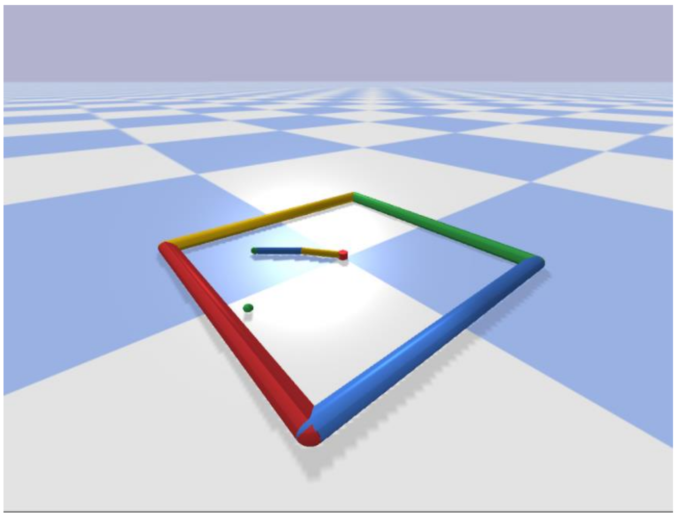
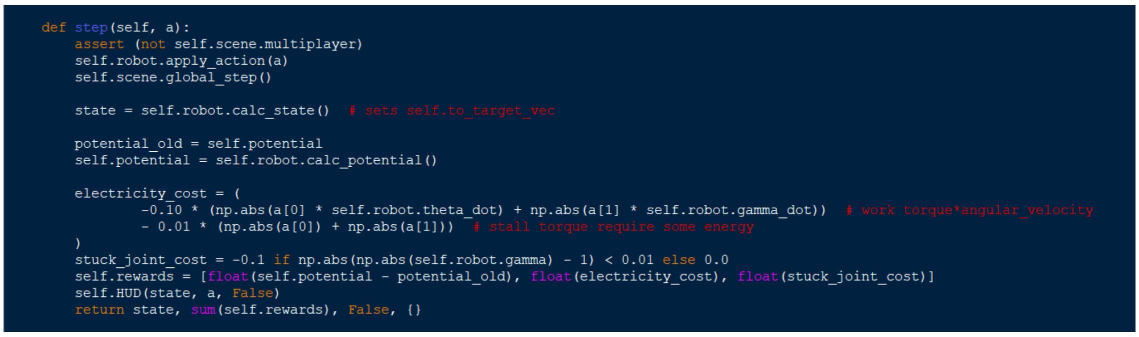


Figure 5. Reacher robot.

Step-function code including the reward function of the Reacher is presented in figure 6. The step function is called by the PPO agent on every timestep, and it performs the action and returns the observation of a single timestep. The environment outputs relevant information about its state such as the joint angles, arm "fingertip" location and location of the goal. It calculates the distance to the goal as "potential". On every step, the potential is assigned as potential_old, meaning the potential on the previous step, and a new potential is calculated representing the current step. The reward is calculated by subtracting the new potential from the old. This way, if the robot has moved closer to the goal compared to the previous step, the reward is positive, and if it has moved further away, the reward is negative. This motivates the agent to take actions which take it closer to the goal. The reward function also has an “electricity cost” meaning that a negative reward is given based on the size of the actions and joint states to motivate the agent to work as efficiently as possible.



```
def step(self, a):
    assert (not self.scene.multiplayer)
    self.robot.apply_action(a)
    self.scene.global_step()

    state = self.robot.calc_state()  # sets self.to_target_vec

    potential_old = self.potential
    self.potential = self.robot.calc_potential()

    electricity_cost = (
            -0.10 * (np.abs(a[0] * self.robot.theta_dot) + np.abs(a[1] * self.robot.gamma_dot))  # work torque*angular_velocity
            - 0.01 * (np.abs(a[0]) + np.abs(a[1]))  # stall torque require some energy
    )
    stuck_joint_cost = -0.1 if np.abs(np.abs(self.robot.gamma) - 1) < 0.01 else 0.0
    self.rewards = [float(self.potential - potential_old), float(electricity_cost), float(stuck_joint_cost)]
    self.HUD(state, a, False)
    return state, sum(self.rewards), False, {}
```

Figure 6. Step function and reward system of Reacher environment (Ellenberger 2018.)

3.2 Leveller robot

The Leveller robot environment, shown in figure 7, was created as groundwork for the specific purpose of this thesis. The robot operates an arm with three joints with freedom to move vertically in two dimensions, representing a simple, excavator-like machine. Its purpose is to manipulate the joints so that the “fingertip” (tip of the red section of the arm) first reaches goal 1 (blue sphere in figure 7), and then move the fingertip to goal 2 (green sphere in figure 7) while keeping the fingertip level on a direct line connecting the two goals appearing in random locations.

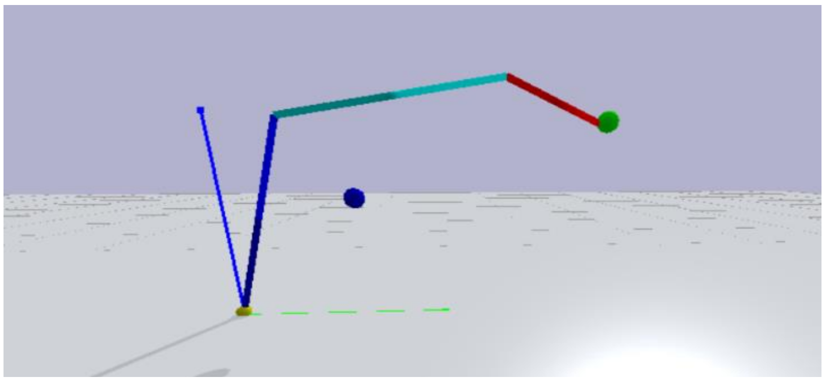
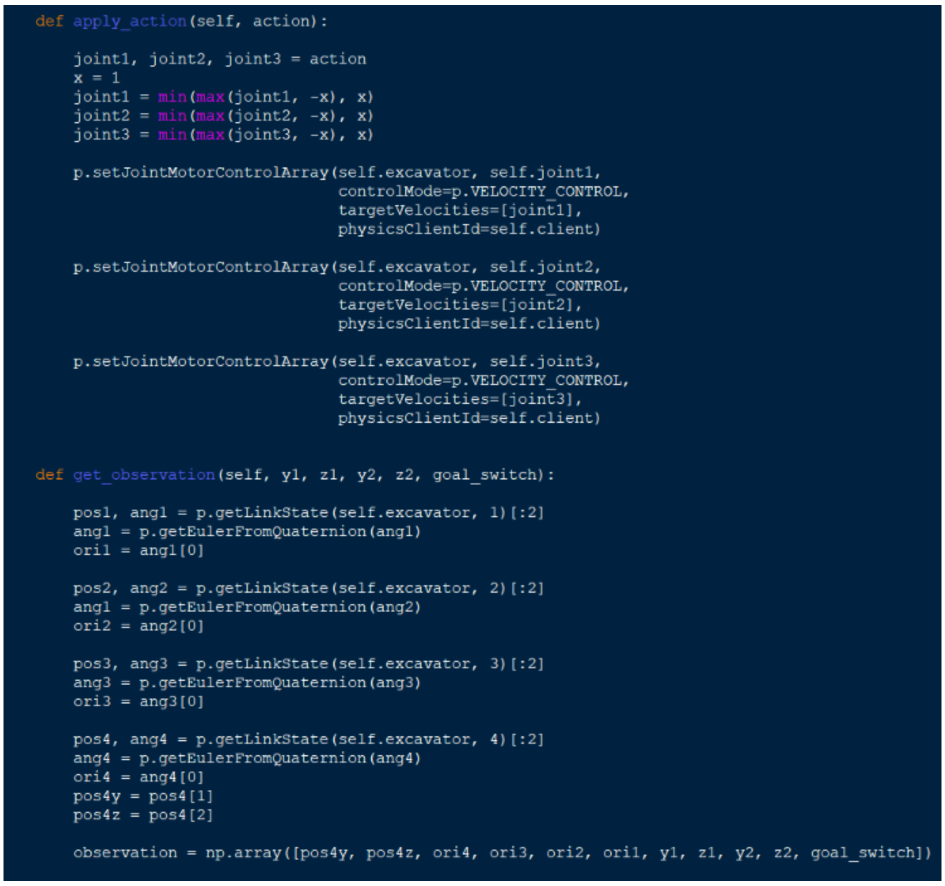


Figure 7. Leveller robot.

The robot itself was created in Unified Robotic Description Format (URDF). URDF defines the dynamics, kinematics, collision model, and the visual aspects of a robot (Kunze, Roehm, Beetz 2011, s1) in a form supported by PyBullet. The environment properties for training are then programmed so that the PPO-agent can control, observe, and learn. The control and observation values of the robot are defined in the file Excavator.py. The robot's joint motors are controlled using PyBullet's velocity controller, meaning that the agent performs actions by applying rotational velocity values to the joints. This action is performed when apply_action -function is called. The file returns the vertical and horizontal location of the fingertip, all relevant joint angles, locations of goals and state of the goal_switch for observation when get_observation is called. Figure 8 shows the apply_action -and get_observation -functions in Excavator.py.

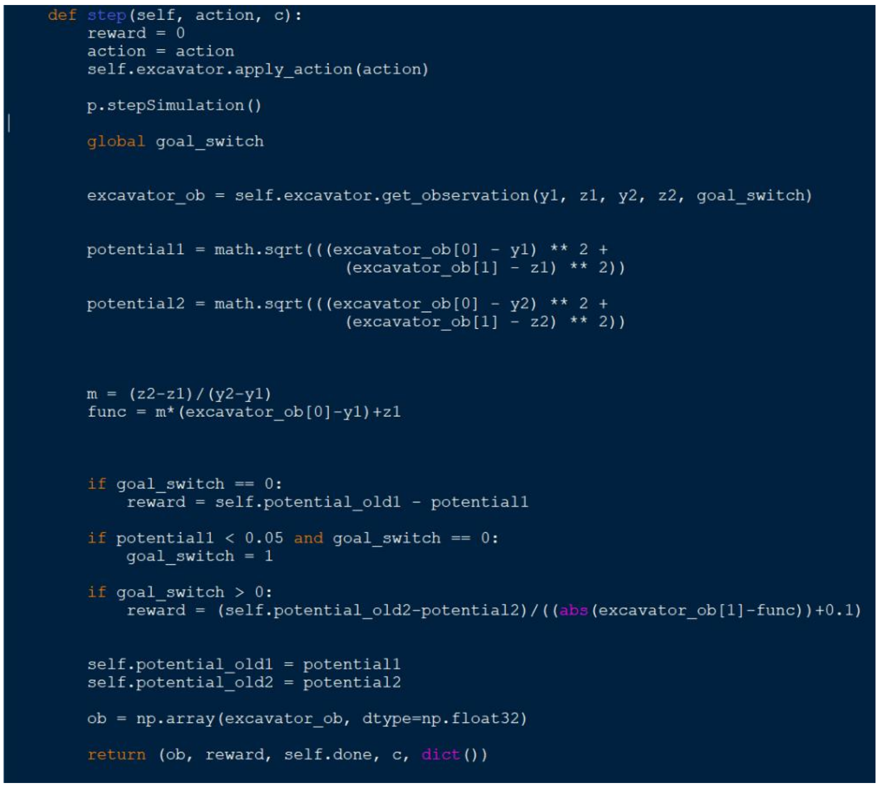


```
def apply_action(self, action):

    joint1, joint2, joint3 = action
    x = 1
    joint1 = min(max(joint1, -x), x)
    joint2 = min(max(joint2, -x), x)
    joint3 = min(max(joint3, -x), x)

    p.setJointMotorControlArray(self.excavator, self.joint1,
                                controlMode=p.VELOCITY_CONTROL,
                                targetVelocities=[joint1],
                                physicsClientId=self.client)

    p.setJointMotorControlArray(self.excavator, self.joint2,
                                controlMode=p.VELOCITY_CONTROL,
                                targetVelocities=[joint2],
                                physicsClientId=self.client)

    p.setJointMotorControlArray(self.excavator, self.joint3,
                                controlMode=p.VELOCITY_CONTROL,
                                targetVelocities=[joint3],
                                physicsClientId=self.client)


def get_observation(self, y1, z1, y2, z2, goal_switch):

    pos1, ang1 = p.getLinkState(self.excavator, 1)[:2]
    ang1 = p.getEulerFromQuaternion(ang1)
    ori1 = ang1[0]

    pos2, ang2 = p.getLinkState(self.excavator, 2)[:2]
    ang1 = p.getEulerFromQuaternion(ang2)
    ori2 = ang2[0]

    pos3, ang3 = p.getLinkState(self.excavator, 3)[:2]
    ang3 = p.getEulerFromQuaternion(ang3)
    ori3 = ang3[0]

    pos4, ang4 = p.getLinkState(self.excavator, 4)[:2]
    ang4 = p.getEulerFromQuaternion(ang4)
    ori4 = ang4[0]
    pos4y = pos4[1]
    pos4z = pos4[2]

    observation = np.array([pos4y, pos4z, ori4, ori3, ori2, ori1, y1, z1, y2, z2, goal_switch])
```

Figure 8. Controls and observation in excavator.py.

The environment itself is defined in a separate env -file, called levelling_env.py. It contains some general setup functions that, for example, define the action and observation spaces, connect the environment to a physics client and load all the necessary URDF-models and reset the environment after an episode. Additionally, the step -function in this file is what contains the reward system, defining the actual purpose of the whole environment so that an RL-algorithm is able to get feedback and learn.

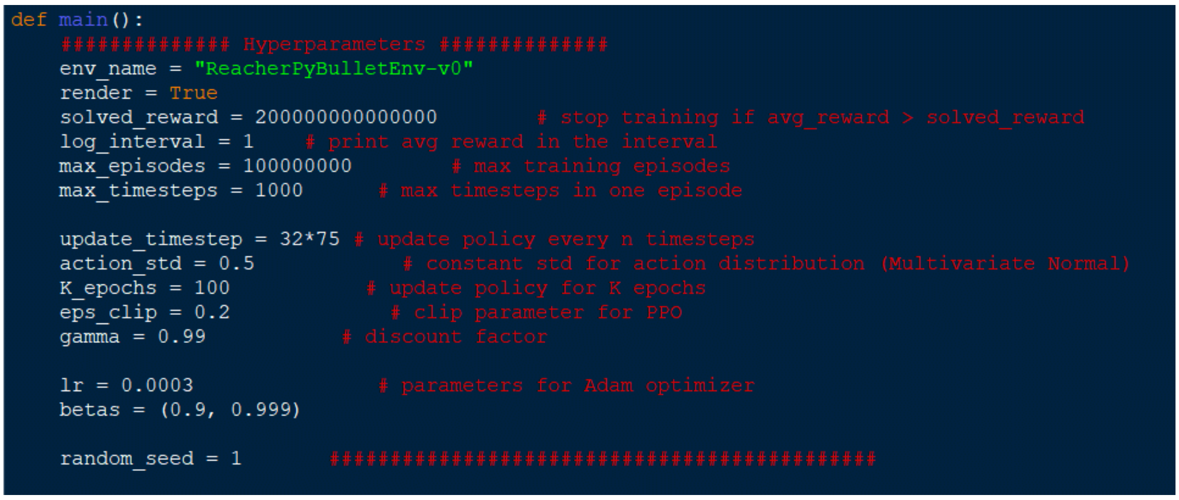
As the purpose of the Leveller robot is to reach locational goals, the reward function has similarities to the one used in the Reacher. As figure 9 of the reward system shows, the previous distance to a goal is used to determine if the fingertip has moved closer to or further away from the goal. As the current distance (potential) is subtracted from the previous, moving closer to the goal makes it smaller than the previous value (potential_old), making the reward positive, and moving further away makes the reward negative. As the goal is to first reach goal 1, and then goal 2, “if” -statements change the reward function depending on if goal 1 has been reached. When the fingertip of the robot gets close to goal 1, it switches to the rewards guiding it towards the second goal. As the route from goal 1 to goal 2 is supposed to be direct, the reward for moving towards goal 2 is divided by a value determined by the distance between the fingertip and the direct route defined as “func”, representing the direct route as a function of the fingertip vertical location. The further away from the direct route, the smaller the reward for moving towards goal 2.



```
    def step(self, action, c):
        reward = 0
        action = action
        self.excavator.apply_action(action)

        p.stepSimulation()

        global goal_switch


        excavator_ob = self.excavator.get_observation(y1, z1, y2, z2, goal_switch)


        potential1 = math.sqrt(((excavator_ob[0] - y1) ** 2 +
                                  (excavator_ob[1] - z1) ** 2))

        potential2 = math.sqrt(((excavator_ob[0] - y2) ** 2 +
                                  (excavator_ob[1] - z2) ** 2))



        m = (z2-z1)/(y2-y1)
        func = m*(excavator_ob[0]-y1)+z1



        if goal_switch == 0:
            reward = self.potential_old1 - potential1

        if potential1 < 0.05 and goal_switch == 0:
            goal_switch = 1

        if goal_switch > 0:
            reward = (self.potential_old2-potential2)/((abs(excavator_ob[1]-func))+0.1)


        self.potential_old1 = potential1
        self.potential_old2 = potential2

        ob = np.array(excavator_ob, dtype=np.float32)

        return (ob, reward, self.done, c, dict())
```

Figure 9. Leveller robot reward system.

Hyperparameters for the training are defined in the “main” -function of the PPO agent, presented in figure 10. Update_timestep defines how many timesteps of data is collected for an update. Action_std is a constant for action distribution. K-epochs defines the number of epochs, meaning the number of passes done for collected batch of learning data. It defines the accuracy on which the policy update is fitted to collected data. Too many epochs cause overfitting, meaning that a single value affects the policy too much and the policy is unable to generalize its learning. Too few epochs on the other hand cause underfitting, meaning that the new policy does not represent the collected data well enough. Clipping parameter eps_clip defines how different the new policy can be compared to the old. Discount factor gamma defines how much the agent values the future rewards in its current state, meaning it defines how motivated the agent is to get good rewards instantly compared to taking an action at a current state to have better rewards in future timesteps. Learning rate lr generally defines the size of a gradient step taken when updating the policy, though when using Adam optimizer, it only defines a maximum value for the learning rate. Betas are Adam optimizer parameters that define exponential decay rates for moment estimates. (Kingma, Ba 2015, p1.) As at the beginning of training, the agent’s behavior is random, setting a random seed (random_seed) makes it possible reproduce results as it gives a starting point for the randomness generator so that the random behavior does not change when reproducing results.



```
def main():
    ############## Hyperparameters ##############
    env_name = "ReacherPyBulletEnv-v0"
    render = True
    solved_reward = 200000000000000        # stop training if avg_reward > solved_reward
    log_interval = 1    # print avg reward in the interval
    max_episodes = 100000000        # max training episodes
    max_timesteps = 1000      # max timesteps in one episode

    update_timestep = 32*75 # update policy every n timesteps
    action_std = 0.5            # constant std for action distribution (Multivariate Normal)
    K_epochs = 100           # update policy for K epochs
    eps_clip = 0.2             # clip parameter for PPO
    gamma = 0.99           # discount factor

    lr = 0.0003               # parameters for Adam optimizer
    betas = (0.9, 0.999)

    random_seed = 1       #############################################
```

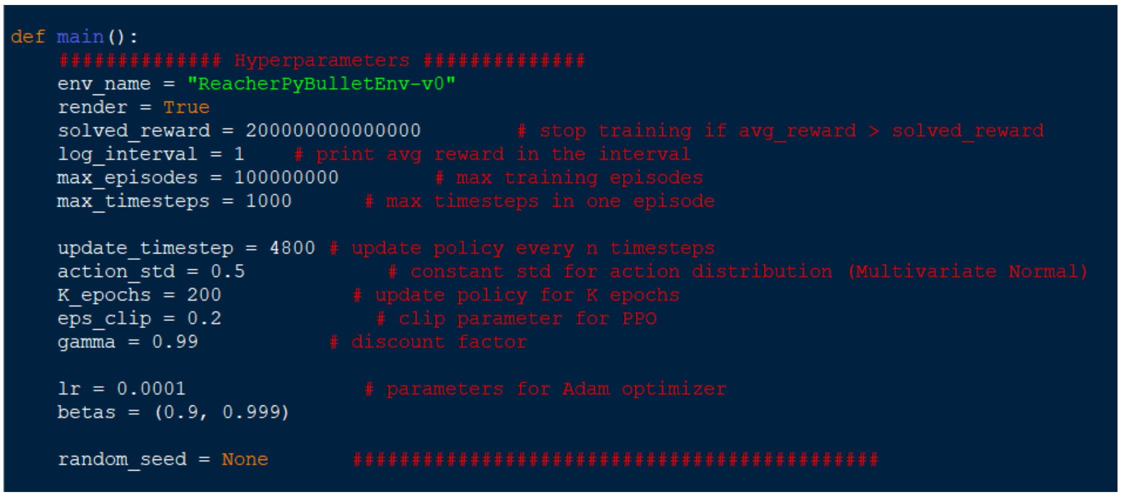
Figure 10. Hyperparameter examples (Barhate 2018.)

4 RESULTS

The results of best learning runs achieved with reasonable amounts of hyperparameter tuning are presented in this section. Episodical rewards for both environments are not solely dependent on performance, but also chance, as the initial state of the environment also affects rewards. For this reason, an average of 32 episodes is used for logging to make the data clearer and less scattered. The hyperparameters tuned were the learning rate, batch size, epochs, and the clipping parameter. The action_std value was left to its default value, and betas were set to default values 0.9 and 0.999 (Kingma, Ba 2015, s1.)

4.1 Reacher environment learning results

The PPO training results were achieved using the hyperparameters presented in figure 11, and the results of 100000 training episodes is presented in figure 12.



```
def main():
    ############## Hyperparameters ##############
    env_name = "ReacherPyBulletEnv-v0"
    render = True
    solved_reward = 200000000000000        # stop training if avg_reward > solved_reward
    log_interval = 1    # print avg reward in the interval
    max_episodes = 100000000        # max training episodes
    max_timesteps = 1000      # max timesteps in one episode

    update_timestep = 4800 # update policy every n timesteps
    action_std = 0.5            # constant std for action distribution (Multivariate Normal)
    K_epochs = 200           # update policy for K epochs
    eps_clip = 0.2             # clip parameter for PPO
    gamma = 0.99           # discount factor

    lr = 0.0001               # parameters for Adam optimizer
    betas = (0.9, 0.999)

    random_seed = None       #############################################
```

Figure 11. Hyperparameters used to train Reacher (Barhate 2018.)

Figure 12 shows 100000 training episodes of the Reacher, and figure 13 shows the behavior in 10000 episodes of the trained PPO agent compared to a PyBullet Gym pre-trained agent. As these figures show, the PPO agent learns stable behavior, but does not overall perform as well as the pre-trained agent, although comparing the two agents is problematic because they use different networks architectures, which may have an impact on their performance. By visually inspecting the simulation behavior, it was observed that the learned behavior does reach the goal well, but not perfectly. The agent does have small problems reaching goals in certain locations and has some small jittering present when the robot should stay completely still. Overall, these problems were present, but in different magnitudes throughout all hyperparameter combinations tested.

Generally, with most tested hyperparameter combinations within normal and reasonable values the algorithm did learn stable behavior, though with varying overall performance with some runs converging to worse policies than others. Naturally extreme values made the algorithm not learn at all or learn poorly, but generally small hyperparameter altering withing reasonable values did not uncover a single crucial hyperparameter that would have had an unexpected positive or negative impact on learning. The effects of hyperparameters were the same as could be expected based on their role in the algorithm. Altering the random seed did not have a significant impact on learning, the algorithm learned consistently regardless of not using a random seed.

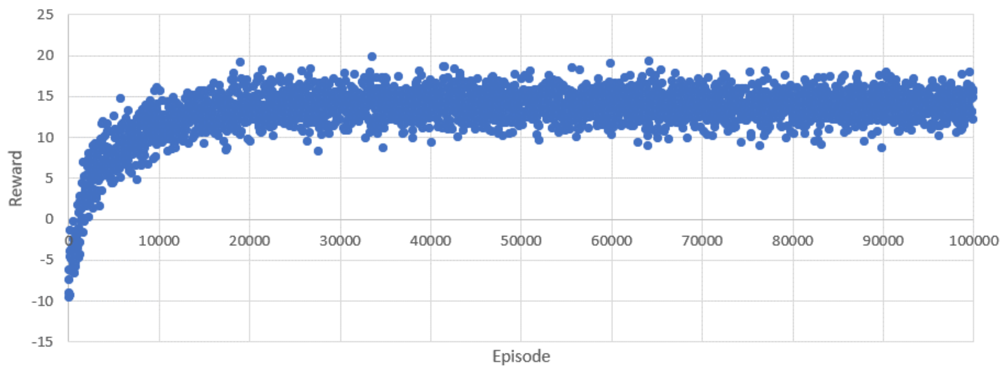


Figure 12. Reacher learning graph.

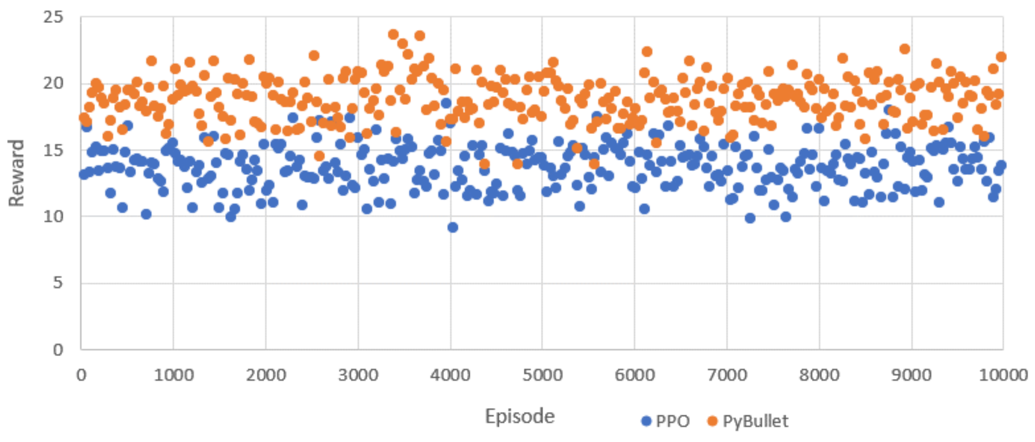
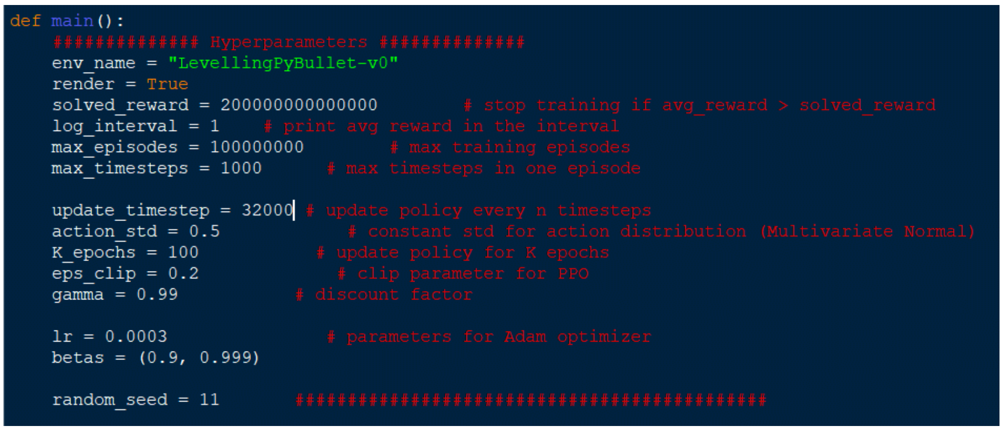


Figure 13. PPO trained agent and pre-trained agent.

4.2 Levelling environment learning results

The Levelling environment was built as groundwork for this thesis. The results of 100000 episodes of training with 1000 timesteps per episode and 32 episodes per batch are presented in figure 14, trained with hyperparameters presented in figure 13. As seen in figure 14, the environment learns stable behavior. By visually inspecting the simulation, it can be seen that these results indeed represent the desired behavior. The robot finds its way to the goals and stays on the direct route between the goals well. After reaching goal 2, the robot also keeps the fingertip in the goal well, but not completely stationary, as some minimal jittering is present.



```
def main():
    ############## Hyperparameters ##############
    env_name = "LevellingPyBullet-v0"
    render = True
    solved_reward = 200000000000000        # stop training if avg_reward > solved_reward
    log_interval = 1    # print avg reward in the interval
    max_episodes = 100000000        # max training episodes
    max_timesteps = 1000      # max timesteps in one episode

    update_timestep = 32000 # update policy every n timesteps
    action_std = 0.5            # constant std for action distribution (Multivariate Normal)
    K_epochs = 100           # update policy for K epochs
    eps_clip = 0.2             # clip parameter for PPO
    gamma = 0.99           # discount factor

    lr = 0.0003               # parameters for Adam optimizer
    betas = (0.9, 0.999)

    random_seed = 11       #############################################
```

Figure 14. Hyperparameters for Levelling environment training (Barhate 2018.)

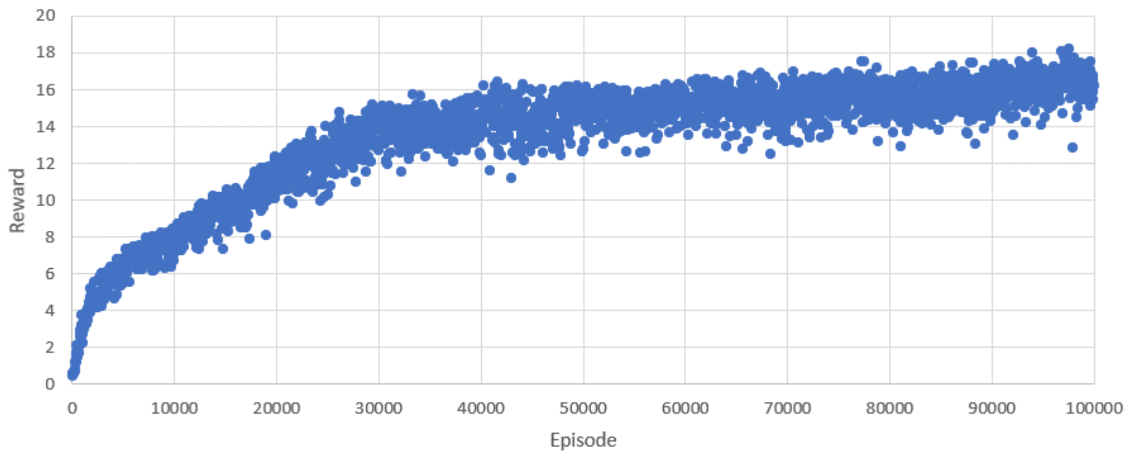


Figure 15. Levelling environment learning results.

Like in the Reacher environment, hyperparameter tuning within reasonable values did not reveal any single crucial hyperparameter that would have drastically changed the learning behavior. Generally, all hyperparameter combinations tried inside reasonable values learned relatively stable behavior with small differences in the final performance, but unlike with the Reacher environment, the random seed used had a major impact on learning. Generally, if a random seed starts the training with very small actions and not enough exploration, it converges to a suboptimal policy as the randomness in the environment is slowly replaced with data collected from insufficient exploration. The badly performing random seeds would often make the agent use only one or two of the robot’s joints sufficiently (visually clearly noticeable) at the beginning of training, leading to one or two of the joints doing nothing. On the other hand, if the random seed chosen made the agent use all three joints during the first batch, the agent seemed to consistently learn the desired behavior among the seeds that made the agent act in this manner during the first batch.

5 ANALYSIS

The results of the two environments suggest that RL used with PPO can learn these kinds of common machinery control tasks. The flexibility and ease of implementation of PPO was well demonstrated, as the PPO agent used for successful training of the two agents required minimal changes to the algorithm between the two environments and was overall flexible and stable in terms of hyperparameter tuning and convergence. Random seeds having a strong impact on learning with the Leveller are not a major issue in these kinds of tasks, as the poor learning results were only caused by a lack of exploration caused by a bad seed. Multiple seeds were tested, and all of them that visually showed enough exploration in the first episode learned stable behavior. On the other hand, in different or more complicated tasks, visually verifying sufficient exploration might be less straightforward.

The reward function used in training the Leveller was a result of multiple different approaches and reward values tested. It was observed that a reward function that would lead to the highest possible reward with correct behavior, does not necessarily mean that correct behavior is learned. For example, motivating the agent to stay on the direct path by using the distance from the path as a negative reward led to the agent using policies that avoided reaching goal 1 entirely and not exploring what would happen by going towards goal 2 in a straight line. This means that it is important to carefully consider what different types and sizes of rewards can motivate the agent to do, and in what situations it is better to use negative reward as “punishment” instead of positive reward as encouragement.

5.1 Confluence to previous research

The stability, flexibility, and ease of implementation of PPO demonstrated in the process of training the two environments is in line with previous research showing similar benefits for PPO in many kinds of tasks.

5.2 Objectivity

The data collected in this thesis is a result of trial-and-error parameter tuning for two, simple environments and use of an open-source optimizer and many libraries and toolboxes for python. Therefor assessing the performance of PPO is limited to a certain set of software.

5.3 Validity and reliability

The results of the research done in this thesis are supported by visually observing the robots' behaviors. The visual footage combined with the results data collected shows, that the RL-scheme clearly does learn.

5.4 Answers to research questions

What are the current state-of-the-art solutions in reinforcement learning?

Making a clear distinction between state-of-the-art and non-state-of-the-art RL-algorithms is not necessarily straightforward as the field of research is developing rapidly. Due to the newness, excellent performance, ease of implementation and stability of PPO, it is, for this thesis, considered as one of the most current and significant state-of-the-art solution.

How hyperparameters affect RL learning?

With the two environments trained, hyperparameters within reasonable values had only a small impact on the end result, though naturally the learning speed was affected by factors such as learning rate, number of epochs and batch size. Generally, the hyperparameters had an expected effect on training and the PPO agent adjusted well to different hyperparameter combinations.

Can reinforcement learning be effectively utilized in machinery control?

Based on the two simulated machinery control tasks, RL could be effectively utilized in machinery control as an asset with precaution. As on-policy RL-algorithms learn by taking random actions and improving behavior based on collected data, a detailed simulation would be, in many cases, needed as making a real machine take completely random actions is not usually a realistic option. For optimization purposes, a pre trained controller could be used to control the machine so that the behavior is safe, but the RL-algorithm could keep learning more optimal behavior with all real-world variables present. For this kind of purpose, the stability of PPO would be crucial, as an unstable algorithm could diminish the pre-trained behavior too. When using PPO for optimization purposes, it should be acknowledged that there is not necessarily a guarantee of convergence, as the motivation of the algorithm depends on the magnitudes of different reward values that can be difficult to determine in advance if there are multiple different factors to optimize. Optimization is more problematic than simply learning a certain task, as visually assessing if the behavior is optimal is not as straightforward as just observing the main purpose of the machine. Overall, RL, especially used with PPO could have great benefits in machine control automation not only because it is able to learn the tasks, but also because it is able to do so with such flexibility regarding hyperparameters, and with such simple reward systems. Teaching several different tasks to the same machine would in best cases only require changes to a few lines of code between the tasks. With the types of machines trained in this thesis, using conventional automation would likely be considerably more difficult.

5.5 Further research

Adding the aspect of efficiency and optimization to the Leveller environment by adding an electricity cost and other optimization parameters to the reward system, and then observing how the presence and magnitude of these parameters affects learning. Overall, more testing with different, more accurate, complicated, and realistic models using different kinds of control types, as well as researching ways to eliminate the unwanted jittering present in both environments. Ultimately, testing the behavior of physical machines, such as miniature models, with pre-trained agents and continuous learning and optimization.

6 SUMMARY

This thesis focuses on exploring the potential of state-of-the-art reinforcement learning in machinery control by training two simulated robots representing common machinery control tasks using proximal policy optimization. It was observed that PPO was able to learn correct and stable behavior with small imperfections. It was concluded in accordance with other available research, that PPO generally is flexible and stable in terms of hyperparameter tuning. Based on these results from the simulated robots, it was concluded that RL has, with precautions, potential as an asset in machinery control automation.

7 BIBLIOGRAPHY

Amiri, R., Mehrpouyan, H., Fridman, L., Mallik, R., Nallathan, A., Matolak, D. 2018. A Machine Learning Approach for Power Allocation in HetNets Considering QoS. IEEE International Conference on communications (ICC). IEEE, 2018.

Barhate, N. 2018. PPO-PyTorch. Open-source repository. [Cited 18.4.2021]. Available at https://github.com/nikhilbarhate99/PPO-PyTorch.

Bhatnagar, S., Sutton, R., Ghavamzadeg, M., Lee, M. 2009, Natural Actor-critic Algorithms. Automatica (Oxford) 45.11 (2009). p. 2471-2482.

Dangeti, P. 2017. Statistics for Machine Learning: Build Supervised, Unsupervised, and Reinforcement Learning Models Using Both Python and R. 1st ed. Packt Publishning, 2017. Print.

Delalleau, O., Peter, M., Alonso, E., Logut, A. 2019. Discrete and Continuous Action Representation for Practical RL in Video Games. Ubisoft La Forge.

Ding, Z., Dong, H. 2020. Challenges of Reinforcement Learning. Deep Reinforcement Learning. Singapore: Springer Singapore, 2020. p. 2049-272.

Ellenberger, B. 2018. pybullet-gym. Open-source repository. [Cited 1.6.2021]. Available at: https://github.com/benelot/pybullet-gym.

Engstrom, L., Ilyas, A., Santukar, S., Tsipras, D., Janoss, F., Rudolph, L., Madry, A. 2019. IMPLEMENTATION MATTERS IN DEEP POLICY GRADIENTS: A CASE STUDY ON PPO AND TRPO. ICLR 2019.

Kingma, P., Ba, L, J. 2015. Adam: A Method For Stochastic Optimization. ICLR 2015.

Kunze, l., Roehm, T., Beetz, M. 2011. 2011 IEEE International Conference on Robotics and Automation, 2011-05. p. 5589-5595.

Levine, S., Kumar, A., Tucker, T, Fu, J. 2020. Offline Reinforcement Learning: Tutorial, Review, and perspectives on Open Problems. UC Berkeley, Google Research, Brain Team.

Nandy, A., Manisha, B. 2017. Reinforcement Learning: With Open AI, TensorFlow and Keras Using Pythin. Berkeley, CA: Apress L. P 2017. Print.

OpenAI 2016. Website. [Cited 23.8.2021]. Available at: https://gym.openai.com/docs/

Coumans, E., Bai, Y. 2021. PyBullet Quickstart Guide. [Cited 23.8.2021]. Available at: https://docs.google.com/document/d/10sXEhzFRSnvFcl3XxNGhnD4N2Sedqw-dAvK3dsihxVUA/edit#heading=h.2ye70wns7io3

Rebala, G., Ajay, R., Sanjay, C. 2019. An introduction to Machine Learning. Chanm: Springer International Publishning AG, 2019. Print.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. 2017. Proximal Policy Optimization Algorithms. OpenAI.

Wollsclaeger, M,. T, Sauter., Juergen, J. 2017. The future of Industrial Communication: Automation Networks in the Era of the internet of Things and Industry 4.0. IEEE insustrial electronics magazine. pp. 17-27.

Zhang, H., Yu, T. 2020. Taxonomy of Reinforcement Learning Algorithms. Deep Reinforcement Learning. Singapore: Springer Singapore, 2020. pp. 125-133.

ANNEX

Repository with Leveller environment files and agents used is available at: https://github.com/shjakebu/Leveller_Reacher

PPO agent licence (Nikhil Barhate 2018)

PYBULLET GYM licence (Benjamin Ellenberger 2018)