DEVELOPMENT OF A ROBUST CONTROLLER FOR 2 WHEELED SELF BALANCING ROBOTS USING DEEP REINFORCEMENT LEARNING

Yash Sahijwani
19095110, B-Tech
Department of Electronics Engineering
Indian Institute of Technology (BHU)
Varanasi-221005
yash.sahijwani.ece19@itbhu.ac.in

Karan Singh Khati
19095051, B-Tech
Department of Electronics Engineering
Indian Institute of Technology (BHU)
Varanasi-221005
karan.singhkhati.ece19@itbhu.ac.in

Supervisor: Dr. Sanjeev Sharma
Assistant Professor
Department of Electronics Engineering
Indian Institute of Technology (BHU)
Varanasi-221005
sanjeev.ece@itbhu.ac.in

ABSTRACT

One of the key challenges in robotics and control systems, is the development of a robust controller capable of handling perturbations in the surrounding environment especially for systems which are inherently unstable. However, in recent years model-free Deep Reinforcement Learning based methods have proven to be quite successful in building robust enough controllers without having to model the perturbations in the environment. In this work, we aim to develop a controller capable of both self-balancing a 2 wheeled robot, also known as TurtleBot, and making it move in its environment by modelling it as a Deep RL optimization problem.

1. INTRODUCTION

Manually controlling a 2 wheeled robot to self-balance is a strenuous task, yet alone make it move. Although there are several controllers like PID[1], LQR[2] and MPC[3] that can be used for this task, they require careful tuning of parameters, such as the proportional gain in PID. These controllers also require a heavy knowledge of the dynamics and the underlying equations. Hence, we use Deep Reinforcement Learning, in particular the DQN[4] (Deep Q Networks) algorithm, to develop a controller policy for making the bot self-balance as well as move forward and backward. We model the problem as a Markov Decision process where the observations are the tilt of the bot with respect to the ground, the angular velocity about the continuous joint which connects the wheels to the torso of the bot and finally the speed of the bot in the direction of motion. We utilize the concept of continual learning by further training the self-balancing policy to give us the policy capable of locomotion.

2. DEFINITIONS AND TERMINOLOGY

a. **Agent** – The decision maker for the entire process. Analogous to the brain in humans.
b. **States** – Observations in the environment that the agent has access to. Can be visual as well as simple data, like readings from an IMU (Inertial Measurement Unit)
c. **Reward** – A scalar feedback signal which indicates how well the agent is doing in accomplishing the necessary task.
d. **Policy** – Agent's behaviour. Mathematically, it is a mapping from the state space to the action space.
e. **Value Function** – A function which predicts the cumulative future reward when the agent is in a particular state.
f. **Action Value Function** – A function which predicts the cumulative future reward when the agent is in a particular state and takes a particular action.

3. BACKGROUND

The goal of solving sequential decision-making problems is to select actions that maximize the total future reward. However, this automatically gives us a constraint that the current best-looking action might not give us the maximum possible total future reward because of the simple idea, that actions may have long term consequences whereas looking only at actions from a single step is a short sighted and greedy approach. Approaches like Dynamic Programming require a proper representation of the model of the environment which is very difficult to obtain in real world scenarios. RL solves this problem by giving algorithms which do not require any form of model of the environment and simply learn by interacting with it. This is analogous to the way a human learns about its surroundings and hence gives us another method to control robotic systems.

The self-balancing 2 wheeled robot that we have used for our experiments is a commonly used robot called TurtleBot[5], which was developed at Willow Garage in 2010 and a large amount of research has gone into understanding its dynamics and optimally controlling the bot. Due to the robustness of RL policies and the simplistic structure of the bot, the learned policies can be easily transferred to real hardware.

4. METHOD

Table of Contents

This project can be roughly divided into 5 parts:
1. Making a gym environment with TurtleBot in simulation.
2. Developing a classical controller with PID to compare with RL algorithms.
3. Training a policy with DQN algorithm to self-balance the TurtleBot.
4. Further continuing the training with changed reward for locomotion.
5. Testing the trained policies in simulation.

Making a gym environment with TurtleBot in simulation

PyBullet[6] is a Python module for robotics simulation and machine learning, with a focus on simulation-to-real transfer. With PyBullet you can load articulated bodies from URDF (Unified Robot Description Format), SDF (Simulation Description Format), MJCF (Multi-Joint dynamics with Contact XML file) and other file formats. PyBullet provides forward dynamics simulation, inverse dynamics computation, forward and inverse kinematics, collision detection, etc. A simulation in the Bullet physics engine can be easily brought into the real world with some minor tweaks. As doing it on hardware was not possible due to lockdown, so we used PyBullet as an alternative. URDF file of

the **TurtleBot** was used in the simulator as it is free, easy to use, and can be easily brought into real-world by 3D printing the STL files of robot parts.

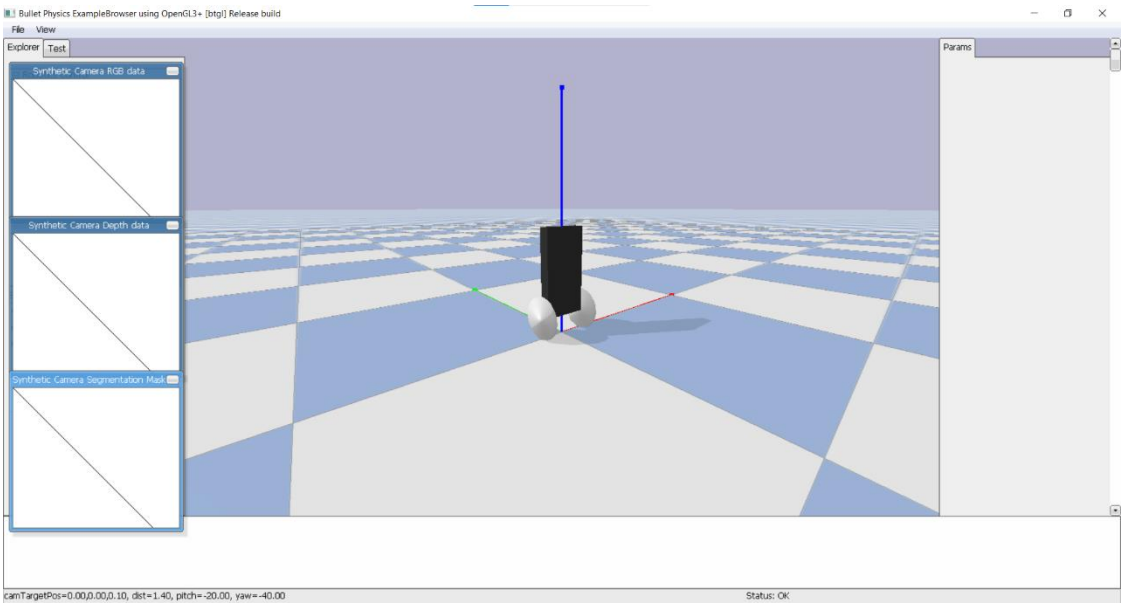

Figure 1: TurtleBot URDF loaded in PyBullet. The hardware of the bot consists of a black torso connected with 2 white wheels by continuous joints, one for each wheel.

State Space, Action Space and Reward Signal

In our environment, the state space comprised of 3 continuous variables – the tilt of the bot with respect to the vertical, the angular velocity of the COM of the bot about the axis of the wheels and the linear velocity of the bot in the direction of motion.

Our action space was kept discrete due to the simplicity of the problem. The dimensionality of the action space was 9 and basically consisted of increments/decrements in the velocity of the wheels. These actions were -1, -0.5, -0.2, -0.01, 0, 0.01, 0.2, 0.5, 1 where negative denotes that the velocity of the wheels was decreased, whereas positive means velocity of wheels was increased.

The reward signal was a weighted average of the cosine of the tilt of the bot with the vertical and the velocity of the bot. The second term was introduced only for locomotion and wasn't involved in the development of the self-balancing controller.

$$R = W_1 \cos(\theta) + W_2 V_t$$

Here, W1 and W2 are the weights, theta represents the tilt of the bot with the vertical and Vt represents the velocity of the bot.

Developing a classical controller with PID (Proportional Integral Derivative)

Proportional-Integral-Derivative (PID) control is the most common control algorithm used in industry and has been universally accepted in industrial control. The popularity of PID controllers can be attributed partly to their robust performance in a wide range of operating conditions and partly to their functional simplicity, which allows engineers to operate them in a simple, straightforward manner. The basic idea behind a PID controller is to read a sensor, then compute the desired actuator output by

calculating proportional, integral, and derivative responses and summing those three components to compute the output. Hence, PID is essentially a closed loop controller.

In this project, we use the PD components to make a controller capable of self-balancing the bot. The error we take here is naturally the tilt of the bot, and hence the derivate of the error equals the angular velocity of the bot about the axis of the wheels. Hence, the controlling equation becomes:

$$V_t = K_p(error) - K_d \frac{\mathrm{d}(error)}{\mathrm{d}t}$$

Here, Vt refers to the velocity of the wheels of the robot, Kp is the proportional gain, Kd is the derivative gain, and error is the angle of tilt of the bot with the vertical.

The proportional component depends only on the angle of tilt of the bot whereas the derivative response is proportional to the rate of change of this tilt. Though this controller is reliable, it takes a lot of tuning of the proportional and derivative gains and still can result in a slight deviation from the required behaviour. Moreover, in real world scenarios, such a controller is not at all robust and is susceptible to a large amount of noise.

Training a policy with DQN algorithm

The DQN algorithm was proposed by Mnih et. al. It is an optimization of the original Q-learning algorithm by using neural networks as function approximators for learning the Q-values. Here, Q-values refer to the action value function which defines the total cumulative future reward that our agent will get given that it is in a particular state and takes a particular action and is represented by Q(s, a), where s is the state and a is the action. Hence, we kept the dimensionality of the action space small, so as to solve the problem of the curse of dimensionality, which can lead to very large training times with not much success.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

In the DQN algorithm, the agent interacts with the environment by using a policy which is epsilon-greedy, i.e. given the Q(s, a) values, it selects the action with highest Q(s, a) with probability ε/m + (1- ε), and rest of the actions with probability ε/m, where m is the total number of actions. These transitions are stored in a replay memory buffer, after which a random minibatch of these transitions is sampled (this sampling is done at every step), and targets are defined according to the equation given, which considers sum of the immediate reward and the discounted value of the maximum Q-value over all the actions for the next state. Note that this Q-value is a fixed Q-target and is not the same Q-value

that is being updated, otherwise it will result in an unstable learning. Then gradient descent is performed by taking the error between the target and the current Q-value for that particular state action pair. Also, the ε is decreased to a small constant over the course of training, so that the search space decreases over time. Hence, DQN uses experience replay and fixed Q-targets. After a fixed set of steps, the fixed Q-targets are replaced by the learned Q-values learned. This is repeated to ensure a stable and sample efficient training.

Neural Network Architecture

In our experiments we have used a Neural Network as a function approximator for the Q-values, which are learned by gradient descent on the parameters of the neural network. The architecture of the neural network is as follows:

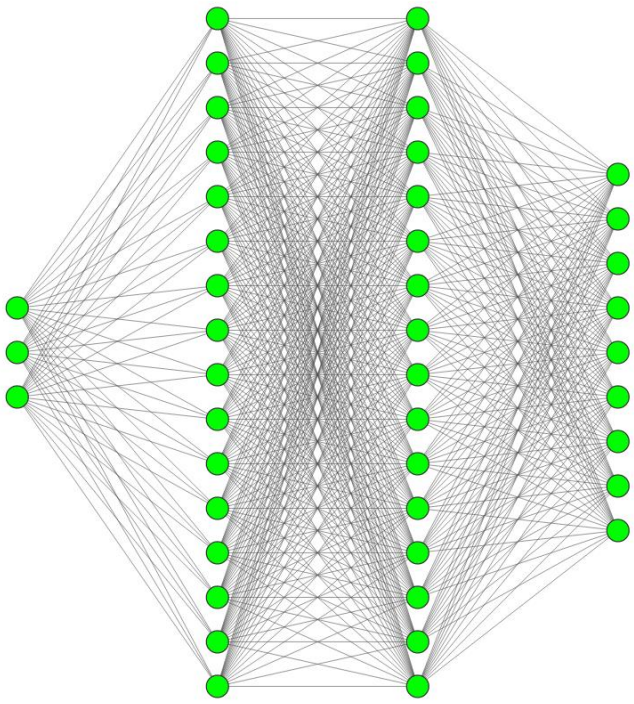

Figure 2: A Fully Connected Network with 3 inputs and 9 outputs, where the 3 inputs refer to the 3 dimensions of the state space, and the 9 outputs are the action values for the different actions, for that particular state.

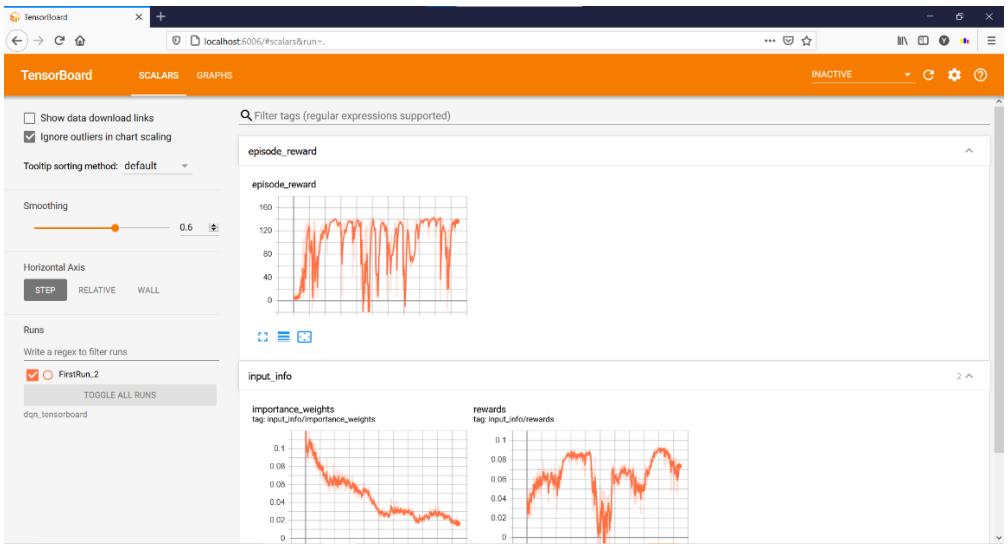

Figure 3: TensorBoard is an easy-to-use framework for visualizing plots of scalars, as well as the computational graph of our network.

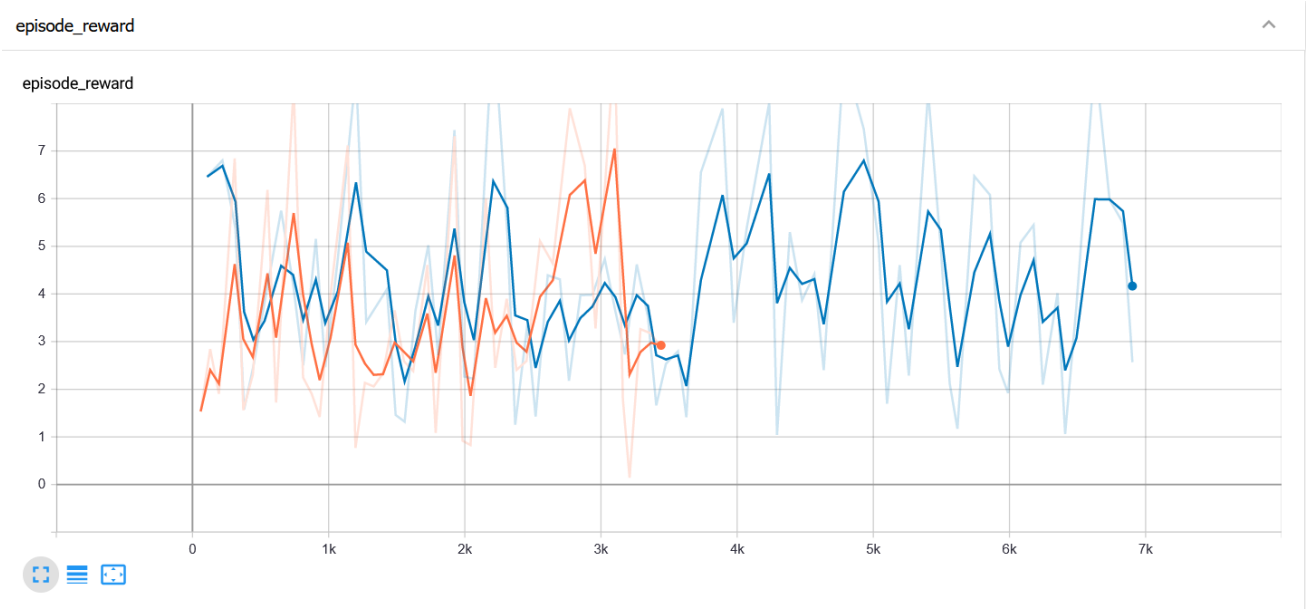

Figure 4: Our training slowly converged, and after nearly 1.4x10^5 steps it was able to get very high rewards.

Continual Learning for Locomotion

For learning a policy capable of locomotion, the previous self-balancing policy learnt by the algorithm was loaded, the initial ε was kept to be 0.4 instead of 1, so that initially it does not take random actions, rather takes actions based on the learnt policy. This helped in speeding up the process of learning locomotion, as the self-balancing part was already present due to the policy.

The reward was changed to include the speed of the COM of the bot as well in a single direction, but apart from that the entire environment was kept the same.

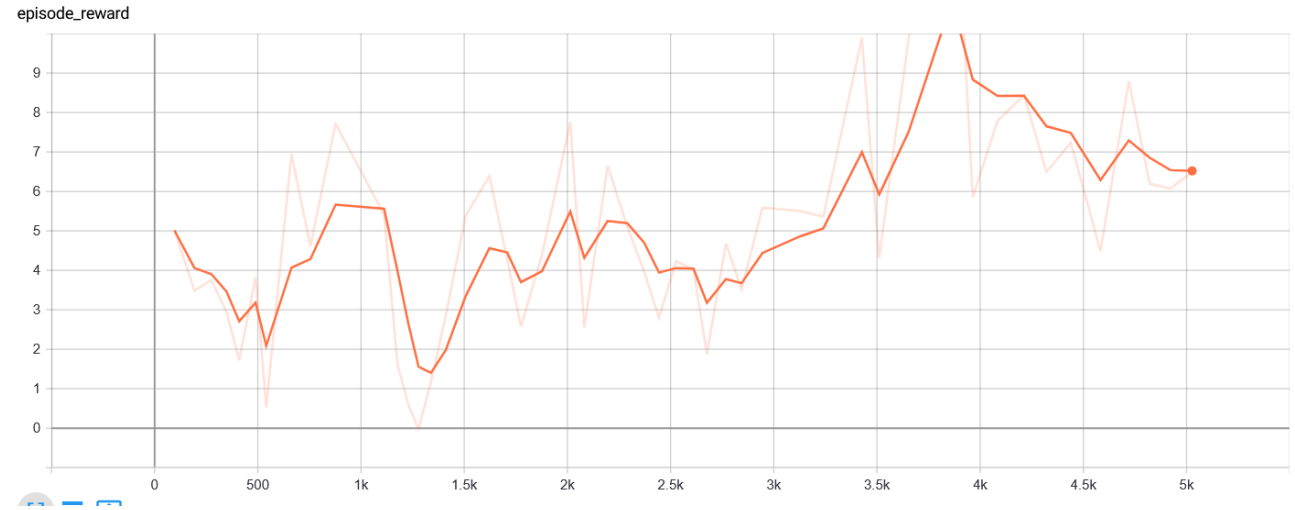

Figure 5: Our training for the locomotion policy slowly converged, and after nearly 5l steps it was able to get very high rewards. This was because the previous policy for self-balancing was further trained, rather than training a new policy from scratch.

Testing the trained policies in simulation

The policies were tested on the bot, and the self-balancing policy was able to balance the bot vertically for the entire duration of the episode. The locomotion policy also gave good results as the bot was able to move in a single direction.

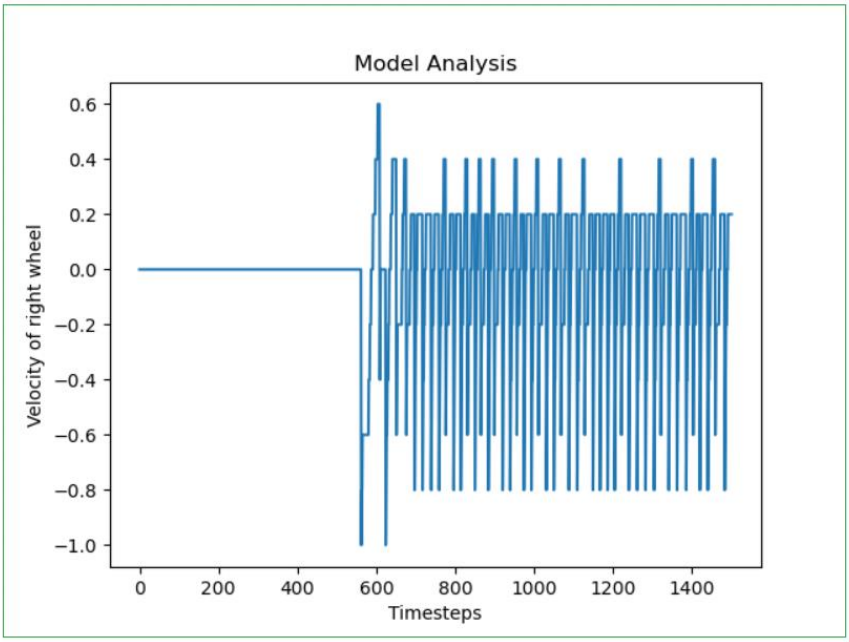

Figure 6: Analysis of the wheel velocity of the self-balancing policy. Initially, the velocity is zero, but as soon as the bot starts falling towards the ground the wheels start moving in order to balance it.

Applications

The controller we have designed can be extended to a large variety of robots, and made more robust by training on uneven surfaces, making it capable of going to places where normal autonomous driving wheeled robots can't go. However, there is still a great unfilled void in reliable automation of such systems. Hence, the need for efficient human control is inevitable and a suitable example would be in the Military – Unmanned Army, bots, etc. These control systems could greatly enhance the capabilities and deployment of robots in hazard prone areas and disaster recovery tasks.

Future Work

The current controller can be extended in order to make the bot make turns, navigate in an uneven terrain (by training it on an uneven surface) and combining the bot with vision systems, such as object detection, pose estimation etc. to improve the navigation system. Sim-to-reality transfer can also be done, in order to transfer this controller to actual hardware.

Conclusion

In this work we take a novel approach, to address the problem of robot controller design. We show the robustness and scalability of our idea by validating our results in the PyBullet physics simulation engine. We are confident that it could be readily transferred to a real-life robot as we experimented with the 3d model of a prefabricated real-world bot named TurtleBot. Though the model was tested on a single robot it is nowhere limited in its capabilities and could easily generalize a variety of such robots irrespective of the design, topology, and other mechanical parameters of the robot. The idea is not limited to just proprioceptive control and could be easily extended towards a vision-based controller by adding a camera structure.

References

[1] Paz, Robert. (2001). The Design of the PID Controller.
[2] Jose, Akhil & Augustine, Clint & M, Shinu & Chacko, Keerthi. (2015). Performance Study of PID Controller and LQR Technique for Inverted Pendulum. World Journal of Engineering and Technology. 03. 76-81. 10.4236/wjet.2015.32008.
[3] Han, Jixia & Hu, Yi & Dian, Songyi. (2018). The State-of-the-art of Model Predictive Control in Recent Years. IOP Conference Series: Materials Science and Engineering. 428. 012035. 10.1088/1757-899X/428/1/012035.
[4] Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David & Graves, Alex & Antonoglou, Ioannis & Wierstra, Daan & Riedmiller, Martin. (2013). Playing Atari with Deep Reinforcement Learning.
[5] https://www.turtlebot.com/
[6] E Coumans, Y Bai, PyBullet, a Python module for physics simulation in robotics, games and machine learning
[7] https://tensorboard.dev/
[8] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., et al.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015) Software available from tensorflow.org
[9] https://stable-baselines.readthedocs.io/en/master/