Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Learning Dynamic System Control on a Data Driven Model

Aleksandr Barinov

Supervisor: Ing. Teymur Azayev
May 2022

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Barinov Aleksandr**	Personal ID number:	**483581**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Learning Dynamic System Control on a Data Driven Model

Bachelor’s thesis title in Czech:

Učení řídicího algoritmu na datově získaném modelu

Guidelines:

The goal of this project is to examine the possibilities of learning useful control algorithms on a (virtual) learned model, with data gathered on a real physical platform.
The student is to specifically fulfill the following goals:
1. Get familiar with the RC buggy platform and given data of manually driven trajectories, learn a forward model which predicts next step velocities given the current state vector and action input. A neural network variant or other suitable model can be used. Create a visual wrapper and environment with a Open GYM (https://gym.openai.com/) interface for the learned model using the Pybullet (https://pybullet.org) physics engine package.
2. Address the issues of input data noise and possible incompleteness of state observability and implement and comment on the use of filtration, as well as at least two of the following variants: a) Adding observations from previous timesteps as input to the predictor b) recurrent neural networks c) temporal convolutions d) Transformers .
3. Use an off the shelf reinforcement learning such as A2C [4] along with a suitable neural network architecture to learn a control policy on the data driven environment created in point 1.
The control policy has to accept a sequence of waypoints and auxilliary inputs such as target velocity, along with the state vector at each timestep and produce the given motor and turning actions which are ready to be sent to the physical motor controllers.
4. Deploy and analyze the performance of the learned control policy on the real platform and compare with the performance of a policy which was learned on a close-matching buggy model available in the Pybullet package
https://github.com/erwincoumans/pybullet_robots.

Bibliography / sources:

[1] David Ha, J. Schmidhuber, Recurrent World Models Facilitate Policy Evolution, Advances in Neural Information Processing Systems 31(NeurIPS 2018).
[2] Ibarz et al., How to train your robot with deep reinforcement learning: lessons we have learned, The International Journal of Robotics Research, 2021.
[3] L Ljung, Nonlinear System Identification, IEEE Control Systems Magazine, 2019, J Schoukens.
[4] Volodymyr Mnih et al., Asynchronous Methods for Deep Reinfrocement learning, 2016.

Name and workplace of bachelor’s thesis supervisor:

Ing. Teymur Azayev   Vision for Robotics and Autonomous Systems FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **12.07.2021**   Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **19.02.2023**

Ing. Teymur Azayev	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I thank my supervisor Ing. Teymur Azayev for an opportunity to work on an exciting project, to learn state-of-the-art machine learning techniques and to conduct research under his guidance.

I am grateful to the CTU for all the knowledge I acquired during my study.

Special gratitude goes to Ing. Pošík Petr Ph.D. for his exceptional compassion and support in my darkest moment.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of the university theses.

Prague, 1 May 2022

Abstract

The aim of this work is to empirically demonstrate that a neural network control policy learned on a data-driven nonlinear dynamical system may achieve better simulation-to-real world transfer than a control policy learned on a state-of-the-art physics engine MuJoCo. Both control policies are at first evaluated in a simulation environment they were trained in and next on a physical platform. In addition, we attempt to learn a robust policy able to control nonlinear dynamics system on random curved trajectories with close to human driver performance.

Experiments show that the performance of the policy learned on the data-driven model suffers significantly less from transfer to the real world than that of the policy learned on the MuJoCo engine. Deployment of learned policies on a validation set of trajectories demonstrates high close to human driver performance and a decent ability to generalize.

Code is freely available on github `https://github.com/barinalex/thesis`.

Keywords: data-driven model, control policy, reinforcement learning

Supervisor: Ing. Teymur Azayev

Abstrakt

Cílem teto práce je empiricky demonstrovat, že založená na neuronových sítích řidicí strategie naučena na datově získaném nelineárním dynamickém systému může dosahnout lepšího přenosu ze simulace do skutečného světa než řidicí strategie naučena na state-of-the-art fyzikálním motoru MuJoCo. Obě strategie nejprv ohodnocene ve simulačním prostředí, na kterém byli trénovani a pak na fyzické platformě. Kromě toho se pokoušíme naučit robustní strategije schopnou kontrolovat nelineární dynamický system na náhodných zakřivených trajektoriích s blízkým ke lidskému řidiču výkonem.

Pokusy ukazují, že výkon strategie naučene na datově získanem modelu trpí vyznamně míň za přechodu do skutečného světa než strategie naučena na MuJoCo enginu. Nasazení naučenych strategií na validační sadu trajektorií demonstruje vysoký blízký ke lidskému řidiču výkon a slušnou schopnost ke generalizace.

Kód je zdarma přistupný na githubu `https://github.com/barinalex/thesis`.

Klíčová slova: datově získaný model, řídicí strategie, zpětnovazební učení

Překlad názvu: Učení řídicího algoritmu na datově získaném modelu

Contents

Figures

Tables

Chapter 1
Introduction

1.1 Motivation

Modern Deep Reinforcement Learning (RL) algorithms allow us to achieve human-level performance in various domains. Control of dynamic systems is one of those.

Despite recent successful applications of RL in the field of nonlinear dynamical systems, e.g., [1], [2], it is still a novelty. Hence, more research and experiments in different areas where autonomous control might be a good fit are needed.

There is a challenging problem with RL, though. In order to learn a useful policy, RL requires a lot of agent-environment interactions. Often these interactions could not be obtained directly from the real platform. At the early stage of policy learning, the risk of damaging the robot or even the environment is quite high. Therefore, we are limited to learning in a simulation environment, which has its disadvantages. The main problem with this approach is a simulation-to-real world transfer at the deployment stage. Physics engines that are broadly used to simulate your specific physical platform may subtly differ from reality and even significantly differ in some edge cases. When deploying the learned policy in the real world, these differences might rapidly lead to failure.

One method to solve this problem is to manually create a highly accurate model of your physical platform. Although, this solution requires a lot of time when dealing with complicated robotic systems.

Another solution is to train a data-driven model that would approximate the dynamics of your platform. In the following chapters, the approach to utilizing Neural Networks for this purpose is described in detail.

1.2 Goals

In this work, a remote control car following a trajectory in a 2-dimensional space is considered. A trajectory is simply a set of points that are called waypoints. The trajectory following the task is demonstrated in the figure 1.1. Red points depict what an agent sees, and green is invisible to the agent, where the agent is an entity that makes decisions or, in other words, acts. As the car moves forward, the visible horizon changes, but the number of waypoints visible to the agent always stays the same. The ultimate goal of the agent is to close as many waypoints as possible while maintaining a minimal deviation from the trajectory, which is to be defined later on.

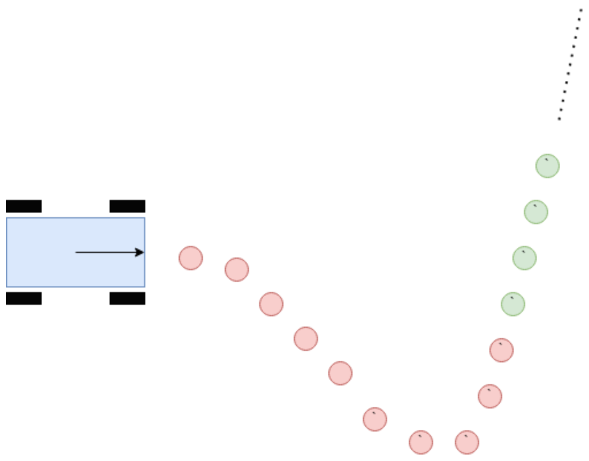

Figure 1.1: Schema of a car following a trajectory that is defined as a set of waypoints (red and green points). Red points form the visible horizon for the agent.

Our final goal is to deploy control policies learned on the MuJoCo engine and on the data-driven model-based engine (the data-driven model-based engine farther in the text is referred to as a **custom engine**) in the real world and analyze the performance of both policies. In order to accomplish that, we first need to learn the data-driven model and prepare environments for Reinforcement Learning.

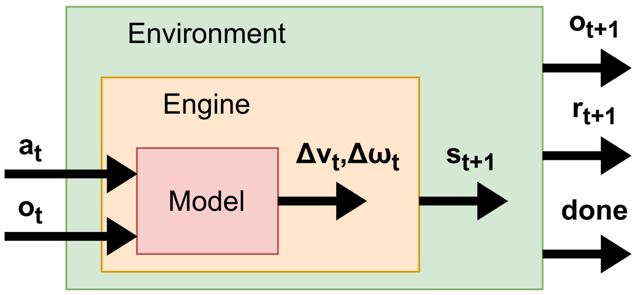

Figure 1.2: The schema of an environment for a policy learning agent-environment loop. The environment includes an engine and a model. The model predicts velocity changes. The engine computes the next timestamp state. The environment returns observation, reward, and a binary variable representing the end of an episode.

The relation between a model, an engine, and an environment is shown in the figure 1.2. The environment is used in an agent-environment interaction loop to learn a policy. Given actions, it provides us with observations, rewards, and the state of an episode depicted as 'done' - if done is true, an episode has ended. The iterative process of feeding the environment or a stand-alone engine with actions is called a **simulation**. The engine is a part of the environment, and it computes the next state given actions. The model is an internal part of our custom engine, and it is responsible for velocity change computations. The model is a neural network trained on the data gathered from the real platform. For clarification, considering the schema 1.2 model would compute velocity changes given new actions that are throttle and steering and observations that are linear and angular velocities, whereas the engine computes new velocities based on changes provided by the model and a new position and orientation at the timestep $t+1$ given new velocities.

The main focus of this work is on the development of the data-driven model that will outperform the MuJoCo engine for our specific physical platform. Different neural network architectures are trained, tuned, and compared for the task. The base model is a Multiple Layer Perceptron (MLP). A more advanced model is a Temporal Convolutional Neural Network (TCNN).

The main contribution of this work lies in empirical confirmation of the hypothesis that neural networks approximating nonlinear dynamics could provide better simulation-to-real world transfer than the state-of-the-art MuJoCo physics engine.

The secondary contribution is a demonstration of a close to the human performance of a control policy learned using Reinforcement Learning on general trajectories of various difficulties. We also contribute by developing pipelines for the data-driven model training and the control policy learning for the real physical platform. Code was made freely available.

1.3 Outline

In the following chapter, we briefly review published papers related to our problem. In the preliminaries chapter, we refresh our knowledge of Reinforcement Learning and the mathematics behind it - Markov Decision Processes.The hardware chapter describes the physical platform we are gathering data from and running real-world experiments on. We continue with this work's task definition, description of our approach to generating random trajectories, and definition of an environment used for learning a control policy. In the data-driven model chapter, our approach to approximate nonlinear dynamics with neural networks is described in detail, from data gathering to model architecture. The next chapter contains a description of conducted experiments and the results we have got. In the last two chapters, we discuss the results and experiments and make the final conclusion.

Chapter 2
Related work

As an introduction to the problem of data-driven modeling of nonlinear systems, we could refer to the paper [3]. It provides a good intuition for an engineer approaching a novel problem along with practical knowledge such as experiment design.

In the second work in our list [4] authors address the problem of learning a dynamics model and then using it as a predictive model for an agent in a model-based Reinforcement Learning. Experiments demonstrate that the task of navigating a race car could be solved using their approach.

The work [1] describes learning of a control policy using Reinforcement Learning for a high-speed drifting. The setup of the problem in this work is quite similar to ours but more restricted. Authors formulate drift control as a trajectory following task and consider the case of high-speed 80-128 km/h drifting on predefined maps. For learning a control policy, they use model-free Reinforcement learning, and this approach delivers promising results. Although, tests are conducted only in a simulation, never in the real world. Therefore, the usability of the used approach is questionable for the real-world use case.

One more work that demonstrates the successes of a control policy Deep Reinforcement Learning is [2]. The approach is again different from ours in that the learning is conducted in a physics simulation without any prior data collection from the real platform. Authors aim to learn a control policy able to generalize well to changing conditions of the real world and robust to a sim-to-real transfer by varying physics engine parameters during the learning process. Their results show state-of-the-art performance for legged robots.

Both of the mentioned above works demonstrate the successful usage of Deep Reinforcement Learning in the field of robot autonomous control. Their methods, though, do not include an approximation of the real platform dynamics by learning a data-driven model.

A new algorithm capable of handling nonlinear dynamics is presented in the work [5]. In this work, a data-driven model approach to approximate nonlinear dynamics with neural networks is researched. The control algorithm used in this work is a model predictive path integral (MPPI) control. Despite the demonstrated successful experiments, the scope of the training was very narrow, with only one simple race track. Therefore, more experiments with nonlinear dynamics approximation using neural networks are required.

Another successful case of neural networks approximating nonlinear dynamics is demonstrated in the work [6]. The aim of this work is to utilize neural networks in model predictive control. The results show that the control performance based on neural networks is similar to manually modeled robot dynamics, which is much more time-consuming. This work demonstrates the great potential of data-driven models for nonlinear dynamics approximation.

Chapter 3

Preliminaries

The aim of the chapter is to briefly refresh the reader's knowledge, mainly of Reinforcement Learning, and to introduce some mathematical notations that are going to be used later on.

3.1 Markov Decision Processes (MDP)

Before we dive into Reinforcement Learning, we should understand Markov Decision Processes. MDP allows us to mathematically formalize sequential decision-making. This formalization is a basis for all Reinforcement Learning problems.

Since we are dealing with decision-making, there should be some entity that makes decisions. In MDP and RL, this entity is called an agent. The rules by which the agent makes its decisions in each particular state are called a policy. The goal of MDP then is to optimize the policy for the most beneficial behavior.

In a classical MDP it is assumed that all states the agent could be in are fully observable. Mathematically MDP is defined as a set

$$M = \{S, A, T, r, \gamma\}$$

where S, A are set of states, actions respectively. Transition operator

$$T_{s',s,a} = p(s_{t+1} = s' | s_t = s, a_t = a)$$

defines the probability distribution of transition between states given an action. Reward for taking an action a_t in a state s_t is defined by a reward function

$$r : S \times A \to \mathbb{R}$$

as $r(s_t, a_t)$. And as a mean to distinguish immediate rewards from distant rewards discount factor $\gamma \in [0, 1]$ is used.

A policy in MDP is a function that maps states to actions probability distribution in a stochastic case

$$\pi : S \times A \to [0, 1]^d \tag{3.1}$$

where d is a dimension of the action space.

Closer formalization of the problem addressed in this work is achieved by a Partially Observed MDP.

3.2 Partially observed MDP (POMDP)

In the Partially Observed Markov Decision Process, the underlying assumption is that the transition between states is defined as an MDP, but for the agent, states are not fully observable. What an agent is able to observe is defined by a probability distribution called emission probability

$$\mathcal{E} = p(o_t|s_t)$$

where o_t, s_t are observation and underlying state at the time t. Figure 3.1 depicts a schema of POMDP.

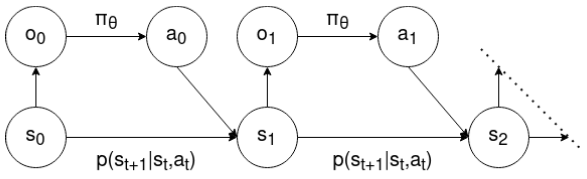

Figure 3.1: The transition schema of a Partially Observed MDP.

Formally POMDP is defined as a set

$$M = \{S, A, O, T, \mathcal{E}, r, \gamma\}$$

where S, A, T, r, γ are the same as in MDP. O is a set of observations.

The goal for an agent then is to maximize an expected total discounted reward

$$\mathbb{E}[G_{t=0}]$$

where

$$G_t = \Sigma_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

is a total discounted reward from timestep t.

The function that maps observations to actions probability distribution is called a policy

$$\pi : O \times A \rightarrow [0, 1]^d \tag{3.2}$$

and it represents a probability of an action a given an observation o

$$\pi(a|o) = p(a_t = a|o_t = o)$$

3.3 Reinforcement learning (RL)

Reinforcement Learning is a framework that allows us to learn the policy function 3.2. There are many kinds of RL. We will focus on the on-policy actor-critic algorithm that is used in this work.

The on-policy learning simply means that during the learning process, the agent utilizes the same policy it intends to learn. That is, it starts with the randomly (or in a different way) initialized policy, and at each following iteration of the learning process, it uses the updated policy from the previous step.

The actor-critic algorithm includes an actor that decides what action to take, which is just another name for a policy function, and a critic that reports to an agent how good his decision was. The critic also has its function representation, that is called a value function, and is defined as follows:

$$V(o) = \mathbb{E}[G_t | o_t = o]$$

The optimization objective in the RL is to maximize the expected sum of discounted rewards:

$$L(\theta) = \mathbb{E}[G_{t=0}] \tag{3.3}$$

One of the subfields of the Reinforcement Learning that solves the optimization problem is Deep Reinforcement Learning.

3.3.1 Deep Reinforcement Learning

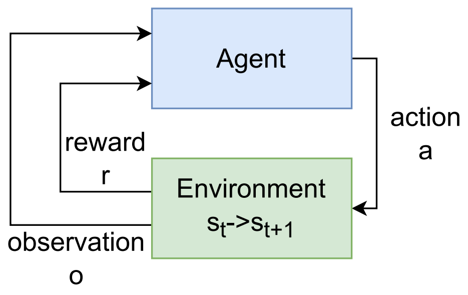

Figure 3.2: Agent-Environment interaction loop.

Deep Reinforcement Learning is a method to approximate both the policy function and the value function with neural networks.

The approximated policy is defined by it's parameters θ that are equivalent with neural network weights:

$$\pi_\theta : O \times A \to [0,1]^d$$

Training of these neural networks is a supervised problem. Training data are obtained in the agent-environment interaction loop shown in the figure 3.2. One of the methods to train the policy neural network is called policy gradients.

3.3.2 Policy gradients

Policy gradient methods allow us to learn policy parameters θ, or using neural networks terminology - weights, by utilizing gradient ascent. The learning process is iterative. At the first iteration, the weights of the neural network are initialized randomly. Then we use our network as a policy in an agent-environment interaction loop to gather new data. This approach is called on-policy learning as opposed to off-policy learning when actions are not chosen by the policy that is being trained but by another stand-alone function. After each episode, or rather a batch of episodes, neural network weights are updated by a backpropagation algorithm. Based on the discounted sum of rewards per episode, we compute gradients that are going to increase the probability of actions that led to positive rewards and decrease the probability of actions that led to negative rewards.

In order to estimate the gradient of the RL objective 3.3 we need to do some derivation. First, let us denote the sequence of decisions made by the agent given observations and gained rewards called:

$$\tau \sim o_1, a_1, r_1, o_2, a_2, r_2, ...$$

then the objective is derived as follows:

$$L(\theta) = \mathbb{E}_\pi[r(\tau)]$$

$$= \int \bigtriangledown \pi(\tau) r(\tau) d\tau$$

$$= \int \pi(\tau) \bigtriangledown \log \pi(\tau) r(\tau d\tau)$$

$$= \mathbb{E}_\pi[\bigtriangledown \log \pi(\tau) r(\tau)]$$

If we make the transition back from the τ notation for the sequence and define an advantage function, which just tells us the difference between the sum of rewards we got and expected as follows:

$$A_t = G_t - V(o)$$

then we get a so-called 'vanilla' policy gradient objective that is defined as follows

$$L(\theta) = E_t[log \pi_\theta(a_t o_t) A_t]$$

The algorithm for policy parameters update is then:

```
for i in n_episodes:
    sample $o_1, a_1, r_1, ...o_T, a_T, r_T$ from $\pi_\theta(a_t|o_t)$
    compute $\bigtriangledown_\theta L(\theta) = \sum_{t=1}^{T}[\bigtriangledown_\theta log \pi_\theta(a_t|o_t) A_t]$
    update $\theta \leftarrow \theta + \alpha \bigtriangledown_\theta L(\theta)$
```

According to [7], using basic policy gradient loss leads to enormous policy updates. In the same paper, an algorithm that deals with this problem was introduced.

3.3.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization [7] is one of the Reinforcement Learning algorithms that uses policy gradients.

In order to overcome large policy updates the PPO algorithm improves on Trust Region Methods [8] and proposes a novel surrogate objective:

$$L_t^{CLIP}(\theta) = E_t[min(\frac{\pi_\theta(a_t|o_t)}{\pi_{\theta_{old}}(a_t|o_t)}A_t, clip(\frac{\pi_\theta(a_t|o_t)}{\pi_{\theta_{old}}(a_t|o_t)}, 1-\epsilon, 1+\epsilon,)A_t)]$$

where ϵ is a hyperparameter, for example, 0.2, the motivation for this loss function is well described in [7].

The final objective in PPO:

$$L_t^{PPO}(\theta) = E_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)]$$

where c_1, c_2 are coefficients, the L_t^{VF} is a squared-error for the value function, S is an entropy bonus [7].

According to [7], PPO outperforms state-of-the-art policy gradient methods while preserving a decent sample efficiency and simplicity of the algorithm.

Chapter 4

Hardware

4.1 Remote control buggy platform

The base for our physical platform is a remote control car shown in the figure 4.1. The scheme of the onboard hardware is depicted in the figure 4.2.

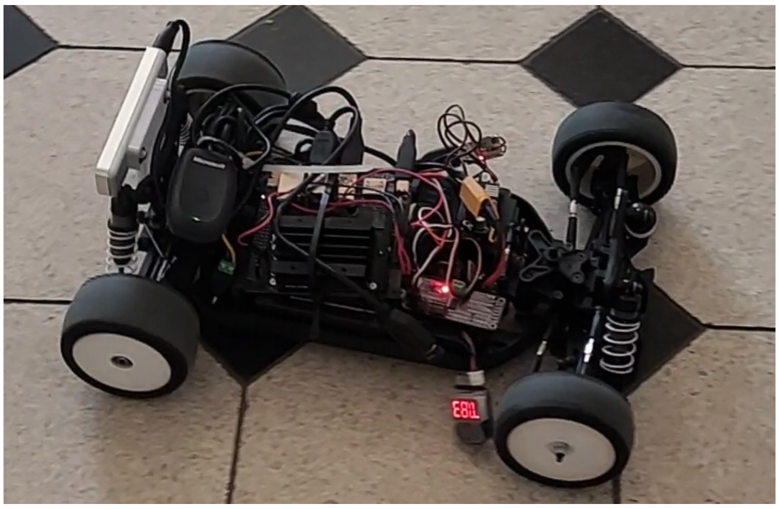

Figure 4.1: Picture of the remote control buggy platform.

The full specification of this platform can be found here [9]. The most important parameters are length - 273mm, weight - 1625g, maximal speed - 16.6 m/s (actually achieved max speed during data gathering is about 8-10 m/s), battery - Li-Po 7.4V 2200mAh, also originally the platform was 4WD, but we downgraded it to rear 2WD.

4.2 Single Board Computer

Our purpose for the hardware is to run an AI workload with low power consumption and reasonable cost. NVIDIA Jetson Nano [10] satisfies our requirements. The computer module size is 70 mm x 45mm module, CPU - Quad-core ARM A57, RAM - 4GB LPDDR4 64-bit, 4x USB 3 ports, onboard

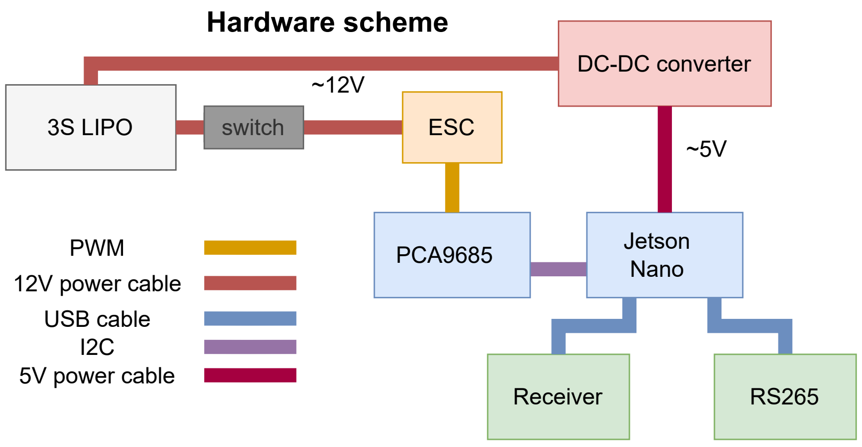

Figure 4.2: Hardware scheme of the remote control buggy platform.

GPU - 128 cores. Offers high-performance computing at 5-10 watts.

4.3 Localization

For our goals, we needed a sensor that provided us with linear and angular velocities along with position and orientation that are going to be used as a definition of a ground truth trajectory. High precision is not required since we attempt to provide a good enough control policy for a rather cheap platform. Low cost, low power consumption, and off-the-shelf usage are three other desirable features. Intel Realsense tracking camera T 265 [11] fitted our requirements quite well. It provided us with reasonable performance ready-to-use V-SLAM (Visual Simultaneous Localization and Mapping) algorithm for a low cost. Based on our own early-stage experiments, we can state that T 265 suffers from vibrations, and its performance outdoors is not acceptable, but since all our included in this work experiments are conducted indoors, and on a flat surface, its usage is well justified. Overtime drift of a position derived from velocities does not bother us since we are working with short-term trajectories mostly, and it is not a goal of this work to provide a control policy that will be capable of autonomous driving for a long period of time.

4.4 Teleoperation

In order to manually drive the platform, Xbox 360 joystick was used with a Microsoft Xbox 360 Wireless Receiver for Windows [12]. Receiving a range up to 10 meters was more than enough for indoor driving. Regarding controllers, the requirement was to have a button to switch between autonomous and manual driving along with controllers for throttle and steering. All requirements were satisfied by a chosen joystick.

Chapter 5

Control Policy

5.1 Task definition

In this work, we attempt to learn a control policy for a car following a trajectory. A control policy maps observations to actions and is defined as follows:

$$\pi : O \times A \to [0,1]$$

Learning a control policy means to optimize it's parameters θ for a maximal expected reward gain $\mathbb{E}[G_{t=0}]$.

Observation and action spaces are defined in the following sections, but let us define the input and an output of the control policy.

Input is a 23-dimensional vector

$$\vec{input} = [v_x, v_y, \omega_z, w_x^{\{1\}}, w_y^{\{1\}}, ..., w_x^{\{10\}}, w_y^{\{10\}},$$

where $w_x^{\{i\}}, w_y^{\{i\}}$ are x, y components of a waypoint i in a car frame.

Output is a 2-dimensional vector

$$\vec{output} = [a_x, a_y]$$

where a_x is a throttle and a_y is a steering action.

We are to optimize the expected sum of discounted rewards

$$\mathbb{E}[\Sigma_{t=0}^{\infty}\gamma^t r_t]$$

where the basic reward function is defined as follows:

$$r_t = \begin{cases} 1 & \text{if a waypoint was closed at the beginning of the timestep t} \\ 0 & \text{otherwise} \end{cases}$$

For the purposes of learning a control policy, we need to define and implement a trajectory generator and an environment that includes an engine, a rewards function, and a way to update the trajectory based on the agent's position. Once all parts are ready, the learning process could be conducted by a Reinforcement Learning algorithm utilizing an agent-environment interaction loop 3.2. The algorithm of choice in this work is the Proximal Policy

Optimization [7] briefly described in the preliminaries chapter. We use the ready-to-use implementation of PPO by the Stable Baselines 3 [13].

But before we dive into details of trajectories generation and environment, let us state one important assumption we make when using Reinforcement Learning.

In this work, we assume that the decision process we are trying to learn a control policy for is a Markov Decision Process.

A decision process is Markov if the Markov Property holds. The Markov Property is a simple claim that the next state depends only on the current state. Mathematically speaking:

$$P(s_{t+1}|s_t, s_{t-1}, ..., s_0) = P(s_{t+1}|s_t)$$

In fact, the underlying decision process in our case is a Partially Observed Markov Decision Process defined in the section 3.2. Particular unobserved states are described in the following section.

5.2 Trajectories

Since the problem we consider is formulated as a car following a trajectory, we need to generate these trajectories somehow as well as have a way to feed the trajectory to the agent during the episode. These two purposes are satisfied by utilizing a trajectory generator and a waypoint closing logic.

5.2.1 Trajectory generator

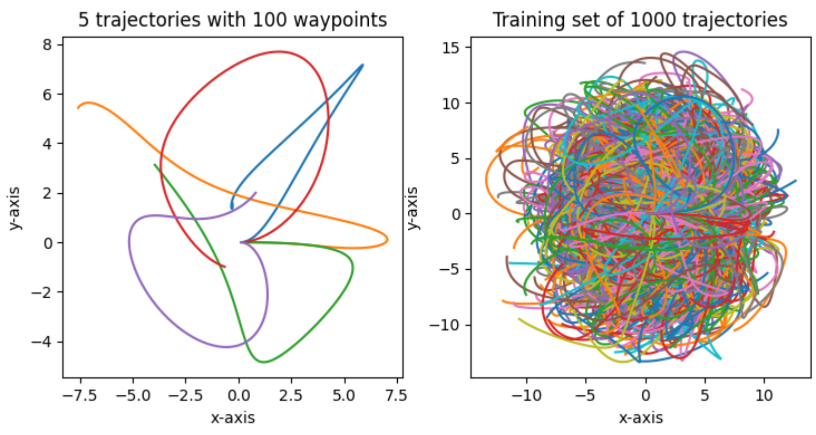

Figure 5.1: Randomly generated trajectories. On the left side, the example of 5 training trajectories is depicted. On the right, the whole training dataset is shown.

For random trajectories generation, we use simplex noise [14]. An example of 1000 generated trajectories with 100 waypoints each is presented in the figure 5.1. As we can see, generated trajectories are quite diverse. On the left side, we present five random trajectories with 100 waypoints each. What could not be seen in these pictures is that the distance between waypoints is longer on the straight sections of trajectories than that on the corners. In other words, waypoints are not uniformly distributed along a trajectory.

5.2.2 Waypoint closing logic

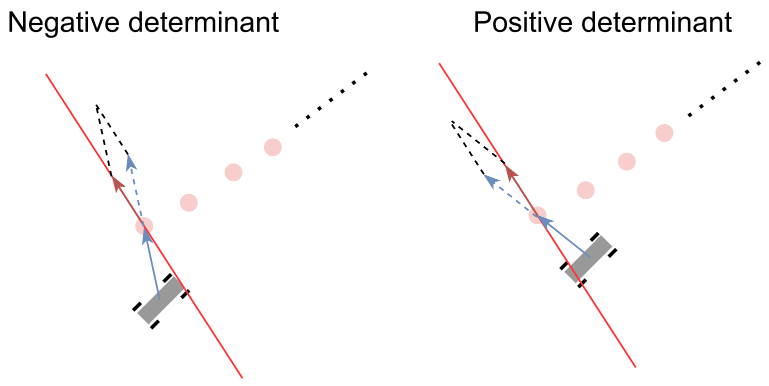

Figure 5.2: The logic to detect closed waypoint using a sign of a determinant. The red vector is a 'left' normal to the trajectory. The blue vector is a subtraction between the next waypoint and the position of the car. The determinant which sign is depicted in the figure takes as a first argument the red vector and as a second the blue vector.

The waypoint is considered closed when the sign of a determinant that takes as a first argument a subtraction between the next waypoint and the position of the car, and as a second argument, the 'left' normal to the trajectory is negative. That is, the condition for the waypoints to be closed:

$$det(\vec{n}_{x,y}, (\vec{p}_{x,y} - \vec{w}^{\{1\}}_{x,y})) > 0$$

where $\vec{n}_{x,y}$ is the 'left' normal to the trajectory.

If the next waypoint $\vec{w}^{\{1\}}$ was passed by the agent, it is discarded, and one more waypoint is added to the end of a waypoints vector. This way, the waypoint vector preserves dimensionality. You can think of it as a fixed size window that moves only forward along the trajectory line. For programmers, a better example would be a queue with a limited size.

5.3 Environment

As was previously mentioned, training data for Reinforcement learning are obtained in an agent-environment interaction loop (figure 3.2). We implement a custom environment based on the OpenAI Gym interface [15].

The environment is defined by its action space, state space, transition function between states, and reward function.

Action space has 2 dimensions - throttle $\in [-1, 1]$ and steering $\in [-1, 1]$.

State-space has 23 dimensions, and these are linear velocity along x and y axes in a car frame, angular velocity, ten waypoints in a car frame, and each waypoint is a 2-dimensional vector $[x, y]$. Velocities are fetched from an engine, and waypoints are updated by a waypoint closing logic.

The transition function is represented by two components: an engine and the waypoint closing logic. There are two types of engines we use, data-driven model-based custom engine 6.1 and MuJoCo engine 5.3.2. How the engine is incorporated into the environment was shown on the schema 1.2.

The rewards function implementation is trivial. The design, though, is a hard part. The design of a reward function in Reinforcement Learning is called reward shaping, and it deserves its own paragraph.

5.3.1 Rewards

The goal of reward shaping is to help an agent learn the desired behavior by providing it with meaningful rewards. There is a danger of intervening in the learning process with custom rewards too much and causing the agent to learn sub-optimal behavior. It is a good practice to start with simple rewards, and in case they do not do their job, increase complexity. Our definition of a reward is as simple as it gets. It is binary:

$$r_t = \begin{cases} 1 & \text{if a waypoint was closed at the beginning of the timestep t} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

Since we want our agent to not only run past the waypoints but also to closely follow the trajectory, the penalty for a deviation from the trajectory was introduced. At each step, we penalized our agent by subtracting from the reward the value of $0.01 * d$ where d is the distance from the current position of the agent to the trajectory. This penalty, though, did not improve the agent's performance and was eliminated during the experiments.

5.3.2 Physics engine MuJoCo

As a state-of-the-art physics engine MuJoCo [16] was chosen to facilitate the research thanks to its fast, accurate, freely available simulation. MuJoCo, in fact, was developed to facilitate model-based control - exactly what we are looking for. According to [17] MuJoCo performs best compared to Bullet and other engines on robotics-related tasks.

MuSHR model [18] resembling our RC buggy was tuned and used to simulate the real platform behavior.

In order to achieve close to the real platform behavior in a simulation environment, parameters in a .xml MuSHR model configuration file such as friction, kv, forcerange for actuator were manually tuned. To evaluate parameters, a few 1-second trajectories from real-world episodes were sampled, and the ground truth trajectory was compared to the resulting trajectory in a simulation after taking the same actions as in the real world. In the figure

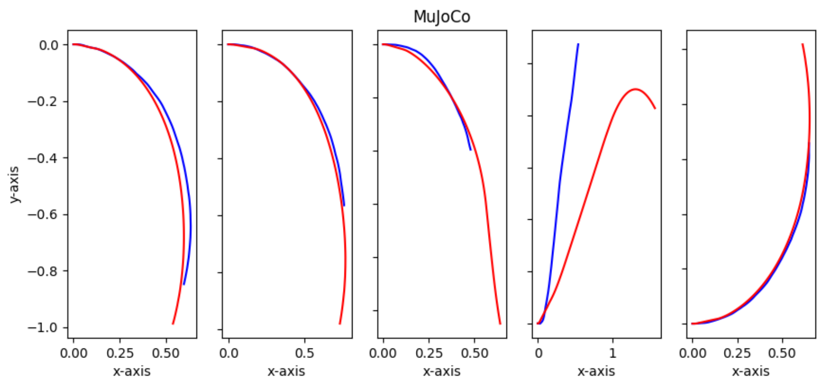

Figure 5.3: Comparison of 1-second ground truth trajectories from the real world episode (blue color) to trajectory from the MuJoCo-based simulation episode. During the simulation episode, the MuJoCo engine was fed with the sequential of actions from the real world episode.

5.3 you can see the result of our attempt to tune parameters, so they fit the reality as close as possible. The blue line is the ground truth trajectory, and the red line is the trajectory of the MuJoCo engine. The mean squared error for these five samples is 0.0370. The mean squared error between trajectories is defined as follows:

$$mse(\vec{p}, \vec{q}) = 1/N \sum_{i=1}^{N} (\vec{p_i} - \vec{q_i})^2 \tag{5.2}$$

where $\vec{p}, \vec{q}$ are vectors defining corresponding trajectories.

5.3.3 Custom Engine

In accordance to the schema 1.2 engine provided with the model output $\delta\vec{v}$ and $\delta\vec{\omega}$ updates velocities, position and orientation of the simulated platform. Linear and angular velocities for the next timestep computed using simple equations:

$$\vec{v_{t+1}} = \vec{v_t} + \delta\vec{v_t}$$

$$\vec{\omega_{t+1}} = \vec{\omega_t} + \delta\vec{\omega_t}$$

Position and orientation updates for the simulated buggy platform depend on a time interval between timesteps that we define as $\tau = 1/100s$. Then computations for the position vector

$$\vec{p} = [p_x, p_y, p_z]$$

and orientation vector in Euler angles

$$\vec{u} = [u_x, u_y, u_z]$$

at a time step $t + 1$ are

$$\vec{p_{t+1}} = \vec{p_t} + \vec{v_t} * \tau$$

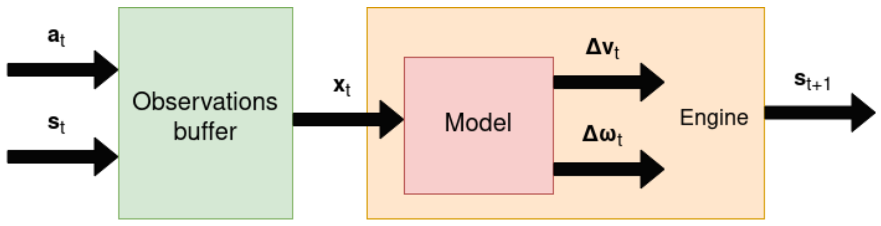

Figure 5.4: The schema represents one step of a simulation loop of a data-driven model-based engine. The observations buffer ensures the history of observations for corresponding models.

$$\vec{u_{t+1}} = \vec{u_t} + \vec{\omega_t} * \tau$$

And since we are in a 2-dimensional space, $p_z, u_x, u_y = 0$.

The repeated process of feeding the engine with actions at every time step of an episode and state updates is called a simulation. Components of a simulation loop are presented in the figure 5.4. Observations buffer is utilized when the history of observations as an input for the model is used.

Chapter 6

Data driven Model

6.1 Mathematical formulation

As we have shown on the environment schema 1.2 model is an internal part of an engine that computes state changes. In our case the output of a model is a 3-dimensional vector

$$\vec{out} = [\delta v_x, \delta v_y, \delta \omega_z]$$

where

$$\delta \vec{v} = \vec{v_{t+1}} - \vec{v_t}$$

is a delta linear velocity

$$\delta \vec{\omega} = \vec{\omega_{t+1}} - \vec{\omega_t}$$

is a delta angular velocity

$$\vec{v} = [v_x, v_y, v_z]$$

$$\vec{\omega} = [\omega_x, \omega_y, \omega_z]$$

are linear and angular velocities respectively. Since we consider 2-dimensional space $v_z, \omega_x, \omega_y = 0$.

That is, our model computes how velocities change at the time step $t + 1$ given the state

$$\vec{state} = [v_x, v_y, \omega_z]$$

and actions taken at the time step t

$$\vec{a} = [a_x, a_y]$$

where a_x is a throttle and a_y is a steering action. Hence our basic model input is defined as a 5-dimensional vector

$$\vec{in} = [v_x, v_y, \omega_z, a_x, a_y]$$

Depending on a model architecture, it might take as an input $k > 0$ previous observations stacked together. In this case input is a $5 * (k + 1)$-dimensional vector

$$\vec{in} = [v_x^{t-k}, v_y^{t-k}, \omega_z^{t-k}, a_x^{t-k}, a_y^{t-k}, ..., v_x^t, v_y^t, \omega_z^t, a_x^t, a_y^t]$$

6.2 Data gathering

Model training is a Supervised Learning problem. Hence we start with collecting labeled data. This is not an easy task, though. The outcome of the training process is highly dependent on the quality of data. But a manual data gathering process is subjected to human biases and limitations. These flaws, though, could be addressed to some degree. But let's not get ahead of ourselves and start with the gathering process.

Data are collected on the RC buggy platform (chapter 4) from a sensor (section 4.3) with a rate of 100 samples per second. The process of gathering data is divided into episodes with a length of $m_i \in \mathbb{R}$ timesteps. Each episode can be described as follows: from an idle state, the platform is manually driven by a human for some time; the driver is not following a predefined trajectory; the driver is restricted by the environment, that is, a room resembling a square with a side size of approximately 8 meters.

The result is a dataset with n entries, where n is a number of episodes. Each entry contain m_i samples of $\vec{v}, \vec{\omega}, \vec{p}, \vec{q}$ vectors. Gathered raw data are consequently divided into train and validation datasets.

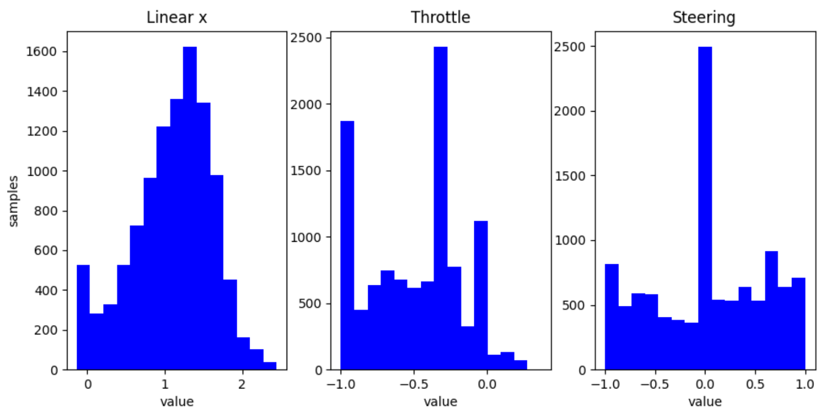

Figure 6.1: Histograms of linear velocity and actions were taken during one real-world episode to demonstrate the distribution of manually gathered data.

Once the data are collected comes the time to assess its quality. A simple way to do so is to plot histograms and manually review the result. Take a look at the figure 6.1. The distribution of steering actions seems quite fair during the depicted episode, but what one could point out is that high throttle actions are missing. Nearly all actions taken were low throttle, less than 0, and in case we would train our model only on this data, it would never be able to approximate the dynamics of the platform with the high throttle. Therefore, for the next iteration of data gathering, we should adjust drivers' behavior so that incompleteness of the dataset is decreased. The reader could easily imagine how this process is done several times in order to increase the quality of a dataset used for the final model training.

6.3 Incompleteness of state observability

We can highlight several states of our platform as directly unobservable. One is a rear-wheel velocity since we have a rear-wheel-drive car, and wheels could slip when the high throttle is applied. This problem could be addressed by adding a wheel velocity sensor, for example. Since we do not have this sensor, we address this problem using a history of actions. Experiments show that the performance of data-driven models with and without a history of actions does not differ significantly. Therefore, we assume the rear wheel velocity to be negligible.

Another unobserved variable is the exact angle at which front wheels are turned. Despite turning angle being correlated with a steering action, the correlation might not be as precise as needed for it to be considered an observed variable. We could address this problem, though, even without additional sensors, by adding a mean of m past steering actions to observations, where m is to be empirically chosen. But the experiments show that the response of steering is almost instantaneous. Therefore, we neglect the turning angle of the front wheels.

The last one is imperfections in the platform itself. When you turn the wheels and then drop the steering controller, the wheels do not return fully to angle 0. There is some bias to the side you turned them previously. This flaw is observable in the history of data, though, since once you start moving forward, there will be a slight turning bias despite the steering action might be zero. But again, experiments show that this unobservable variable is not of high importance, and one could neglect this platform flaw if the steering bias is not too high.

6.4 Data preprocessing

6.4.1 Noise

Raw data are noisy, as we can see in the figure 6.2, where a part of an episode linear velocity data is shown. Noise affects training performance as well as the resulting model. Therefore, we need a way to reduce noise in the training dataset.

Since data from the sensor, when the control policy is deployed in the real world, have to be filtered the same way training data were filtered, only filters that use past observations could be used. In other words, data at the time step t could not be changed based on the future data $t + k$, only on the past data $t - k$.

6.4.2 k-past mean filter

The basic way to filter the input data is to simply take the average of k previous entries. This approach suffers from reducing extremes, though, which is undesirable when dealing with high and low speeds. The results of filtering

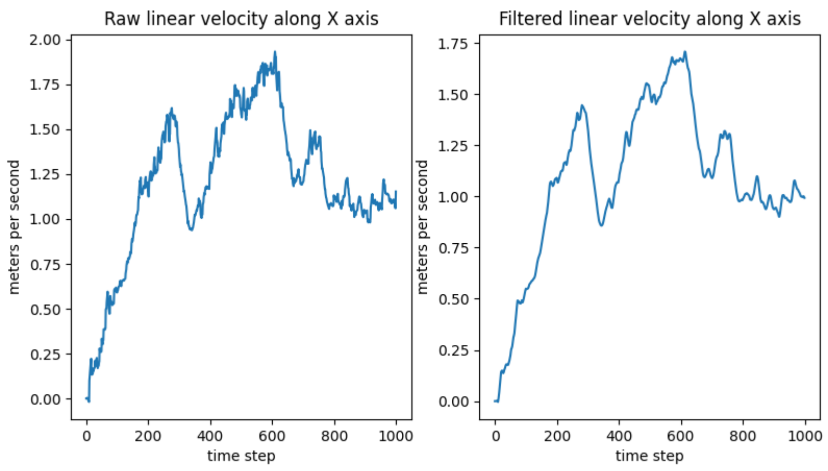

Figure 6.2: Raw linear velocity data on the left and filtered with a k-past mean filter with $k = 10$ on the right.

linear velocity with $k = 10$ are shown in the figure 6.2 on the right side. Since we take an average of previous timesteps, extremes on original raw data are reduced. But if we apply the same filter when deploying policy learned on the engine utilizing a model trained on these filtered data, we should be fine.

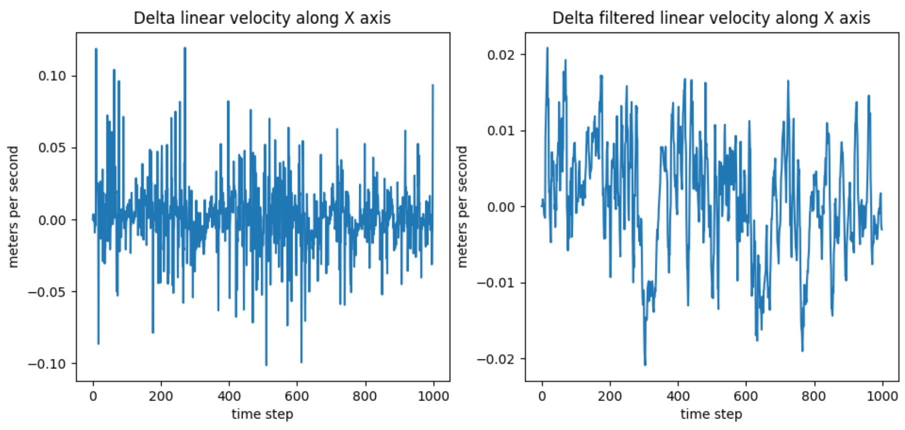

Figure 6.3: The example of unfiltered and filtered labels. Delta linear velocity data on the left and delta filtered linear velocity with the k-past mean filter with $k = 10$ on the right.

Examples of computed labels based on raw and filtered data are in the figure 6.3.

6.4.3 Normalization

For numerical reasons before training we need to normalize our labels. Normalization is done by dividing label values by the standard deviation

$$\sigma = \sqrt{1/N \sum_{j=1}^{N} (x_j - \mu)^2}$$

where μ is a mean value

$$\mu = 1/N \sum_{j=1}^{N} x_j$$

The normalization factor is stored, and the output of our model is multiplied by this factor during deployment.

6.4.4 Data augmentation

Due to the manual process of data gathering, the resulting dataset might be arbitrarily biased. For example, due to the environment constraints or driver's preference, more left turns than right turns might be present in the data. Another example is a throttle distribution. The latter was addressed to some extent in the section 6.2. The mentioned approach was to gather more data in order to obtain a decent distribution of throttle actions. It would be much more difficult to do so by augmenting the dataset. It is not so with steering actions, though.

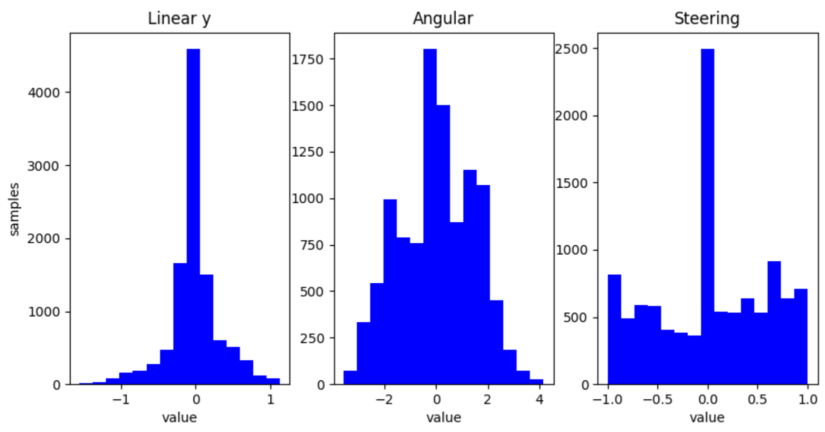

Figure 6.4: Histograms demonstrate the distribution of training data related to turning before augmentation.

If we assume that our platform is symmetrical, it is easy to balance turns in the dataset by mirroring actions and corresponding observations: a_y, v_y, ω_z. That is, we obtain the mirrored data

$$\vec{in} = [v_x, -v_y, -\omega_z, a_x, -a_y]$$

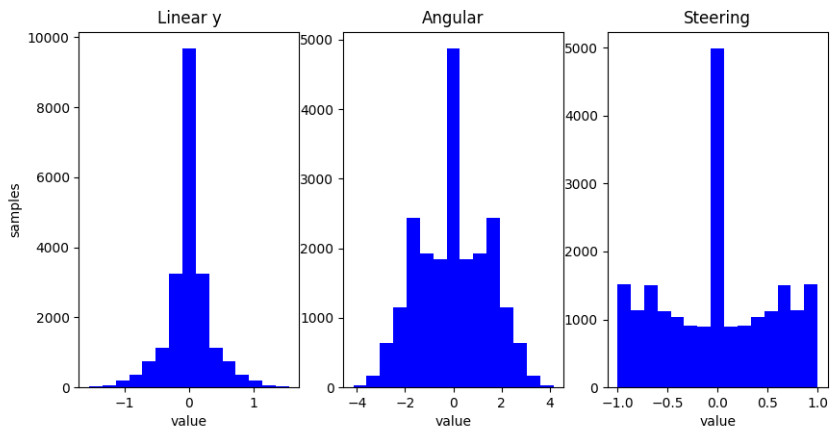

Figure 6.5: Histogram demonstrating the distribution of augmented training data related to turning.

and add it to the original dataset doubling it's size.

An example of histograms computed on training data before augmenting and after is shown in the figures 6.4, 6.5.

The effect this approach has on the model's performance is described in the chapter 7.

6.5 Model's architecture

6.5.1 Multi Layered Perceptron (MLP)

The base model for the task is a feed-forward neural network with fully connected layers called Multi-Layered Perceptron. A simple schema of an MLP architecture where the input vector does not contain a history of observations is shown in the figure 6.6. All layers of the model are fully connected without dropout. The parameters such as the number of hidden layers and the number of neurons per layer are to be optimized. In the figure, we specify these only as an example.

6.5.2 MLP with history of observations

MLP with history is basically the same model as an MLP. The only change is that the input contains observations from k previous timesteps. Hence the input layer will have more neurons. The idea behind this model is to continue with a simple model at the start but provide it with more data in order to get better results. The comparison of the base MLP model and model with the history of observations is made in the chapter 7.

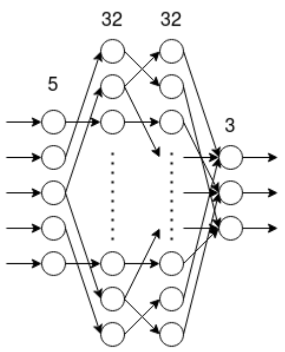

Figure 6.6: The feed-forward neural network model with fully connected layers is also called the Multi-Layered Perceptron.

6.5.3 Temporal convolutional neural network (TCNN)

Thanks to successful usages of TCNN on time series data, for example, [19], [20] etc., and it's relative to LSTM simplicity, the decision to use it in our case was made.

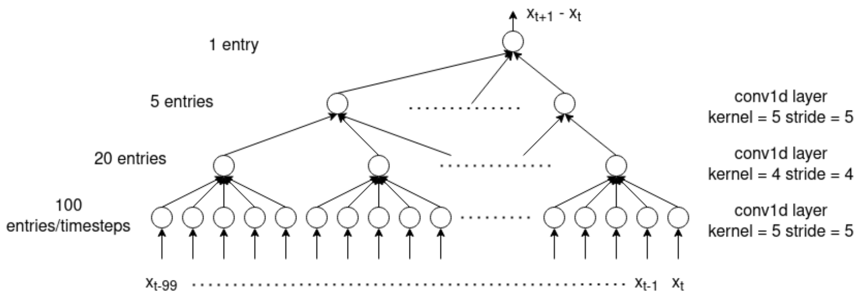
$x_{t+1} - x_t$
1 entry
5 entries
20 entries
100 entries/timesteps
conv1d layer kernel = 5 stride = 5
conv1d layer kernel = 4 stride = 4
conv1d layer kernel = 5 stride = 5
x_{t-99}
x_{t-1}
x_t

Figure 6.7: The feed forward process of a time series data through convolutional layers of a Temporal CNN.

The key feature of a TCNN is that the output of a convolution is only dependent on the previous observations on each convolutional layer. An example of TCNN convolutional layers scheme with inputs sequence of size 100 is presented in the figure 6.7 in order to better understand time dependencies. With specified kernel and stride values for each layer the computations correspond to the following time dependencies schema for the output: layer 1 $x_{t-99} : x_{t-95}, .., x_{t-4} : x_t$; layer 2 $x_{t-99} : x_{t-75}, .., x_{t-24} : x_t$; and finally layer 3 $x_{t-99} : x_t$.

The final TCNN architecture that was used in the experiments is presented in the figure 6.8. We start with five channels for the input features that are defined in the 6.1 and represent forward velocity, lateral velocity, angular velocity, throttle, and turn actions. At the last convolutional layer, we have 16 channels that are mapped to a fully connected linear layer with 16 neurons that are connected to the output layer with three neurons corresponding to delta velocities that we aim to predict.

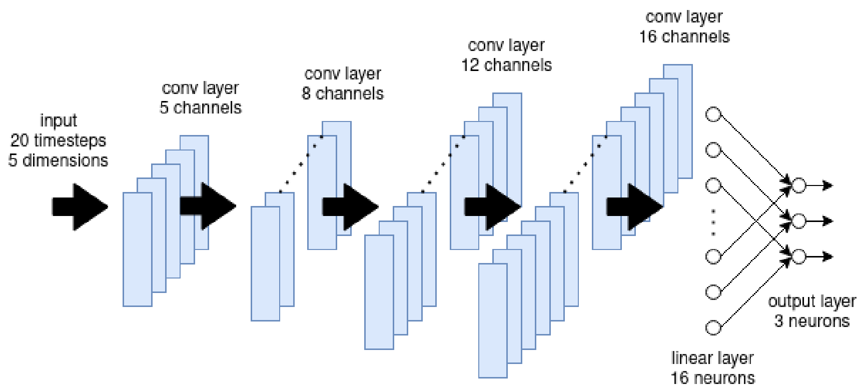

Figure 6.8: Temporal convolutional neural network architecture.

Chapter 7

Results and experiments

7.1 Data Driven Model

7.1.1 Experimental setup

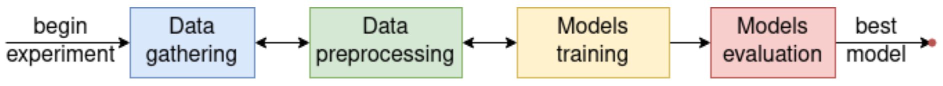

Figure 7.1: Schema of an experimental setup for the data-driven model learning.

The first type of experiment conducted in this work pursues the goal of learning the data-driven model approximating nonlinear dynamics. The iterative process of achieving the stated goal is depicted in the figure 7.1. It starts with the data gathering step, then through data preprocessing and models training, we come to the models' evaluation final step. If we are satisfied with the observed performance, we proceed to the control policy learning. During the experiment, we might return to the previous step if needed. This case is depicted by a bi-directional arrow.

Since our model approximates the nonlinear dynamics of the real platform, it is necessary to compare its predictions to the ground truth data gathered from the mentioned platform. The way we accomplish this is by iteratively feeding the same actions we took in the real-world episode and updating the simulated platform state by the model's predictions. The resulting trajectory of a simulated platform is then compared to the ground truth trajectory from the real-world episode. We compare only short episodes 1-seconds long since, due to the differences between the reality and our model, the trajectories will inevitably diverge. Ground truth 1-second trajectories are sampled from the longer real-world episodes randomly. The initial state of the platform in these trajectories is set in a simulation before we start feeding it with actions taken in the real world. For comparison purposes, we use the mean squared error metric 5.2.

The second kind of evaluation is not so well reproducible and conducted in order to reveal some edge case flaws of the learned model. Instead of feeding the model with actions from the real-world episode, we manually drive the platform in the simulation and try to assess its ability to approximate the real platform dynamics.

7.1.2 Data preprocessing

To understand the effect of the data preprocessing on our model, we compare the training performance of the MLP with and without a particular preprocessing technique. In the figure 7.2 an example of such a comparison is presented. The MLP model trained on data preprocessed with the k-past mean filter achieves noticeably lower test loss. The variable $k = 5$ of the k-past mean filter was optimized separately. The augmentation that is done in addition to the data filtering does not seem to help. If we measure the mean squared error between real-world trajectories and simulated trajectories by all three models, we come to the conclusion that filtering without augmentation is the best way to preprocess data. Measured mean squared errors could be found in the table 7.1.

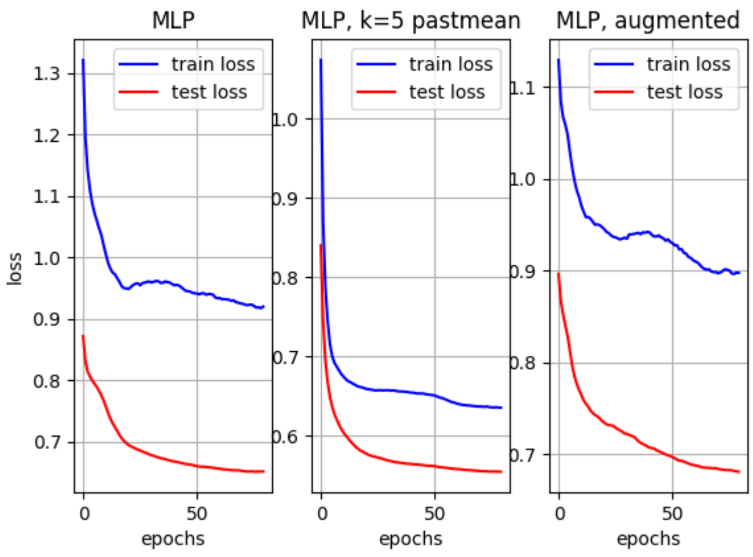

Figure 7.2: Comparison of different data preprocessing methods based on the train/test loss for the MLP. From left to right: no data preprocessing, filtering training data by the k-past mean filter with $k = 5$, filtering training data by the k-past mean filter with $k = 5$, and augmenting the dataset by mirroring left and right turns.

7.1.3 Training

Training of all chosen neural network architectures is conducted on the same training data preprocessed in the exact same way.

Train and test losses per epoch for training on data filtered with $k = 5$ past mean filter are shown in the figure 7.3. MLP with a history of observations achieves the lowest test loss as well as train loss. These losses, though, are not

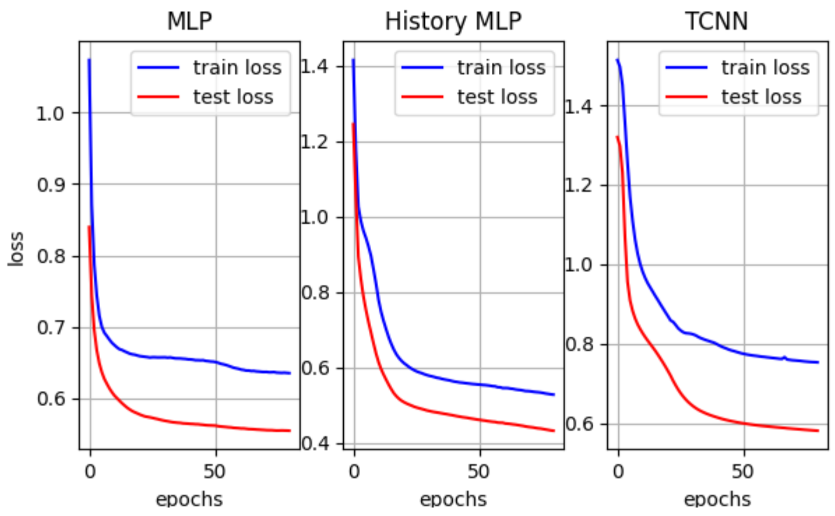

Figure 7.3: Learning curves for an MLP, MLP with history of observations, TCNN data-driven models.

representative enough to make the final decision. Therefore, an evaluation of each trained model's performance must be conducted.

7.1.4 Evaluation

In order to properly compare all three models, we run ten 1-second episodes in a simulation as was previously described. The results for 5 episodes are shown in the figure 7.4, mean squared error (MSE) scores 5.2 for each model are in the table 7.1.

According to the figure 7.3 the MLP with a history of observations has the lowest test loss, but the lowest MSE score is obtained by the TCNN. Therefore, further tests are needed to distinguish between those models.

Predictions of all models diverge in time because they are deployed on a different distribution of data they were trained on. This distribution is generated by the model itself, since at each step of the simulation, we feed the model with observations that are affected by all its previous predictions, whereas the training distribution corresponds to the real-world environment

Engine	MSE
MLP	0.0596
History MLP	0.0592
TCNN	0.0376
MuJoCo	0.0371
MLP, augmented	0.095
MLP, unfiltered	0.105

Table 7.1: Results of a comparison of data-driven models using the mean square error metric.

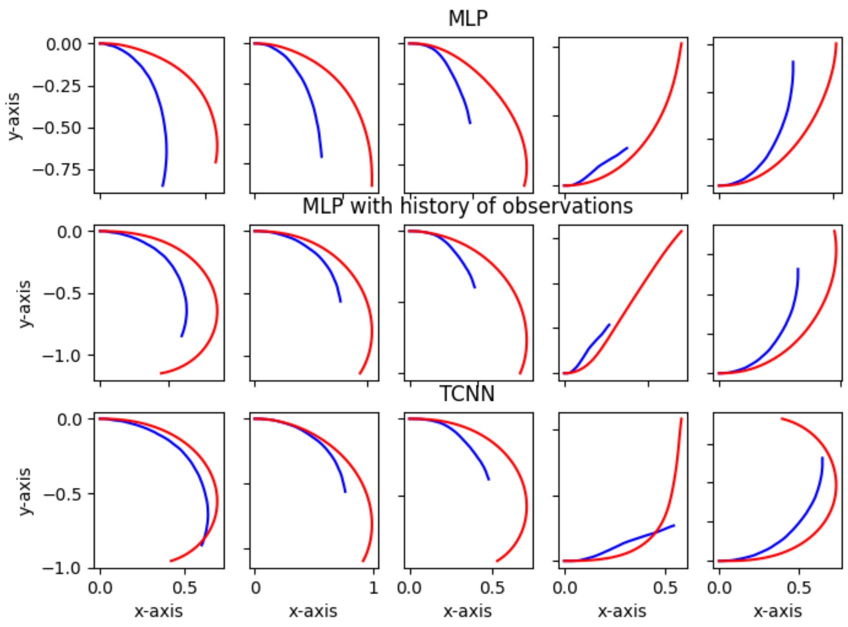

Figure 7.4: The comparison of the real-world 1-second trajectories to the trajectories simulated by different data-driven models fed with the same sequence of actions as was taken in the real-world episode. The blue line depicts the ground truth trajectory from the real-world episode. The red line depicts the resulting trajectory from a simulation.

and the real platform. This effect is demonstrated in the figure 7.5 where 2-second ground truth and simulated trajectories are compared.

Despite the fact that on 1-second trajectories, the TCNN performs best, its predictions diverge worse than other models when deployed for longer episodes. The TCNN seems to be more sensitive to the change of data distribution according to conducted manual tests.

During manual driving of the platform simulated by the TCNN, significant instabilities in the behavior were discovered. For example, when applying high throttle from the idle state with a high steering action, the platform would start unnaturally drifting in circles.

When TCNN's parameters were optimized in order to avoid this undesirable behavior, another problem occurred. When we increase the input sequence length for TCNN it becomes less responsive to the current action, probably due to the insignificance of one action to the whole observations sequence.

MLP with a history of observations, on the other hand, suffers less from TCNN problems. During manual tests, there was no unnatural behavior, and its responsiveness to a new action resembles the real platform quite well.

Hence, the final choice of the data-driven model was MLP with a history of observations.

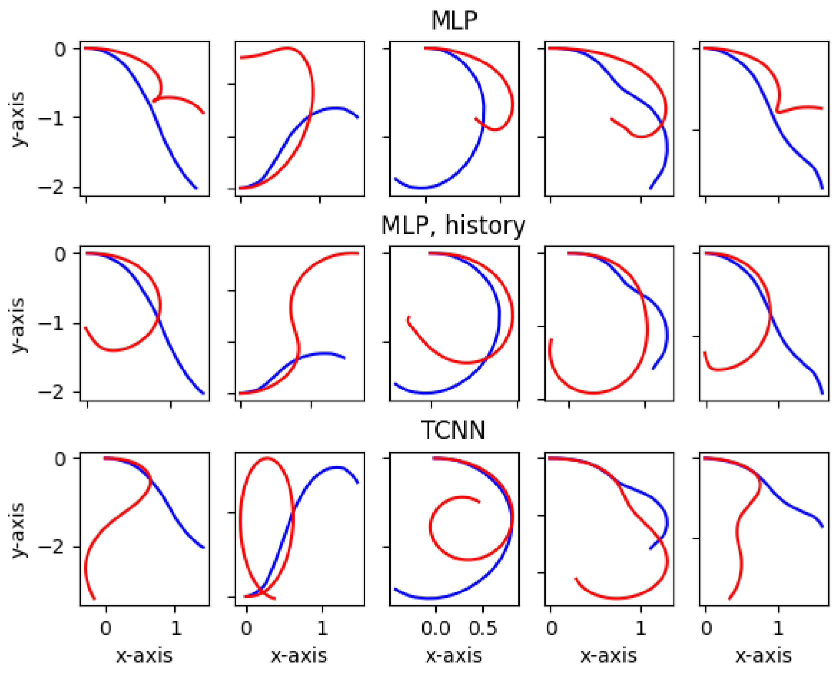

Figure 7.5: The demonstration of divergence due to the different data distribution during the model deployment. Real-world 2-second trajectories compared to trajectories simulated by different data-driven models fed with the same sequence of actions as was taken in the real-world episode. The blue line depicts the ground truth trajectory from the real-world episode. The red line depicts the resulting trajectory from a simulation.

7.2 Control Policy

7.2.1 Experimental setup

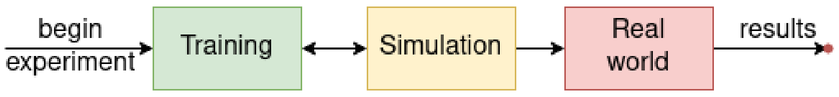

Figure 7.6: The schema of a control policy learning experiment.

Experiments with a control policy could be divided into training, evaluation in a simulation, and evaluation in the real world parts. In case we are not satisfied with the policy performance in a simulation, there is no sense in proceeding to the experiments on the real platform. Therefore, we return to training (bi-directional arrow in the figure). During the experiments, we consulted the paper addressing the issues of using the Deep Reinforcement Learning [21].

Now with the picture of the experiment setup in mind, we can proceed and describe the results we got at each step.

7.2.2 Training

Control policies are trained on two engines: MuJoCo and our custom engine based on the MLP with a history of observations model that was chosen as the best in the previous section. For policies based on both engines, we always have the same training setup for the sake of fair comparison. That is, all hyperparameters for the PPO algorithm are the same, as well as a set of trajectories and the reward function.

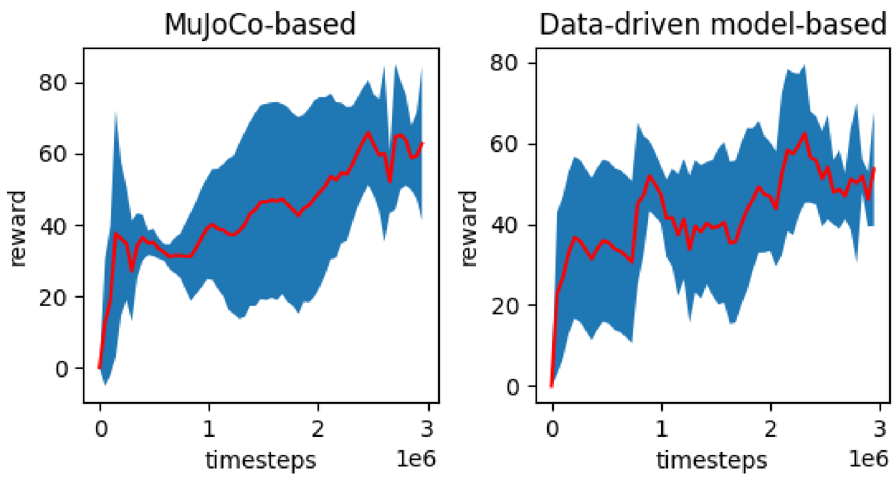

Figure 7.7: The comparison of learning curves of the MuJoCo engine-based policy and the data-driven model-based (or custom-based) policy. Each policy was trained for a total of 3 million timesteps. Read line depicts the mean of 3 training runs. The blue area depicts the standard deviation of 3 runs.

In addition to the main goal of this work, we are also trying to learn a robust policy that is going to generalize well for a wide variety of trajectories. The training set of trajectories was shown in the figure 5.1. As can be seen in the figure, there is a good distribution of difficulty between trajectories and decent coverage of possible directions for a trajectory to take.

In the figure 7.7 an example of policies performance during training is demonstrated. We mostly care about the development of the learning process and the overall trend of improvement of the policy, which both policies show with a local downtrend in the middle of learning.

7.2.3 Validation

In order to test the performance of learned policies and the ability to generalize, we must have a validation dataset.

For tests in a simulation environment, our validation dataset consists of 10 randomly generated trajectories that an agent has not seen during training. Generator parameters for validation trajectories were set the same as for the training dataset. Validation set is shown on the figure 7.8 where trajectories are cut to 100 waypoints.

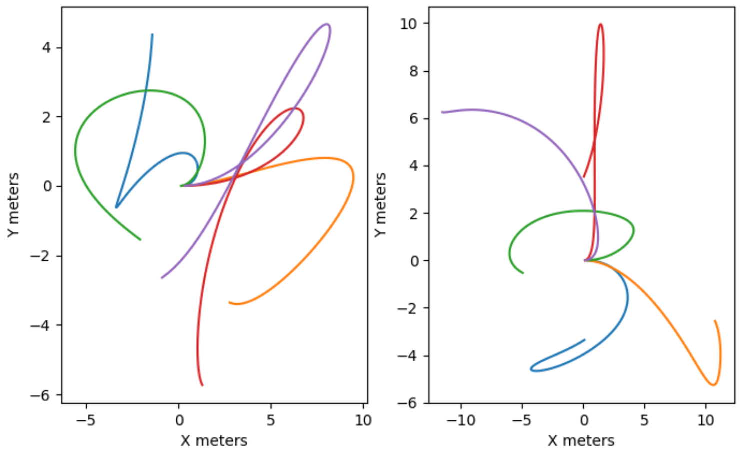

Figure 7.8: Ten (five on the left and five on the right) randomly generated validation trajectories of 100 waypoints each.

The validation process itself is quite simple. We take two learned policies, one on the MuJoCo engine and one on the custom engine, and run 20 episodes 1000 timesteps long (ten seconds) for each policy, one episode per trajectory. Ten episodes are run on the subset of 10 training trajectories, and another ten episodes are run on the validation set. Results for the training dataset could be found in the table 7.2 and for the validation dataset in 7.3. We also switch the engines for both policies to assess how well both policies can adapt to new conditions. In the tables, rows represent engines on which the simulation is running, and columns represent policies that are being deployed. Results are presented as a mean sum of rewards ± standard deviation.

From the results of the experiments, we could deduce that the custom engine-based policy is significantly more robust to the environment changes

Engine|Policy	Custom-based	MuJoCo-based
Custom	88.2 ± 21.23	34.9 ± 21.93
MuJoCo	67.2 ± 29.04	81.5 ± 29.91

Table 7.2: Results of policies comparison in a simulation on a subset of 10 training trajectories for 1000 timesteps per trajectory. Results are presented in the following format: mean sum of rewards ± standard deviation.

Engine|Policy	Custom-based	MuJoCo-based
Custom	59.67 ± 26.98	18.91 ± 22.03
MuJoCo	52.02 ± 16.61	78.73 ± 25.30

Table 7.3: Results of policies comparison in a simulation on a set of 10 validation trajectories for 1000 timesteps per trajectory. Results are presented in the following format: mean sum of rewards ± standard deviation.

since when deployed on the MuJoCo engine, its performance is only slightly worse compared to that of the MuJoCo based policy when deployed on the custom engine. The custom-based policy on the validation set lost 7.65 rewards on average when deployed on the MuJoCo engine, whereas the MuJoCo-based policy lost 60.54 rewards on average when deployed on the custom engine.

We could also point out that both policies generalize quite well. On the validation set compared to the training set, the custom engine-based policy lost 28.53 rewards on average, which is 32% of performance on the training data, and the MuJoCo-based policy lost only 2.77 rewards on average, which is only 3.3% of performance on the training data.

Our main goal, though, is to test the hypothesis that the transfer to the real world from the simulation environment is done better with the data-driven model-based engine. Therefore, we need to conduct additional tests for both policies on the suitable for the real world environment trajectories.

7.2.4 Evaluation in a simulation

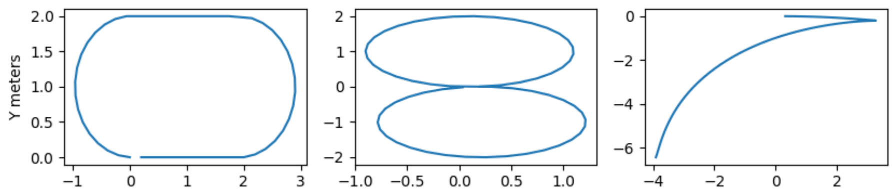

Figure 7.9: The set of validation trajectories for real-world tests. The lap is shown on the left, the infinity in the middle, and the random trajectory on the right.

Evaluation of the transfer to the real world is a bit trickier because we are limited by the size of the room we conduct our experiments in. Random trajectories used for validation would not fit in this room. Therefore, two manually created trajectories and one created by the same method as the training dataset but manually chosen are used for the purpose of policy evaluation. Trajectories are shown in the figure 7.9. The purpose of this manually chosen random trajectory is to assess the ability of the policy to cope with a situation when highly nonlinear behavior (drift) is required.

The results are summed up in the table 7.5. As a reference point, we provide the results of human driver's performance on the same set of trajectories in the simulation in the table 7.4. Here human driver is the author of this work who is definitely not a professional driver, but also not an amateur, since he spent at least several hours controlling the real platform during the data gathering as well as the simulated platform on both engines during the debugging process, reward shaping etc.

7.2.5 Evaluation in the real world

Once we are satisfied with the performance of learned policies in the simulation, we proceed to the tests on the real hardware.

For the tests on the real hardware, we had to scale throttle and steering actions for the policy learned on the MuJoCo engine due to the difference in response of the motors in the simulation and on the real platform. It is not a big problem since the scaling is linear. We applied a 0.95 coefficient to the throttle action and 0.9 to the steering. That is, actions of the policy learned on the MuJoCo engine were

$$0.95a_x, 0.9a_y$$

where a_x, a_y are predictions of the policy.

As was stated in the previous section, we conduct tests on the three chosen trajectories. On each trajectory, we run each policy five times. Test results are in the table 7.6. The format for results is as follows: mean sum of rewards $\pm$ standard deviation.

Trajectory|Policy	Human on Custom	Human on MuJuCo
lap	58.33 ± 4.18	68.66 ± 5.31
infinity	64.66 ± 10.53	70.33 ± 9.03
random	32.66 ± 2.86	31.0 ± 4.96

Table 7.4: Human driver performance on the real-world validation dataset in a simulation environment. On trajectories 'lap' and 'infinity,' the episode lasted 1000 timesteps, and on the 'random' trajectory - 500. On each trajectory, three episodes are conducted. Results are presented in the following format: mean sum of rewards $\pm$ standard deviation.

Trajectory|Policy	Custom-based	MuJoCo-based
lap	87	106
infinity	75	72
random	36	22

Table 7.5: Comparison of policies in a simulation on both engines. On trajectories 'lap' and 'infinity,' the episode lasted 1000 timesteps. On the 'random' trajectory - 500. The result is a sum of rewards since in a simulation environment, policy is deterministic.

Trajectory|Policy	Custom-based	MuJoCo-based
lap	60.0 ± 16.39	47.25 ± 14.25
infinity	64.4 ± 10.8	42.2 ± 8.68
random	31.6 ± 4.62	11.5 ± 1.5

Table 7.6: Results of tests in the real world. On trajectories 'lap' and 'infinity,' the episode lasted 1000 timesteps on the 'random' trajectory - 500. Results are presented in the following format: mean sum of rewards $\pm$ standard deviation.

The difference between the simulation and the real-world rewards are summed up in the table 7.7. For the custom engine-based policy, the difference in the mean rewards is -13.66. For the MuJoCo-based policy, the difference in the mean rewards is -33.02.

Video examples of both policies deployed in the simulation and in the real world on the trajectory 'lap' are provided on YouTube.com via the link `https://www.youtube.com/channel/UCb7esjH7TSxUHZ_tABx2E-w/videos`. The simulation time might not correspond to the real-time due to the slow rendering on the personal hardware.

Trajectory|Policy	**Custom-based**	**MuJoCo-based**
lap	-27 ± 16.39	-58.75 ± 14.25
infinity	-10.6 ± 10.8	-29.8 ± 8.68
random	-3.4 ± 4.62	-10.5 ± 1.5

Table 7.7: The difference between the real-world and the simulation rewards for the set of real-world validation trajectories. Results are presented in the following format: subtraction between the mean sum of rewards in the real world and the sum of rewards in the simulation $\pm$ standard deviation of the sum of rewards in the real world.

Chapter 8

Discussion

8.1 Data-driven model

Despite a decent performance of the MLP with a history of observations model was achieved, there were several flaws with all data-driven models and with the TCNN in particular that we either did not address at all or did not have time to solve due to the time constraints.

The problem we did not address occurs when forward velocity is close to zero. In this state, the predictions of the model cause the simulated platform to drift when the throttle is not applied. Also, if you apply steering action, the platform will drift in this direction from the state with zero forward velocity and with no throttle applied. This problem was not addressed since, in our task, we are not interested in states with close to zero forward velocity and no throttle applied.

The major problem well described in [22] is a violation of i.i.d. assumption. This happens because the observations in a simulation environment depend on previous predictions of a model. Hence, the distribution of observations in a simulation differs from that of training data that were collected in the real world. This leads to divergence between the behavior of the real and the simulated platforms even though the model fits training data quite well. We partially address this problem by clipping observations to the training data extremes in order to avoid unlimited divergence of the model predictions. But it does not solve the underlying problem completely since some observations are out-of-distribution of training data but by absolute value do not exceed extremes at any dimension of the observation space.

We also did not succeed at optimizing the TCNN hyperparameters. The TCNN is a promising model for the task, and it partially outperformed the MLP model, but we failed at tuning it to approximate the general dynamics of the real platform.

8.2 Control policy

The learned control policies have their flaws as well. If we take a look at the videos accessible via the link `https://www.youtube.com/channel/`

`UCb7esjH7TSxUHZ_tABx2E-w/videos`, we could notice that even in a simulation environment, the agent deviates a lot from the given trajectory. It is due to the rewards definition. We do not penalize the deviation from the trajectory because we failed to optimize the penalty that would not harm the training performance. Also, the agent overuses steering when following a straight line. This problem could be addressed by steering actions penalization, but in this work, such a penalty was not introduced.

Another problem is poor simulation-to-real world transfer. Despite the fact that the data-driven model-based policy performs better according to the 7.7, both policies closed a lot fewer waypoints on average compared to the simulation. For the MuJoCo-based policy, it is not that surprising since it is hard to optimize the physics engine parameters to closely match your real platform. But from the data-driven model-based policy, we expected better performance in the real world. Our explanation for this poor performance is the distribution shift problem described in the previous section that happens due to the violation of the i.i.d. assumption.

8.3 Future work

In order to thoroughly research the possible advantages of the data-driven model-based engine, some future work has to be done.

The primary concern is the optimization of MuJoCo parameters to resemble the behavior of the physical platform. It is challenging to fit the physics engine parameters to the physical platform and the environment. Hence, the performance of the MuJoCo engine could be arguably better, although, as was shown in the table 7.1 the MSE between real-world trajectories and simulated on MuJoCo trajectories was the smallest compared to any of the data-driven models.

However, on the other hand, the models we used for nonlinear dynamics approximation could be improved as well. A possible direction of improvement for the data-driven model is data collection and data preprocessing since Neural Networks heavily rely on data quality.

Another direction that could be pointed out is the architecture and training of Neural Networks. Because of the inexperience of the author choices made in this work were probably flawed. Therefore, it makes sense to rethink the used approach in order to get better model performance.

The performance of learned control policies could also be further enhanced. An unsolved task is the optimization of hyperparameters of the Reinforcement Learning algorithm. Due to the hardware limitations, it was unfeasible to run the automatic optimization. Hence, in the optimization lies the potential improvement. Another potential lies in the reward shaping process. Control policies demonstrated in this work were trained with a simple reward 5.1. There were also attempts to use a penalty for a deviation from the trajectory, but they were unsuccessful.

Chapter 9
Conclusion

This work experimentally demonstrated that a control policy trained on a data-driven model using Deep Reinforcement Learning performs better on the corresponding physical platform than a control policy trained on the close matching platform simulated by the physics engine MuJoCo.

According to the results, the control policy learned on the MuJoCo engine performed significantly worse in the real world compared to the simulation. On average, MuJoCo-based policy closed 33.02 fewer waypoints in the real world than in the simulation for the validation trajectories. At the same time, the policy learned on the custom engine based on the MLP with a history of observations lost only 13.66 waypoints on average.

Also, the total reward in the real world for the policy learned on the data-driven model is higher than that of the MuJuCo policy even when, in a simulation, the MuJoCo-based policy performed better. The MuJoCo-based policy got 106 rewards in the simulation versus 47,25 on the trajectory 'lap' in the real world. At the same time, data-driven model-based policy got only 87 rewards in the simulation and 60 in the real world.

The results are consistent with the initial hypothesis that the data-driven model provides more accurate simulation-to-real-world transfer as a learning environment for the control policy than the MuJoCo engine.

We also demonstrate that a control policy for nonlinear systems learned with Deep Reinforcement Learning algorithms can achieve high performance on general trajectories. The performance is comparable with the performance of a human driver with driving experience of several hours in total that was gained during this work. See tables 7.4, 7.5.

Specific goals that were fulfilled during the course of this work are:

- Dataset for the data-driven model learning gathered from the RC buggy platform. Data were analyzed and preprocessed, and issues of noise and incomplete state observability were addressed.

- Data-driven models of different architecture approximating nonlinear dynamics of the platform were learned, compared, and evaluated. The training pipeline was developed along with the visual wrapper for the learned model using Pybullet package [23] and MuJoCo engine [16].

- The pipeline for a control policy learning using the state-of-the-art Reinforcement Learning algorithm Proximal Policy Optimization was developed. Policies based on the MuJoCo engine and on the data-driven model-based engine were learned. The performance of both policies was evaluated in the simulation environment, and then they were deployed on the real platform.
- Results of experiments were described and analyzed.

Bibliography

[1] P. Cai, X. Mei, L. Tai, Y. Sun, and M. Liu, "High-speed autonomous drifting with deep reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 1247–1254, apr 2020. [Online]. Available: https://doi.org/10.1109%2Flra.2020.2967299

[2] A. Kumar, Z. Fu, D. Pathak, and J. Malik, "Rma: Rapid motor adaptation for legged robots," 2021. [Online]. Available: https://arxiv.org/abs/2107.04034

[3] J. Schoukens and L. Ljung, "Nonlinear system identification: A user-oriented road map," *IEEE Control Systems Magazine*, vol. 39, no. 6, pp. 28–99, 2019.

[4] D. Ha and J. Schmidhuber, "Recurrent world models facilitate policy evolution," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/2de5d16682c3c35007e4e92982f1a2ba-Paper.pdf

[5] G. Williams, N. Wagener, B. Goldfain, P. Drews, J. M. Rehg, B. Boots, and E. A. Theodorou, "Information theoretic mpc for model-based reinforcement learning," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 1714–1721.

[6] M. T. Gillespie, C. M. Best, E. C. Townsend, D. Wingate, and M. D. Killpack, "Learning nonlinear dynamic models of soft robots for model predictive control with neural networks," in *2018 IEEE International Conference on Soft Robotics (RoboSoft)*, 2018, pp. 39–45.

[7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. [Online]. Available: https://arxiv.org/abs/1707.06347

[8] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of

Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 1889–1897. [Online]. Available: https://proceedings.mlr.press/v37/schulman15.html

[9] RCprofi, “Rtr model bxr.s1, specifications,” Accessed: 2022. [Online]. Available: https://www.rcprofi.cz/rtr-brushless-buggy-4wd-hobbytech-bxr.s1?gclid=EAIaIQobChMI9c_I-c3X9gIVUY9oCR2p9ABlEAAYASAAEgKRmfD_BwE

[10] NVIDIA, “Nvidia jetson nano,” Accessed: 2022. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/

[11] Intel, “Intel realsense tracking camera t 265,” Accessed: 2022. [Online]. Available: https://www.intelrealsense.com/tracking-camera-t265/

[12] Amazon, “Microsoft xbox 360 wireless receiver for windows,” Accessed: 2022. [Online]. Available: https://www.amazon.com/Microsoft-Xbox-Wireless-Receiver-Windows/dp/B000HZFCT2

[13] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, “Stable baselines,” https://github.com/hill-a/stable-baselines, 2018.

[14] Wikipedia, “Simplex noise,” Accessed: 2022. [Online]. Available: https://en.wikipedia.org/wiki/Simplex_noise

[15] Open AI, “Open ai gym library,” Accessed: 2022. [Online]. Available: https://gym.openai.com/

[16] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 5026–5033.

[17] T. Erez, Y. Tassa, and E. Todorov, “Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 4397–4404.

[18] MuSHR, “Mushr platform,” Accessed: 2022. [Online]. Available: https://mushr.io/tutorials/quickstart/

[19] Y. Chen, Y. Kang, Y. Chen, and Z. Wang, “Probabilistic forecasting with temporal convolutional neural network,” *Neurocomputing*, vol. 399, pp. 491–501, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231220303441

[20] C. Lea, M. D. Flynn, R. Vidal, A. Reiter, and G. D. Hager, “Temporal convolutional networks for action segmentation and detection,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[21] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, "How to train your robot with deep reinforcement learning: lessons we have learned," *The International Journal of Robotics Research*, vol. 40, no. 4-5, pp. 698–721, 2021. [Online]. Available: https://doi.org/10.1177/0278364920987859

[22] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Gordon, D. Dunson, and M. Dudík, Eds., vol. 15. Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011, pp. 627–635. [Online]. Available: https://proceedings.mlr.press/v15/ross11a.html

[23] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2021.