ON THE MODELING OF DYNAMIC-SYSTEMS USING SEQUENCE-BASED DEEP NEURAL-NETWORKS

A Dissertation Proposal
Presented to
The Academic Faculty

By

Antoine Richard

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2022

ON THE MODELING OF DYNAMIC-SYSTEMS USING SEQUENCE-BASED DEEP NEURAL-NETWORKS

Thesis committee:

Prof. Cédric Pradalier
Computer Science
Georgia Institute of Technology

Prof. Bloch Matthieu
Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Alexandre Locquet
Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Geist Matthieu
Brain Team
Google Research

Dr. Kira Zsolt
Computer Science
Georgia Institute of Technology

Date approved: 4/29/2022

ACKNOWLEDGMENTS

I would like to thank my supervisors, Prof. Cédric Pradalier and Prof. Matthieu Geist, for giving me the unique opportunity of doing a thesis with them. Their support, guidance, and mentorship have enabled me to explore many exciting topics. During this Thesis, I have learned tremendously about my research field, but also about me. I have had a blast, and for that, I am deeply grateful.

I would like to express my appreciation to the rest of the committee members, Prof. Matthieu Bloch, Dr. Zsolt Kira, and Prof. Alexandre Locquet, for the precious time they spent reading my thesis.

I am grateful to my amazing friends and coworkers at GeorgiaTech without whom this thesis would not have been possible. Stéphanie Aravecchia, Antoine Mahé, and Luis Batista, thank you for helping me with my experiments in the cold of the winter, shivering while we waited for “Kiki” to perform its tasks. Pete Schroepfer, Dr. Assia Benbihi, thank you for your mentoring, advice, and support. Your experience and our talks had a tremendous impact on me and this thesis. I would also like to thank Dr. George Chahine, Othmane Ouabi, and Laura Monnier, for our interesting discussions and the time we shared.

I would also like to thank Dr. Offer Rozenstein for inviting me to Israel. I had a great time doing research in your lab, and I hope we will have the opportunity to collaborate again in the future. Lior Fine, thank you for the warm welcome, your assistance on the irrigation methods, and conducting the experiments with the farmers.

Finally, I would like to thank my parents François and Florence Richard, my girlfriend Camille Gardelle, and my brother Maxime, for their boundless love, support, and understanding. I would also like to thank my childhood friends, and the Calpe for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ARMA Auto Regressive Moving Average

CNN Convolutional Neural Network

DA Domain Adaptation

DL Deep Learning

DNN Deep Neural Network

DR Domain Randomization

EC Eddy-Covariance

ELU Exponential Linear Unit

ET EvapoTranspiration

GCRL Goal Conditioned Reinforcement Learning

GRU Gated Recurrent Unit

KL Kullback-Leibler

KNN K-Nearest Neighbour

LAI Leaf Area Index

LReLU Leaky ReLU

LSTM Long Short-Term Memory

LUT Look Up Table

MBE Mean Bias Error

MBRL Model Based Reinforcement Learning

MBRL Model Free Reinforcement Learning

MDP Markov Decision Process

MDS Marginal Distribution Sampling

MDV	MeanDiurnalVariation
MIMO	multiple-input multiple-output
ML	Machine Learning
MLP	Multi Layer Perceptron
MPC	Model Predictive Control
MPPI	Model Predictive Path Integral
MSE	Mean Squared Error
NLG	Natural Language Generation
NLP	Natural Language Processing
NN	Neural-Network
PER	Prioritized Experience Replay
POMDP	Partially Observable Markov Decision Process
PPK	Post Processing Kinematic
PReLU	Parametric ReLU
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network
RTK	Real Time Kinematic
SELU	Scale ELU
SISO	single-input single-output
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
USV	Unmanned Surface Vehicle
VAE	Variational Auto-Encoder

SUMMARY

The objective of this thesis is the adaptation and development of sequence-based Neural-Networks (NNs) applied to the modeling of dynamic systems. More specifically, we will focus our study on 2 sub-problems: the modeling of time-series, the modeling and control of multiple-input multiple-output (MIMO) systems. These 2 sub-problems will be explored through the modeling of crops, and the modeling and control of robots. To solve these problems, we build on NNs and training schemes allowing our models to out-perform the state-of-the-art results in their respective fields.

In the irrigation field, we show that NNs are powerful tools capable of modeling the water consumption of crops while observing only a portion of what is currently required by reference methods. We further demonstrate the potential of NNs by inferring irrigation recommendations in real-time.

In robotics, we show that prioritization techniques can be used to learn better robot dynamic models. We apply the models learned using these methods inside an Model Predictive Control (MPC) controller, further demonstrating their benefits. Additionally, we leverage Dreamer, an Model Based Reinforcement Learning (MBRL) agent, to solve visuomotor tasks. We demonstrate that MBRL controllers can be used for sensor-based control on real robots without being trained on real systems. Adding to this result, we developed a physics-guided variant of DREAMER. This variation of the original algorithm is more flexible and designed for mobile robots. This novel framework enables reusing previously learned dynamics and transferring environment knowledge to other robots. Furthermore, using this new model, we train agents to reach various goals without interacting with the system. This increases the reusability of the learned models and makes for a highly data-efficient learning scheme. Moreover, this allows for efficient dynamics randomization, creating robust agents that transfer well to unseen dynamics.

CHAPTER 1
INTRODUCTION

1.1 About the PhD dissertation topic

The proposed dissertation topic emerged from the wish of evaluating and developing machine-learning algorithms to model dynamic systems and deploy them in the field. We will explore three different domains: the modeling of low-dimensional dynamic systems, the application of control to these systems, and finally, the modeling and control of high-dimensional systems. To test the modeling capacities of NNs, we chose to apply them on two types of "systems": crops, and robots. Across these two domains, we will meet the same problems: modeling a dynamic system with few data points and a strongly unbalanced dataset. As we solve the different problems we will use similar methods starting from Multi Layer Perceptron (MLP) to Long Short-Term Memorys (LSTMs) to Transformers. Furthermore, in all the problems that we will encounter, the inputs are always processed sequentially. Finally, all the methods developed here are built and designed to be applied and deployed in the field, on real systems.

1.2 Low-Dimensional Systems Modeling: Application to Crops-Water-Consumption

Crops and their water consumption are a compelling, low-dimensional, subject of study. Indeed, they present cycles that are often based on the time of the day, but also the seasons. We designed NNs capable of natively leveraging the cycles in crops to improve their accuracy. The NN presented here have been developed to solve two tasks: a forward prediction task, and a gap-filling task. In practice, our architectures learn to model the crops' water consumption, also known as EvapoTranspiration (ET). This task is particularly interesting as crops exhibits day/night cycles and rapidly changing dynamics.

1.2.1 Achieving Expert Level Irrigation using DNN based Crop-Modeling

Initially, to evaluate the potential of NNs in the field, we tested them on a forward prediction task. This is the usual application case of NNs where they are given a set of inputs and tasked with predicting an output. With this experiment, we wanted to see if a NN could estimate the ET of crops using only recordings from two tomato fields. To do so, we designed an algorithm to forecast the growth of the crop canopy. Then, using the forecasted canopy growth and a carefully selected set of meteorological variables, we trained a NN to estimate the water consumption at a half-hourly rate. We then developed a mobile app and a server-backend to predict the ET. This application was then used by farmers to irrigate their fields in real-time. In the real field experiment, we showed that our model was capable of performing as well as an expert with access to soil moisture levels, and better than standard irrigation practices in Israel. This algorithm is about to be tested on a large scale in the Hula Valley by Tomato farmers.

1.2.2 Filling Gaps in Evapotranspiration Measurements

Then, we moved on to a more challenging task: filling gaps in measurement streams. In this task, we have measurement streams with 10 to 30% missing values. The data losses occur over a continuous period of time, creating large gaps in the data.

Our goal is to reconstruct the missing values inside these gaps. To account for the day/night cycles present in crop ET measurements, we designed an NN architecture inspired by Transformers. To fill gaps in ET measurements, most often, the literature uses an interpolation technique called Marginal Distribution Sampling (MDS). This method leverages the day/night cycles present in the data to fill the gaps without modeling the crop itself. In the end, our architecture demonstrated superior capacities when compared to the state-of-the-art methods while being highly sample efficient. In particular, it better estimated local trends and offered a gain of around 15% in Root Mean Squared Error (RMSE) on real-world datasets[1].

1.3 Data-Efficient Modeling & Control for Robotic Systems

When learning robots' dynamic models, one of the key limiting factors is interacting with the physical system to collect the data required to learn. Indeed, collecting data on real systems is often a very long and costly process; even simulators are relatively expensive to run. In the following, we focus on improving the training efficiency of NNs. Furthermore, we investigate if the techniques used to make NNs more efficient can also improve the quality of a controller using said NNs.

1.3.1 Prioritized Data Sampling for Improved Modeling

To assess the performance of the different prioritization schemes, we first applied on a system identification task two techniques: the Prioritized Experience Replay (PER) and the gradient upper bound. These methods draw samples with higher error more often. This can improve the performance of NNs. However, if there are outliers in the data, it may also degrade their performance. Hence, to better understand how they can be applied to learn robots' dynamic models, we studied the impact of their hyperparameters. To do so, we recorded datasets from different systems, with or without data-inbalance, and performed grid searches to find the most optimal parameters. In the end, we showed that these methods were capable of properly identifying the models even with poor action-space/state-space exploration.

1.3.2 Prioritization applied to control

With the characterization of the prioritization methods' parameters done, we then applied them to the control of robotic systems. In particular, we used the dynamic models learned using the prioritization inside an MPC controller, and analyzed their impact on the controller performances. In simulation, we demonstrate that these methods can provide a significant performance uplift in particular when the system has not fully explored its action/state space, or when the model was learned on a strongly unbalanced dataset. Finally, we applied the lessons learned in simulation on a real boat tasked with following lakeshores at a given velocity. This boat successfully performed its task, however, it had a high computational cost and required for the dynamic model to be learned on the real system.

1.4 Model Based Reinforcement Learning for mobile robots

To improve upon the results of the MPC controller, we explored model-based policy learning. A computationally lighter alternative to MPC. We use MBRL with policy learning to control and model robotic systems using both low and high dimension inputs, such as velocity readings or laser scanners.

1.4.1 MBRL for navigation in GPS denied environments

Using state-of-the-art MBRL with policy learning, we studied its performance on the same shore following task as the MPC. However, this algorithm had only access to the laser-scanner measurements. With this test the objective was twofold. First, we wanted to compare the performance of the two approaches in simulation and make sure the Reinforcement Learning (RL) agent was capable of controlling our robot as well as the MPC. Second, we studied the transferability of the learned policy from the simulation to the real world. In [2], we showed that not only it was capable of doing so, but our system out-performed the state-aware MPC controller even though the RL agent had never been trained on the real system. We further assessed the robustness of the system by increasing its drag, deploying it in flooded environments, and an frozen lake where the robot had to break through a 2 to 5mm thick ice layer.

1.4.2 Dynamic model separation for modular MBRL

Building on the same MBRL algorithm, we modified it to be better suited to mobile robots. In general, when learning how to solve a task, MBRL algorithms learn a so-called world model. It models the environment in which the robot evolves as well as the robot itself. Here, we chose to split this model into two submodels, one learns the dynamic model of the robot, while the other learns the residual. This allows changing the dynamic model while preserving environment knowledge and vice-versa. In this model, we add the physical state measurement to the inputs of the model. Overall, in [3], we show that this model performs better and is more robust than its predecessor.

1.4.3 Improving the reusability, robustness and transferability

So far, all the models learned to navigate at a single velocity. In order to improve the flexibility of our models, we introduced a mechanism to allow agents to reach multiple velocities efficiently. Our method was extensively tested both in simulation and on a real system. During these experiments, policies trained with this method showed superior power efficiency and strong robustness. Moreover, we have investigated efficient ways to further improve the robustness of the learned policies. These showed promising simulation to real transfer results.

1.5 Publications

The proposed thesis was the motivation behind the following first author publications:

- **Richard A.**, Fine L., Malachy N., Pradalier C., Tanny J., & Rozenstein O., **Data-Driven Estimation of Actual Evapotranspiration to Support Irrigation Management**, AAAI, AIAFS Workshop, 2022.
- **Richard A.**, Aravecchia S., Geist M., & Pradalier C., **Learning Behaviors through Physics-driven Latent Imagination**, Conference on Robot Learning (CORL), 2021.
- **Richard A.**, Aravecchia S., Schillaci T., Geist M., & Pradalier C., **How To Train Your HERON**, ICRA RA-L, 2021.
- **Richard A.**, Fine L., Rozenstein O., Tanny J., Geist M., & Pradalier C., **Filling Gaps in Micro-Meteorological Data**, European Conference on Machine Learning (ECML

PKDD), 2020.

- Mahé A.*, **Richard A.***, Aravecchia, S., Geist M., & Pradalier C., **Evaluation of prioritized deep system identification on a path following task**, JINT, 2020. (*Shared first authorship)

- Mahé A.*, **Richard A.***, Geist M., & Pradalier C. **Importance Sampling for Deep System Identification**, ICAR, 2019. (*Shared first authorship)

The work aforementioned has also led to the following submissions as second author:

- Fine L., **Richard A.**, Tanny J., Pradalier C., Rosa R. & Rozenstein O., **Introducing State-of-the-Art Deep Learning Technique for Gap-Filling of Eddy Covariance Crop Evapotranspiration Data**, Water, 2022.

- Pradalier C., Schroepfer P., **Richard A.**, **A Graph-based Approach to the Initial Guess of UWB AnchorSelf-Calibration**, IROS, 2022. (Submitted)

- Khazem S., **Richard A.**, Constant T., Fix J., & Pradalier C., **A modern Toolbox for high precision Trees' Xray analysis**, Computer and Electronics in Agriculture, 2022. (Submitted)

- Venkataramanan A., **Richard A.**, & Pradalier C., **A Data Driven Approach to Generate Realistic 3D Tree Barks**, Graphical Models (GMOD), 2022. (Submitted)

CHAPTER 2
FUNDAMENTALS & LITERATURE SURVEY

Black box modeling using neural-network has been around for a long time [4]. However, until recently, their usage was limited due to their huge data requirements, their lack of reliability, and the computational power required to train and infer them. Yet, today, these problems are starting to fade away, as the amount of data and compute available on edge devices has kept increasing over the last decade. Because our application domains are fairly different, each chapter in this thesis will include its own literature survey. In this chapter, we will focus on the elements that are commonly shared across each application field. First, in section 2.1 we provide a brief introduction on NNs as well as a presentation of the architectures relevant to this thesis. Then, in section 2.2 we present part of the literature on NN architectures designed for sequential processing and system identification. This section also discusses works on estimating the uncertainty in neural-networks predictions on both regression and classification tasks. Finally, Section 2.3 focuses on work related to making the neural-networks training more efficient.

2.1 Fundamentals on Deep Neural Networks

This thesis makes extensive uses of NNs we invite the readers that want to refresh their memory to follow this short summary about Deep Neural Networks (DNNs).

2.1.1 What is Deep Learning?

Deep Learning (DL) is a subfield of Machine Learning (ML) that enables an algorithm, or model, to solve a task by learning from a set of data. The reason behind its name is that the models are usually deep, i.e. they have a lot of "layers" stacked together. This subfield is usually separated into three categories: supervised learning, unsupervised learning, semi-supervised learning. In supervised learning, the model is learned by providing for each element of the dataset the complete answer to the problem it must solve. This implies having a dataset that has been labeled. Conversely, in unsupervised learning, the algorithm does not have access to the solution of the problem. Hence, no labels are needed. Semi-supervised learning methods sit in between these two categories. In this scenario, only a small portion of the data has been labeled. Or, alternatively, the algorithm could start with an empty dataset. It would then collect its own data to learn from. This setup is commonly found in reinforcement learning.

2.1.2 Vocabulary

- Generalization: The ability of a NN to work on data outside its training distribution. This is a highly desirable feature for a model.
- Regularization: A process that consists in modifying the learning algorithm to improve its generalization capacities. An exhaustive list of regularization techniques can be found in [5].
- Overfitting: The fact that a NN over-learned on its training data. This results in a model with a high precision on its training set, but with a poor accuracy on the real data. I.e. a model that does not generalize.
- Underfitting: The fact that a NN fits poorly to the training data. This results in an unusable model.
- Latent space: is an embedding of a set of samples. Within that space, the samples that are the most similar to each other are positioned closer.

2.1.3 Commonly used Layers

The most basic element of any DNN is the artificial neuron. While it is called an artificial neuron it does not realistically model a biological neuron. A more realistic approach to biological neurons is spiking neural networks [6] that we will not cover here. This neuron takes as an input $x \in \mathrm{I\!R}^{\mathrm{N}}$, and outputs a $y \in \mathrm{I\!R}$. The neuron itself is composed of a weight vector $w \in \mathrm{I\!R}^{\mathrm{N}}$ and a scalar bias $b \in \mathrm{I\!R}$. The model of the neuron is given by Equation 2.1, where σ is the activation function.

$$y = \sigma(w^T x + b). \tag{2.1}$$

σ can either be a linear or non-linear function, but most of the time they are non-linear functions, enabling the NN to learn complex non-linear mappings. Among the most common activation functions one can find Rectified Linear Units (ReLUs) or softmax, more on them in subsubsection 2.1.3.

Dense Layers

Dense layers, or fully connected layers, are with convolutions the main building blocks of neural-networks today. The dense layer is the simplest layer. It is composed of a multitude of artificial neurons organized in a computationally friendly fashion. A dense layer with $N \in \mathrm{I\!N}$ neurons and an input $x \in \mathrm{I\!R}^{\mathrm{M}}$ has N weights $w \in \mathrm{I\!R}^{\mathrm{M}}$ organized in a matrix of weights $W \in \mathrm{I\!R}^{\mathrm{N \times M}}$. Similarly, the biases are organized in a single array $B \in \mathrm{I\!R}^{\mathrm{N}}$. The output $y \in \mathrm{I\!R}^{\mathrm{N}}$ is given by the equation in Equation 2.2, where σ is applied to each element of the resulting array.

$$y = \sigma(W^T x + B). \tag{2.2}$$

Convolution Layers

A convolution layer [7] is a function dedicated to the analysis of grid-structured inputs. This function processes three types of inputs, time series $x_{1D} \in \mathrm{I\!R}^{\mathrm{L \times C}}$, images $x_{2D} \in \mathrm{I\!R}^{\mathrm{H \times W \times C}}$, and voxels or videos $x_{3D} \in \mathrm{I\!R}^{\mathrm{H \times W \times D \times C}}$, where L is the length of the time series, W and H are respectively the width and height of the image or voxel, and D is the depth of the voxel or the number of frame in the video stream. C is the number of channels in the input. Unlike dense layers whose weights match the size of the input, convolution layers have a kernel of fixed size, unrelated to the size of the input. A typical 2D convolution has its kernel defined by $K \in R^{N \times M \times O}$, where $N, M \in \mathbb{N}$ are the size of the kernel, and O is the number of learnable filters. N and M define the "reach" of the convolution, the local region within which the convolution will analyze the data. In addition to their kernel, convolutions use a bias $b \in \mathbb{R}^{\mathrm{O}}$. Once the convolution is applied it returns an output $y \in \mathbb{R}^{\mathrm{H \times W \times O}}$, the 2D convolution operation is given in Equation 2.3, where $i, j \in \mathbb{N}$ are the position in the 2D grid, and o is the current filter that is being applied.

$$y_{i,j,o} = \sigma\left(\sum_{c=1}^{C} \sum_{n=1}^{N} \sum_{m=1}^{M} K_{n,m,o} x_{i-n,j-m,c} + b_o\right). \tag{2.3}$$

This process is applied on every single cell of the grid O times, the number of filters. Here, for the sake of simplicity, we do not consider border effects that reduces the size of the output by half of the kernel size. However, to prevent this from happening, one can use padding.

Recurrent Layer

Recurrent layers, also referred to as cells, are different from the previously seen layers as they have a feedback connection. Their previous output is connected to their input. They rely on their "state" also known as "hidden-state" to detect patterns in sequences of inputs. The term sequence implies that the data are chronologically ordered, it can be time-series, video frames, etc... An exampled of a basic Recurrent Neural Network (RNN) layer is given in Equation 2.4, where $x(t) \in \mathbb{R}^{\mathrm{N}} \forall \mathrm{t} \in [1, \mathrm{T}]$ is the input with T the size of the sequence. $s \in \mathbb{R}^{\mathrm{M}}$ is the state of the recurrent layer, $W_1 \in \mathbb{R}^{\mathrm{N \times M}}$ and $W_2 \in \mathbb{R}^{\mathrm{M \times M}}$ are the weights of the model.

$$\begin{aligned} s(t) &= y(t-1). \\ y(t) &= \sigma(x(t)W_1 + s(t)W_2). \end{aligned} \tag{2.4}$$

More complicated and commonly used recurrent cells can be found in subsubsection 2.1.3 and subsubsection 2.1.3.

Long-Short-Term Memory

LSTMs[8], are a more advanced type of RNN. They feature good memory capacities while being relatively light-weight. LSTMs are explicitly designed to remember information for long periods of time. To do so, an LSTM unit is composed of four components: the cell c, the input gate i, the output gate o, and the forget gate f. The gates regulate the flow of information into and out of the cell while the cell remembers past values. The equations that rule the behavior of the LSTM are given in Equation 2.5. W are the inputs weights and U are the hidden state weights, B is the bias. x is the input, y is the output and hidden-state, c is the cell state.

$$\begin{aligned} f(t) &= \sigma_s(W_f x(t) + U_f s(t-1) + B_f). \\ i(t) &= \sigma_s(W_i x(t) + U_i s(t-1) + B_i). \\ o(t) &= \sigma_s(W_o x(t) + U_o s(t-1) + B_o). \\ \tilde{c}(t) &= \sigma_h(W_c x(t) + U_c s(t-1) + B_c). \\ c(t) &= f(t)c(t-1) + i(t)\tilde{c}(t). \\ y(t) &= o(t)\sigma_h(c(t)) \end{aligned} \tag{2.5}$$

The key idea behind how the LSTM work can be seen in Equation 2.5 with $c(t)$. In this equation, the previous cell state $c(t-1)$ is factorized with forget gate. This step is used by the LSTM to remove information from the cell state. Then, some “fresh” information is added through the input gate, using the past hidden state $y(t-1)$ and the new input $x(t)$. This relatively simple process allows the LSTM to remember or discard information.

Gated Recurrent Unit

Gated Recurrent Units (GRUs)[9] are a simpler version of the LSTMs. Their output gate was removed, and they have fewer parameters. Their simpler structure allows for faster execution, and eases the learning on small datasets, as fewer parameters need to be learned. Their performance on most task are akin to the one of the LSTMs. The GRU are made of two gates, the update gate u, the reset gate r, and a cell that is also the output y. The equations that rule the behavior of the GRU are given in Equation 2.6. W are the inputs weights and U are the hidden state weights, B is the bias, x is the input.

$$\begin{aligned} u(t) &= \sigma_s(W_u x(t) + U_u s(t-1) + B_u). \\ r(t) &= \sigma_s(W_r x(t) + U_r s(t-1) + B_r). \\ \tilde{y}(t) &= \sigma_h(W_h x(t) + U_h(r(t)y(t-1)) + B_h). \\ y(t) &= (1-u(t))y(t-1) + u(t)\tilde{y}(t) \end{aligned} \tag{2.6}$$

Activation functions

Activation functions are arguably one of the most important component in a NN, they come in many flavors, and it can be hard to keep track of all existing functions. In Table 2.1, we present a non-exhaustive list of the most commonly used functions.

In general, we can split the activation function into two groups: the saturated activation functions, and the non-saturated activation functions. Over the last decade, non-saturated activation functions have slowly replaced their saturated counterparts. Indeed, saturations can lead to the vanishing gradient problem: preventing the network from learning correctly. Yet, the strong non-linearity of saturated activation functions can be better for some application domains. Among the non-saturated function we find the sigmoid, the softmax, and

Table 2.1: A list of commonly used activation functions.

Activation	Reference
`Mish`	[10]
`Swish`	[11]
`GELU`	[12]
`ReLU`	[13]
`ELU`	[14]
`Leaky ReLU`	[15]
`SELU`	[16]
`SoftPlus`	
`SReLU`	[17]
`ISRU`	[18]
`TanH`	
`RReLU`	[19]

the hyperbolic-tangent. The sigmoid activation function features a smooth non-linearity, as shown in Equation 2.7

$$f(x) = \frac{1}{1+e^{-x} \forall x \in \mathbb{R}}. \tag{2.7}$$

The hyperbolic tangent, is simply the mathematical function often shorten as `tanh`. This function is very similar to the sigmoid. The main differences are a negative mapping and a steeper gradient. Another commonly used function in classifier is the softmax, its equation is given in Equation 2.8.

$$f(x)_j = \frac{e^{x_j}}{\sum k = 1^K e^{x_k}} \forall j \in 1, ..., K. \tag{2.8}$$

All of these functions are "dense" functions. This means that their gradient is more expensive to compute than, for instance, the ReLU gradient, which is "sparse".

The large ReLU family is composed of three primary functions : the standard ReLU, the Leaky ReLU (LReLU), and the Parametric ReLU (PReLU). First applied to neural networks in [13] the ReLU is defined as in Equation 2.9.

$$y_i = \begin{cases} x_i, & \text{if } x_i \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.9}$$

Introduced in [15], the LReLU is defined as in Equation 2.10 where α is generally a fixed value.

$$y_i = \begin{cases} x_i, & \text{if } x_i \geq 0 \\ -\frac{x_i}{\alpha}, & \text{otherwise} \end{cases} \tag{2.10}$$

The LReLU prevents some “locked” neurons from blocking the learning of the whole stack by propagating some of the gradients to the upper layers. PReLUs are akin to the LReLUs with the exception that the parameter α is learned along with the rest of the network parameters.

Among the many activations functions, computer vision architectures seem to favor ReLUs style functions as well as Swish/Mish. In reinforcement learning, architectures tend to prefer Exponential Linear Units (ELUs), Scale ELUs (SELUs), SoftPlus, and Swish.

2.1.4 Common Architectures

The field of machine learning has produced a wide variety of architectures over the last years. However, they all adhere to similar construction principles. In the following we provide the basic elements to understand the different architectures that will be discussed in this Thesis.

Multi-Layer Perceptrons & Convolutional Neural Networks

MLPs, or Feed-Forward, are the most primitive type of neural-network. They are made of a succession of dense layers (subsubsection 2.1.3), sometimes separated by batch-normalization, and dropout layers. It is also common to find them at the end of encoders.

Autoencoder

An auto-encoder is a type of architecture designed to perform unsupervised feature extraction. Its objective is to learn a representation, or encoding, of a dataset. This is usually done to reduce the dimension of said data, and ease its manipulation. This type of architecture features two main elements, the encoder, and the decoder. The data is sent through the encoder, which compresses the data and projects it onto a compact latent space. The compressed data is then decompressed by the decoder, whose goal is to reconstruct the original data. This process is similar to a lossy compression algorithm. Regarding the encoder and decoder, they can be made of any layer. Often they are made using Convolutional Neural Networks (CNNs), but it is not a defining element. The one thing that matters is the dimension reduction. Formally, the auto-encoding process can be written as in Equation 2.11. Where x is the input, $\hat{x}$ is the reconstructed input, F_{enc} is the encoder, z the encoded variable, and F_{dec} is the decoder.

$$\begin{aligned} \hat{x} &= F_{dec}(F_{enc}(x)) \\ &= F_{dec}(z) \end{aligned} \tag{2.11}$$

To learn how to reconstruct the input multiple option are possible. One can use a standard Mean Squared Error (MSE), or more complicated loss functions. Auto-encoders style architectures are particularly popular in Generative Adversarial Networks. In these networks, the loss is often the combination of the prediction of another neural-network, and one or more human designed cost functions.

Variational Auto-Encoder

The following description gives an overview of Variational Auto-Encoder (VAE), the reader can refer to [20] for a more thorough description. VAEs[21] are often assimilated to regular auto-encoders as they share the same type of structure. However, there are significant differences, most notably on their goal and mathematical formulation. Similarly to auto-encoders, VAEs encode the data onto a reduced latent space. However, the main goal of that latent space is to make the generation of novel realistic data easier. The term "variational" comes from the similarities this architecture shares with variational inference methods in statistics.

Initially, one might believe with a given learned decoder, generating new content should be as easy as sampling a random point from the latent space and decoding it. However, for this to be true, the latent space must be regular enough. Yet, when we train an auto-encoder, nothing is done to enforce the regularity of the latent space. Hence, to ensure that this is the case, VAE leverage regularization techniques. The main difference between a VAE and an auto-encoder, is that the former encodes the input into latent distributions, when the latter encodes them into a latent space. The encoding/decoding process is given in Equation 2.12. Where x is the input, $\hat{x}$ is the reconstructed input, F_{enc} is the encoder, μ and σ the mean and standard deviation of a set of normal distributions, $p(z|x)$ the latent distribution, z the encoded variable, and F_{dec} the decoder.

$$\begin{aligned} \mu, \sigma &= F_{enc}(x) \\ p(z|x) &= \mathcal{N}(\mu, \sigma) \\ z &\sim p(z|x) \\ \hat{x} &= F_{dec}(z) \end{aligned} \tag{2.12}$$

In Equation 2.12, the sampling step is not differentiable, hence this equation must be reworked to allow the gradients to be propagated. Equation 2.13 shows the reparametrization trick, allowing gradients to be back-propagated inside the network.

$$\begin{aligned} \varepsilon &\sim \mathcal{N}(0,1) \\ z &= \sigma\varepsilon + \mu \end{aligned} \tag{2.13}$$

To ensure that the network learns a well regularized latent space, the loss needs to be more constrained than the one used with auto-encoders. From a formal perspective, this problem can be rewritten within the variational inference framework. With x our input, the goal is to find z, a random vector jointly-distributed with x. To do so, we learn an encoder who approximates a posterior distribution $q_\phi(z|x)$, and a decoder who learns the likelihood distribution $p_\theta(x|z)$. Hence, the loss function must make sure that $p_\theta(x|z)$ is as close as possible to $q_\phi(z|x)$. This is commonly done using the Kullback-Leibler (KL) divergence. However, most often, to optimize this type of network the literature uses the Evidence Lower Bound loss, or ELBO (Eq. Equation 2.14). The complete derivation of the ELBO is given in [20].

$$\mathcal{L} = E_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)||p_\theta(z)) \tag{2.14}$$

2.1.5 Optimization

For the architectures and layers presented earlier to be applied to a problem, we first need to find a set of parameters that can solve it. In NNs this is done using gradient-based optimization algorithms. To apply these algorithms, we need to compute an error between the current results of the algorithm and its targets. The goal is to refine the value of the weights inside the network to improve the performance of the model. Using the fact that all the operations inside the network are differentiable, the optimization schemes can leverage the chain rule to compute the gradient for each operation within the network. This process of computing gradients is commonly referred to as backpropagation. Finally, the only thing missing to optimize the networks' weights is the learning rate. The learning regulates the learning process by weighting the gradient update.

While there exists a wide diversity of optimizers, most of the modern literature relies on Adam [22]. Its main advantage over optimizers such as the Stochastic Gradient Descent (SGD) is that its learning rate is auto-adaptative. It uses the average of the gradients' moments to automatically tune the learning rate for each parameter of the network. Moreover, it has a built-in decay, which means that over time the learning rate will decrease, making the convergence faster and more accurate. In this thesis, we used Adam for the whole of the trainings, as it is particularly robust [23].

2.2 Survey on Neural-Network based Modeling & Sequence Processing

This survey is focused on black-box modeling using NNs, while there exists many ways of modeling non-linear systems[24], we chose to rely on one of the most flexible methods: neural-networks[4].

The past ten years have seen NNs achieve exceptional results across a broad range of applications, in particular in Computer Vision, and Natural-Language-Processing. Additionally, recent advances in computing-hardware brought multi-teraFLOP performances to a credit card format with power-draw inferior to 15W. All this, coupled to the broader availability of data make deep-learning-based approaches extremely compelling.

When modeling dynamic systems, the models often rely on a sequence[1] of inputs to predict a sequence of data-points or a single data-point. Among the NNs architectures used to model dynamic systems we can find MLPs[25, 4, 26], 1D CNNs [27, 28, 29], RNNs [30], LSTMs[8, 31, 32, 33], and GRUs [9, 34]. All in all, those models can be categorized into two categories: non-recurrent-models, and recurrent-models.

[1]The term sequence implies that the used data are chronologically ordered

2.2.1 Non-Recurrent Architectures

Non-recurrent models include MLP and 1D CNN. The MLP is one of the most common NN architectures used for system identification. It relies on stacked dense layers, and their activation functions. The MLPs' relative simplicity makes them very interesting, as they can be linearized and integrated inside controllers. Furthermore, they are quick to infer, making them ideal for real-time execution. Their main drawback is that they process the input sequence indifferently of the order of the sequence elements. Hence, they are unaware of time dependencies and cannot model them.

1D CNNs[35, 28, 27], which consists of stacked 1D convolutional layers, partly solve this issue. By construction, a 1D CNN processes its input sequence using local features (depending on the size of the kernel), which means that even though it was not designed for that purpose, it processes the data in an ordered fashion. This observation is at the root of network like WaveNets[28] which enhance this behavior by using atrous convolutions[36] of larger sizes as the network gets deeper. This temporal processing is compelling as it behaves a bit like a recurrent neural network but without the added complexity of those networks. Also, when MLPs are mostly used for sequence to next point predictions, 1D CNNs can be used for seq2seq processing. In this case, the 1D CNN takes as an input a sequence and then outputs another sequence. This process would be analog to image segmentation, and similar architectures could be used. Their main disadvantages lie in the extra amount of hyper-parameters, increased complexity, and the fact they are still not processing the data with respect to its temporality.

2.2.2 Recurrent Architectures

Despite the fact that from a physical standpoint, the dynamic of a system depends on time. None of the previously cited architectures genuinely account for the temporality of the data. On the contrary, the family of RNN architectures is designed to explicitly depend on time. To do so they process the sequence elements iteratively.

The LSTMs were invented to capture long-term dependencies better. They rely on three gates (the input gate, the output gate, and the forget gate) used to predict the output and update their hidden state. Those gates are a form of regulation allowing more or less information to flow through them. GRUs, as introduced in [9], are somewhat similar to the LSTMs, but they combine the forget and input gates into one gate called the update gate. The main drawback of the recurrent architectures is their hidden state that needs to be handled expertly to maximize the performances of the networks.

2.2.3 Attention & Exotic Architectures

Recent advances in the field of Natural-Language-Processing have seen the rise of a new type of model that, like MLP, is non-recurrent, while still being to account for the temporality of the sequence like a recurrent network would. These models called Transformers are faster to train and infer that their recurrent counterparts while reaching higher performances in Natural-Language-Processing tasks. To achieve those results, models such as [37, 38] leverages a mechanism called self-attention to draw global dependencies between input and output. Self-attention, the mechanism at heart of these model is an attention mechanism that puts in relation different elements of a single sequence in order to compute a representation of that sequence.

On top of these generalist architectures new studies have shown promising physics-ruled neural-network such as [39, 40, 41]. This architecture draw inspiration from the Lagragian and Hamiltonian mechanics to design and constrain neural-networks such that they are more robust and learn systems that abide by the law of energy conservation. The main drawback of these contributions is that they often make constant energy assumption and/or consider perfect observability of the system which can be problematic in system where the cause for energy losses are not observable. Also, one could consider [42, 43], yet, these approaches based on Restricted Boltzman Machine scale poorly to high dimension inputs such as 2D lasers or images.

2.2.4 Neural-Networks & Uncertainty Estimation

In addition to neural network architectures, recent works[44, 45, 46] focused on evaluating the uncertainty of the NNs predictions. The lack of confidence on the NNs predictions is one of the main factors holding back the widespread adoption of these approaches in real-systems [44, 47]. While Bayesian NNs[46, 48, 49, 50] have been around for a couple years, their measure of the uncertainty was unreliable in particular on regression tasks. New studies[51, 52] based on evidential deep learning opened new research perspectives but were so far limited to classification tasks. Fortunately [44] introduced a novel formulation of evidential deep learning that can be applied to continuous problems such as system modeling.

One of the first method used to evaluate the confidence of the network was the Monte-Carlo dropout[46], its main advantage relies in its simplicity. The network is train with dropout layers, and is also inferred with the dropout layers active. In this method, the network is executed n times. The average of the predictions gives the network's results, and the entropy or the standard deviation of the predictions gives the uncertainty. However, it is known that neural-network are over-confident by nature[53]. Hence, this method can still lead to inaccurate uncertainty estimations. Also, this method is not good at catching out-of-distribution samples[44].

More recently, [44] showed that non-Bayesian neural-networks can be trained to predict a continuous model and estimate both data (aleatoric) and model (epistemic) uncertainties. Deep evidential learning places the uncertainty over a likelihood function, instead of the network weights as in Bayesian NNs. With evidential learning, each sample adds support to a learned higher-order evidential distribution. Sampling from this distribution yields instances of lower-order likelihood functions from which the data was drawn. By training a NN to output the hyperparameters of the evidential distribution, a representation of both epistemic and aleatoric uncertainty can then be learned without the need for sampling.

2.3 Survey on Sampling Schemes for Efficient Neural-Network Training

A NN is as good as its dataset[54]. This means that to achieve optimal performance, a lot of data-points are necessary. The sample inefficiency of NNs leads to the need to generate large demonstration datasets. These are often very unbalanced [55] and the training is saturated with common samples while interesting data points have little impact on the learning process. Usually, in classification tasks, this problem can be handled using class balancing methods such as balanced sampling, weighted sampling or applying a bias to the loss of the network. However, in tasks such as system modeling this is not that trivial: there are no classes and if we take the example of robotic systems, sampling uniformly over the state or action space is not always feasible.

All in all, properly learning hard cases is hindered by the imbalance of the data. The capacity of NNs to keep improving over new data by continuously training has been used to alleviate this problem [25]. However, this implies that we keep training the network on very well-known situations while new pieces of information are only seen once in a while. Identifying the informative samples on which to focus the learning makes it more efficient[56]. This is the philosophy behind one of the most used algorithm in RL: the prioritized experience reply. This method uses the loss of the model on the training set's samples to estimate how important those samples are. Then, it generates a sampling distribution based on the samples' importance to focus the training on the samples where the model has a higher loss.

While this method proved itself in RL, the PER suffers from major drawbacks. It necessitates the tuning of multiple parameters: α, β, and how often the sampling distribution must be updated. Tuning α and β makes [57]'s work unpractical, as a large grid searches have to be performed to acquire optimal prioritization parameters. Then, the update frequency of the sampling distribution is hard to tune, if it is refreshed too often the training loses efficiency, if it is not refreshed often enough the samples' importance remains the same even though their usefulness has evolved. However, one of the main advantages of this method is that it permits a fine control over how the prioritization is done. This is particularly interesting as it allows putting a low amount of prioritization at the beginning of the training to grasp the general modeling concept, and then increase the prioritization as the training reaches its end to maximize the learning of hard cases.

In [58] they study different sampling method and show that the loss of the network on a given sample can be used as an indicator of the sample's importance. However, [59] also outlines that using the loss can results in degraded learning performances. Fortunately, they show an interesting mechanism that alleviates both the tedious parameter tuning present in the prioritized experience replay and the need to update the sample importance based on the most recent network state. In their experiments the loss is no longer used as an estimate of a sample importance. Instead, they rely on an estimation of the gradient norm. Note that one could use the real gradient, but the computational cost is prohibitive. The prioritization weights are no longer saved for the time of the epoch but recomputed at every iteration on a super-batch[2]. This has two advantages: first as the super-batch is sampled uniformly which naturally reduces the over-fitting risks; second, it ensures that the scores are accurate for this instance of the network, thus preventing the risk to keep giving a high importance to samples that have become easy and further mitigating the over-fitting risk. Nevertheless, these advantages come at a cost: because the scores have to be recomputed at every iteration, the training process is slower than before. To get over that problem, a trigger is introduced to perform importance sampling only when we estimate that gradient acceleration is possible. This trigger is computed for free when the backward pass is performed. All in all, they showed that their approach out-performed all previous prioritization methods on classification tasks.

[2]A super-batch is a batch *n* times larger than the batch size used for training. In their work, it is suggested to chose a super-batch 3 times larger than the batch size

CHAPTER 3

LOW-DIMENSIONAL SYSTEMS MODELING: APPLICATION TO CROPS-WATER-CONSUMPTION

This chapter focuses on the modeling of low-dimension systems using neural-networks. The models are learned and applied to crops. The goal is to model the water usage of fields based on their current state and the weather. Figure 3.1 shows the type of fields our methods are applied to.

Figure 3.1: A tomato field in the Hula Valley, Israel.

Crops and their water consumption are a compelling, subject of study. Indeed, they present cycles that are often based on the time of the day, but also the seasons. In the following sections, we present NNs capable of natively leveraging the cycles in crops to improve their accuracy. The models presented here have been developed to solve two tasks: a forward prediction task (section 3.1), and a gap-filling task (section 3.2).

We applied our method in a specific region of the world, the Middle East. The rapid temperature change in this region is particularly interesting for irrigation experiments and crop modeling. Indeed, the plants grow very quickly, and the water consumption can increase drastically over short time periods, making modeling them hard.

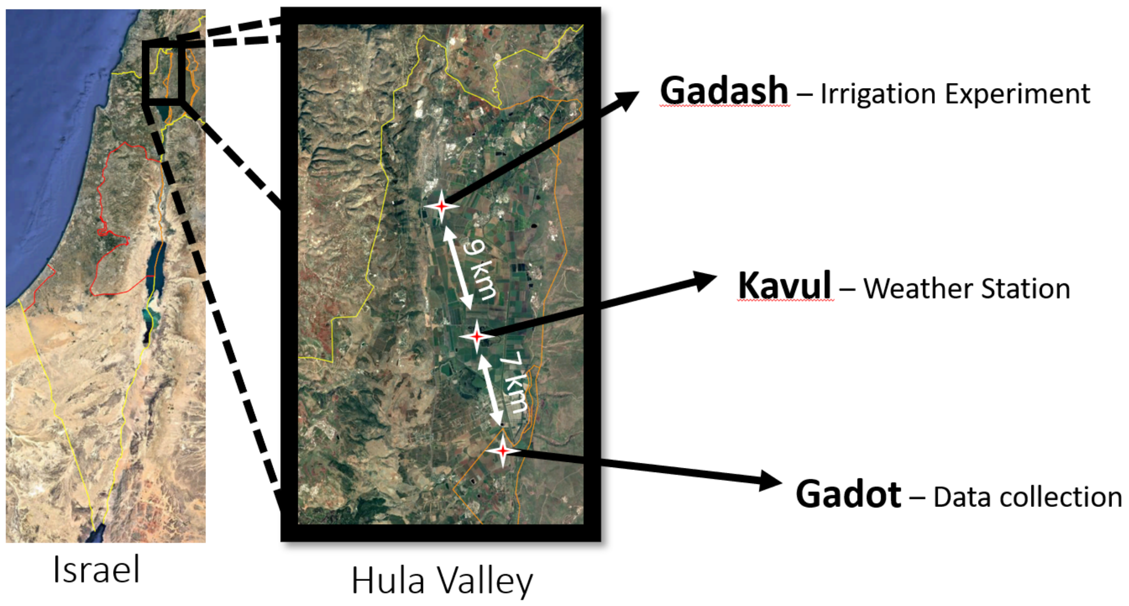


Figure 3.2: A map of Israel and our main testing in Gadot.

Figure 3.2 shows our main testing site in northern Israel. This site was used to collect data as well as a test area for our irrigation techniques. The work carried out in this chapter was done in collaboration with our Israeli partners, in particular Lior Fine, and Offer Rozenstein. They acquired the data and helped us better understand the systems we wanted to model. Without further ado, let's get modeling.

3.1 Achieving Expert Level Irrigation using DNN based Crop-Modeling

3.1.1 Motivation

Water demand is expected to increase by 55% globally between 2000 and 2050, mainly for manufacturing, electricity, and domestic use [60]. This will leave a small margin to increase water use in agriculture, and therefore, it is imperative to optimize the irrigation process. A precise estimation of crop water consumption, or evapotranspiration (ET_c), can improve irrigation management and lead to similar yields while reducing water usage throughout the growing season. Tomato *(Lycopersicon esculentum Mill)* is one of the most important vegetable crops globally, with production estimated by 180 million tons in 2017 [61]. It is also one of the most demanding in water [62]. Accordingly, improvement in tomato irrigation could result in significant water savings. Therefore, tomato is a suitable model crop for the evaluation of irrigation strategies.

A key challenge to achieve precise irrigation, i.e. using just the right amount of water, is to be able to estimate the water need of the plant as well as the evaporation related to the environment. Hence, when modeling the model of a crop, one needs to model the environment demand as well crop's. Another interesting effect, is that as the plant grows, the environment demand decreases and the crop's water need increases. This is due to plant canopy getting larger over the growing season. Thus, estimating the crop water consumption is not trivial, it depends on many variables such as the meteorological variables, the size of the crop's canopy, or their age. Please note that the plant canopy growth depends on the crop type, the amount of energy received. It is not just a function of the time.

For these reasons, precision irrigation is expensive, and often require experts. In Israel, the current best practice to determine the correct irrigation dose requires an agromet scientist, as well as an array of tensiometers. This is impractical for small farms or developing countries.

In this section we propose two methods, with pros and cons, to address the aforementioned issues. Specifically, the estimation of K_c from UAV multispectral imagery, and the application of an NN trained to predict latent heat fluxes based on meteorological data. These methods were tested against the current best practice in processing tomato irrigation and were shown to perform as well while being cheaper and simpler to use.

3.1.2 Related Work

The FAO-56 crop coefficient approach is one of the most commonly applied irrigation management methods [63]. Using this approach, ET_c is estimated based on the reference evapotranspiration from a hypothetical crop (ET_0) and is given by $ET_c = K_c \times ET_0$, where ET_0 is commonly derived using the Penman-Monteith method, while K_c for specific crops in specific environments is empirically determined in water consumption experiments to isolate the atmospheric evaporative demand from the plant reaction. Standard K_c tables based on such experiments may not be sufficiently accurate when the regular crop development is inhibited by stress factors or irregular weather conditions. Therefore, remote sensing estimations of K_c based on vegetation indices that reflect the ground cover and crop development level in near-real-time, can serve as surrogates of K_c that overcome this limitation [64, 65].

In a previous study [66], K_c estimation models were developed for processing tomatoes based on Sentinel-2 imagery that is available at a frequency of 5 days at 10–20m spatial resolution, and Venµs imagery that are available at a frequency of 2 days at 5–10m spatial resolution [67]. This development facilitates estimating K_c at a high enough temporal resolution for irrigation decisions that well capture within-field variability. However, in cloudy environments, even such a high revisit time may not be enough to support near-real-time estimations of K_c from optical satellite imagery. In addition, satellite pixels are too coarse to properly estimate K_c in narrow experimental plots. However, low flying Unmanned Aerial Vehicles (UAVs) can overcome such limitations. An UAV can capture imagery on days not covered by satellite overpass, and even under clouds [68]. Moreover, the spatial resolution of imagery from low altitude remote sensing is better suited for small plots [69]. K_c estimations from UAVs equipped with multispectral cameras have been previously used.

In parallel with the increased use of UAV, in recent years there is an upsurge in the use of machine learning for system modeling, not only for remote sensing data but also for irrigation management (e.g.,[70, 71]). Recently, [72] highlighted the potential of using deep learning techniques in geoscience for modeling dynamic time series. Wide research was done on prediction of reference evapotranspiration using machine learning ([73] and references therein), but few on actual evapotranspiration measurements over agricultural crops. Some work focused on using MLP to gap in ET_c time series [74, 75, 1]. However, here we want to make running predictions of the ET_c values. This is significantly harder as we do not have access to the past or future ET_c values. A key issue when trying to estimate running crop's ET_c values is that as the crop grows, its evapotranspiration increases as well. This means that we need to learn two things: The impact of the meteorological variable on the crop evapotranspiration, and how to model the plant growth.

3.1.3 Proposed Methods

We present two different methods aimed at maximizing tomato yield by irrigation management. To do so, we developed two original methods and built a mobile application to instructs farmers with irrigation recommendations based on our models. The different methods used are detailed in the following sub-sections.

Irrigation Estimation from UAV

The first method we present here consists of flying a UAV equipped with a multispectral camera to acquire images and estimate K_c to compute the required irrigation dose. The drones used can be seen in Figure 3.3. Figure 3.4 shows one of the eddy covariance tower used to collect data

The K_c estimation model developed for Sentinel-2 [66] was applied to multispectral

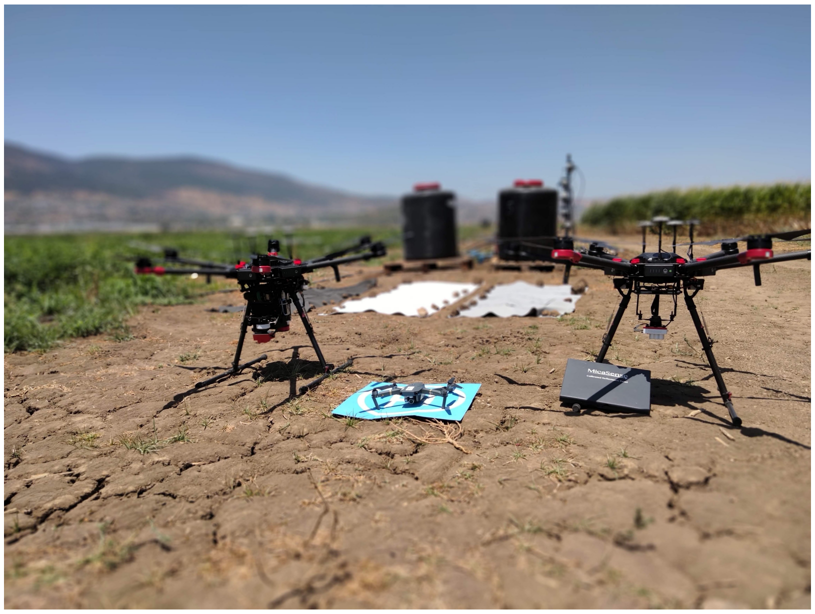


Figure 3.3: The UAVs used when collecting aerial images in the fields.

imagery acquired with a Micasense[1] RedEdge-MX Sensor. This work by [66] used eddy covariance measurements of the actual crop water consumption during three growing seasons to calculate the actual K_c and model it using spectral vegetation indices derived from spaceborne multispectral imagery. To apply this model to imagery acquired by the RedEdge-MX Sensor, a relative calibration between Sentinel-2 Level-2A products and RedEdge-MX imagery was carried out. We used co-acquired imagery of agricultural fields from four different dates and crops. UAV imagery was processed into orthomosaics using Pix4Dmapper (Pix4D S.A., Prilly, Switzerland). Satellite and UAV images were then resampled to 10 m resolution, and the area of the field was masked. Subsequently, linear regression models were fitted for overlapping pixels of Sentinel-2 and RedEdge-MX bands with similar central wavelengths. The result was transformation equations from Sentinel-2 to RedEdge-MX reflectance values for which the K_c estimation models could be applied to.

UAV flights took place at the irrigation experiment site in Gadash Farm in the Hula Valley (33°10'55"N 35°34'57"E) every 5-10 days during the 2020 growing season from an altitude of 50 m above the ground, with front and side overlap was 85% to facilitate the generation

[1]Seattle, Washington, USA

of a good equality orthomosaic. The average K_c value in UAV-treatment replicates together with ET_0 data from a nearby meteorological station (Kavul station; 33°06'03"N35°36'34"E) was used to calculate the actual ET_c in this experimental treatment and to instruct the grower with an irrigation recommendation via the mobile application.

Irrigation Estimation using a NN

The second method consists in using an NN to predict the ET_c. To do so, the NN used meteorological variables collected from local weather stations, and the average Leaf Area Index (LAI) of the control treatment. The LAI is measured on a fortnightly basis while the weather variables can be acquired at a ten-minute sampling rate or higher. Since the irrigation was applied every day, we first needed to build an algorithm capable of forecasting the LAI while taking into account past measurements.

Given that we only had LAI recordings from four past experiments, we chose to use a K-Nearest Neighbour (KNN) algorithm to predict future LAI values with a two days sampling rate. This meant that at the beginning, the LAI was modeled as the mean of all the past measurements, and then as we started collecting measurements, the extrapolated points followed the growth curve of the most similar examples in the database. To interpolate between the predicted points we fitted a spline on top of them, relaxing the shape of the predicted curve, and making the sampling process more convenient.

To estimate the daily irrigation dose, we trained a neural-network to predict the latent-heat-flux, a common proxy for ET_c. To do so, we used the LAI, acquired with the method outlined above, in combination of the following variables: the net-radiation, the temperature, the humidity, the wind speed, the time-of-the-day, and the days since germination. These variables are acquired by scrapping data from local weather stations, while the latent-heat-flux was measured using the eddy-covariance method during in previous collection campaigns in the region. Figure 3.4 shows of this tower installed in a freshly sowed field. We sampled all these variables at a 30 minutes rate and used them to train an MLP. This MLP

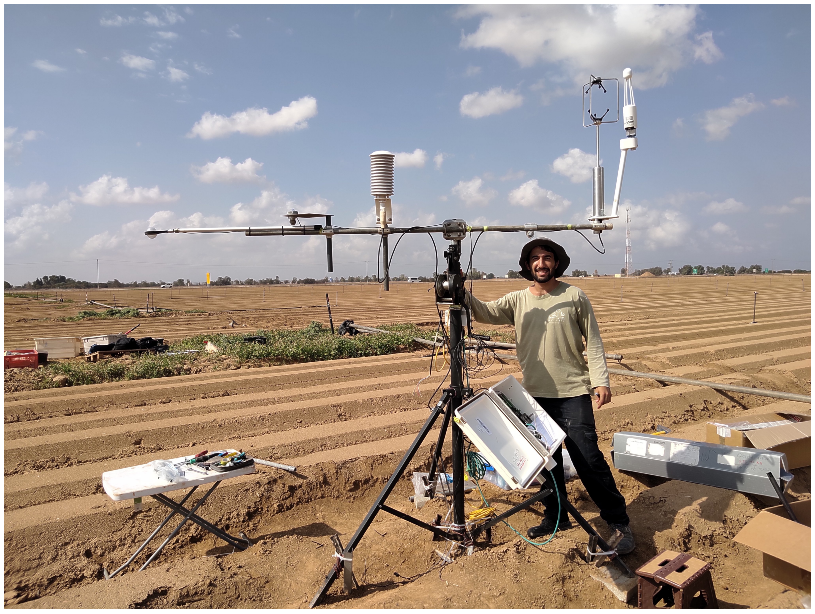


Figure 3.4: An eddy covariance tower in a field. Coworker for scale.

was composed of two dense layers with 48 neurons each, and a last dense layer with a single neuron. To minimize overfitting, we added dropout layers in-between each dense layer. All layers used LReLU activations [15], except for the last layer, which had no activation. The network weights were regressed using the adam optimizer, with a learning rate of 1e-5, and the drop rate was set to 0.3. Regarding the optimization function, we used a huber-loss as it made the training less rigid, and allowed to account for the inaccuracies in the variables fed to the network. To make our training more efficient, we also relied on a PER training scheme [57]. In the end, this models allowed us to predict the ET_c at a half-hourly rate. To get the daily treatment recommendation we integrated the predicted values over a whole day.

3.1.4 Baseline, Experiment & Evaluation

Baseline

To compare our methods, we used the current best-practice irrigation in Israel. This control treatment consisted of an expert relying on a set of soil tensiometers to determine the irrigation dose. The water tension in three depths was used as feedback to confirm the correct irrigation; if a desired water tension threshold was not reached, the next irrigation could be supplemented to reach the target value. Three more treatments were derived from the control treatment as ratios of 50%, 75%, and 125% of the control irrigation dose.

Experiment

To evaluate the different methods, we conducted an irrigation trial in an experimental crop-farm close to some of our previous data-collection campaigns (33°10'55"N 35°34'57"E). We selected the processing tomato cultivar H-4107 and transplanted them with a plant density of 2500 plant/dunam. After transplanting, the entire field was irrigated with 30mm water in order to fill the soil profile. Then it was irrigated according to the irrigation expert guidance for two months after which the irrigation trial began. In total, we tested six different irrigation treatments: 1) "Control" – the 'best-practice' irrigation, our baseline.

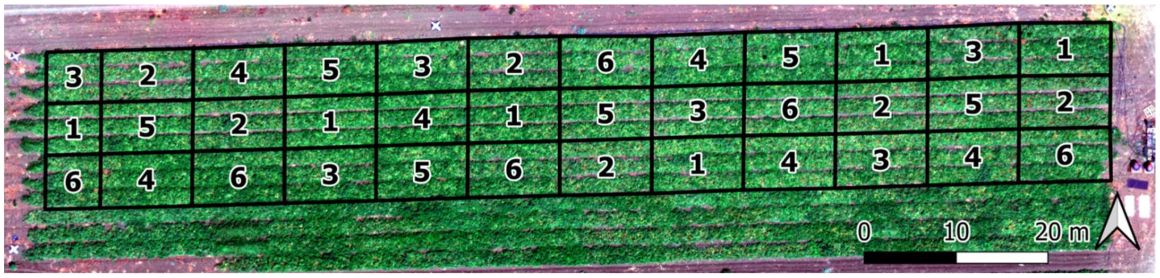


Figure 3.5: An overhead imagery of our experimental field and the different plots corresponding to each treatment.

2) 50% of the control. 3) 75% of the control. 4) 125% of the control. 5) "ANN" – irrigation based on the trained machine learning model. 6) "UAV" - the irrigation based on the K_c estimated from the UAV. The main assumption in this experiment was that the natural variability, which originates in environmental conditions, genetic material, equipment and management, is considerably smaller than the differences from the different irrigation treatments. Each treatment had six repetitions as can be seen in Figure 3.5 where each repetition was comprised of three 10 m by 2 m rows (60 m^2). The effects of the environmental conditions were also be mitigated by the scattering of the different repetitions across the

field.

Evaluation

To evaluate the performance of the different methods we used three evaluation metrics: the yield, water use efficiency, and brix. The yield is the most important metric, it is a measure of the fresh biomass of harvested tomato fruit per unit of area (e.g., ton / dunam).

This metric is supplemented by the water use efficiency, calculated by dividing the total yield (kg) with the total applied irrigation (m^3) in each treatment. The overall goal of the irrigation experiment was to maximize the yield while maximizing the water use efficiency at the same time. However, the optimization of the water use efficiency should not be done to the detriment of the yield. Hence, for now, having a higher yield is more desirable than having a higher water use efficiency.

Finally, Degrees Brix is a measure of the sugar content in an aqueous solution. One brix degree corresponds to one gram of sugar for 100 grams of liquid. In the case of the Tomatoes, the brix level is a common way to quantify their quality; higher is better, but there is usually a trade-off between quality and quantity. In general, less irrigation typically results in higher Brix but lower yield.

3.1.5 Web Platform

To enable farmers to apply our irrigation recommendation, we have developed a backend server using an Amazon Web Services EC2 instance[2]. The backend was used to extract data automatically from the weather station[3] using the selenium web driver[4], as well as running the neural-networks. On top of this, we have built a website on which the farmers could connect to get the irrigation dose for the day. Figure 3.6 shows the original website in 2020.

In 2021, for our new testing campaign, the website has been remodeled to allow farmers

[2]https://aws.amazon.com/?nc2=h_lg
[3]https://meteo.co.il/home/map?TargetIds=9,3,0,1
[4]https://www.selenium.dev/documentation/webdriver/

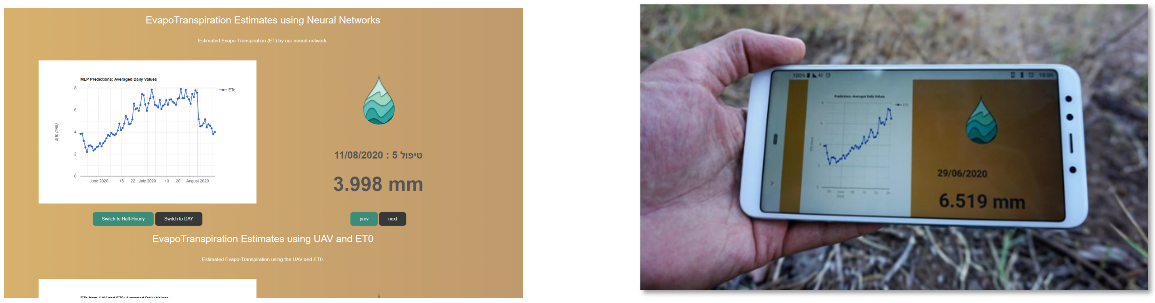

EvapoTranspiration Estimates using Neural Networks
11/08/2020 : 5 טיפול
3.998 mm
EvapoTranspiration Estimates using UAV and ET0
29/06/2020
6.519 mm

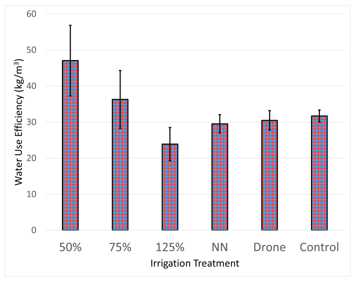
Figure 3.6: The irrigation website on a web browser and a smartphone.

to insert field measurements themselves. This new website, built with the help of the ministry of agriculture, can be seen here: https://irrigation-dss.agri.gov.il/.

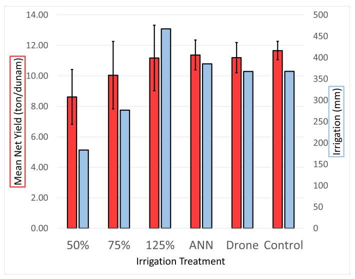
3.1.6 Irrigation Results

In this section, we compare the different treatments using the metrics defined earlier. Figure 3.7, shows the metrics for each method throughout the season. The histograms show both the average performance of the treatments across the six repetitions and their standard deviation. As can be seen on Figure 3.7b, the control, the NN, and the UAV, all achieve a similar yield, around 12 tons/dunam. The 125% treatment achieves the highest yield, while the 50% treatment achieves the lowest yield, 8.5 ton/dunam. On the same graph, we can also see that the 125% treatment consumed a lot more water than the control, while its yield was not significantly higher than the other methods. This shows, that our methods and the control are performing near the optimal yield/irrigation ratio. At the same time, the 50% treatment and 75% treatment consumed much less water but their yield is drastically reduced. The same pattern can be seen in Figure 3.7a, the 50% and 75% treatment both have a high water use efficiency but this comes at the cost of the yield, which is not desirable as we are first and foremost interested in the yield. On the other side of the spectrum, we can see that the 125% treatment has the lowest water use efficiency while it does not have a significantly better yield than the other treatments. In the end, we can see that from the irrigation and yield perspective our methods performed as well as the best-practice irrigation (control).

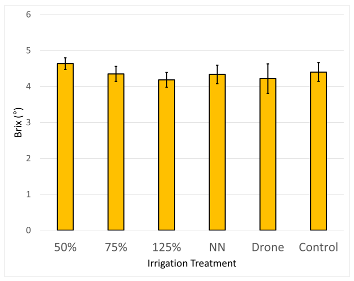
The brix measurement shown in Figure 3.7c, displayed very large variance across the



(a) Water use efficiency, higher is better.



(b) Yield against irrigation. Red, higher is better; blue, lower is better.



(c) Brix, higher is better.

Figure 3.7: Results of the different treatments over the whole season.

different repetitions of each treatment. The 50% treatment was slightly higher then the rest but this difference was not statistically significant.

Overall, this experiment, in which we delivered live recommendation to the farmer, was successful. The whole of the pipeline, from data-scrapping, to predicting, and sending the prediction worked reliably for the full summer season. This allowed us to show that the irrigation recommendation from the NN and the UAV almost perfectly agreed with the best practice, both in the total amount and rate of irrigation throughout the season. Moreover, they resulted in a similar yield and brix levels.

3.1.7 Discussion

In this experiment, we showed that methods based on NN could be used to achieve expert level irrigation. The main advantage of this method compared to the baseline resides in its low running cost, and ease of use. However, this method is timely to set-up as it requires careful calibration of the system. This calibration is unique to the area and crop-type, which means it has to done again for every new locations. Furthermore, it requires no supervision and is transparent to use. This makes it particularly interesting in developing countries, where the cost of advanced equipment and the availability of domain experts remain a key limitation to a wider adoption of efficient irrigation methods.

The other method, the UAV, is more expensive to run but does not require region-specific calibration and allows for highly accurate irrigation recommendation. It can be used anywhere on earth, deployed quickly, and does not need expert supervision.

3.1.8 Conclusion

Both novel approaches to determine the irrigation dose in processing tomatoes were found to perform equally to the control treatment of the best common practice for processing tomato irrigation. While the control treatment relied on an experienced agronomist specialized in vegetable crops cultivation that had the benefit of feedback from soil tensiometers, the experimental approaches, the estimation of K_c from an UAV, and the NN, did not. This makes these methods particularly interesting as they alleviate the need of crop experts and hence make efficient irrigation more affordable, which is crucial to broader the usage of high precision irrigation techniques. In our experiments, the multispectral imagery-based K_c estimation model, originally calibrated for Sentinel-2, was successfully transferred to work from a UAV with a multispectral camera payload. The trained NN model demonstrated its validity by estimating ET accurately. There were no significant differences in the yield quality and quantity between the approaches in the irrigation experiment. Although the study included only one irrigation experiment, the results illustrate the capacity and ease-of-implementation of novel techniques based on UAVs and NNs for irrigation management.

3.2 Filling Gaps in Evapotranspiration Measurements

3.2.1 Motivation

A precise estimation of crop ET is of importance to quantify terrestrial water budgets for irrigation purposes and to understand evaporation, transpiration, and photosynthesis processes. Most ET estimation methods are indirect and thus provide only an approximate estimation. Direct methods are expensive and technically complex [76] but provide a fairly accurate and reliable estimate of ET. The EC method is a direct approach for measuring field-scale ET over crops [77]. Most EC systems provide a time series of half-hourly average fluxes of latent and sensible heat and CO2 as well as momentum fluxes. Unfortunately, the percentage of missing data is between 20 and 60% of the original dataset [78] due to gaps of various lengths caused by different factors, including sensor malfunction, power breaks, and data quality filtering. To calculate daily, monthly, or yearly sums, these gaps must be

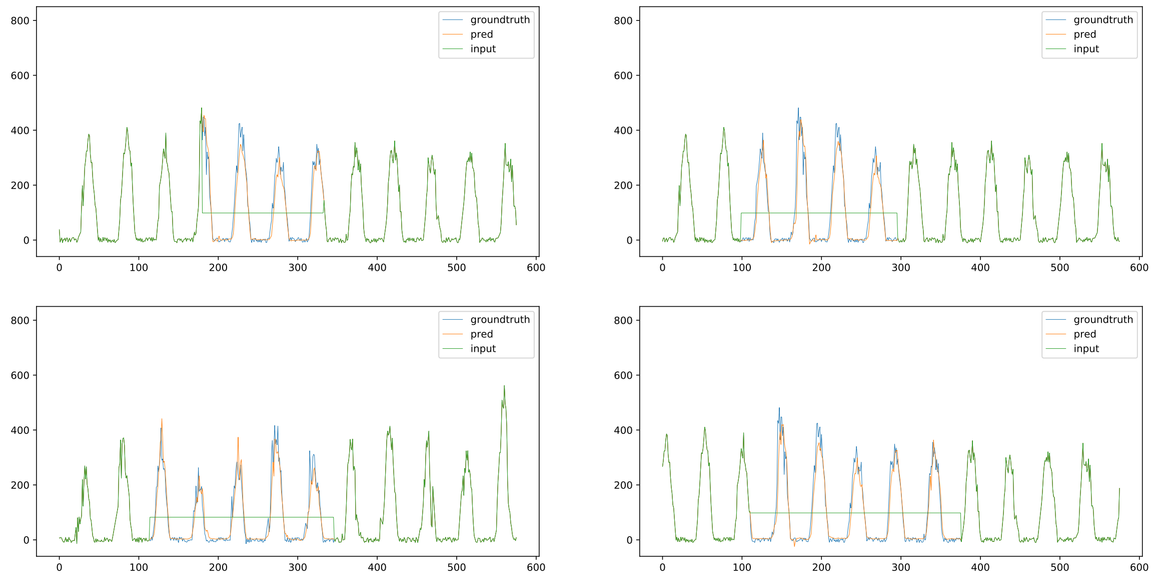


Figure 3.8: Four gaps in the eddy covariance data (green), and the gap filled by our neural network (orange) as well as the real value (blue).

reliably filled. Furthermore, when gaps occur in the data (e.g., when studying the ET diurnal curve or when comparing to other high-temporal-resolution ET measurement methods) accurate gap filling is required. The EC method is also widely used to measure fluxes over various crops [79]. These crops are managed ecosystems with rapid change throughout a growing season. In a warm climate, as illustrated in this article, a typical crop cycle could last 3-4 months, during which the conditions in the field change dramatically, mainly due to variations in the leaf area index and canopy structure. As a result, the boundary layer properties and the albedo change during the cropping period and large temporal variations of measured heat fluxes are observed [80]. Additionally, due to the warm climate, ET fluxes reach much higher values (and, as a result, higher errors when estimating them) than natural ecosystems like forests and grasslands in European climates. Further-more, gaps occurring during the short growing season lead to a limited amount of valid data, putting a strain on modeling the data in the gaps. These characteristics of EC measurements over field crops make gap-filling a challenging task.

3.2.2 Formal Description of the Problem

The system we want to model can be formulated as in Equation 3.1, where $x_i(t)$ is the set of observed variables such that $\forall i \in [1,n], \forall t \in [1,T]$, $x_i(t) \in \mathbb{R}$ and $z_i(t)$ is the set of non-observed variables such that $\forall i \in [1,m], \forall t \in [1,T]$, $z_i(t) \in \mathbb{R}$.

$$\begin{aligned} y(t) = f(x_1(t), .., x_n(t), z_1(t), .., z_m(t)) \in \mathbb{R} \\ \forall i \in [1,n], \exists \tau_i \in \mathbb{R}\,\mathrm{s.t.}\,\mathrm{x_i(t) = x_i(t+\tau_i)} + \varepsilon \\ \exists \tau \in \mathbb{R}\,\mathrm{s.t.}\,\mathrm{y(t) = y(t+\tau)} + \varepsilon \end{aligned} \tag{3.1}$$

We will denote as x the “sequence of observations” of our system, and y the “sequence of targets” of our system. The systems modeled are pseudo periodic with x_i and y values that are relatively similar from one day to another. The goal of our model will be to recover the missing values from the target sequence using the sequence of observations and the sequence of targets with missing values. The model is deployed offline, hence it can use values before, during, and after the gap. As such, it takes as inputs a sequence of observed variables $\mathcal{X}_i(t) = [x_i(t-T), x_i(t+1-T)..,x_i(t)]$, the sequence of targets with missing values $\mathcal{Y}_{deg}(t) = [y(t-T), y(t+1-T),..,y(t)]$, and a mask indicating where the data is degraded $\mathcal{M}(t) = [y(t-T), y(t+1-T),..,y(t)]$. T being the length of the sequence processed. The function g our models will learn is given in Equation 3.2.

$$\hat{\mathcal{Y}}(t) = g(\mathcal{X}_1(t),..,\mathcal{X}_n(t),\mathcal{Y}_{deg}(t),\mathcal{M}(t)) \in \mathbb{R}^{\mathrm{T}} \tag{3.2}$$

3.2.3 Related Work

Machine Learning Applied to Fill-Gaps

There are many methods applicable to fill gaps in data-streams, for instance, early methods used to replace missing data using Look Up Table (LUT). It consists in using the previous occurrence that resembles most the point we are trying to recover. A different approach to this problem, is to use KNN. Here, we consider that only the target data $y(t) \in \mathbb{R}$ is corrupted, but that the remaining data $x_i(t) \in \mathbb{R}$ is correct. Hence, we can fit a probability distribution over $Y|X$, and the missing values can be recovered by sampling the marginal distribution over X and computing $P(Y|X)$. In practice, it amounts to performing a local weighted average over available values of $Y|X$. The weights are proportional to their distance from X, and only samples in the vicinity of the gap are used. The main drawback of those methods is that they solely rely on the neighboring points. Hence, if the weighting window is too small, there may not be occurrences similar to the points to be filled. However, when the horizon increases, the impact of non-observable variables (latent variables) may significantly deteriorate the filling quality. This is particularly true in systems with high paced dynamics, where the latent variables are responsible for amplitudes changes.

With the rise of deep neural-networks, these methods are being challenged. Standard MLPs have been successfully used to fill gaps in slowly changing time-series. Those time-series have no hidden local trends and very large data-banks which make the system identification process easy. In this case, MLPs are often using the observation of the system

$x_i(t)$ at a given point to predict the value of $y(t)$. Yet, it is not uncommon to see MLPs using $\mathcal{X}_i(t)$ to predict $y(t)$ or $\mathcal{Y}(t)$. However, when processing sequences, MLP are suboptimal as they cannot natively embed temporal relations, and cannot extract patterns over long sequences. This means that if those networks were used for rapidly changing systems, they would not be able to extract local trends to re-scale their forecasts.

RNNs are a good alternative to MLPs when processing sequences. The recurrent nature of RNNs allows them to remember past information, and architectures such as LSTMs [8, 81] and GRUs [9] are commonly used to solve time-series forecasting problems. Unfortunately, to the best of our knowledge, in the case of gap filling, the machine learning literature has little to offer. While these methods address the problems encountered with MLPs, they have limitations of their own. Indeed, LSTMs and GRUs suffer from limited memory capacity and range. Because they are recurrent architectures, the hidden-state of the network, which is comparable to a memory, will go through n transformations, with n the number of elements in the sequence. This means that correlating information from elements at the beginning of the sequence with those at the end will get harder as the sequence gets longer. Additionally, properly initializing the RNNs' hidden state is tricky and can lead to prediction errors, especially in the case of small datasets.

This memory capacity is one of the key points alleviated by Transformers [37] and more recently Bert [38]. The mechanism at their root, self-attention, offers the benefits of recurrent models without their downsides. In this section we make heavy use of Transformers, hence, we provide an introduction to this type of networks in subsection 3.2.4. It also presents why their properties make them particularly well suited for gap-filling.

Gap Filling in Irrigation

In order to solve the aforementioned problem, various gap-filling methods that use available data to reconstruct the missing parts have been developed [78, 82, 83, 84, 74, 85, 86, 87, 1]. Most of the methods are based on empirical techniques that derive and parameterize the relations between certain drivers and the fluxes. Usually, the drivers are meteorological variables (e.g., air temperature, solar radiation, etc.) measured on-site or at a nearby meteorological station. Several methods have been suggested in the literature, from basic methods like Mean Diurnal Course and Lookup Tables [82], and their integrated version (Mean Distribution Sampling; [83]), to more sophisticated ones such as nonlinear regression [78, 82, 84], artificial neural networks [74, 85, 86] and random forests [87]. A comprehensive comparison of 15 gap-filling techniques based on ten benchmark datasets showed that different methods performed almost equally well, suggesting little room for improvement [78]. However, these datasets are year-long and represent European forests, which are characterized by conditions that are much less time-varying than those of the rapidly changing croplands. Furthermore, most of the methods presented in that comparison and the broader literature have been developed and tested on CO2 flux rather than on ET.

Unlike the shallow networks, used in most studies on gap filling and modeling of EC flux data [74, 85, 86], DL models can learn to extract features and take advantage of the spatial or temporal structure of the data streams in a hierarchical way. EC and additional bio-meteorological measurements from flux towers generate multi-dimensional time series, and these, in general, have been the topic of application of these new classes of neural networks [88]. Recently, Reichstein et al. [72] highlighted the potential of using DL techniques in geoscience for modeling dynamic time series. In the context of ET estimation, DL has been evaluated for interpolating local ET prediction to regional scale by correlating terrain appearance with ET [89]. However, as far as we know, this kind of model has never been examined in the context of gap-filling of EC data.

Only a few papers have dealt with gap-filling of crop ET [82, 90, 91, 92]. These studies, used various methods, e.g., MDS, Kalman Filter, Multiple Linear Regression, and MeanDiurnalVariation (MDV). In recent years, however, new advances in machine learning, specifically DL, were developed, with high capabilities in time series modeling. Nevertheless, so far, none of the studies on ET gap-filling available in the literature employed NN. Hence, the major goal of this research was to examine the suitability of such DL approached to fill gaps in eddy covariance ET data over crops.

3.2.4 Fundamentals on Transformers

A Transformer is an NN architecture introduced by [37] which makes use of the self-attention mechanism, weighting differently each element of the input sequence based on its significance. Similarly to RNNs, transformers are designed to handle sequential data. Unlike the RNNs, transformers do not necessarily process the data iteratively or in order. This makes them faster, as they can process a whole sequence at once. While their original intended application was primarily Natural Language Processing (NLP) and Natural Language Generation (NLG), the last two-year have seen increased usage of Transformers in computer vision, where they seem particularly performant on large datasets [93, 94].

Overview

The original Transformer architecture is structured similarly to an auto-encoder. It features an encoder that projects a sequence $[\mathcal{X}_1(t), ..\mathcal{X}_n(t)] \in \mathbb{R}^{\mathrm{NxT}}$ into a latent space $\mathcal{Z}(t) \in \mathbb{R}^{\mathrm{MxT}}$ with $M >> N$. And a decoder that decodes this latent space to create a novel sequence $[\mathcal{Y}_1(t), ..\mathcal{Y}_k(t)] \in \mathbb{R}^{\mathrm{KxT}}$. If we take the example of translating a sentence in French into English, the encoder takes in a given French sentence, encodes it into a latent sequence, and the decoder turns it into an English sentence. The latent space represents a shared language between the encoder and the decoder. Figure 3.9 shows the original transformer architecture.

To achieve this, the first thing the networks does is “embedding” the input, or projecting it to a higher dimension space. Then it adds a positional encoding, allowing the network to know the position of each element in the sequence. With all this done, it can now extract contextual information from the embedded data. To do so, it uses its “multi-head attention” blocks, which contain the self-attention layers. Once this step is done, we can get the latent space by mixing the output of the different attention heads through some dense layers (Feed-

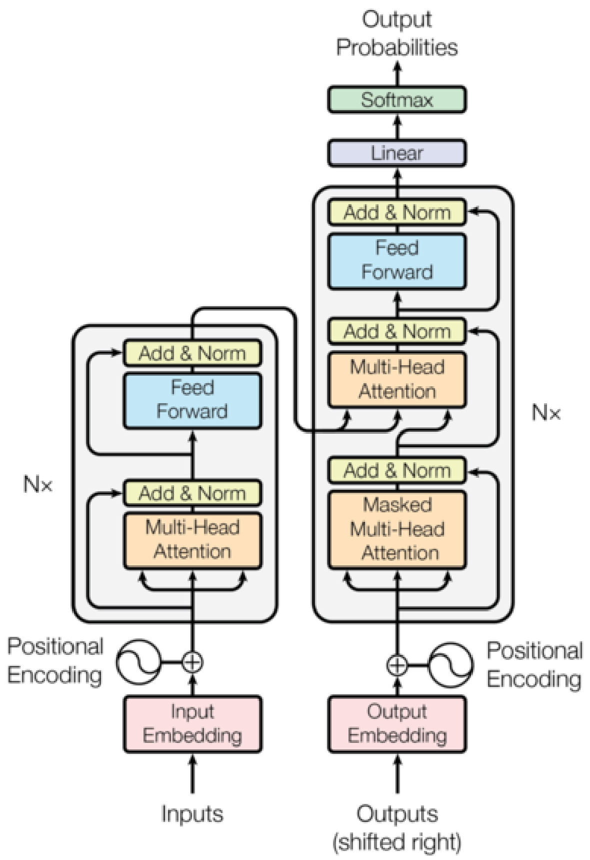


Figure 3.9: The transformer architecture. The encoder is on the left, the decoder is on the right. (Image from [37])

Forward). The decode operation is a bit more complicated. Instead of simply decoding the latent space, the current output is encoded, and sent through a masked attention layer. The mask prevents the network from accessing the real data at training time. When inferring the model, this data does not exist yet. This results of the masked-attention process is then used to pull information from the latent space, which are then mixed through some more dense layers, and finally projected to the desired output shape.

There are two elements that make transformers so performant at processing sequences: the positional encoding, and the attention mechanism. So let us start by exploring how the attention works in transformers.

Scaled Dot-Product Attention

In transformer, the Scaled Dot-Product Attention computes the correlation for every combination of elements pairs in the sequence under the form of an attention matrix. The process of the attention is given in Equation 3.3, where *input* is the encoded input. W_{k_i}, W_{q_i} and W_{v_i}, are projection weights of a given attention-head i, used to compute the key $K \in \mathbb{R}^{\mathrm{d_k}}$, the query $Q \in \mathbb{R}^{\mathrm{d_k}}$, and the value $V \in \mathbb{R}^{\mathrm{d_k}}$. A is the attention matrix, or attention weights, which once applied onto the value V gives the output of the Scaled dot product. Please note that in some instances, V can be acquired using a different input. This can be seen on the second attention mechanism in the decoder of transformer.

$$\begin{aligned} K &= W_{k_i} input \\ Q &= W_{q_i} input \\ V &= W_{v_i} input \\ A &= softmax\left(\frac{KQ}{\sqrt{d_k}}\right) \\ output &= AV \end{aligned} \tag{3.3}$$

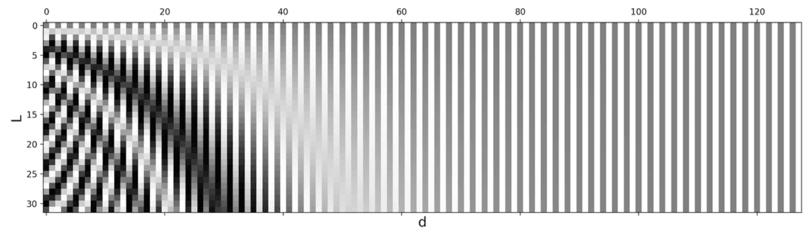
Overall, the attention weights in A are defined by how each element of the sequence projected to Q are influenced by all the elements in the sequence projected to K. The softmax is applied to make an even distribution between 0 and 1 and ease the learning process. When applying multi-head attention, the idea is that while the input remains the same, the weights in the W_{k_i}, W_{q_i}, W_{v_i} matrices will be different for each head. Hence, the network can learn with each head different type of element association.

Positional Encoding

Unlike RNNs, which recurrently process the elements of the sequence, attention models cannot know where the different elements are positioned inside the sequence. To alleviate this issue, transformers use a positional encoding, which gives a unique value (an identifier) to each element of the sequence. Despite the potential benefits of learning this encoding [95], which could improve the performance, [37] shows that there are no benefits from using a learned positional encoding. Instead, they used sine and cosine functions given in Equation 3.4 to encode the position. $pos \in \mathbb{N}$ is the position in the sequence, i is the dimension of the encoded input.

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \end{aligned} \tag{3.4}$$

A visual representation of this function in given in Figure 3.10. The left axis represents the position of each element of the sequence, while the top axis is the depth of the encoding. It can be seen that each element has a unique encoding.



[*] https://lilianweng.github.io/lil-log/2020/04/07/the-transformer-family.html

Figure 3.10: The positional encoding of a sequence of size 32 with a depth, d_{model}, of 128. (Image from Lil'Log[*])

Benefits for gap filling

Now that we know how transformers works, let us have a look at what makes them well suited to fill gaps in data. If we recall the one of the key limitation of RNNs is their memory, and the difficulty to correlate elements that are far apart in a sequence. With transformers, this is not a problem anymore. The positional encoding coupled to the attention mechanism allows associating elements across the whole sequence without any loss. Furthermore, because each attention heads works in parallel to one another, increasing the capacity of the network is relatively cheap. Another advantage of the transformers is the ability to visualize the attention matrices. These matrices can give insights about what is important in the input data. Finally, the positional encoding can be manipulated to easily account for cyclicity in the data, such as day/night cycles.

3.2.5 Proposed Approach

When building our model dedicated to filling gaps, we took inspiration from the canonical attention architecture: Transformer[5]. However, instead of using the whole model, we only relied on a single part of the transformer model: the encoder. Our architecture can be seen in Figure 3.11 This simplification of the encoder/decoder architecture reduces the number of parameters within the model, reducing its complexity and making it easier to train. Similarly to transformer, our architecture features multiple attention-heads which process embedded inputs.

Initial results with a single head showed poor performances: the network was having a hard time separating the different variables. Indeed, in the case of the $\mathcal{X}$ sequence, the model should look everywhere, but in the case of the $\mathcal{Y}_{deg}$ sequence, it should not use values inside the gap. Hence, based on that observation, and in an effort to minimize the learning complexity, we chose to manually assign different variables to our network heads. We used three attention heads: one head processes the observation sequence, another head processes the target sequence, and finally, the last head processes a concatenation of observation and target sequences. Ideally, we would let the model learn how to separate the variables on its own, but, due to the limited training samples, this was not feasible on our evapotranspiration datasets.

After applying the multi-head attention mechanism, the output heads are concatenated

[5]Complete details of the architecture is given in subsection 3.2.4.

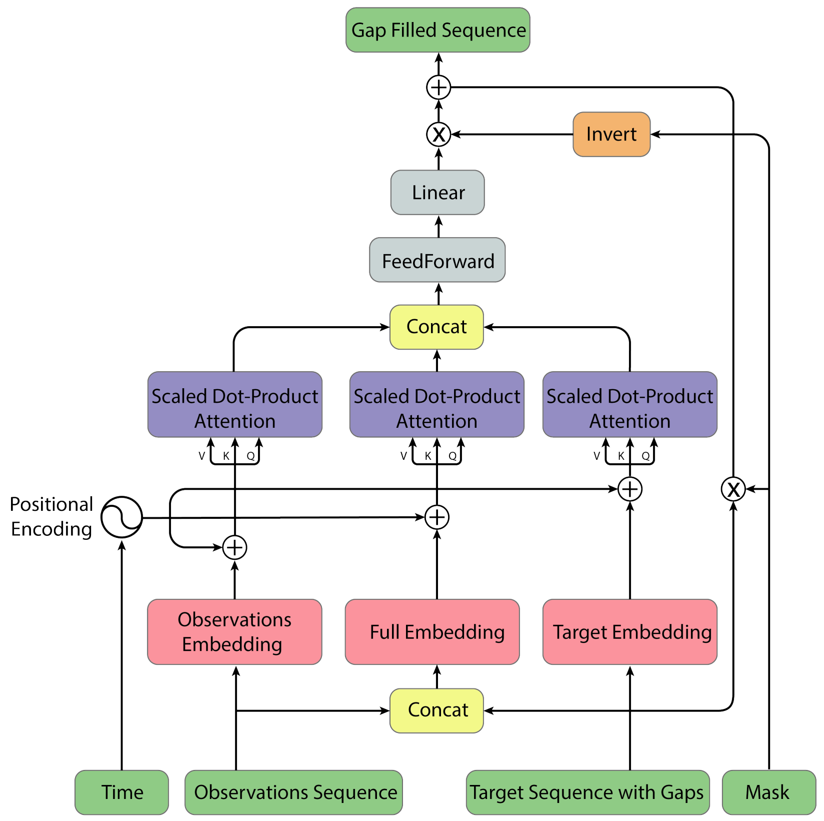


Figure 3.11: Our architecture for gap filling

and passed to a feed-forward layer: two dense layers with LReLU [15] activation function, and a dropout layer. Similarly to [96], our models performed approximately 10% better when using LReLUs over normal ReLUs. After the feed-forward layer, a dense layer is used to project back the output of our feed-forward layer to a one-dimensional sequence: the target sequence with its missing data filled.

Furthermore, to improve the performance of the network we use a mask $\mathcal{M}$ in the overall structure to copy the non-gap-points, and set the values of $\mathcal{Y}_{deg}$ inside the gap to -1. This tells the network it should not consider these values, -1 was chosen as the inputs are $[0, 1]$ normalized, so they clearly stand out. Using that same mask, we implemented a full skip-branch, directly copying the original target value onto the output. This prevents the network from learning a complicated function, where part of the target sequence is copied, and the rest is changed to fill the gap. As of now, the detection of the gaps in the data is handled by the EddyPro software[6]. This software is one of the most prevalent in the field and features a wide panel of failure detection methods.

To natively account for the cycles, we modified the positional encoding function. Instead of using the position of the elements in the sequence to perform the positional encoding, we chose to use the value of the primary periodic variable: in the case of the micro-climatic data, the time of the day. Hence, the positional encoding becomes cyclic. This lets us account for the periodic time dependencies natively. Similarly to the Transformer model, we use sine and cosine functions with the modifications mentioned above, as can be seen in Equation 3.5. $pos \in \mathbb{N}$ is the position in the sequence, i is the dimension, and $t(pos) \in \mathbb{N}$ is period.

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(\frac{t(pos)}{10000^{\frac{2i}{d_{model}}}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(\frac{t(pos)}{10000^{\frac{2i}{d_{model}}}}\right) \end{aligned} \tag{3.5}$$

Despite the appeal to learn the encoding [95], which should, in theory, allow the network to learn the frequencies that make most sense for the problem at hand, we chose not to. Firstly, based on the conclusion of [37], it seems that there are no benefits from using a learned positional encoding. Secondly, this lets us reduce the complexity of the learning process in regard to the limited amount of training examples at our disposal.

From a practical perspective, the positional encoding depends on the period from the different sequences of the batch, which have different time offsets. This means that it cannot be computed ahead of time. Thus to maximize performance, the positional encoding is computed directly within the network's graph.

[6] https://www.licor.com/env/products/eddy_covariance/software.html

3.2.6 Evaluation Datasets

We tested our model on two cases: a real use case from the field of agriculture and a toy case to demonstrate that our approach also scales to other related problems. For the application on real data, we aim to demonstrate that our methodology is capable of strong generalization by learning on a growing season in different crop fields and using this knowledge to fill gaps in situations that were not encountered before: a different crop at a different season at a different location.

Toy Problem

To test our architecture, we developed a small toy problem that features similar construction to our general problem. We create 3 variables $x_i(t) \in \mathbb{R}, \mathrm{t} \in \mathbb{N}$ s.t. $x_i(t) = \alpha_i + \sin(t * \gamma + \tau_i) * \beta_i + \varepsilon_i$ and γ is computed such that the periodicity of the variables is 48 points. Additionally, we create a latent variable (which will not be observed by our methods) $z_1(t) \in \mathbb{R}$ defined as $z_1 = \sin(t * \omega)$ where ω is set such that the periodicity of z_1 is 720 points. We chose 720 as this is larger than the maximum scope of our neural-networks. We then combine those variable to form Equation 3.6.

$$y(t) = \left| x_1(t)^2 \times e^{x_2(t)} \times \log\big(x_3(t)\big) \right| \times \big(z_1(t) + 2\big) + \varepsilon \tag{3.6}$$

This problem is interesting because the cyclicity of the positional encoding of our networks is set to 48 points and not 720 points. Hence, our model will have to adapt to the amplitude change created by z_1 and extract local trends to estimate the correct values. A total of 70,000 points were generated. On this problem, we compare ourselves to a KNN method with a maximum window of 300 points on each side of the gap. It is set to use up to 15 points as long as their L_∞-norm is below 5% error. If no points matching this condition are found, it then works as a LUT. We are also comparing ourselves to an MLP model that we acquired doing a grid-search for the optimal set of dense layers.

Evapotranspiration

To evaluate our model on a real-world scenario, we chose to apply it to ET measurements. These measures of ET are acquired using an EC device. An EC tower measures latent heat flux (*i.e.* water vapor flux) from a crop. This can be used to infer the crop water consumption and hence its evapo-transpiration. Additionally, the tower also records the relative humidity in the air, along with the sun radiation, the wind speed and the air temperature. These 4 variables, which we will call meteorological data, are the only variables our neural network has access to, to reconstruct the missing point in the ET.

Please note that sometimes the tower had sensors failures as well. In order to avoid having gaps in our input data, we used values from a nearby meteorological station to fill in the missing meteorological values. The files used to train and evaluate our model come from six different measurement campaigns and feature two growing seasons in three different types of crops: processing tomatoes, cotton, and wheat. These recordings were acquired at different seasons: winter and summer, and in different regions in Israel: north and south. The direct consequence is that the amplitudes of the variables change significantly between the different recordings. Figure 3.12 shows how the variable we want to fill gaps in fluctuates. Each recording has 5 variables, the latent heat flux (i.e. our target), the net radiation, the relative humidity, the air temperature, and the wind speed. Figure 3.13 shows typical values for different crops. Those files are recorded with a half-hourly rate over a period of 3 to 4 months. In total, this makes for about 3000 to 4000 continuous gap-free-points per recording.

To evaluate our networks on those crops, we could not train and test on each recording individually. This would result in too little data to train or evaluate our model properly. Also, it would mean that our model would be tuned for this specific recording, and would not be able to generalize to other crops, making our approach impractical. We verified this hypothesis using tomato crops and then chose to do 6 different train/test sets. To do so, we put all our recordings but one in the training and the remaining one in the test. We ran the 6

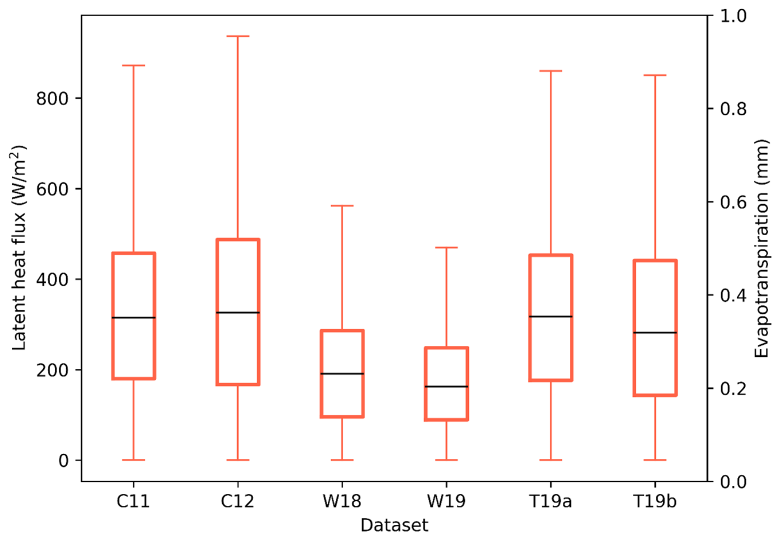


Figure 3.12: The different sites where the data has been acquired. C stands for cotton, W stands for wheat, T stands for tomato. The number gives the year of the recording.

possible combinations and obtained 6 different datasets.

This dataset exhibits interesting behaviors when compared to similar problems, for instance in forestry. Here, the hot middle-eastern climate coupled to the spring season creates rapid changes in the plant canopy, increasing its leaf area index which in-turns increases the water it consumes. Ideally, we would include a vegetation variable, like the leaf-area index to our model, but this variable is very tedious to acquire, and in most cases is not measured. Using vegetation indices (e.g. NDVI - Normalized Difference vegetation index) derived from remote sensing data[97] could also be considered, but they often exhibits gaps and are not applicable to small fields. This is why in the end, we did not consider any vegetation variables in our experiments. On this dataset we only compare ourselves to REddyProc. As the MLP performed worsed than the REddyProc its results are not presented here.

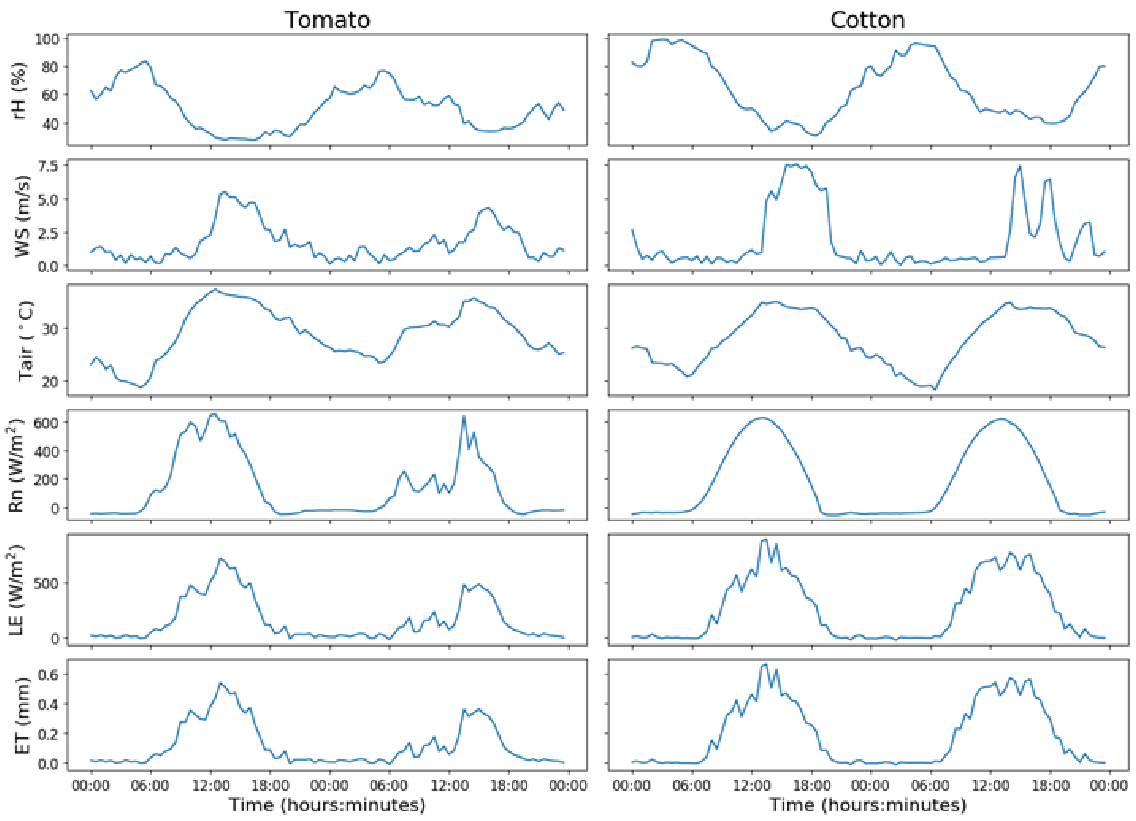


Figure 3.13: The different variables accessible to the network to make its predictions.

3.2.7 Evaluation Metrics

We chose to evaluate our approach on different sequences and gap sizes. To do so, we generated sequences with 3 different lengths: 192, 384, and 576 points, equivalent respectively to 4, 8, and 12 days on the real data. For each of these lengths, one model is learned. We will refer to sequences of 192 points as small (S), 384 points as medium (M), and 576 as large (L).

When considering our real data, the limited quantity of data-points was a problem. Even for small sequences, we only had 82 unique non-overlapping sequences. To increase that number, we generated sequences using a moving window with a stride of one. This allowed us to generate about 2500 sequences out of one recording. This high redundancy in our data explains why we aimed to minimize the network's parameters: to prevent overfitting. Additionally, we chose to generate gaps of random size, and at random positions when sampling batches during both training and testing. This further increases the quantity of available data and further mitigates the risk of overfitting. In the small sequences, the gaps ranged from 24 to 72 points; in the medium ones, the gaps ranged from 72 to 144 points, and in the large ones, the gaps ranged from 144 to 288. This is slightly higher than the usual 30% missing data in average in EC measurements.

To evaluate our approach on the real data, we compare ourselves to the most prevalent tool in the field: REddyProc [98]. This tool, developed by the Max-Planck Institute, is strictly dedicated to fill gaps within flux measurements by EC systems. Embedded as an R package, it relies on the MDS, and in some extreme cases on the MDV to recover the missing points. The main restriction of this tool comes from the techniques it uses. As it is based on MDS, a window of at least 7 days on each side of the gap is being used. Since the size of this window cannot be changed, the two approaches were compared using different sequence lengths to fill the gaps: Our network sees a much smaller horizon of points due to memory limitation of our GPUs. Also, using 14-day windows (7 on each side) and 6-day gaps would be equivalent to use sequences of 960 points, which we could not do with our data-recordings as it would result in too few sequences.

Finally, to evaluate the results of the different methods, we sample 100 gaps from the test set. On these gaps, we compute the RMSE and the Mean Bias Error (MBE) between the methods results and the ground-truth EC measurements. The MBE is the mean of all the errors on a given gap. For each gap the percentage of improvement is computed and the mean and standard deviation of the improvement is also reported. The metrics are computed per gap and then averaged. Additionally, we compute the standard deviation for each of the metrics. The objective of the MBE metric is to make sure that the model has a zero centered error. In irrigation, it is used to indicate the bias induced on the daily or seasonal sums of the water loss. Finally, to ensure the repeatability of our results, all our models are trained 3 times with a different optimizer seed, different test samples, and different training batch orders. The presented results are an average of the 3 runs.

Table 3.1: RMSE and MBE of our model and KNN on the toy problem. Our problem easily bests the KNN approach.

Seq Size	Methods	RMSE		MBE	
		mean	std	mean	std
S	Ours	**0.09**	**0.021**	**0.008**	**0.021**
	MLP	0.16	0.041	0.0016	0.044
	KNN	0.48	0.21	-0.016	0.21
M	Ours	**0.12**	**0.04**	**-0.004**	**0.026**
	MLP	0.25	0.08	-0.007	0.096
	KNN	0.54	0.21	0.011	0.23
L	Ours	**0.25**	**0.09**	-0.007	**0.033**
	MLP	0.41	0.09	-0.016	0.15
	KNN	0.69	0.19	**0.04**	0.34

3.2.8 Results

Toy Case

On the toy problem, our approach outperforms both MLP and the KNN algorithm despite the larger view horizon of the KNN. As can be seen in table Table 3.1, which summarizes the results for the different gap-size, our approach is consistently better than KNN and MLP across all metrics except for the MBE on large sequence sizes. This can be explain by the nature of the MBE metric: it is the mean of the error, thus one value can slightly change the overall result even more here since the error values are very small. Hence it is more important to focus on the variance of the MBE as it depicts how is fluctuates over various gaps. Fig Figure 3.14 compares the KNN method (in blue) to our architecture (in orange). We can see that our approach better fits the data. Our attention-based model is able to extract the local trend, whereas KNN is being tricked by the long term amplitude changes created by the latent variable z_1. [ht]

Evapotranspiration Data

Table Table 3.2 summarizes our models' performance on the EC datasets. Based on the RMSE results, our model performs statistically better or as well as the reference method:

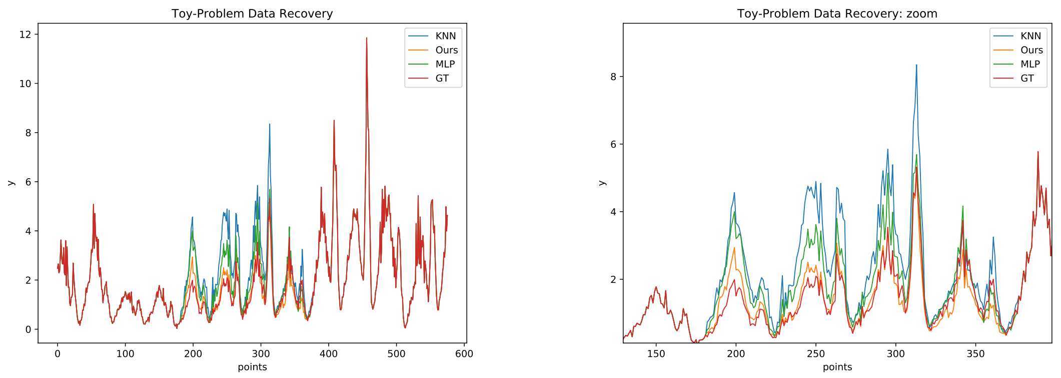


Figure 3.14: Gap filling quality comparison of our model (NN), with KNN on the toy problem on large gaps.

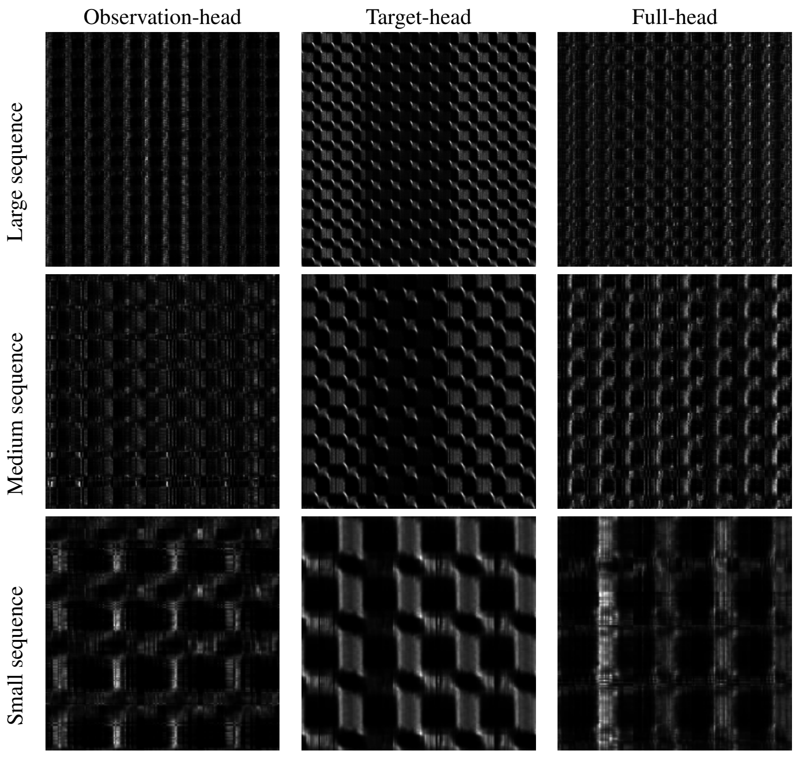


Figure 3.15: Attention weights for the different sequence sizes on real data. The periodicity of the data was understood and leveraged by the attention mechanism.

REddyProc. Only on large gaps on C12 does it perform slightly worse in average, but the difference is not statistically significant. On another hand, our model performs much better than the baseline on the Tomato crops. This is particularly visible on the Tomato 2 experiment, which yields an average improvement of 30% across all gaps. This is interesting

Table 3.2: RMSE and MBE of of our model and REddyProc applied on real EC data (lower RMSE and MBE values indicate better model performance). Values range from -50 to 800.

Crops	Seq Size	Methods	RMSE		RMSE Improvements	MBE	
			mean	std		mean	std
C11	S	Ours	**44.2**	**13.2**	+12% ± 17%	**-0.9**	**8.4**
		REddyProc	51.8	17.4		2.1	13.8
	M	Ours	**53.9**	**13.4**	+7% ± 14%	**-3.9**	13.9
		REddyProc	57.0	13.5		9.1	**12.2**
	L	Ours	**46.7**	**7.62**	+10% ± 10%	**-0.5**	7.4
		REddyProc	52.9	12.7		0.9	**5.7**
C12	S	Ours	**43.2**	**13.9**	+10% ± 15%	-0.7	**12.0**
		REddyProc	48.1	13.9		**0.1**	14.0
	M	Ours	**48.0**	**11.9**	+4% ± 22%	**0.1**	12.1
		REddyProc	51.0	12.0		-5.5	**9.7**
	L	Ours	49.5	14.6	-10% ± 49%	2.2	12.3
		REddyProc	**47**	**10.9**		**-1.1**	**8.6**
T19a	S	Ours	**30.9**	**14.0**	+29% ± 19%	-10.5	**11.7**
		REddyProc	54.6	24.0		**-4.1**	19.3
	M	Ours	**31.0**	**4.2**	+35% ± 12%	-2.6	**4.8**
		REddyProc	49.9	14.9		**0.6**	10.7
	L	Ours	**44.6**	**9.1**	+35% ± 12%	-4.4	**5.4**
		REddyProc	71.9	23.8		**-3.3**	12.8
T19b	S	Ours	**32.4**	**9.1**	+19% ± 25%	**-1.2**	**9.1**
		REddyProc	42.1	13.2		-3.4	16.6
	M	Ours	**36.4**	**6.88**	+14% ± 20%	**-1.6**	**8.5**
		REddyProc	44.2	12.34		-5.9	13.0
	L	Ours	**37.8**	**4.1**	+13% ± 15%	**1.0**	**5.9**
		REddyProc	44.4	9.2		-1.6	9.8
W18	S	Ours	**37.9**	**12.2**	+7% ± 30%	-8.9	**9.4**
		REddyProc	46.44	13.58		**-7.6**	22.5
	M	Ours	38.0	**8.1**	+0% ± 25%	**-4.3**	**4.8**
		REddyProc	**37.5**	11.4		6.5	5.8
	L	Ours	**37.9**	**4.6**	+0% ± 13%	-8.9	**4.2**
		REddyProc	38.7	6.6		**-1.7**	8.5
W19	S	Ours	**26.0**	**7.2**	+0% ± 30%	4.5	**6.3**
		REddyProc	28.42	13.6		**-1.8**	10.3
	M	Ours	**27.9**	**4.8**	+2% ± 21%	7.5	**7.4**
		REddyProc	29.9	9.1		**-0.8**	9.2
	L	Ours	**29.9**	**3.3**	+7% ± 17%	7.2	**4.9**
		REddyProc	31.9	7.0		**-2.8**	6.9

as tomatoes are summer crops with a quick canopy growth. These particularities lead to faster ET dynamics than the one encountered in the other crops present in this dataset. The superior performance of our network on this crop shows that our approach performs well on a system with high dynamics, which was our original goal. Additionally, the constant performance of our network demonstrates that, despite its smaller time horizon, our architecture is more reliable than REddyProc. Regarding the MBE our model performs as well as REddyProc. Overall, the MBE quantifies the irrigation bias but not the accuracy of the prediction, hence our prediction keeps similar performances.

Figure Figure 3.15 shows examples of attention matrices for the different attention heads. Each of those matrices translates the cross-correlation between the elements of the same sequence. Let us define A, an attention matrix for a sequence of size k s.t. $A \in R^{k \times k}$ and $i, j \in [1, k]$, the position of two elements inside that same sequence; then the value $A(i, j)$ is a measure of how strong the correlation between the elements i and j is. The brighter the pixels in the image, the stronger the correlation. The attention matrices of our networks present periodic patterns. The periodic patterns show that our architectures are leveraging the periodicity of the data to fill the missing values in our sequences. One can also see that on the target-heads, the center of the matrix is less bright than on the other heads. This is due to the fact that the target heads avoid using the values inside the gaps.

If one looks at the last row of figure Figure 3.15, one can see some patterns and at some point a variation in that pattern, this can easily be seen on the target-head's weights where a black band appears. This is where the gap is located inside the data. What this black band means, is that the network learns not to use data where there are gaps in order to fill the missing values. Similar things can be seen on the medium and large sequences (on the target head), but the gap is not fully black. Instead, during the nights, the network is still trying to make use of the data. This is probably due to the network having difficulties detecting the gap or to a lack of training data. On the other heads, a similar pattern can be seen, but the gap is not clearly visible. However, what is visible are the nights: during the nights the values are homogeneous and the variables all have the same weights. This is represented by this darker bands that can be seen in the different heads weights.

Finally, using this training-testing split, we show that our network achieves solid performances without even training on the dataset which we aim to fill. This demonstrates the strong generalization capacities of our method and makes it almost as convenient as the MDS since its application would be training-free.

3.2.9 Discussion & Conclusion

In this section, we examined a novel deep learning gap-filling method for crop ET measured by eddy covariance. The method was tested on a database containing different crops and demonstrated superior performance than the widely-used gap-filling method, MDS. Overall, the DL method showed a significant decrease in RMSE throughout all datasets and gap lengths Table 3.2. The DL method examined here was able to "learn" from a relatively small database containing only six ET measurement campaigns over agricultural fields and success-fully fill gaps of up to six days. Across all different datasets and gap lengths, the DL method showed a more accurate (lower RMSE) and precise (lower standard deviation) prediction of ET flux than did MDS.

Moreover, the present DL method has some crucial advantages over past MLs methods. First, using a model developed for a certain crop and season, the DL method does not need to be trained on the specific dataset being gap-filled. Secondly, the DL includes natural embedding of time – i.e., the method learns the data as a time sequence rather than as individual points. The benefits are (1) faster runtime, in the order of seconds versus minutes, (2) easier implementation after the model is trained, and (3) minimal data input of about 12 days instead of a full-year as required in [78], making the method more suitable for short growing seasons of annual crops. The minimal data input is possible because our DL model relies mainly on the values of the neighboring days and in addition on modeling of the relations between the meteorological variables and the flux. The ML methods [78] require a whole year because they are based on modeling the relationship between the meteorological variables and the flux only, without an understanding of the time or the sequence of the values. Therefore, our approach is advantageous compared to previously developed techniques, especially for cases with limited training data.

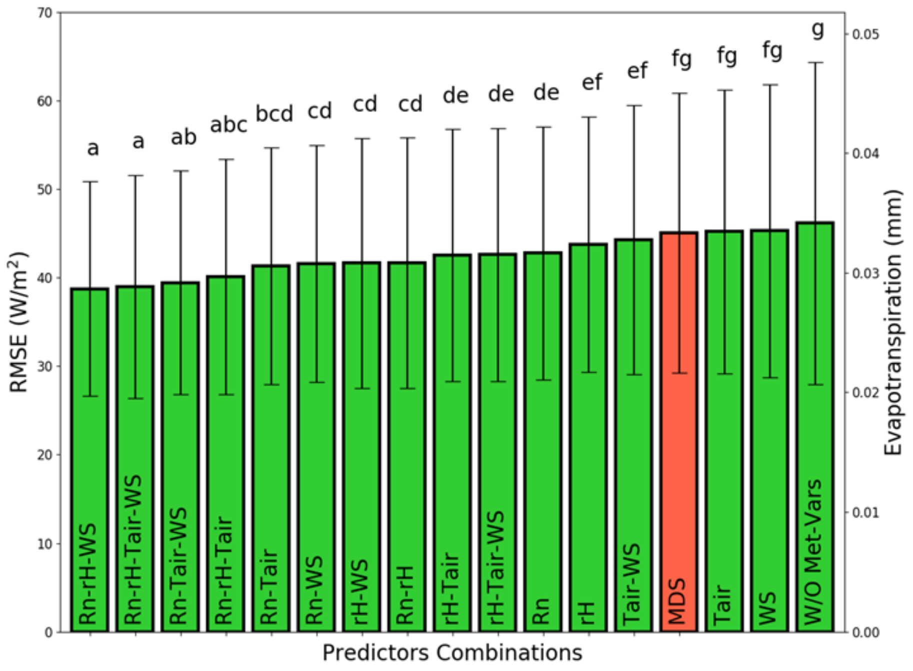


Figure 3.16: Deep learning (DL) model performances using combinations of different meteorological variables (Rn – net radiation; rH – relative humidity; WS – wind speed; Tair – air temperature; 'W/o met. vars.' – without meteorological variables, i.e., time of day only) as model inputs compared to the common gap-filling method, marginal distribution sampling (MDS). Bars represent the standard deviation of the mean.

The sensitivity analysis on the predictors, given in Figure 3.16, shows an increase in RMSE when net radiation was excluded. Excluding any other variable do not result in a significant increase in RMSE. Furthermore, using net radiation as a single predictor shows a significantly lower RMSE than any other single variable, even when comparing to the MDS method. The high correlation between latent heat flux and net radiation in relation to the other meteorological variables, can be seen in the diurnal courses (Figure 3.13) where LE and Rn curves are well aligned. These results indicate that net radiation is the most important predictor of ET, relating to the fact that radiation is the primary source of energy for the ET process, and therefore is the limiting factor in the system. This is in agreement with [99], who showed that solar radiation was the most essential factor when tested on different reference ET calculation methods. Additionally, they showed that air temperature was the second most important factor, although this is not evident in this section, perhaps due to auto-correlation of the data (the similarity between points at the same time of day but on neighboring days), which reduces the importance of the predictors, in general. Some limitations of the suggested gap-filling method should be noted. Although this section suggests that the method is insensitive to adding data from different crops, the model must be trained on data similar enough to that of the gap-filled dataset (similar crops, climate, ecosystems, etc.), because of its dependency on previously trained datasets. Additionally, the method was tested on a relatively small database with specific characteristics, and more testing in different conditions, ecosystems, and ET fluxes is needed. The method proposed here is based on recent advances in deep learning methodologies [37]. Hence, as is, it cannot be used at this stage as a convenient off-the-shelf tool for routine gap-filling like the MDS tool developed by [100].

3.3 Conclusion on Irrigation Modeling

In conclusion, in this chapter, we presented two methods that achieved state-of-the-art results on two different tasks. Despite the limited amount of data available, they clearly showed the superiority of NNs over traditional methods. However, these studies have only been conducted on a small scale. To validate these results, the methods should be applied to more crops, in more countries. Future work should focus on deploying these approaches onto more diverse data around the world. We believe that NNs are outstanding function approximators that are currently under-utilized in agronomy.

Another key element of the NNs that is not addressed here is their reliability. In real-world applications, it is critical to know how much one can trust the estimations of its model. While most white-box models do not provide this, the fact that they are easy to understand allows practitioners to know when they can be applied. With NNs, due to their black-box nature, knowing the region of applicability of the model is harder. Hence, we believe that future work should also focus on the estimation of the uncertainty. While this problem has been studied on classification problems, in regression few works have reliable uncertainty estimation. This area of DL is one of the rare that is not crowded and has ample room for high-impact contributions.

CHAPTER 4
DATA-EFFICIENT MODELING & CONTROL FOR ROBOTIC SYSTEMS

4.1 Introduction

In this chapter, we study how to efficiently model robotic systems. In particular, we apply importance sampling to learn the dynamic model of robots. Indeed, when learning robots' dynamic models, the main limiting factor is data collection. The amount of data required to learn NNs-based models is often fairly high, requiring to run the robot for a long time. Moreover, on some naturally unstable robots, collecting many sample next to the instabilities is impossible. This will result in the data containing only a few samples from these hard-cases. Conversely, the large majority of the data will come from stable states. This creates an unbalance. In turn, it can lead to learning a model that works well in most cases but fails as the robot ventures near these unstable states. Thus, it is critical to find ways to leverage every bit of data at our disposal.

In classification, this task is relatively easy as we can sample data with respect to classes, but in regression setups it is not that trivial. This is why we propose to use importance sampling, a group of method that uses the performance of the network to compute a dataset's sampling distribution. This enables to learn model even when the data is strongly unbalanced. The drawback is that these techniques may put a lot of emphasis on outliers. Thus, we first evaluate their modeling capacities on different scenarios, with balanced, unbalanced, and real noisy-data. Then, we apply them on a MPC problem. We evaluate if the gains in modeling accuracy also translates into improved performance. Finally, we apply this knowledge on a real system where we use the MPPI to control an USV. The work presented in this chapter was done in collaboration with Antoine Mahé. He helped to acquire the data on the real systems, brought is robotics experience and knowledge regarding the PER.

4.2 Prioritization Applied to Dynamics Modeling

4.2.1 Motivation

Model identification is often the first step in designing the command of a dynamical system. Nowadays, the most commonly used method is Auto Regressive Moving Average (ARMA). Its simple implementation and light computational weight made of ARMA the *de facto* standard in the field of system identification for the last fifty years.

Yet, recently, building on the boom of deep learning, and dedicated hardware capable of inferring simple models more than a thousand times per second, NNs have made a remarked entrance in this field. They have produced impressive results with their high versatility, and their capacity to learn continuously over time [25]. Unfortunately, those methods require a massive amount of data to converge properly.

Regrettably, collecting data from a robotic system naturally leads to the construction of a dataset with command distribution issues [55]. The distribution of the commands on real systems can be strongly unbalanced, as we experienced in our data acquisition where most of the samples are generated with commands near the origins. Most commands sent during data acquisition comes from states where the operator (or the controller) is comfortable. This leads to some areas of the model's action space being visited only a few times while others are continually experienced. This results in limited generalization capabilities of the network.

To cope with those problems there has been some recent research proposing to improve the quality of deep neural networks model training by sampling a subset of the data based on how they perform on them [57, 58, 101]. The common idea is to evaluate how useful samples are for the training, then increase or decrease their weight based on their importance. In order to improve the ability to train NN onunbalanced datasets we use two different sampling mechanisms. Those methods evaluate which samples are most suited to improve the model performances at a given time during training.

In this section, we take advantage of these ideas to propose a model-identification method based on NN that are able to learn a complex dynamic despite noisy and unbalanced datasets. We demonstrate the method on different conditions and systems and compare it to other identification methods such as ARMA or more classical NNs training.

4.2.2 Related Work

Since the 1970's ARMA [102] has been the way to go for most black-box system identification. Such models are particularly simple to understand and implement. They rely on linear difference equations (their transfer functions are rational fractions), based on the past states and inputs of the system. The linearity of these equations enables the computation of the system parameters with a least-squares method. As a result, the fitting of an ARMA model is not computationally expensive; this makes it possible to perform efficient grid-search over the orders of the filter (orders of the numerator and denominator of the transfer function).

The last decade has seen Deep Learning achieving outstanding results in many tasks, over a very large field of applications, ranging from computer vision to natural language processing. Control and system identification are no exceptions. section 2.2 provide an overview of the litterature on NNs applied to system identification.

To train these networks a lot of data is necessary which implies an important amount of demonstration of the system in its environment. The sample inefficiency of NNs leads to the need to generate large demonstration datasets. These are often very unbalanced [55] and the training is saturated with common samples while interesting data points do not have any impact on the learning process. All in all, properly learning hard cases is hindered by the imbalance of the data. section 2.3 presents the current state-of-the-art on efficient NN learning.

In this section, we apply various importance sampling methods to system identification. Those approaches are evaluated on standard datasets [103] and custom datasets that exhibit various degrees of non-linearity, unbalancing and noise. Building on the promising simulated result of [56] we expand the prioritize experience replay method to real world data and show its applicability of on standard datasets. Moreover, we propose to use an other prioritization scheme that alleviate the hyperparameter complexity of the previous algorithm.

4.2.3 Model Identification

Linear system identification of the ARMA family have been used for decades with success. When $u(t)$ and $x(t)$ respectively denote the system's input and output at time t, ARMA's model of the system is given by the following discrete-time linear difference equation:

$$x(t) + \sum_{k=1}^{p} a_k x(t-k) = \sum_{k=1}^{q} b_k u(t-k) \tag{4.1}$$

It is more intuitive to consider this equation as a way to determine the next output value given previous observations and a set $\theta = \{a_1, ..., a_p, b_1, ...b_q\}$ of parameters:

$$x(t) = -\sum_{k=1}^{p} a_k x(t-k) + \sum_{k=1}^{q} b_k u(t-k) \tag{4.2}$$

The linearity of the model makes it easy to compute the optimal parameters θ^* using the linear least-square method.

More recently, the ARMA methods have been challenged by NNs, as they expanded the range of system that can easily be modelled from data. In particular, non-linear systems are no longer an issue using a NN [104].

In this section we chose to use a standard MLP, with 2 hidden layers. Not only has this kind of networks been extensively used and studied in the last decades [25], but their simplicity also highlights the performances of our algorithms and their impact on the final results. To properly learn the dynamic of the system, the historic of the twelve previous commands and states is used. This time horizon gives the MLP a memory of the previous events which proved to be sufficient for the considered systems. Thus, it made sense not to use LSTM as the required time horizon can easily be known. Additionally, LSTMs add unnecessary complexity to this study where our focus is on the data and the optimization, not the architecture of the networks.

To learn the model, a multi-parameter regression is computed. The cost-function used is the MSE, but its implementation within TensorFlow [105] has been slightly modified to allow easier computation of the network's gradient.

4.2.4 Importance Sampling

Learning a dynamic which is poorly represented in the dataset is hard. This is particularly true on datasets which have not been acquired for the specific purpose of model identification. Indeed, large model identification datasets tend to be filled with redundant information. To cope with this issue, we apply importance sampling mechanisms to the training process. Some dynamic systems are more difficult to identify than others. For instance, the velocity variations of the drone used in our experienced are way easier to identify in a straight line than during a 180-degree turn. Moreover, as operators, we tend to demonstrate systems in conditions where we already have a good understanding of the system behavior. Thus, the interesting data where the network should learn the most is the rarest. This leads traditional learning approaches to fail to train on those seldom seen events.

Prioritize Experience Replay

To answer this problem, we propose to adapt the prioritization scheme introduced in [57] for reinforcement learning to the context of system identification. Indeed, prioritization forces the training on harder samples even if they are scarce. The adaptation of this sampling strategy to system identification yielded encouraging results, as illustrated in [56]. In practice, we use the loss of the network prediction to estimate the training value of a sample. The samples that lead to the highest errors are the one where the network has the most learning to do. Hence, the network prediction errors are collected to compute a probability distribution over the samples, which is then used for sampling the dataset for the next training session. This process is detailed in the algorithm algorithm 1.

Algorithm 2: Data prioritization from [57] adapted to system identification.

```
$data$
$K$ : number of trials
$MLP$ : neural network model
$trainingData \leftarrow data$
$sampleWeight \leftarrow \emptyset$
for $k = 0$ to $K$ do
    $MLP \leftarrow Train(trainingData)$
    N number of samples in $data$
    for $i = 0$ to $N$ do
        $\delta_i \leftarrow \|Y_i - MLP(X_i, U_i)\|$
        $P(i) \leftarrow \frac{\delta_i^\alpha}{\sum_k \delta_k^\alpha}$
        $w_i \leftarrow \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta$
        $sampleWeighted \leftarrow \{w_i\}_{0 \leq i \leq N}$
        $trainingData \leftarrow$ sample data $d_i \sim P(i)$
```

One of the limitations of this approach is that it focuses on a small subset of samples. Although that focus improves its data efficiency, it also increases the risk of over-fitting. Noisy datasets are also hard to learn from, as it is harder to make the distinction between complex cases and outliers. To mitigate those problems and make the method practical, hyperparameters are introduced. Those parameters allow choosing how much the training should focus on hard cases. The probability of choosing the sample *i* during the sampling is given in Equation 4.3 where δ_i is the score of sample *i*, in our case the error between the NN prediction and the actual observation.

$$P(i) = \frac{\delta_i^\alpha}{\sum_k \delta_k^\alpha} \tag{4.3}$$

However, the sensibility to the hyperparameters make the approach difficult to apply and makes a systematic, tedious grid search mandatory to find optimal values for these parameters.

Gradient Upper-Bound

Another way to prioritize samples is to use the gradient upper-bound, as explained in [59]. As its name implies, the gradient upper-bound method relies on an approximation of the norm of the network's gradient. [101, 106] showed that the gradient norm represents what a network can learn from a data-point. In comparison, the loss of the network on which the prioritized experience replays relies is a poor approximation of it. As a result, drawing from a loss-based distribution is less efficient than using a distribution homogeneous to the norm of the gradient.

Yet, computing the gradient norm is prohibitively expensive. In order to alleviate this issue [59], introduce an accurate and computationally inexpensive estimation of it, that is the gradient upper-bound. This so-called gradient upper-bound is obtained by computing the norm of the gradient between the loss and the last activation layer of the network. Furthermore, this approach, which constantly updates the probabilities of drawing the samples, has fewer hyperparameters than the prioritized experience replay. Indeed, instead of updating the weights at some arbitrary training step, or epoch, this technique samples *super-batches*, i.e. n-time larger batches than the standard training batches. From these *super-batches*, a distribution based on the gradient upper-bound is computed, and a standard batch is sampled from it. This as two implications: because the super-batches are sampled uniformly, it reduces both the risk of over-fitting and the risk of focusing on outliers. However, depending on the size of the *super-batches*, the training time can be significantly extended. Algorithm algorithm 3 shows our implementation of the gradient prioritization scheme.

Algorithm 4: Data prioritization from [59].

```
$data$
$N$ : number of steps
$ssbs$ : super-batch size
$bs$ : batch size
$trainingData \leftarrow data$
for $i = 0$ to $N$ do
    $super\_batch \xleftarrow{\text{ssbs}} \mathcal{U}(trainingData)$
    $g = get\_gradient\_upper\_bound(super\_batch)$
    $\mathcal{G} \leftarrow distribution\ from\ g$
    $weights \leftarrow \frac{1}{Bg}$
    $batch \xleftarrow{\text{bs}} \mathcal{G}(super\_batch)$
    $train\_step(batch, weights)$
```

4.2.5 Experiments

Evaluation

To evaluate the model, we consider two metrics. The first one which will be referred as single step accuracy expresses the next-point prediction accuracy. Its value is computed using the MSE of all the prediction made on the test set. This metric is used to evaluate the instantaneous prediction accuracy. The second one is a trajectory prediction accuracy, later it will be referred as multistep accuracy. In this case a time horizon is selected (15 samples for the drone, 30 for DaISy), and the network iterates over its own predictions. The model is run over 60 different trajectories from our test set. The final metric is the MSE of those trajectories. The latter evaluation score is the one we use to select our hyperparameters during the grid search.

Datasets

After validating the concept on a simulated drone, we test our approach on standard system identification dataset collected from real systems. Finally, we move onto a more challenging dataset, consisting of a real drone fitted with an Real Time Kinematic (RTK) GPS.

Parrot Bebop Drone In order to test the validity of the new optimization method and the importance sampling, we first evaluate our approach using a simulated drone. The simulation is done in Gazebo [107], using ROS [108] and a drone emulated by the rospackage tum simulator[1]. The system includes the simulated drone and its low-level controller. The dataset created for this experiment is a combination of two trajectory generation algorithms. The first one only moves the drone in the plan in straight line, there is no vertical and no angular command sent. The second algorithm move the drone such as it uniformly explores the action space while not crashing. The aggregation of samples from both data generation schemes produce an unbalanced dataset that we use to train our model. On the real drone, the data is acquired by the uniform exploration algorithm detailed earlier on. As such this dataset is balanced. To acquire the position of the drone an RTK GPS has been set up on it. The acquisition was made using the RTK mode, and not Post Processing Kinematic (PPK)[2] as this would not be representative of a real time system. Also, we rarely reached a fix solution state but were most of the time in a good float solution state. Hence, the precision is roughly 20cm, and we can see some discontinuities in the data. Moreover, the drone crashed into some trees. This implies that the data contains strong outliers.

DaISy DaISy [103] is a large open source system identification database of real and simulated systems. Because our scope is robotics, we chose, to study two robotic systems, namely the flexible robot-arm and the CD-player arm.

- Flexible robot arm is a single-input single-output (SISO) system, the input command is the reaction torque of the structure, while the measured state is the acceleration of the arm. This dataset is balanced.
- For the CD-player arm, a MIMO system, the inputs commands are the actuators forces, while the output is the tracking accuracy of the arm over the x, y coordinates. This dataset is balanced.

[1] http://http://wiki.ros.2org/tum_simulator

[2] The PPK mode is computed offline after that the system has run.

The size of this datasets is small: about a thousand samples. They are here to illustrate that our methods work at least as well as previous methods even when the data is well-balanced and contains few samples.

ARMA

In order to enable comparisons between our algorithm and a more standard system identification method, we modeled our system with an ARMA filter. The AR and MA orders are found through a grid search. The goal is to find the lowest orders providing the most accurate results.

Neural Networks

As described earlier we chose to use MLPs to learn the model of the different systems. To emphasize the versatility of our method, we used the same architecture to fit all of the models depicted in this section. This architecture is similar to the one used in Auto-Rally [25]: it has two hidden layers with a width of 32. The input is made of a state vector X, and a command vector U, for all of the datasets the commands and states history are the same length. It has been set to twelve samples. In the case of the drone, this represents 2.4 seconds, enough to catch the dynamics of the model. All of the models are trained using TensorFlow [105] with the ADAM [22] optimizer and a learning rate of 0.001. All the networks trained for a total of 25,000 iterations, with a batch size of 16. Finally, the input data are normalized individually, that means that each command input and state input have been normalized independently. Thus, the networks predict normalized states.

Prioritization

In this section, we evaluate two prioritization schemes: first we try the sampling based on the prioritized experience replay which relies on the loss of the samples to assess the importance of a sample, and then we use the gradient upper-bound to estimate the importance of a sample.

Prioritized experience replay In all of our experiments, we retrain our networks five times while sampling on a loss-based distribution obtained with the previous iteration of the network. In order to choose the hyperparameters, we perform a grid search where we look for the combination of α, β that yields the best accuracy. α is comprised between 0 and 1, this parameter allows increasing the prioritization of hard cases. At the same time, we search for the β parameter also included within 0 and 1. Its goal is to compensate for the bias introduce by the prioritization. High values of alpha rapidly lead to over-fitting and focusing on outliers. Details about the roles of α and β can be found in [57].

Gradient prioritization When training using the gradient upper-bound, the gradients are computed between the loss and the last activation layer. To do so, the original implementation of TensorFlow's MSE is modified to allow access to the per-example MSE instead of the batch MSE. The only parameter to tune is the super-batch size: to select it we perform a grid search where the super batch size varies between 60 and, 10000 samples per *super-batches*.

Grid Searches

To appropriately select the hyperparameters on all of our approaches, we use a k-fold-cross-validation methodology. The dataset is split in five equal parts, four for training and validation, one for testing. The data allocated to the training is split into five more sets, four for training and one to validate the hyperparameters. Furthermore, the datasets are normalized: the mean and the variance of each command and state are extracted from the training set and used to normalize both on the input data and output data. Finally, each experiment is run ten times to average the results. The best hyperparameters for a given test set are selected using the trajectory-based accuracy.

4.2.6 Results

Drone Simulations

The simulated dataset has the particularity to have little noise, but is unbalanced. As such, we expect our prioritization scheme to perform better than the regular training. The results of the experiments on the simulated drone can be seen in table Table 4.1. It shows that the importance sampling scheme bring a 5 to 10% on single step accuracy. The gradient prioritization scheme noted GRAD in table Table 4.1 consistently outperforms the baseline, *i.e.* the training without any form of prioritization noted as STD. We can see that the Prioritized Experience Replay results are less constant, this is most probably due to the complexity of properly choosing the hyperparameters. In multistep accuracy we can draw similar conclusion as the Gradient outperforms the non-prioritized approach. Here again, the edge provided by our method allows for a 5 to 10% increase in overall performance. Please note that those results do not rely on hand-picked hard cases, but rather on trajectories taken at random in the test set. As such, this shows that our methods improve the performance of the networks in general. Additionally, we can see that the traditional method: ARMA is nowhere close to the performances of the NNs.

Table 4.1: Simulated Drone k-fold cross validation results. Lower is better.

Test set	0	1	2	3	4
	Single Step Accuracy				
ARMA	0.836	0.873	0.850	0.855	0.883
STD	0.311	0.346	0.109	0.329	0.481
PER	**0.298**	**0.328**	0.122	0.316	0.492
GRAD	**0.298**	0.331	**0.100**	**0.315**	**0.473**
	Multi-Step Accuracy				
ARMA	1.003	0.986	1.056	0.983	0.958
STD	0.214	**0.263**	0.166	0.257	0.709
PER	0.227	0.271	0.179	0.250	**0.685**
GRAD	**0.202**	0.271	**0.155**	**0.239**	0.705

DaISy

Those datasets are short, and their command distributions are balanced. This means that our methods should have little impact on the training results, but more importantly, we are interested in assessing that in the case of a small and/or balanced dataset they do not end up degrading the training performances.

SISO: Robotic Flexible Arm As shown in table Table 4.2 the training using Prioritized Experience Replay (PER), gives better results on both single-step accuracy and multistep accuracy. This result is interesting as it shows that even on small balanced datasets our methods can increase the performance. Only on 1 out of the 5 test sets, the training without any prioritization (STD) gave better results. On the other hand, the gradient based sampling scheme (GRAD) consistently made marginally worse predictions than the two others.

Table 4.2: Flexible robotic arm k-fold cross validation results. Lower is better.

Test set	0	1	2	3	4
	Single Step Accuracy				
ARMA	0.0454	0.0364	0.0671	0.0432	0.0485
STD	0.00031	0.00162	**0.00503**	0.00610	0.00144
PER	**0.00012**	**0.00135**	0.00518	**0.00565**	**0.00038**
GRAD	0.00052	0.00188	0.00623	0.00660	0.00123
	Multi Step Accuracy				
ARMA	2.220	3.379	1.359	1.921	3.173
STD	0.00065	0.00422	**0.0104**	0.0141	0.00239
PER	**0.00033**	**0.00347**	0.0115	**0.0117**	**0.00074**
GRAD	0.00097	0.00489	0.0138	0.0148	0.00214

MIMO: CD Player Arm As table Table 4.3 shows, the results here are blurrier, the PER still outperforms its counterparts in single-step accuracy. Yet, in multi-step accuracy, we can only conclude that the methods are equivalent as we cannot pick a method that consistently performs better than the other. However, this clearly shows that despite the small dataset and the fact that the data are balanced our method do not decrease the training performances.

Table 4.3: CD Player Arm k-fold cross validation results. Lower is better.

Test set	0	1	2	3	4
	Single Step Accuracy				
STD	0.0197	0.0185	0.0149	**0.0173**	0.0173
PER	**0.0191**	**0.0181**	**0.0148**	0.0180	**0.0171**
GRAD	0.0199	0.0184	0.0154	0.0175	0.0183
	Multi Step Accuracy				
STD	**0.0933**	0.0845	**0.0685**	0.0773	0.0896
PER	0.0948	**0.0825**	0.0717	0.0799	0.0850
GRAD	0.0961	0.0834	0.0751	**0.0710**	**0.0842**

Real Drone

Table 4.4: Real drone K-fold cross validation results. Lower is better.

Test set	0	1	2	3
	Single Step Accuracy			
ARMA	0.778	0.812	0.783	0.804
STD	0.179	**0.549**	0.144	0.116
PER	0.190	0.563	0.162	0.134
GRAD	**0.165**	0.560	**0.137**	**0.108**
	Multi Step Accuracy			
ARMA	1.054	1.012	1.046	1.014
STD	0.863	0.738	0.543	0.804
PER	**0.861**	**0.720**	**0.519**	**0.783**
GRAD	0.890	0.728	0.534	0.822

On the real system, our model and training methods are put to the test. The acquisition periods have to be short due to limited battery life. The weather and in particular the wind make proper model identification complicated. Additionally, we recall that we never achieved RTK fix only a good float[3] resolution. Yet, despite the aforementioned issues, our approaches do not degrade performances on this balanced dataset. To the contrary, they slightly improve the performance showing that those methods are resilient to outliers and do not over-fit on noisy data.

[3]In fix mode the RTK GPS as an accuracy bellow 5cm, in float the localization is comprised between 1m and 5cm, in our case the localization was around 20cm

4.2.7 Conclusion on Prioritization

In this section we demonstrated that the gradient-upper bound method is a viable alternative to the PER for system identification. We showed that even without complex hyperparameter fine-tuning it achieves comparable result to previous methods on unbalanced dataset of a simulated drone. Furthermore, we expand those methods on standard dataset as well as on a dataset we collected with a real drone. We provide a thorough comparison of different identification methods:ARMA, standard NNs training, prioritized experienced replay NNs training, and gradient upper-bound NNs training. Furthermore, we show that even on small datasets, our approaches do not degrade the performances.

4.3 Prioritization and Model Predictive Controllers

4.3.1 Motivation

Inspection tasks are more and more reliant on autonomous robotic systems. Precision agriculture, building inspection and river monitoring are such tasks that benefit greatly from the improvement of unmanned vehicles [109, 110]. However, most robotic applications require an expert to accomplish the missions of the system. This dependency is often a limitation: in situations where communications are limited, such as underwater exploration or underground mines monitoring, autonomy becomes a requirement.

In order to provide this much needed autonomy, control algorithms are continuously being developed and improved upon. One controller commonly used in this context is the MPC. It relies on a model of the system for optimizing a cost function over a receding horizon. This family of controllers has been successfully used in a variety of applications [111, 112], and is now widely adopted in the industry [113].

By definition, MPCs require a model of the system's dynamics to be able to control it, and plan its movement. This modeling of the robotic system is still an active area of research. The well known ARMA algorithm is the de facto method to perform system identification. Its light computational cost and simplicity made ARMA the way to go for black-box modeling.

Yet, recent development in machine learning have pushed the search for new modeling schemes able to more easily cope with complex non-linear systems. Moreover, the ability of NNs to learn over time [25] offers interesting possibilities, increasing the relevance of deep learning for system identification. Unfortunately, this approach suffers from its computational cost and its data inefficiency.

While recent advances in hardware and well optimized frameworks addressed the high computational cost requirement, the lack of per-sample efficiency of the NNs-based methods remains a major problem when dealing with robotic system [55]. Indeed, collecting data is often a tedious and costly process that tends to produce unbalanced datasets. On robots, it is particularly hard to explore exhaustively the state and action spaces, as the robot may not be stable and exploring these state could result in damaging the robot. Also, if the dynamics of the robot is learned from commands sent by an operator, the robot may only explore a subspace of the state-action space, leading to a biased and incomplete dataset.

This problem can also be commonly found in image classification and RL. Alas, as system identification is a regression task, most of the common methods from the computer vision fields are inapplicable as they rely on the classes to re-balance the datasets. Nonetheless, recent work in the field of RL [57] suggests focusing the NNs training on the samples that are most useful to their convergence. [57, 58, 101] showed that using prioritization schemes, NNs are capable of learning from highly unbalanced datasets on both RL tasks and image classification tasks. In this section, we study if these schemes are also applicable to the field of system identification. To evaluate the performance gain brought by those schemes, we will apply them in an MPC on a track-following task.

In this section, we use the MPPI [114] controller. This controller can work with any dynamic model and has been shown to work well [115, 25] coupled to NNs on robotic systems. This section presents a thorough study of the importance sampling scheme, along with their application on an MPC. We study the impact of different parameters of the MPPI and show how they influence the robustness of the control algorithm across the different learning schemes. Finally, we apply the MPPI to a real system.

4.3.2 Related work

This work leverages both Neural-Networks and Prioritization schemes to learn dynamic models. A complete review of the state of the art on these problems can be found in section 2.2, and section 2.3.

To demonstrate the benefits of the samples prioritization when applied to system identification, we use them on a “race against the clock” task as part a of an MPC. MPCs have been used to control drones [116, 117] and rovers [118, 119] with impressive success. MPC algorithms optimize the trajectory of a system such that it follows the trajectory that yields the lowest cost. Unfortunately, most of the well-known MPC algorithms such as the LQR or H_∞ controllers require the model of the robot to be written in a closed-form equation. This is something that we do not have as the model is a NN. Hence, in our study we use the MPPI, an MPC controller first defined in [115] and then refined in [114]. The particularity of this controller is its high flexibility, it can work with almost every cost function or model. For instance, the cost function can implement both objectives and constraints, which is very useful in autonomous control where both mission and security are often in competition. However, this controller is very expensive to run when compared to LQR or H_∞ controllers. Because it relies on both a NN and a monte-carlo optimization scheme, this algorithm requires to run on a GPU which limits its application to fairly large robotic systems.

In this section, we apply various importance sampling methods to system identification. Those approaches are evaluated on a custom dataset that exhibit a strong non-linearity, and unbalancing. We expand the prioritize experience replay and gradient upper-bound method to an MPC task and show its advantage over non-prioritized models. We then demonstrate the ability of these models to perform robotic tasks of various difficulties on a USV. We explore how the MPPI behaves as its main parameters change, and how these changes translate to the different prioritization schemes. We conclude this section by applying our findings on a real system, and provide insights on the main challenges that arose when we deployed the MPPI in the field.

4.3.3 MPPI

To correctly steer the robot, the MPPI controller is built on two main components: a dynamic model and a cost-function. The dynamic model, which uses a NN, is used to infer future trajectories. The cost-function measures three metrics: a position cost, inferred from the cost map defined in the world frame, a velocity cost, which is computed based on the system's velocities in the robot frame, and a heading cost which is based on the heading of the USV.

Hence, to be able to accurately compute these costs, we rely on a state X_t. More specifically, in our setting, our state X_t is composed of the 2D pose of the robot in the world frame x_t, y_t, and θ_t along with its velocities in the robot frame: the linear velocity v_{lin_t}, the lateral velocity v_{lat_t} and the angular velocity ω_t. The position update is done using a kinematic update as shown in Equation 4.4, while the next velocity is given by the NN. It is worth noting that the dynamic model predicts the next velocities in the robot frame. Hence, they have to be projected into the world frame to update the pose of the robot.

$$\begin{cases} v_{x_t} = -\sin(\theta_t)v_{\text{lin}_t} + \cos(\theta)v_{\text{lat}_t} \\ v_{y_t} = \cos(\theta_t)v_{\text{lin}_t} + \sin(\theta)v_{\text{lat}_t} \\ x_{t+1} = x_t + v_{\text{lin}_t}dt \\ y_{t+1} = y_t + v_{\text{lat}_t}dt \\ \theta_{t+1} = \theta_t + \omega_t dt \end{cases} . \tag{4.4}$$

To find the optimal trajectory to follow, the MPPI samples N sequences of commands over a time horizon of T time-steps. Using these sequences of commands and running them through the dynamic model gives N trajectories from which the cost-function can infer the costs. Based on the cost of the trajectories, the optimal set of commands found at the previous optimization step is updated and applied to the system until the next optimization step. In our implementation, we reduced the update frequency to 5Hz to match the relatively slow pace of our system. For comparison, in [115], the update rate is set to 40Hz. Because of this, the commands are no longer smoothed using the Savitsky-Golay filter as it is in [114, alg. 2]. A pseudo-code of our implementation can be seen in algorithm 5. Even though we sample our commands at 5Hz, we send commands to the system at a rate of 20Hz. To do so we apply a linear interpolation on the set of optimal commands found by the MPPI.

As briefly mentioned earlier, the cost function used to optimize the MPPI trajectory is composed of 3 components:

- A cost-map: this component of the cost function makes sure that the robot remains on the track. In our case the cost-map is computed based on the quadratic distance from the track.
- A velocity cost: this component of the cost function makes sure that the USV moves on the track at the desired speed. It is computed using Equation 4.5, where v_{target} is the desired speed and v the actual speed of the USV.

$$\text{vel\_cost} = \frac{\left|v_{\text{target}} - v\right|}{0.0001 + v}. \tag{4.5}$$

- Finally, the last element of the cost function is a heading cost. It is computed based on the difference between the heading of the robot and the heading of the track. This cost is the error between the desired heading and the actual heading. The desired heading is set for each segment and follow the track counterclockwise direction. This cost is given by : $head\_cost = \left|\theta_{target} - \theta\right|$. This element of the cost is only used in the square track (see subsection 4.3.4).

In the end, the total cost is given by Equation 4.6, with α_1, α_2 and α_3 regulating the weights of the different components of the cost function.

$$cost = \alpha_1 \text{pos\_cost} + \alpha_2 \text{vel\_cost} + \alpha_3 \text{head\_cost}. \tag{4.6}$$

Experiments

In this section we explain how the experiments are performed, what is evaluated, and how it is evaluated.

Algorithm 5: Model Predictive Path Integral [114]

```
$F$ : Dynamic model
$T$ : number of timesteps
$K$ : number of sampled trajectories
$\phi$ : cost function
$u_t$ : commands sent at step $t$
$s_t$ : state at step t
$U = \boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_T$ : initial control sequence
Sample $\varepsilon_k = \varepsilon_k^1, \varepsilon_k^2, \ldots, \varepsilon_k^T \sim \mathcal{N}(\mu, \sigma^2)$
for $k = 0$ to $K-1$ do
    for $t = 1$ to $T$ do
        $u_t = \boldsymbol{u}_t + \varepsilon_k^t$
        $s_{t+dt} \leftarrow F(s_t, u_t)$
    $S_k = \{s_t$ for $t$ in $[0, T]\}$
    $C_k \leftarrow \phi(S_k)$
$\beta \leftarrow min_k[C_k]$
$\eta \leftarrow \sum_{k=0}^{K-1} exp(-(C_k - \beta))$
for $k = 0$ to $K-1$ do
    $w_k \leftarrow \frac{1}{\eta} exp(-(C_k - \beta))$
for $t = 1$ to $T$ do
    $\boldsymbol{u}_t = \boldsymbol{u}_t + \sum_{k=1}^{K} w_k \varepsilon_k^t$
return $U$
```

4.3.4 Simulation Setup

We tested our approach in simulation using the Gazebo software[4], a simulator that allows creating complex simulations with custom robots and hardware. The simulation of the USV itself is done using the heron package[5] along with the uuv-simulator[6]. The first one provides a simulated version of Clearpath Robotics's Heron an USV, while the latter provides advanced water buoyancy simulation, and realistic thrust non-linearities that imitates the real USV behavior. All the experiments were carried out using ROS[7], a well known robotic middleware.

To evaluate the impact of the different parameters and models, we created two different tracks with a width of half a meter. The first one, a simple one, features smooth curves and

[4]gazebosim.org
[5]github.com/heron/heron_simulator
[6]github.com/uuvsimulator/uuv_simulator
[7]www.ros.org

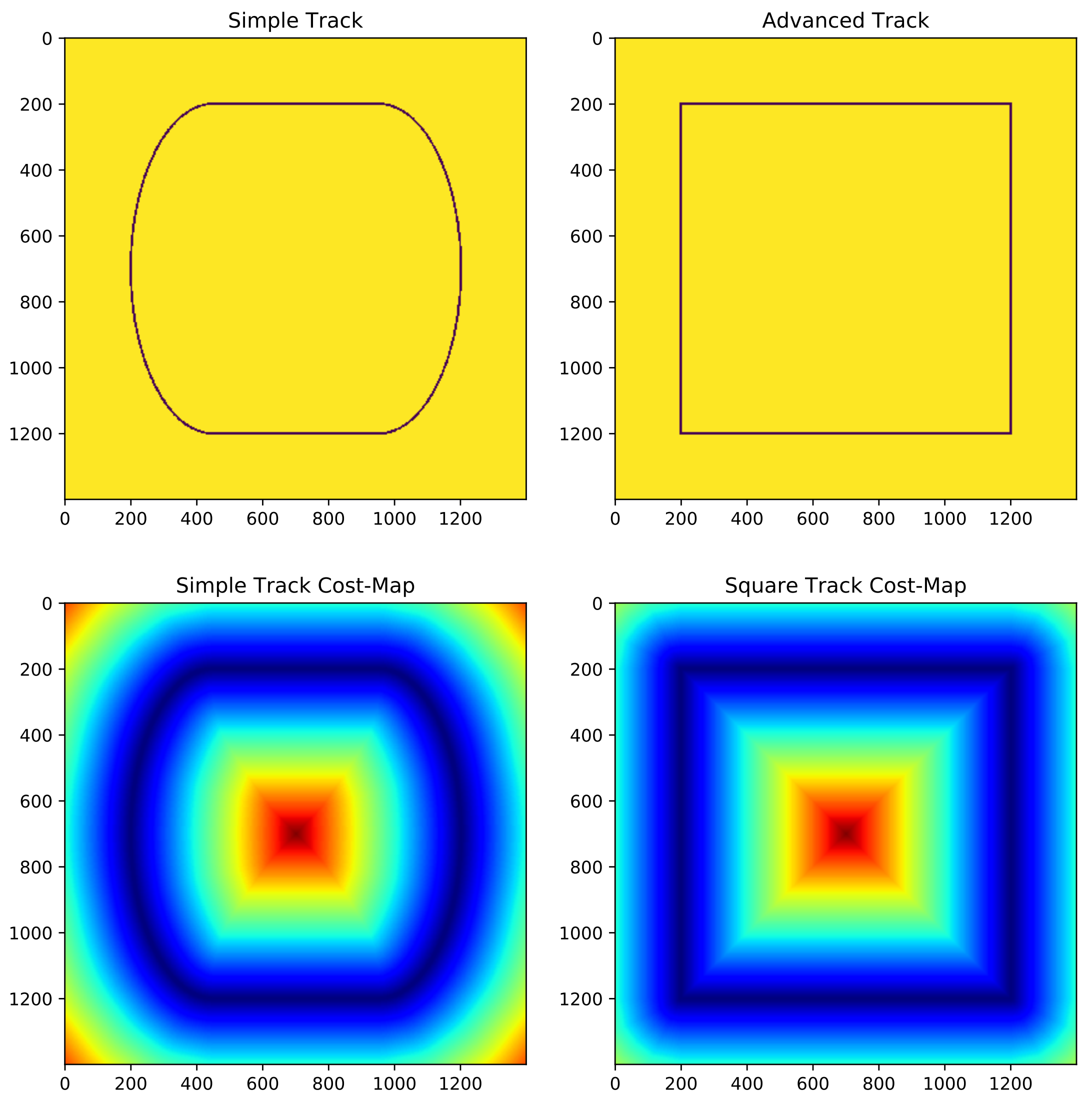


Figure 4.1: The two tracks used to evaluate the different learning paradigm and parameters of the MPPI (first row), associated with their respective cost maps (second row).

slow changes, while the second one has abrupt orientation changes. The two tracks can be seen in Figure 4.1. From these tracks we compute a cost map used by the MPPI to plan its trajectories. In our case, the resolution of the cost map is 10cm/pixel, and their values are computed from the following rules: if the pixel is on the track then its cost is 0, if the pixel is outside the track then its cost is determined by its squared distance in pixels from the track. The two cost maps can be seen in Figure 4.1. This is different from the implementation of the cost-map in the original MPPI [25, 114] code. In the latter, the cost map was binary with 0 for the track and 1 outside of it. In our experiments, we have found that having a gradient around the track helps the USV to stay on it: if it ventures outside of the track, the gradient helps the USV to come back to it.

Neural Networks

In the following subsection, we detail the dataset used to train the NNs along with how the networks are trained and evaluated.

Dataset

To train the NNs that are used to predict the dynamics of the system, we need to create a dataset. To create a system identification dataset, the most efficient method is to sample random commands and send them to the system for a random amount of time. Unfortunately, even though this method works perfectly in simulation, in the real world it does not work for obvious reasons. With this in mind, we created a dataset that is a combination of straight lines and turns at different velocities that we mixed with twenty percent of random commands. In addition to being closer from what a real dataset looks like, this dataset should also show how the PER and the gradient upper-bound can leverage the random samples to improve their prediction performances.

Training parameters

The NNs used in this section are simple MLPs. More precisely, these MLPs feature two dense layers with 32 neurons, and a final layer with 3 neurons: one for the linear velocity, one for the lateral velocity, and finally one for the angular velocity. The activation function used are LReLU [15], and there is no activation function in the final layer. The input of the network is a flattened sequence of the six previous states and commands. While these networks can look simplistic, this answers a performance need: with up to 1,000,000 forward passes per seconds, the networks need to be light enough to run in real time on an embedded platform. Before training, the dataset is normalized by subtracting its mean and dividing it by its standard deviation. To perform the regression, an L2 loss is used. Finally, we use the Adam optimizer [22] with a learning rate of 0.001. All the networks are trained using Tensorflow [105] version 1.15.

Evaluation

To evaluate the performance of the networks, we build two datasets. The "full-random" dataset: a balanced dataset, and an "unbalanced" dataset. Both the "full random" dataset and the "unbalanced" dataset are split into three subsets: a training set, a validation set and a test set. In the "full random" dataset, all the subsets are comprised of samples obtained by generating random commands. However, for the "unbalanced" dataset, only the test set is composed of samples acquired using random commands. Both the training and the validation set are made of a mix of straight line and random commands, as defined in subsubsection 4.3.4. Overall, both datasets include about 1 millions samples. The goal here is to see if the networks trained with PER or gradient upper-bound will achieve better performances than the standard training procedures. To evaluate the different schemes, we trained them with all the combinations of parameters 5 times and averaged the results for each combination of parameters. In the case of the PER, we trained 5 networks, for each combination of α and β, with α and β ranging between 0.1 and 0.9 with a 0.1 increment. For the gradient upper-bound, we trained 5 networks for different super-batch size values. Specifically, we took as super-batch size every power of 2 between 32 and 8196. Additionally, to show how those networks would perform in the case of a dataset acquired only by applying random commands, we also included the results of these comparisons on a fully random dataset in addition to the mix one detailed in subsubsection 4.3.4. In this case, we want to see if these methods perform worse than the standard one when applied to a well-balanced dataset, for instance by focusing the networks' training on outliers.

To evaluate the performance of the networks, we consider two metrics:

- The Root Mean Squared Error (RMSE) of the network when predicting the next state of the system. It will be referred to as single-step accuracy.

- The RMSE of the network over a trajectory of 15 points. In this case the network iterates over its own predictions 15 times. It will be referred to as multi-step accuracy.

For both of these metrics, we report the average and the standard deviation of the RMSE over the 5 runs.

In our experiments, we also evaluated the impact of the different parameters of the MPPI. We studied the impact of the following parameters:

- The number of samples: this is the amount of trajectories that are sampled in the Monte-Carlo optimization process. A small amount of samples means that the trajectories generated will most likely not sufficiently cover the area of space that is interesting. On the other hand, a large amount of samples means that the trajectory will cover a broader space and that the chances of having scattered trajectories is lower. The main drawback of having a large amount of samples is an increase in computational cost. In this study, we vary the number of samples between 500 and 6000.
- The number of time-steps: this is how far the MPPI predicts in the future. Too few time-steps, and the sampled trajectory will not go far enough in the future. This means that the algorithm will not be able to account for the slow dynamics of the boat and its high slippage; the algorithm will not anticipate enough and may not be able to turn correctly. However, with too many time-steps the problems comes from the dynamic model learned using the NN: for every time-step, the model iterates on its own predictions, thus increasing the prediction error over time. In this study, the number of time-steps vary between 5 and 40.
- The variance of the sampling: this parameter rules how new trajectories are sampled. As shown in algorithm 5, the new commands are sampled by taking the optimal commands found at the previous optimization step, and adding noise onto them. The variance itself is how much noise will be applied. Too much noise, and the trajectories will be sparse requiring a high amount of samples to compensate; not enough noise, and the

trajectories will be generated in a very small cone leading to a suboptimal solution. All in all, we tested different variance values ranging from 0.15 to 2.0.

All these experiments were carried out on the square track. Its abrupt turns helped better differentiate the parameters. On these experiments, the results are the average of 15 runs, along with the standard deviation between these runs. We also compared how the different learning paradigms impact the evolution of the parameters. To do so, we tested all the networks described previously, and reported the results of the best performing networks across all parameters.

When evaluating the MPPI we monitor two distinct metrics: its performance in terms of how well it stays in the track, and its average velocity. While we could have studied the cost on the velocity, we chose not to as it is very noisy, and no useful information can be taken out of it. This is due to the exponential penalty added to the velocity cost as the USV slows down.

4.3.5 Results

Neural Networks training results

First, we present the results of the NNs training, with and without a sample prioritization scheme. Here, we expect the different prioritization schemes to perform better than the baseline, in particular on the unbalanced dataset. The main question is which of the PER or gradient upper-bound perform better. We evaluate them on two datasets: a dataset solely comprised of random commands and an unbalanced dataset. All the results presented in the tables and figures below are reporting the average of 5 trainings with different optimizer seeds.

Figure 4.2 and Figure 4.3 show the grid-search results of the gradient upper-bound and the PER prioritization schemes on the unbalanced dataset. Figure 4.4 and Figure 4.5 show the grid-search results of the gradient upper-bound and the PER prioritization schemes on the fully random dataset.

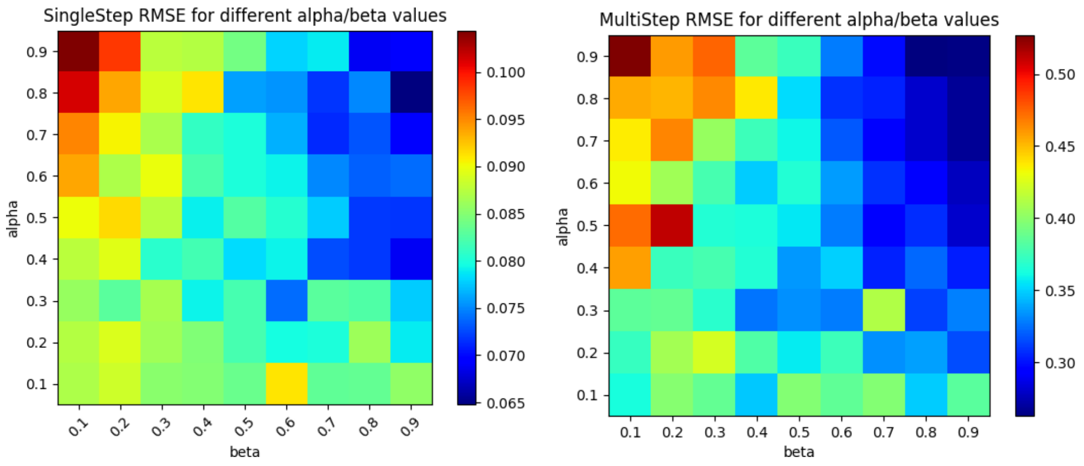


Figure 4.2: The PER results on the unbalanced dataset. Left: single-step accuracy. Right: multi-step accuracy. The colder the color the lower the RMSE. The lower the RMSE the better.

Table 4.5: Neural networks overall performance. The PER and gradient upper-bound networks were selected as the best performing parameters in multi-step accuracy. Lower is better.

	full random dataset				unbalanced dataset			
	single-step RMSE		multi-step RMSE		single-step RMSE		multi-step RMSE	
	mean	std_dev	mean	std_dev	mean	std_dev	mean	std_dev
PER	0.060	**0.006**	0.18	**0.021**	**0.068**	0.0048	0.26	0.088
GRAD	**0.052**	0.007	**0.17**	0.024	0.070	0.0029	**0.25**	**0.024**
STD	0.066	0.014	0.27	0.101	0.082	**0.0028**	0.40	0.1

Let us first have a look at Figure 4.2: it shows that the best results are obtained with both a high α and a high β. This means that to achieve the best results, the PER must put the emphasis on the more difficult samples (α) but also compensate as much as possible the bias that they introduce (β). Additionally, if we look at the color distribution, it is always better to pick a large β, while α seems to be less important. If we now look at Figure 4.4, we can see that on the fully random dataset, the single step prediction is more homogeneous in performance, with the notable exception of selecting a high α coupled to low β. This behaviour, which can also be found on the unbalanced dataset indicates that if the PER does not compensate for the bias, then it most likely overfits on outliers, hence degrading the general model performance. Interestingly, when considering the multi-step accuracy of the

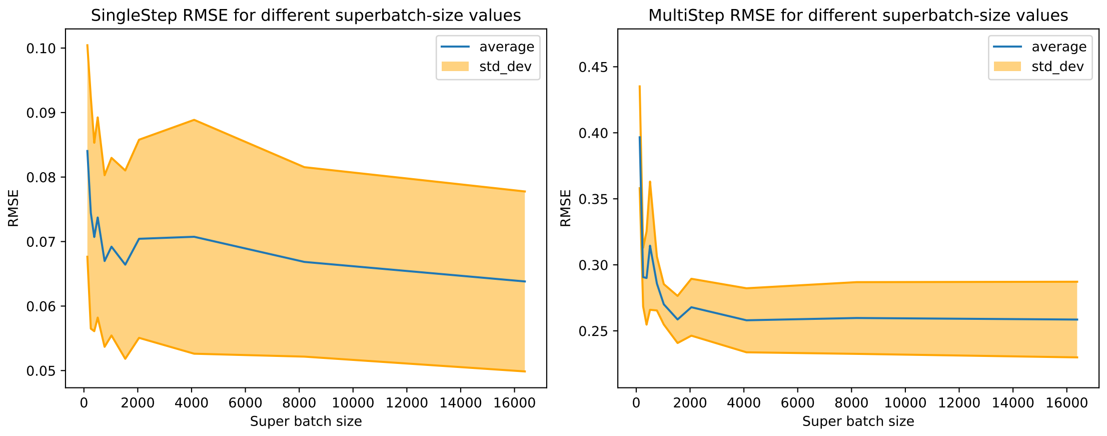


Figure 4.3: The gradient upper-bound results on the unbalanced dataset. Left: single-step accuracy. Right: multi-step accuracy. The lower the RMSE the better. The narrower the orange area the better.

full random dataset (Figure 4.4), we can see that the color distribution is similar to the one of the unbalanced dataset. This indicates that the PER scheme helps improve multi-step performances in general, which is confirmed by the results shown in Table 4.5.

We can now move on to Figure 4.3 showing the impact of the gradient upper-bound parameters on the unbalanced dataset. On this figure, it can be seen that, as the super-batch-size increases, the single-step performance also increases. However, on the multi-step RMSE, the accuracy is saturating once the super-batch-size exceeds 2048 samples. Yet, as the super-batch-size further increases the variance diminishes. When comparing these results to the fully random dataset (Figure 4.5), the multi-step accuracy appears to be ruled by the same phenomenon with a saturation of the performances after 2048. Surprisingly, on the single-step accuracy, the increase in super-batch size initially penalizes the accuracy, and as it reaches a value larger than 2048, the accuracy remains somewhat constant with a variance slightly increasing. This could be due to a focus on irrelevant samples that degrades the performances.

Finally, table Table 4.5 compares the different learning schemes. As can be seen on this table, as expected, both the PER and the gradient upper-bound consistently perform better than the standard training method on the mean RMSE. The most interesting element of this

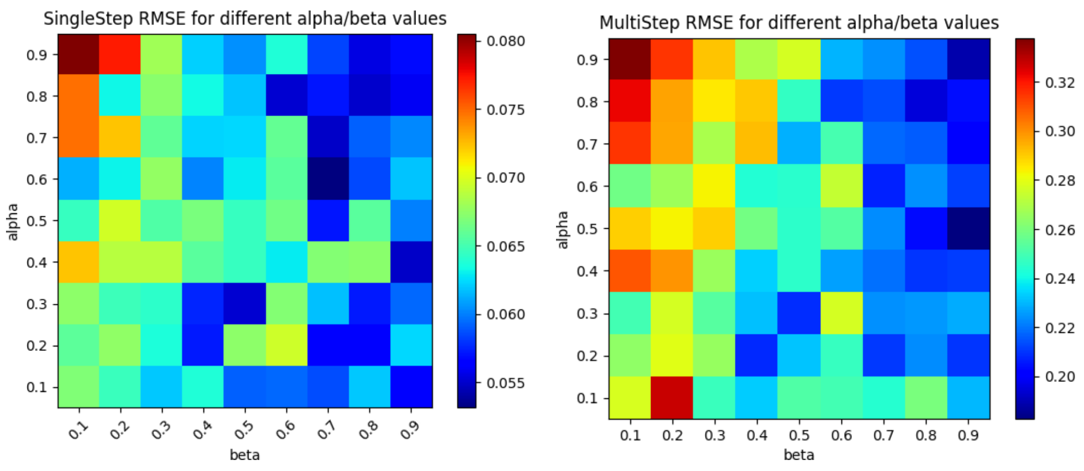


Figure 4.4: The PER results on the fully random dataset. Left: single-step accuracy. Right: multi-step accuracy. The colder the color the lower the RMSE. The lower the RMSE the better.

table is the large performance boost that these methods offer on the multi-step accuracy, with almost 30% of performance increase on both datasets. Furthermore, from the standard deviation on the different metrics, one can see that the prioritization schemes reduce the variance among the trainings. This is particularly interesting as it means that training with these methods provides models which are more reliable.

MPPI results on the cost-map

Here, we first present the results of the different models when applied in the MPPI on a “race against the clock” task. We then compare the results of the two tracks and discuss how some of the MPPI parameters influence the robustness of the control. As detailed in subsubsection 4.3.4, the results reported in the figures are averaged over 15 runs.

Comparing the different schemes on the simple track

First, the controller is tested on the simple track. It is composed of straight lines followed by arcs forming a loop. The main difficulty on this track arises from the discontinuity in the track’s curvature where a straight line and an arc meet. Figure 4.6 shows how the different learning schemes performed after 670 time-steps of 0.2 seconds. The models shown are

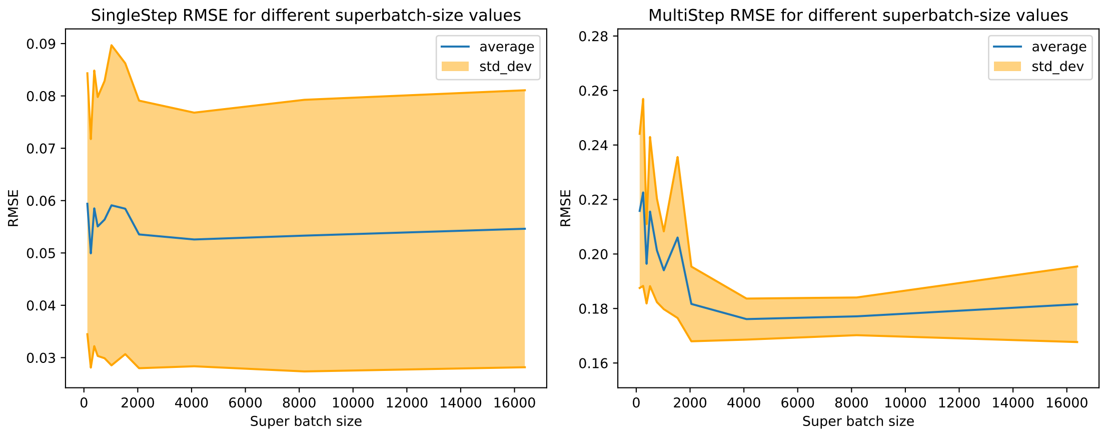


Figure 4.5: The gradient upper-bound results on the fully random dataset. Left: single-step accuracy. Right: multi-step accuracy. The lower the RMSE the better. The narrower the orange area the better.

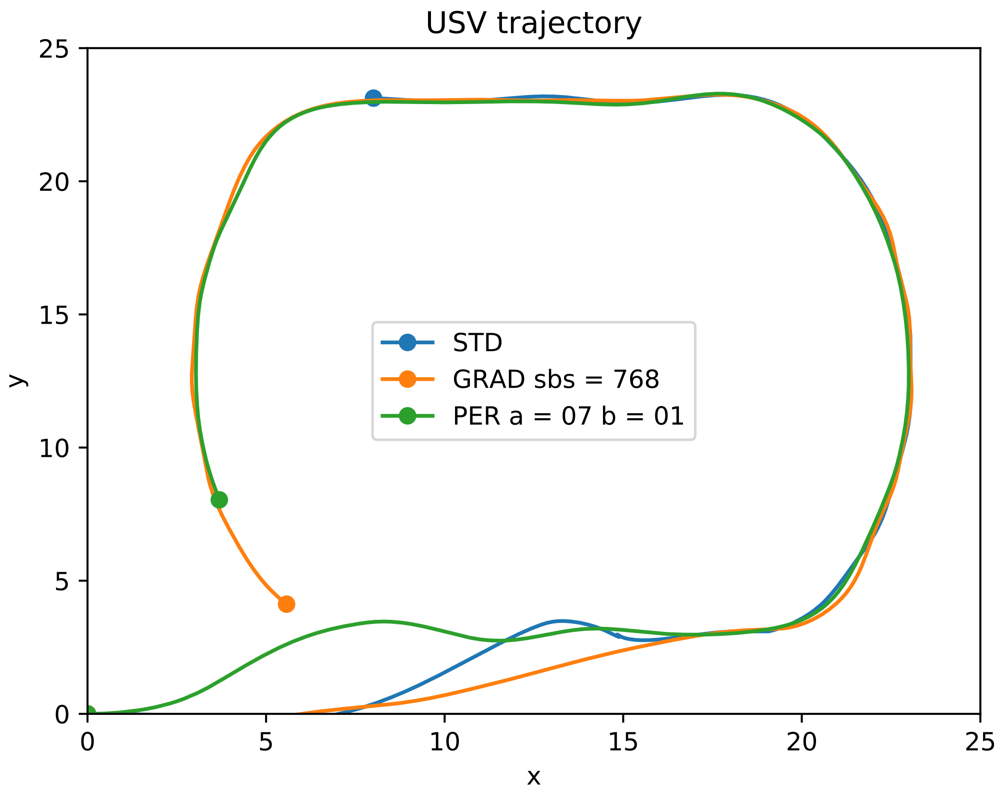


Figure 4.6: Comparison of standard neural network (STD) and prioritized (PER/GRAD) version on a composite track

the best performing ones for their category. The best PER model is obtained $\alpha = 0.7$ and $\beta = 0.1$ while the best gradient model is obtained for a suberbatch size of 768.

We can see here, that the models using prioritized sampling perform better than the one using the standard training procedure. When using the standard model, the controller overshoots as it reaches the track, and struggles to keep up with the pace of the prioritized networks. The network trained using the gradient method overshoots on the first curvature change at $x = 20; y = 4$ but manages to follow the track and beats the other models in the race. Finally, the PER method manages to stay on the track rather well and is closely following the gradient based method.

Comparing the different models on the advanced track

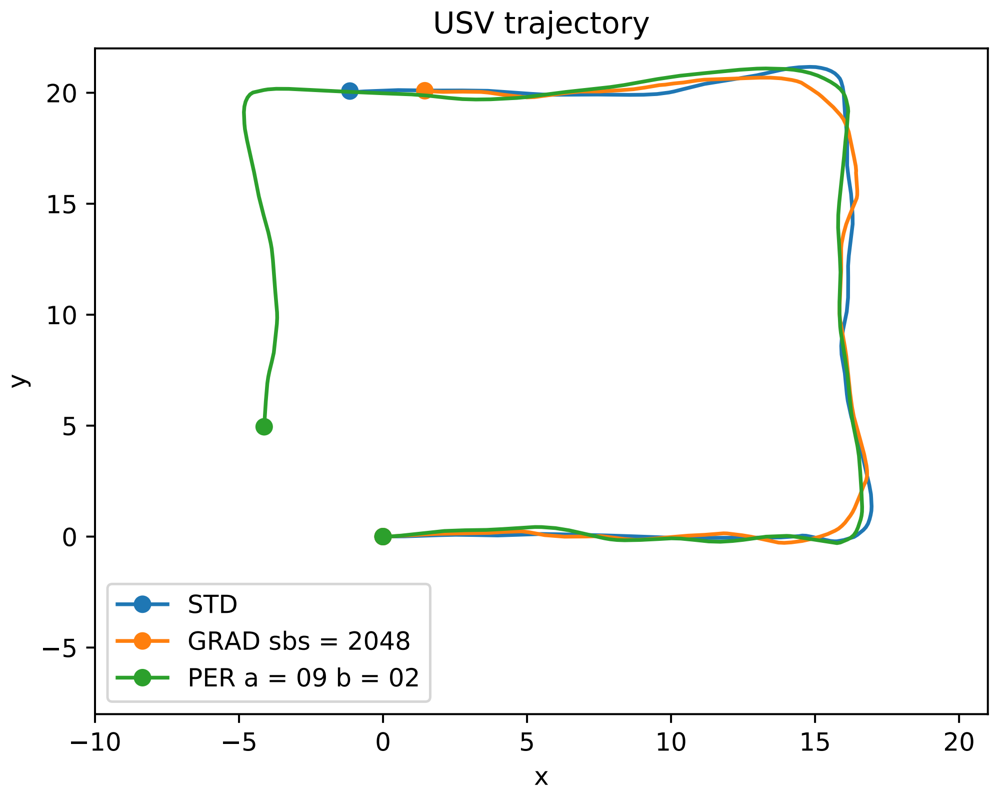


Figure 4.7: Comparison of standard neural network (STD) and prioritized (PER/GRAD) version on a square track

To test the capacity of the algorithms, we repeat the previous experiment on a much more challenging track: a square track. The 90 degrees turns present major difficulty for the USV due to its slow dynamics and high lateral slippage. Despite the complexity of this task, all the models managed to follow the track once we added the heading cost. Without it, they used to stay stuck in the corners. Figure 4.7 shows the trajectories followed by the different models on their best run. On that run, the PER is the fastest, but the gradient-based model is the one with the lowest map-cost, showing that it respects the track better. The detail of the costs is as follows : the gradient achieves an average map-cost of 3.9 and an average global cost of 20.0, the PER achieves an average map-cost of 6.9 and an average global cost of 18.7 while the standard method gets an average map-cost of 6.7 and an average global cost of 22.3.

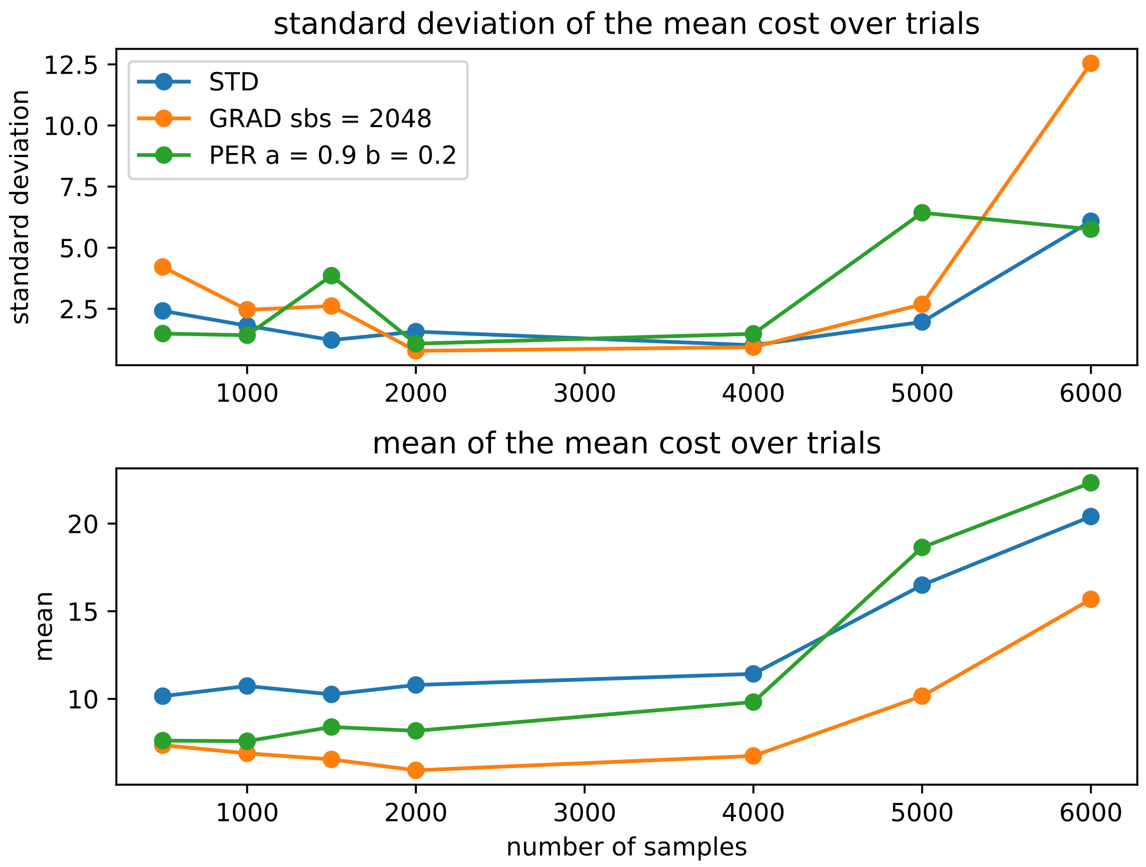


Figure 4.8: Average map-cost and variance over several trials for different number of sample trajectories. Lower is better.

Figure 4.8 shows how the mean of the map-costs evolves as the amount of samples used in the optimization process increases. In our specific setup, using less than 500 samples makes the controller highly unstable, leading to the failure of the track following task for all the models. On the other hand, after 4000 samples we are reaching the limit of what our python implementation can achieve in real time. Above 6000 samples, the controller cannot work properly anymore as it is no longer running in real time, and hence suffers from a delay between its observations and the optimal trajectories it finds.

Overall, the average map-cost is decreasing as the number of samples increases, up to the point where the computational cost becomes a limiting factor. We can see on the mean graph that on average the prioritized versions perform better than the classic ones. The gradient method in particular obtains very good results compared to the other methods. In Figure 4.7, we see that the PER is faster than the gradient. However, Figure 4.8 shows that the gradient is more reliable than the other approaches, with a better average cost. It is important to note that the comparison in Figure 4.7 is on the best run only, while Figure 4.8 show the results over 15 trials.

Figure 4.9 shows the influence of the length of the trajectories being evaluated by the MPPI. Below 5 steps, the trajectories are so short that they cannot take into account the dynamics of the USV, making the controller useless. Above 40 steps, the computational cost becomes so high that, as encountered before, the optimal trajectories cannot be given in real time, leading first to degraded performances and then to the divergence of the controller. In terms of average map-cost, the standard model is worse than the prioritized models. We can also see that increasing the number of time-steps improves the performance of the controller until 25 steps, after which the performances decrease slightly. This may be caused by the fact that as the trajectory gets longer the position error of the model increases, as explained in subsubsection 4.3.5.

In Figure 4.10, we show the results for different variance values when sampling new commands. With too little variance, below 0.15, the controller can not find any relevant

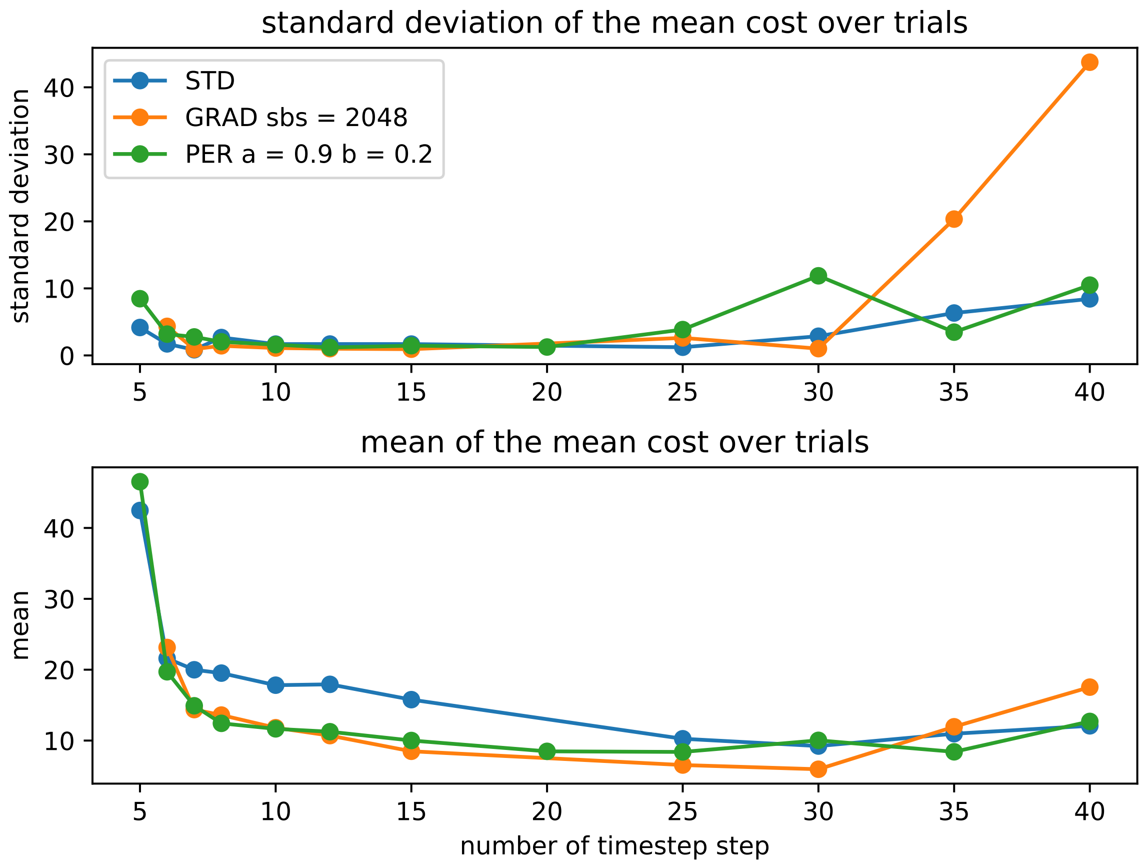


Figure 4.9: Average map-cost and variance over several trials for different number of time-steps per trajectory

trajectory and fails completely. As the variance grows beyond 1.5, the performances decrease, to the point where the algorithm fails and the costs diverge. Figure 4.11 helps better understand the behavior of the system with a low and a high variance. With a low variance, the system does not explore enough and finds a suboptimal trajectory. This can be observed from the many small oscillations on the system trajectory. Also, because the MPPI uses the previous optimal command to sample new commands, the acceleration will be slower using a small variance. On the other hand, with a variance of 2.0 we sample over the whole of the action space. While it can be interesting, it also means that the trajectories will be sparser. This is visible on the oscillations in the straight line that are not present on figure Figure 4.7 that was acquired with a variance of 0.6. The gradient model encounters some issues at the 0.9 variance mark due to one run that diverged. Otherwise, the consistency

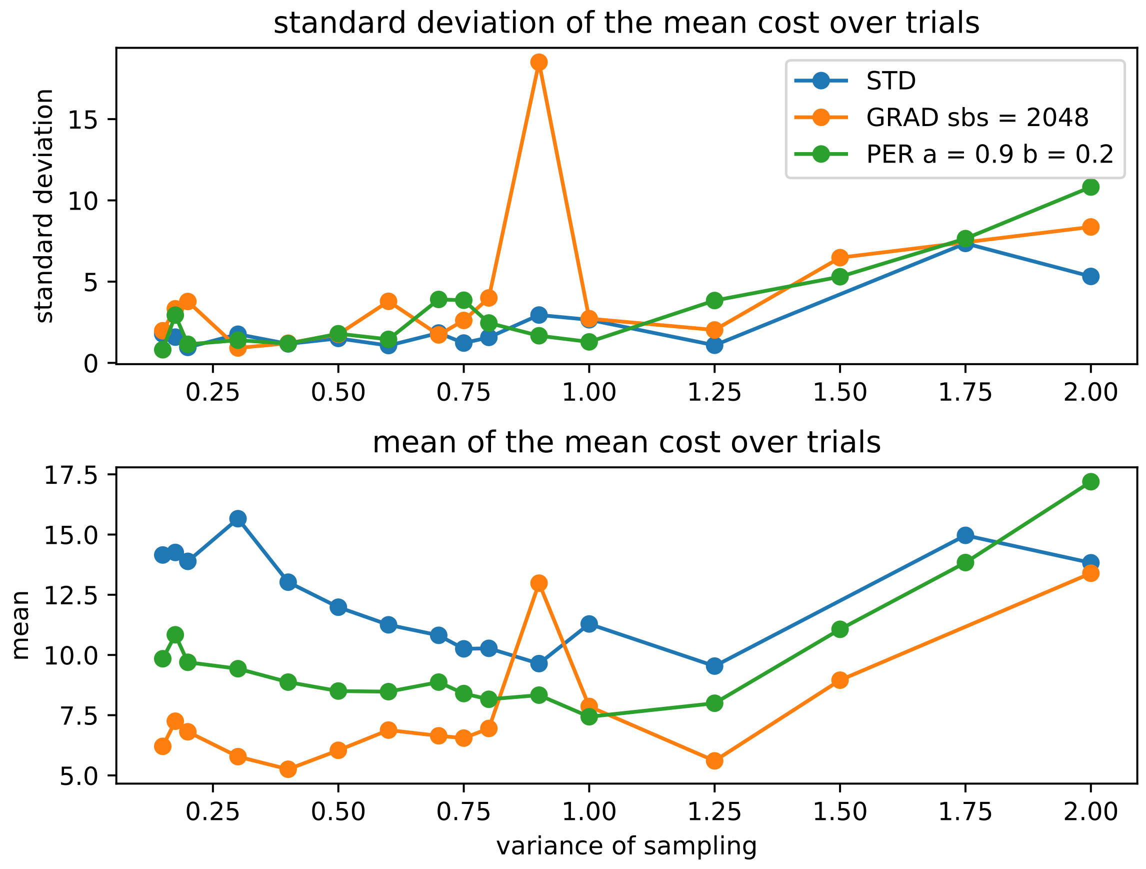


Figure 4.10: Average map-cost and variance over several trials for different sampling variance

of prioritized methods observed in the previous graph is still true. We can also see that an increased variance slightly improves the performance of the standard and PER version.

Overall, it is observed that the prioritized models lead to better results in terms of map-cost. Moreover, for this experiment the MPPI worked best for a number of samples between 1500 and 4000, a number of time-step between 15 and 30 and a variance between 0.4 and 0.8.

MPPI velocity results

In this section, we have a look at how the different parameters of the MPPI impact the average velocity of the USV. Furthermore, we compare the velocity of the different learning schemes.

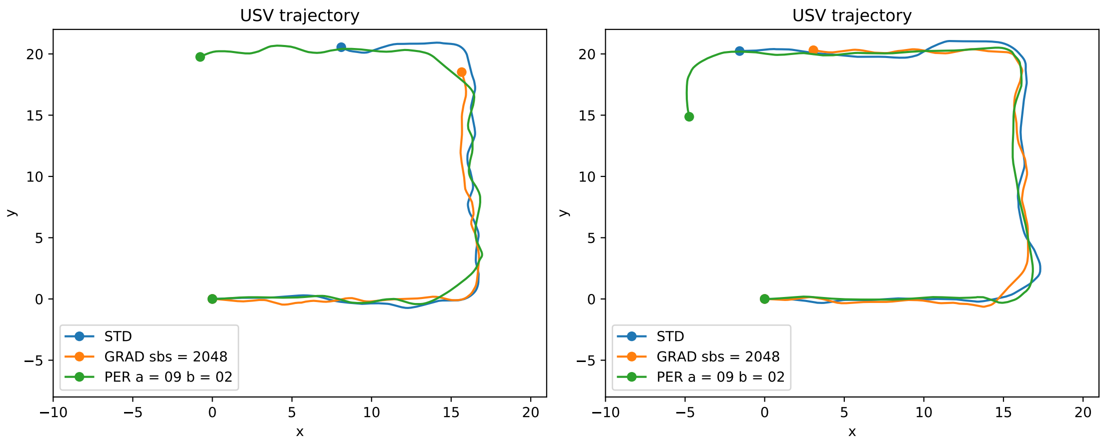


Figure 4.11: Snapshot of the robot position (dot) and their trajectory (line). Command sampling variance at 0.15: left; and variance at 2.0: right.

Let us start by analyzing the impact of the number of times-steps on the mean velocity around the track. From Figure 4.12, it can be seen that, as the number of time steps increases, the velocity of the USV decreases. This makes sense, since with more time-steps the network can plan farther ahead, and anticipates the sharp corners of the track. Another interesting point is that the standard model is going faster than the prioritized models. This indicates that the estimation made by the standard model is not as good as the prioritized model. Hence, it believes it can go faster and will end-up going outside the track. This behavior can be seen in Figure 4.9, Figure 4.8, Figure 4.10 where the standard model almost always has a higher map-cost than the prioritized models. Similar behavior can be seen on Figure 4.13, Figure 4.14 where the standard model is constantly faster than the prioritized approach. In the end, while the standard model goes faster on average, this extra velocity is misused and leads to worst performance on the track-following task.

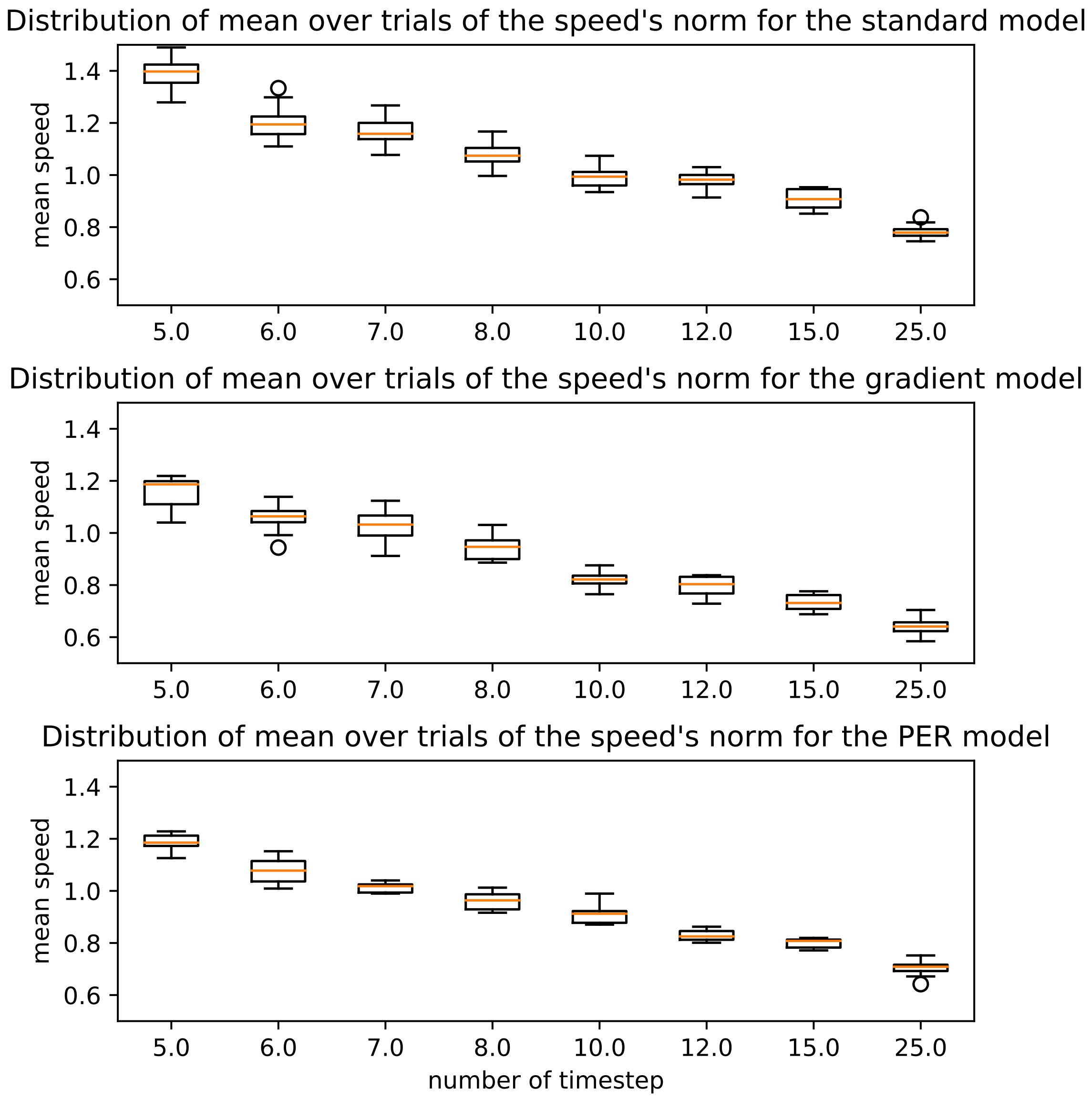


Figure 4.12: Average velocity over several trials for different number of time-steps. Higher is better.

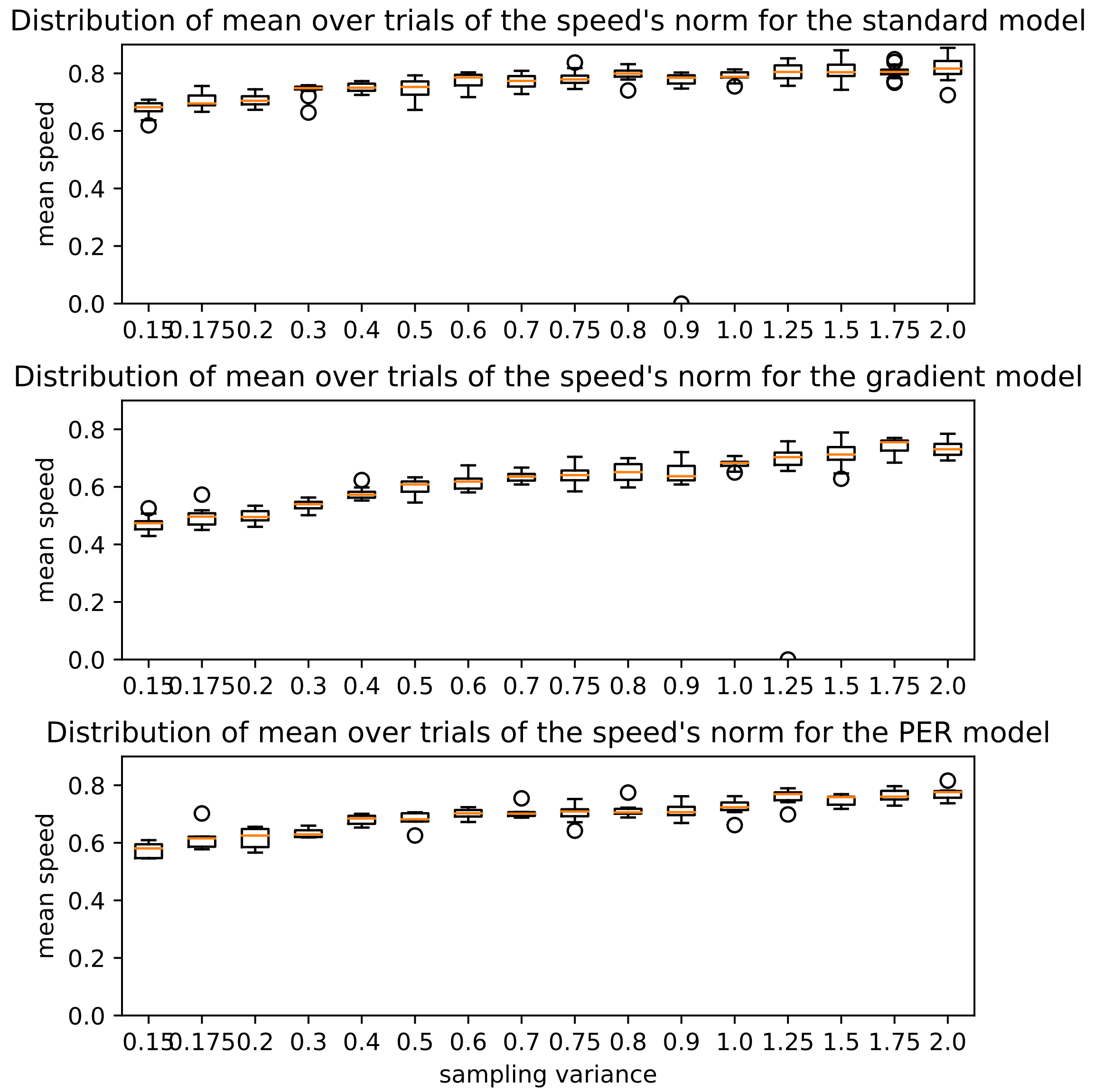


Figure 4.13: Average velocity over several trials for different values of variance. Higher is better.

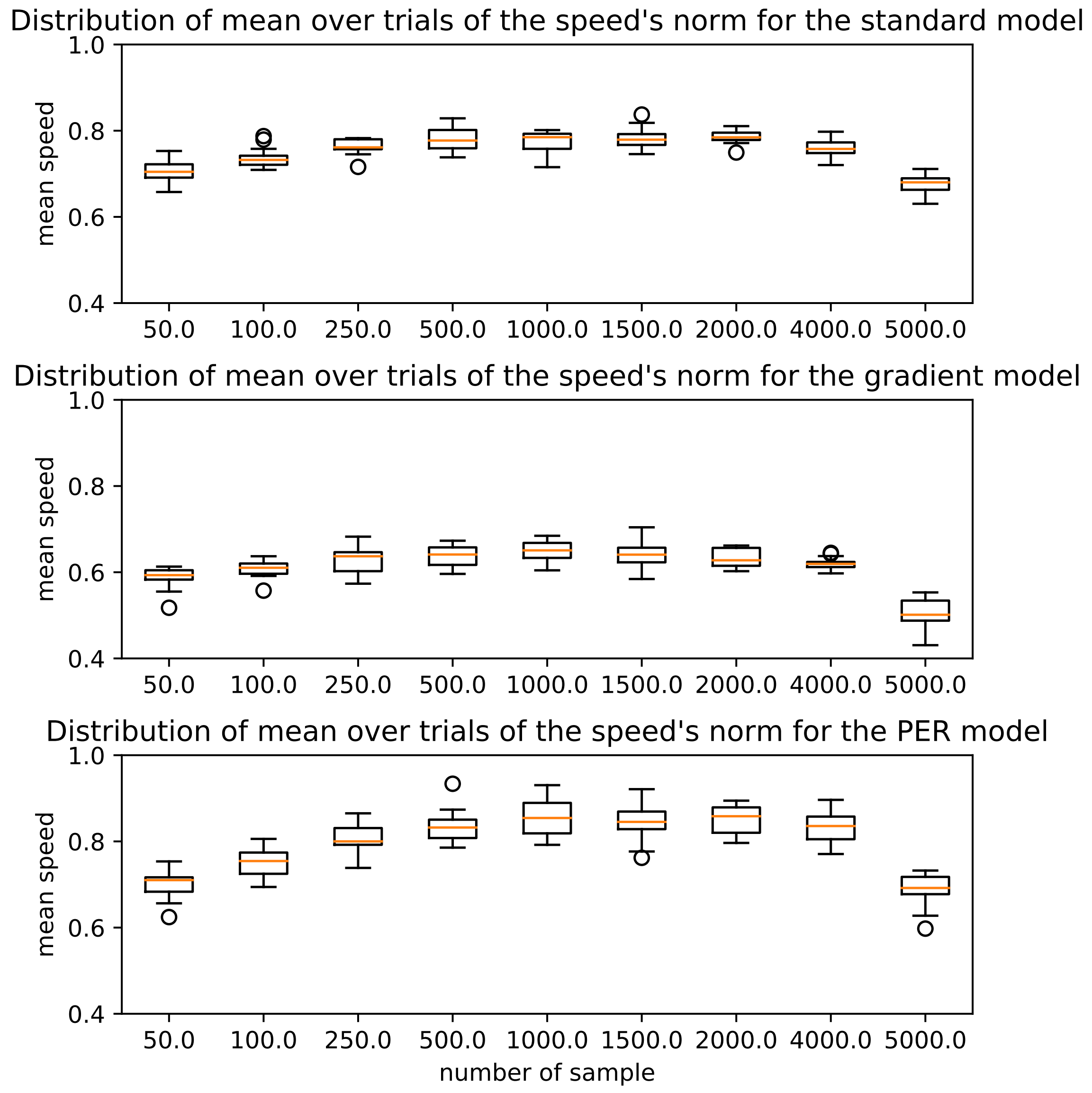


Figure 4.14: Average velocity over several trials for different number of samples. Higher is better.

4.3.6 Application to a real world scenario

In the following we apply the MPPI to a real robotic system, in a real-world scenario: following a lake shore. In this use-case, unlike in simulation, the trajectory to follow is not known a priory. Instead, a track is inferred in real-time using the onboard sensors of the robot.

Problem definition and setup

In this subsection, we used the know-how presented previously to make an USV autonomously follow lake shores. To tackle this task, we relied on the Kingfisher, a 1.5 meter long catamaran from Clearpath Robotics. We fitted our USV with a 20 meters-range SICK LMS111 2D-LIDAR, and an EMLID Reach RS+ RTK-GPS. The GPS was used to acquire the boat state which usually provides a localisation with a precision of at least 5cm at 5Hz. On the computational side, an Intel Atom was used for low-level computations, and an NVIDIA Xavier was used to run the MPPI. Figure Figure 4.15, shows our real system and the test environment.

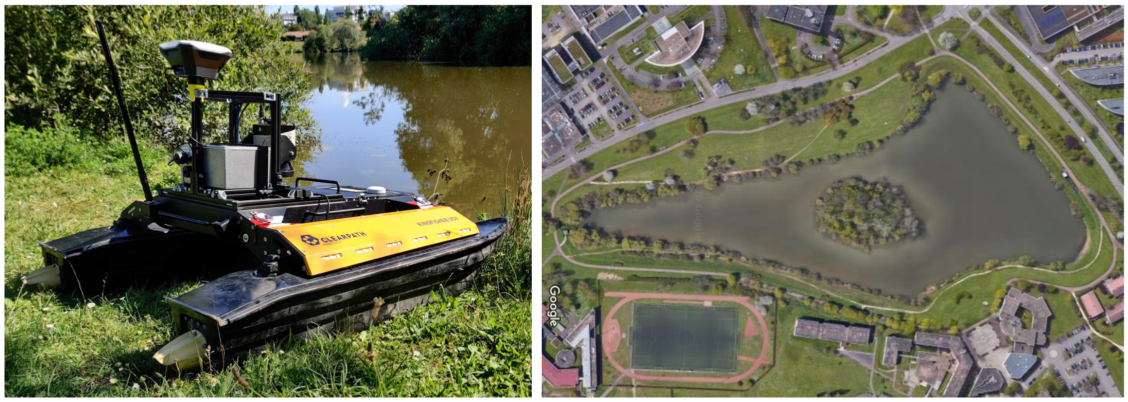


Figure 4.15: Left our USV equipped with its sensors, right top view of our test environment: soccer field for scale. (Lac Symphonie, 57000 Metz, France, Google Maps, 2020)

Unlike the previous subsection, we do not follow a virtual GPS track; instead, we build a track that follows the shore at a 10 meters distance. To do so, we convert the output of the 2D-LIDAR into a local track. We then use this local track as the cost-map for the MPPI.

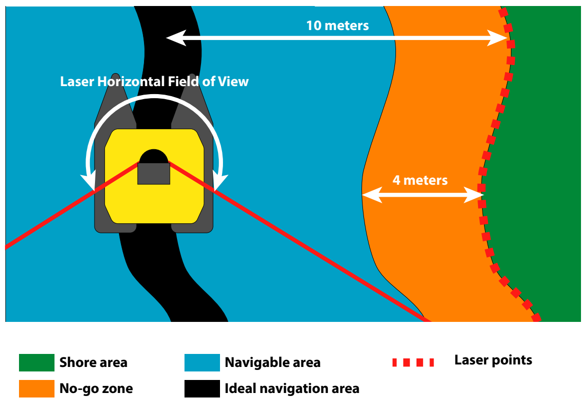


Figure 4.16: The USV and its shore-following task. Black is a zero cost area, blue has a gradually increasing cost as we move away from the track, orange has a positive cost of 500, green is collision and has a positive cost of 10000.

The main difference between this method and the one presented before, is that the map is no longer fixed but changes with every new MPPI step. An illustration of the system and its task can be found in Figure 4.16.

In the end, the cost function that we used to solve this task is similar to the one shown in Equation 4.6, and the target velocity was set to 0.7m/s: the maximum velocity the system could reach while performing its task reliably. The heading term was removed from the cost function as we do not have a smooth and well-defined trajectory to follow, but rather a track to stay on. Regarding the MPPI parameters, we leveraged the previously found results and chose the parameters that made most sense in an embedded application. As a result, we opted for 1500 samples and 20 time-steps. As for the variance, we found that 0.3 was a good value for the real system, as we will see in subsubsection 4.3.6. Finally, we trained our NN using a PER scheme on a dataset collected in the real environment with the method described in subsubsection 4.3.4. For this dataset, we also used a grid-search to find the optimal PER parameters.

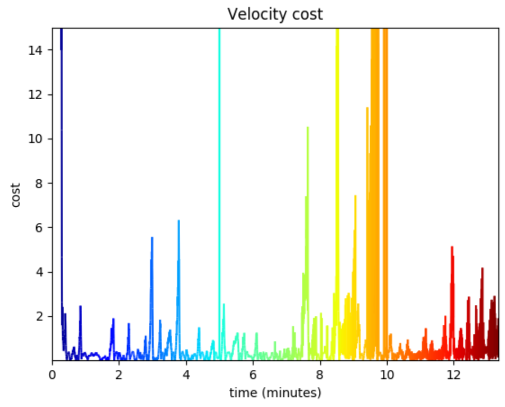
Results

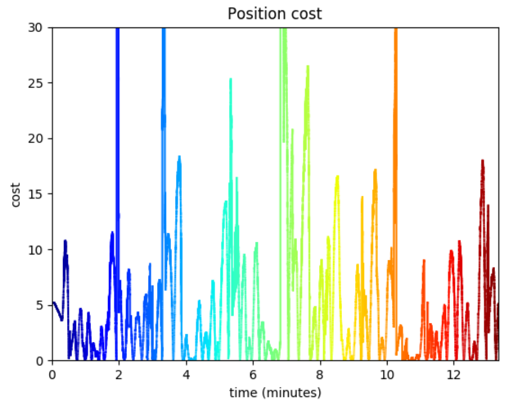
Because performing grid-searches on a real system is prohibitively expensive, we chose to only report one experiment in the result section and explain how we tuned some parameters.

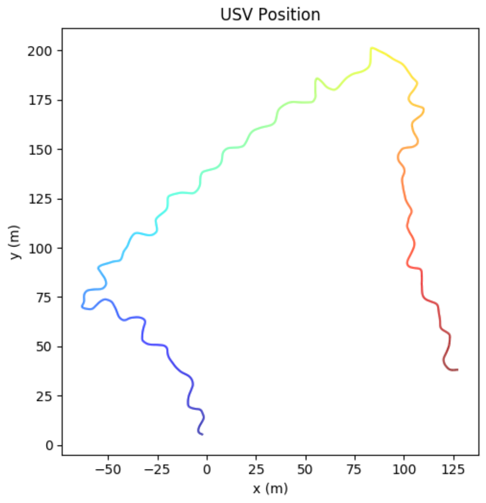
Figure 4.17 shows the trajectory followed by the USV along with the cost in position and velocity associated to this trajectory.

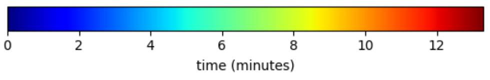
Here we only show a third of the run, the remaining part having been frequently interrupted by fishermen. During this run, the USV maintained a velocity of $0.83m/s \pm 0.19m/s$, and a distance from the shore of $9.5m \pm 2.5m$. As we can see on the velocity cost, this constraint is fairly well respected. We can observe a few high spikes: these happen when the robot goes backward because it came too close to the shore. Regarding the distance from the shore, the cost values may seem large, but this is because LIDARs measurements are noisy in natural environments. Since the leaves do not reflect the laser beams well and the branches are small objects, the laser beams only hit them partially, and it results in incorrect distance measurements. Finally, because our laser only has a 270° field of view, there is a blind spot which leads to a deformation of the local path when the robot is not parallel to the shore. Nonetheless, the USV performed well, and across our testing it never collided with the shore and managed to go around most of the obstacles. The main limitation of the approach was that when facing obstacles that have an acute angle relatively to the shore the boat could turn back.

On the real system, we found that the sweet-spot for the command variance sampling was around 0.3, when in simulation values from 0.5 to 1.0 seemed to work best. Having a larger variance would lead to instability. This could be explained by the constraints faced by our embedded system. In simulation, we can see that optimal amount of samples range from 2000 to 4000 and the ideal horizon was around 25. This was not achievable with our implementation on the Xavier. Hence, we reduced the number of particles, which in-turn forced us to reduce the variance of the command sampling to prevent the MPPI from becoming unstable.


Velocity cost
cost
time (minutes)


Position cost
cost
time (minutes)


USV Position
y (m)
x (m)


time (minutes)

4.3.7 Conclusion Prioritization Applied to MPC

Following the work done in section 4.2, this section investigated the potential benefits of using importance sampling scheme for deep model identification. After showing the advantage of our method when modeling the USV, we showed that it translated into good performance on a track-following task for two different tracks. Then, we studied the reliability of the controller for the different models depending on the algorithms parameters. We showed that the prioritization improves the controllers' performances. The gradient-based method outperformed every other methods in most cases. Even though these methods bring significant performance boost, both the PER and gradient upper-bound require grid searches to work optimally. This can be seen on datasets where there are outliers to which the PER is very sensitive to. Additionally, the main drawback of the MPPI and the NNs in general comes from their inability to correctly estimate their uncertainty. Yet, recent advances in the field [44, 46, 120] show promising results.

4.4 Conclusion

In this chapter, we have shown that prioritization methods can be used to improve the accuracy of dynamic systems modeling. We compared two different method, the PER and the gradient upper-bound. We showed that even though the gradient upper-bound has fewer hyperparameters, it performs better than the PER in average. When applied inside an MPPI controller, we proved that these methods improve the performance of the controller and its reliability. However, we believe that the main limitation of these methods is their inability to estimate the uncertainty. Recent advances in this field [44, 46, 120] show promising results. Future work should focus on implementing these inside the MPPI framework. An interesting application could consist in leveraging the uncertainty in the state to increase the variance or the number of particles inside the MPPI. For instance, when the model is certain of its prediction, the variance could be smaller than when the uncertainty is large. At the same time, if the variance is smaller, then the controller does not need as many particles, reducing the computational cost of the MPPI. Another interesting area of study could be to learn a function similar to the value function in reinforcement learning. This function could then be used to adjust the sampling variance at run-time, improving the accuracy of the controller in low-variance scenarios, while allowing the controller to get out of complex situations by greatly increasing the sampling's variance.

CHAPTER 5
MODEL BASED REINFORCEMENT LEARNING FOR MOBILE ROBOTS NAVIGATION

5.1 Introduction

In this chapter, we study how model-based reinforcement learning can be used to efficiently teach robots to solve navigation tasks in outdoor environments. Understanding how to efficiently teach robots to solve tasks is critical, as collecting samples from real robots is prohibitively expensive. Furthermore, simulating robots in complex environments is not that cheap as well. While novel simulators like IsaacSim[1] promises faster GPU enabled simulations, training still has a cost. This is why we used MBRL[121] in this thesis. An efficient form of RL that leverages a learned transition model to learn how to act. In the following sections, we will start with a quick overview of what RL is along with a presentation of Dreamer[122], a state-of-the-art MBRL algorithm that we built on in this thesis. Then, we will see the type of robots we used as well as the task they solved. We explore how we can apply Dreamer on a real robot without training it on the said robot, but on a simulated version instead. We start by modeling the system using high dimension exteroceptive inputs. Then, we integrate the robot's physical state into the loop. In this configuration, we use both high-dimension and low-dimension inputs to model the system. To improve the reusability of the models, we learn two transition functions, one for the dynamics, one for the rest of the environment. This allows changing the dynamic model while preserving environment knowledge and vice-versa. Finally, we leverage this model to efficiently learn how to reach different velocity goals.

[1] https://developer.nvidia.com/isaac-sim

5.2 Fundamentals on Reinforcement Learning

5.2.1 What is Reinforcement Learning

Reinforcement Learning (RL) can be seen as the machine learning solution to the control problem. In this paradigm, an agent is interacting with the world by taking actions and observing the response of that world (in a sequential manner). This agent receives (numerical) rewards (given by some oracle) that are local information about the quality of the control. The aim of this agent is to learn to take action-sequences that will maximize the sum of the rewards over this sequence. One major difference between RL and control, is what the reward does. In control, the error is used as active feedback to steer the system in the right direction. However, in RL, the reward is only used as a means to learn. Once the system has learned, it does not need to know what the reward is. This short section provides an introduction to the field of reinforcement learning. We invite the reader to refer to reference textbooks to learn more about RL [123, 124, 125, 126].

In RL, the agent is interacting with a system, often referred to as the environment or the world. If we wanted to train a robot to solve a manipulation task, for instance, this "world"

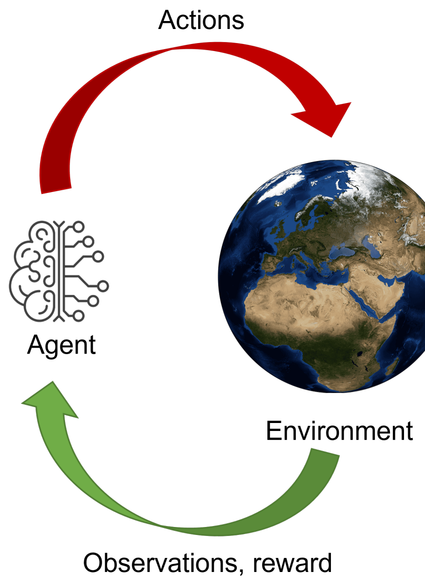


Figure 5.1: An agent interacting with its environment.

would encompass both the robot and everything that is related to the task it is trying to solve. An example of an agent interacting with a system can be seen in Figure 5.1.

For each time step, the system observes the world and acquires its state. Using this state, the agent can pick an action. This action is then applied to the world, which in turn returns a reward. This reward can be any numerical value. If the reward is predominately null, then the reward is called sparse. Sparse reward problems are often harder to learn. With the action sent to the world, the environment updates itself and this process can be repeated. The goal of deep RL is then to learn a function called "policy" that will maximize the sum of rewards over a trajectory.

Let us illustrate that with a robotic car that must follow a road. In this case, the state of the car could be an image provided by a forward-facing camera. The actions of the agent can be defined as the amount of throttle and steering sent to the car. The goal of our agent could be to stay on the right side of the road. The reward could consist of one point for staying on the right side of the road, minus one point if the car is not there, and minus ten points if the car has an accident. The agent would control the car at a given frequency, and for each step, it would receive a reward.

To formalize this type of problem mathematically, reinforcement learning leverages Markov Decision Process (MDP). So let us start by defining them.

5.2.2 Markov Decision Processes, Policy & Value Functions

A Markov Decision Process or MDP is a tuple S, A, P, r, γ where:

- S is the (finite) state space
- A is the (finite) action space
- $P \in \Delta_S^{S \times A}$ is the Markovian transition kernel. The term $P(s'|s,a)$ denotes the probability of transiting in state s' given that action a was chosen in state s. The transition kernel is Markovian because the probability to go to s' depends on the fact that action a was chosen in state s, but it does not depend on the path followed to reach this state s. This

means that to be Markov, s must contain all the information necessary to get s'. This assumption is at the core of everything presented here.

- $r \in \mathrm{I\!R}^{\mathrm{S}\times\mathrm{A}}$ is the reward function, it associates the reward $r(s,a)$ for taking action a in state s. The reward function is assumed to be uniformly bounded.
- $\gamma \in (0,1)$ is the discount factor. Depending on its value, it will give more or less importance to long-term rewards. The closer is γ to 1, the more importance we give to rewards far in time. Usually, this parameter is set to a value close to 1 such as 0.99.

So, the system is in state $s \in S$, the agent chooses an action $a \in A$ and gets the reward $r(s,a)$, then the system transits stochastically to a new state s', this new state is drawn from the conditional probability $P(.|s,a)$. Please note that if the agent has only a partial observation of its world, for example, if our car driver cannot differentiate between the left and right lane, the dynamics are no longer Markovian. This is known as Partially Observable Markov Decision Process (POMDP).

The function that rules the way an agent chooses its actions is called a policy, noted $\pi \in A^S$. In state s, an agent using policy π chooses the action $\pi(s)$. Hence, the problem RL tries to solve is finding the best policy. However, to do so, we need to be able to evaluate the quality of a policy. A common evaluator of a policy quality is the value function: $v^\pi \in \mathrm{I\!R}^{\mathrm{S}}$. This so-called value function provides the discounted cumulative sum of rewards obtained by following the policy π from the current state s.

$$v^\pi(s) = \mathrm{I\!E}[\sum_{t=0}^{\infty} \gamma^t r(S_t, \pi(S_t))|S_0 = s, S_{t+1} \sim P(.|S_t, \pi(S_t))] \tag{5.1}$$

Equation 5.1 gives the analytic expression of the value function for an agent in state s with a policy function π. The concept of this function is to estimate how well a policy is currently doing. If the policy, the reward, and the transition functions are known, then the value function can be computed analytically. In this calculation, one can see that the rewards acquired along the trajectory are discounted by γ. There are other means to evaluate a policy, but the value function is the most common one. Because a value function can evaluate the quality of a policy, it also allows comparing them Equation 5.2.

$$\pi^1 \geq \pi^2 \Longleftrightarrow \forall \in S, v_{\pi^1}(s) \geq v_{\pi^2}(s). \tag{5.2}$$

To solve an MDP one needs to find the optimal policy $\pi^* s.t. v^{\pi^*} \geq v^{\pi} \forall \pi \in A^S$. Hence, the optimal policy satisfies Equation 5.3.

$$\pi^* \in \underset{\pi \in A^S}{\arg\max}(v^{\pi}) \tag{5.3}$$

These equations can then be used to solve the MDP. However, we will not explore these derivations here. Our goal was to provide definitions of key terms in RL. We invite the curious reader to refer to [123]. Yet, before carrying on with the rest of the thesis, let us take the time to explain the differences between two branches of RL: model-free and model-based RL.

5.2.3 Model-Free & Model-Based Reinforcement Learning

Reinforcement Learning is usually separated into two fields: Model-Free, and Model-Based. The main difference between these two approaches lies in their training philosophy. In Model Free Reinforcement Learning (MBRL) the actor is learned by interacting with the system. This is the only thing that is learned along with the value function, as is required by most deep RL techniques. The core idea behind MBRL is that learning a policy function is sufficient, and thus, it rejects the concept of planning for most real-world scenarios.

In MBRL the first thing that is learned is a transition function. This transition function can serve multiple purposes. With it, one can create a MPC controller, a good example of that is the MPPI[25], or some of the work of Chelsea Finn[127]. The benefit of this method is that once the transition function is learned, one can apply any cost function and the controller should work. Here, we say should because the system may wander outside the training distribution of the transition function, which could cause the system to fail. Nonetheless, this is extremely practical as the actor does not have to be learned, and can easily be adapted. This is particularly desirable in robotic setups. Another way of using this known transition function can be to learn a policy from it. In Planet[128] or DREAMER[122, 129] they use a learned transition function to train a model-free actor. The advantage of this method is that it allows generating billions of so-called imagined trajectories, greatly increasing the sample efficiency of model-free methods. Finally, another way to use them is a combination of policy learning and planning. A great example of that can be seen in MuZero[130], where they learn both a policy and a transition function. Figure 5.2 shows the difference between a model-free agent and a model-based agent with policy learning. In this thesis, we focus on model-based reinforcement learning with policy learning.

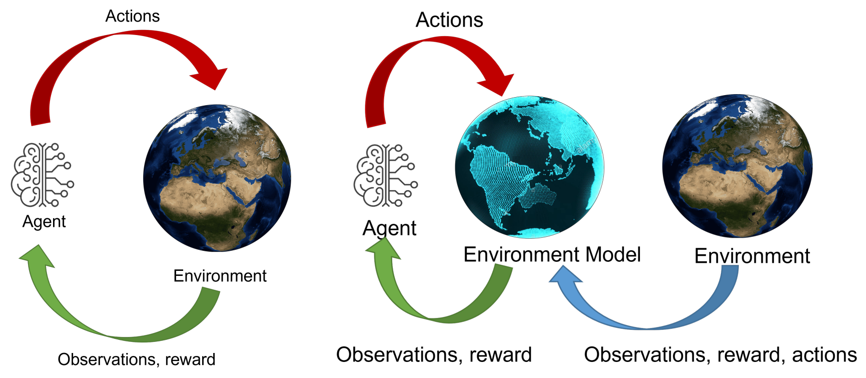


Figure 5.2: Left model free learning, right model based policy learning.

5.3 Fundamentals on Dreamer

In this thesis, we build on Dreamer, a strong MBRL agent. In the following section, we present how this agent works.

5.3.1 Recurrent State Space Model

Dreamer leverages a process called latent imagination to learn behaviors. Latent imagination consists in predicting a sequence of latent states of fixed length. In Dreamer, this is done by using a Recurrent State Space Model or RSSM. The RSSM is a VAE that acts as a Bayesian filter. As such, it has two functions: an observe function, which allows using observations to update its internal state; and an update function, which predicts the next internal state given an action. As their name suggests, RSSMs are recurrent models. They use a GRU to propagate and encode prior information. More precisely, the core mechanism behind RSSMs is that their state is made of two components. A deterministic part, which is the hidden state of the GRU, and a stochastic part, which is the variational part of the model. To understand their relationship, we need to see how the model works. Figure 5.3 shows the update function. On it, we can see that the first feedforward layer acts as a transition function, taking in the action and the past stochastic state that are then sent to the GRU. The GRU updates the deterministic state and acts as a regularizer preventing the stochastic state from changing too fast. The output of the GRU, is then sent through a feedforward layer, after which the reparametrization trick is applied to update the stochastic state.

As shown in Figure 5.4, to update the state of the RSSMs using observations, we start by applying the update function. Then the resulting deterministic state, and the new observations are concatenated and sent through a feed-forward layer, whose output gives the stochastic state after applying the reparametrization trick. A more detailed explanation of RSSM can be found in [128, Sec.3].

To learn its world model, Dreamer uses input reconstruction like a VAE would. A

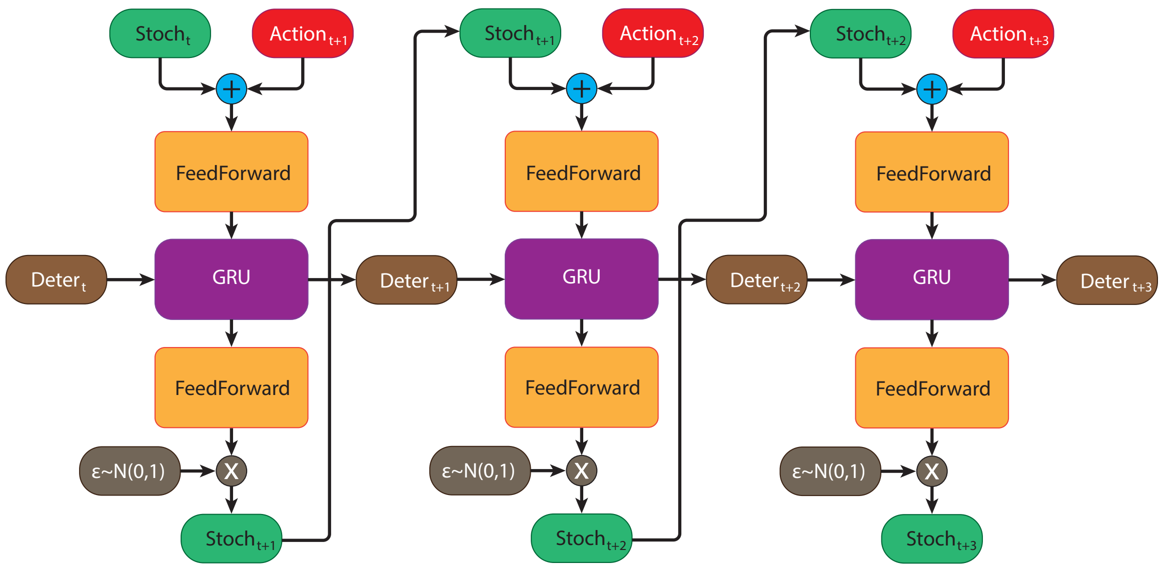


Figure 5.3: The update step of the RSSM. "stoch" is the stochastic state, while "deter" is the deterministic state. The blue "+" is a concatenation operation, the gray "x" is the reparametrization trick (recall Equation 2.13).

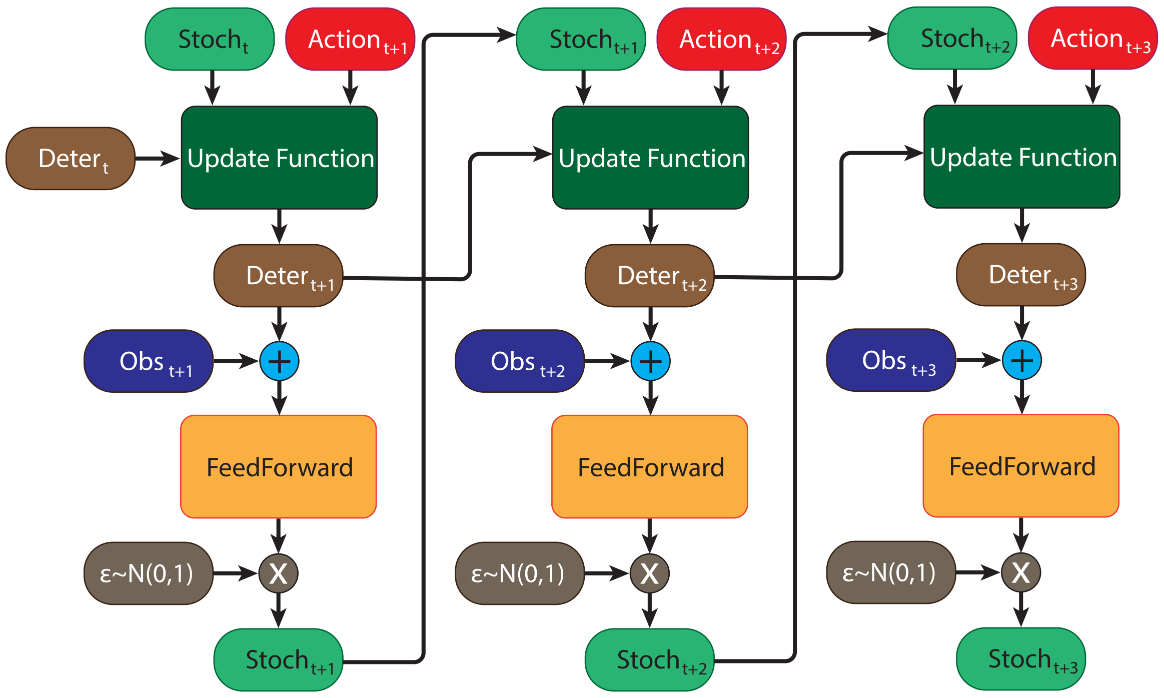


Figure 5.4: The observation step of the RSSM. "obs" is the observation of the system, "stoch" is the stochastic state, "deter" is the deterministic state. The blue "+" is a concatenation operation, the gray "x" is the reparametrization trick (recall Equation 2.13).

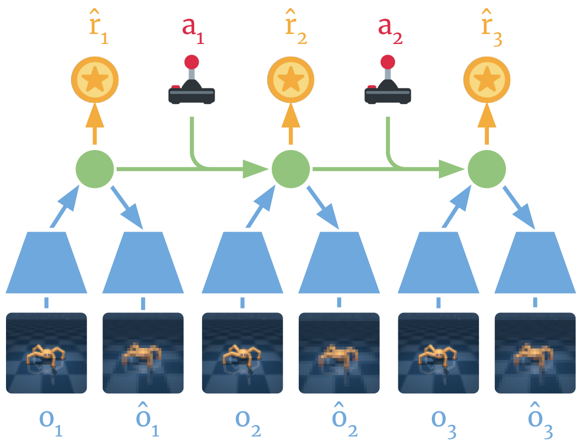


Figure 5.5: An illustration of the reconstruction process. Image from [122].

sequence of images is sent through the RSSM. Each image o_t is first encoded, then recurrently processed by the observation function. This gives us a sequence of states $s_t \iff (stoch_t, deter_t)$, which are then concatenated and sent to a decoder network. The output of the decoder is the reconstructed input $\hat{o}_t$. Similarly to a VAE, this network is trained by minimizing the ELBO loss. The reconstruction ensures that the agent learns a compact representation of the world that it can then use to learn how to play. The last thing that is missing here is a reward signal r_t. Thus, on top of the input reconstruction, the RSSM also learns to reconstruct the reward from its latent state. A depiction of this process can be seen in Figure 5.5, where $\hat{r}_t$ is the reconstructed reward, and a_t is the action sent at time t. Now that we know how an RSSM works, let us see how we can leverage them to learn behaviors.

5.3.2 Learning Behaviors

To learn behaviors, Dreamer uses an RSSM to generate novel trajectories. But why is an RSSM interesting in the first place? There are at least three big advantages to using this kind of model. First, an RSSM can be viewed as a simulator, a blazing fast one. Indeed, because the RSSM is a neural network, it can be "batchified". This means that a single GPU can now simulate thousands of trajectories concurrently. In Dreamer, they use this to generate 2500 trajectories per batch with a horizon of 45 points. An outstanding number of simulations ran in parallel in less than a second. Second, the stochastic state inside the RSSM can generate previously unseen situations. This should make the agent more robust to out-of-distribution events. Third, this model can be used to learn offline. One can deploy its robot, collect data, update the RSSM, and use them to update its policy. This is ideal for most robotic setups.

To learn a policy, Dreamer uses the well-known actor-critic framework. We recall that Dreamer uses the following quantities: o an observation, $r \in \mathbb{R}$ the reward, a an action. Using o_t, a_t and the RSSM one can get s_t. Using s_t one can also obtain $\hat{r}_t$, and $\hat{o}_t$. This dynamic model is formally defined by Equation 5.4.

$$\begin{aligned}
&\text{representation model:} && p(s_t \mid s_{t\text{-}1}, a_{t\text{-}1}, o_t) \\
&\text{transition model:} && q(s_t \mid s_{t\text{-}1}, a_{t\text{-}1}) \\
&\text{reward model:} && q(r_t \mid s_t) \\
&\text{decoder model:} && q(o_t \mid s_t).
\end{aligned} \tag{5.4}$$

Using this transition model, and a policy $a_\tau \sim q(a_\tau \mid s_\tau)$, we can generate a sequence of state, and a sequence of imagined rewards. These sequences can then be used to learn a value function who will approximate the lambda return of the policy from a given state, as in Equation 5.5.

$$v(s_t) \approx \mathbb{E}_{q(.|s_\tau)}\left(\sum_{\tau=t}^{t+H} \gamma^{\tau-t} r_\tau\right) \tag{5.5}$$

The goal of the agent is to maximize the value. The algorithm used to learn behaviors using Dreamer is given in algorithm 6. A depiction of the learning process is given in Figure 5.6. On this figure, we can see that using an observation we get a latent state, and then using the learned transition function and our policy we get a sequence of latent-states. Then for each of this state we compute the reward, and the value. Which we in turn use to refine the policy. More details about Dreamer can be found in [122, Sec.3].

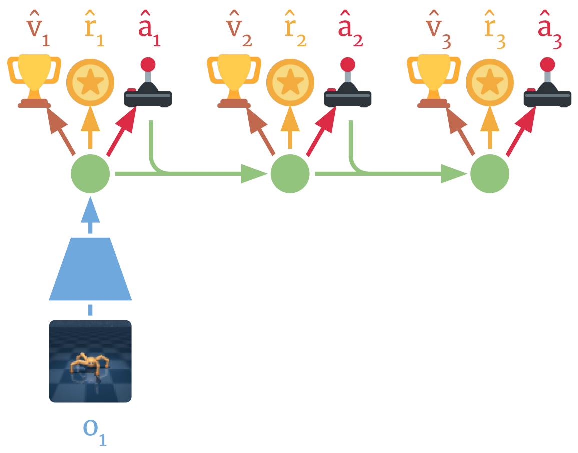


Figure 5.6: The learning process of Dreamer. Image from [122].

Algorithm 6: Dreamer (Algorithm from [122])

```
Initialize dataset $\mathcal{D}$ with $S$ random seed episodes.
Initialize neural network parameters $\theta, \phi, \psi$ randomly.
while not converged do
    for update step $c = 1..C$ do
        // Dynamics learning
        Draw $B$ data sequences $\{(a_t, o_t, r_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
        Compute model states $s_t \sim p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$.
        Update $\theta$ using representation learning.
        // Behavior learning
        Imagine trajectories $\{(s_\tau, a_\tau)\}_{\tau=t}^{t+H}$ from each $s_t$.
        Predict rewards $\mathrm{E}\Big(q_\theta(r_\tau \mid s_\tau)\Big)$ and values $v_\psi(s_\tau)$.
        Compute value estimates $\mathrm{V}_\lambda(s_\tau)$.
        Update $\phi \leftarrow \phi + \alpha\nabla_\phi \sum_{\tau=t}^{t+H} \mathrm{V}_\lambda(s_\tau)$.
        Update $\leftarrow -\alpha\nabla_\psi \sum_{\tau=t}^{t+H} \frac{1}{2}\left\| v_\psi(s_\tau) \mathrm{V}_\lambda(s_\tau)\right\|^2$.
    // Environment interaction
    $o_1 \leftarrow$ env.reset()
    for time step $t = 1..T$ do
        Compute $s_t \sim p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$ from history.
        Compute $a_t \sim q_\phi(a_t \mid s_t)$ with the action model.
        Add exploration noise to action.
        $r_t, o_{t+1} \leftarrow$ env.step($a_t$).
    Add experience to dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^{T}\}$.
```

Model components

Representation	$p_\theta(s_t \mid s_{t\text{-}1}, a_{t\text{-}1}, o_t)$
Transition	$q_\theta(s_t \mid s_{t\text{-}1}, a_{t\text{-}1})$
Reward	$q_\theta(r_t \mid s_t)$
Action	$q_\phi(a_t \mid s_t)$
Value	$v_\psi(s_t)$

Hyper parameters

Seed episodes	S
Collect interval	C
Batch size	B
Sequence length	L
Imagination horizon	H
Learning rate	α

5.4 Robotic Systems & Task

5.4.1 Robotic systems

Heron

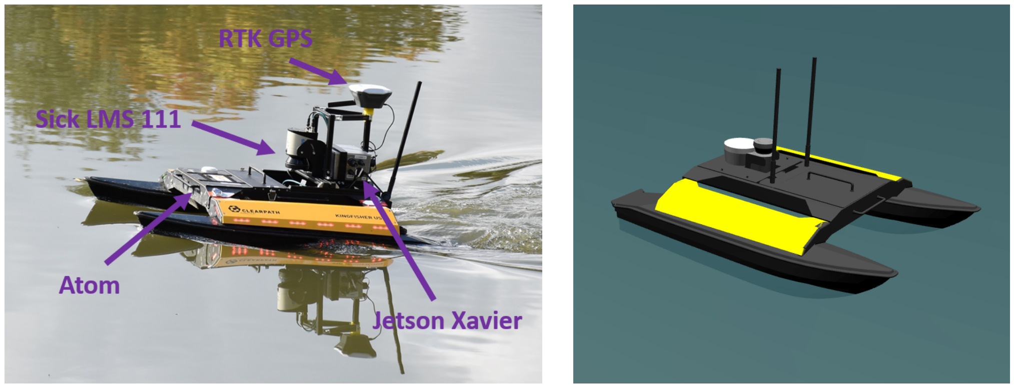


Figure 5.7: Left the real Kingfisher, right the simulated Heron.

The simulation and the real experiments were performed using similar systems. The real robot is a Clearpath Robotics Kingfisher, while the simulated robot is a Clearpath Robotics Heron (the Kingfisher's new version). In both simulated and real experiments, our USV is equipped with a SICK LMS111, a 20 meter-range and 270° field of view 2D-LiDAR running at 50Hz. To acquire the pose of our robot we use a REACH RS+, an RTK GPS from Emlid coupled to a pair of IMU that we use to get the heading of our robot. These informations are then fused inside an EKF that provides pose and velocity estimates. The weight distribution of the real and simulated systems are different: our real system was adapted to carry an NVIDIA Jetson Xavier and the RTK GPS. On the real system, an Intel Atom is used for low-level computations, while the Jetson Xavier at base clocks is used to compute and apply the agent policy. Both computers are running ROS with a single master.

The main challenge of this system is its inertia. With its current configuration, our Kingfisher's weight is around 35 kg, and it only has two 400 W motors (one left, one right). Hence, if an agent wants to take turns correctly, it needs to anticipate.

On the real system, the RL agent runs on Ubuntu 18.04 with ROS melodic, CUDA 10.0, python 2.7 and TensorFlow 2.1 [105]. The agent model has not been converted using Tensor RT or any other DNN compilers and can run easily at 12Hz on the Xavier.

Husky

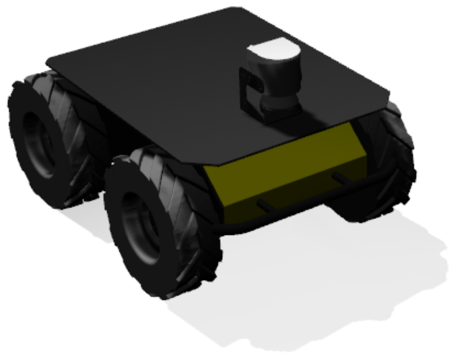
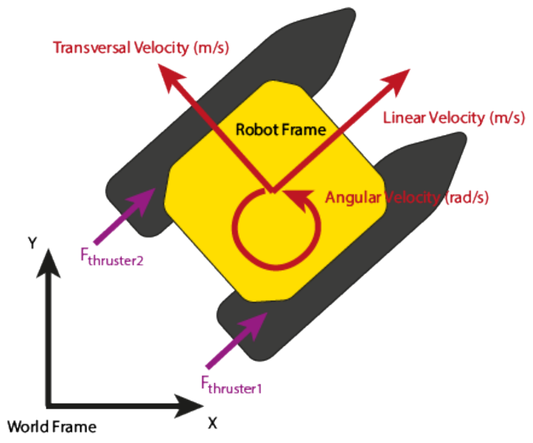


Figure 5.8: The simulated Husky.

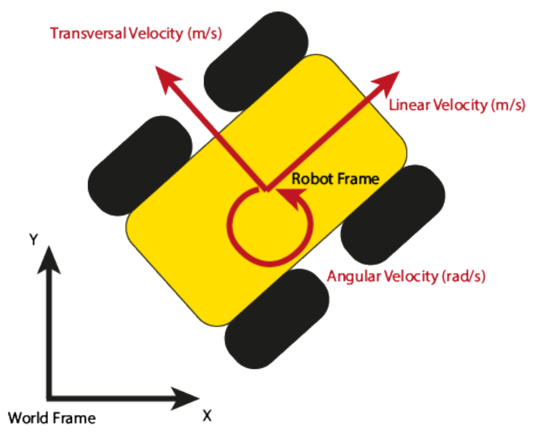
The second system we tested our method on is an Unmanned Ground Vehicle (UGV): a Clearpath Robotics Husky. This USV is a four driving wheels robot designed for outdoor applications. Because our real robot is not equipped with any GPU yet, we only tested it in simulation. The Husky in simulation is equipped with the same SICK LMS111 than the Heron.

State-Space

In our experiments, our robots are given access to three proprioceptive variables: their linear velocity, their transversal velocity, and their angular velocity. All these velocities are given in the robot frame, as illustrated in Figure 5.9. Regarding the action-space of the robots, the commands are values in the range $[-1,1]$. The Kingfisher has a two-dimensional action space, where each dimension controls the amount of current sent to the turbine in each of its floats. As for the Husky, the first component of its action space requests a linear velocity in m/s while the second one requests an angular velocity rad/s.



(a) Kingfisher/Heron



(b) Husky

Figure 5.9: State-space/Action-space illustration.

5.4.2 Task definition

This appendix gives more details on the task the reinforcement learning agent were trained to do. The task that we solved is a sensor-based navigation task. More precisely, we teached a robot to follow lake and river shores at a fixed distance and a fixed velocity. This is motivated by applications involving long-term monitoring of natural environments (for example, to assess that the shores are not moving over time). To follow the shore, our agent relies on a laser scanner and has access to its velocities (forward velocity, lateral velocity, angular velocity). Figure 5.10 shows the robot and the task it has to accomplish.

The reward of our agent is based on two independent metrics: Δ_d, the distance of the agent from the target shore distance d_t, and Δ_v, the difference between the agent's velocity v and the target linear velocity v_t. In all our experiments, we set $v_t = 1.0$ and $d_t = 10$. Furthermore, we penalize the agent if it gets too close to the shore or if it goes backward. The reward R is defined as follows:

$$R = 1.0 \times R_v + 2.5 \times R_d$$

with R_v and R_d given by:

$$R_d = \max(-20, 1 - 0.5(d_t - d)^2)$$

$$R_v = \begin{cases} \frac{1-|v_t - v|}{v_t}, & \text{if } v \geq 0.05 \\ -0.625, & \text{otherwise} \end{cases}$$

This reward is computed when training the model, and learned as part of it. The agent is trained on the learned reward (estimated solely from the laser-scan measurements, through the learned embedding).

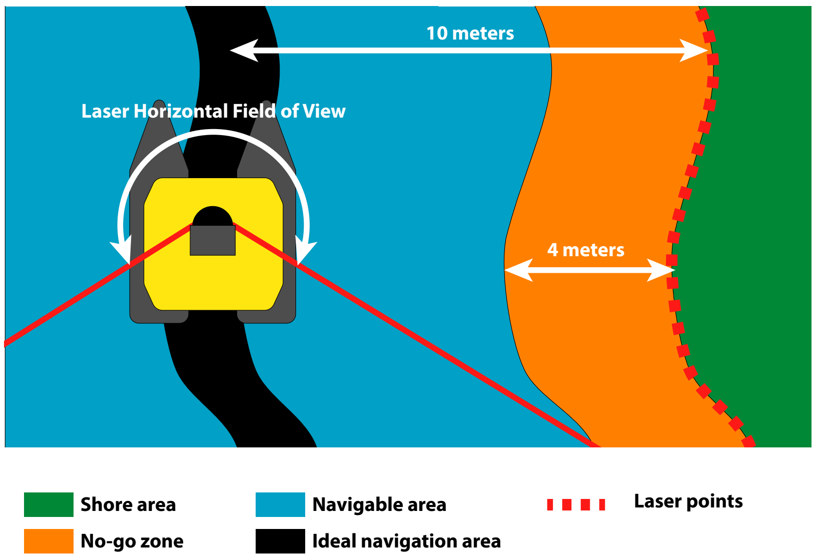


Figure 5.10: The USV and its shore-following task. (Colors are illustrative.)

5.4.3 Training

To train our robots, we used Gazebo: a robotic simulator, coupled to ROS, a robotic middleware. The USV dynamics simulation is handled by the `uuv-simulator` package. Each agent was trained in TensorFlow [105] for 1000 episodes of 500 steps amounting to 0.5 million interactions with the simulated environment. All our agents were trained purely in simulation, and were never fine-tuned on real-data. During the training, we applied a simple fixed-step curriculum learning. At the beginning of the training, the robot spawned close from the requested distance to the shore. As the training progressed, the agent spawned farther from that position with different headings. More details about the hyper-parameters, our training setup, the curriculum and others can be found bellow.

Curriculum

The curriculum learning of the robot is done by spawning the robot in increasingly difficult positions. To do so, we use a normal density probability function $f(x)$ with $x \in [0,1]$. The mean of the probability density function μ, starts at 0 and shifts towards 1 as the training progresses. The equation is given in Equation 5.6 and examples of the probability function can be seen in Figure 5.11, where $\sigma = 0.25$.

$$f(x) = e^{\frac{-(x-mu)^2}{2\sigma^2}} \tag{5.6}$$

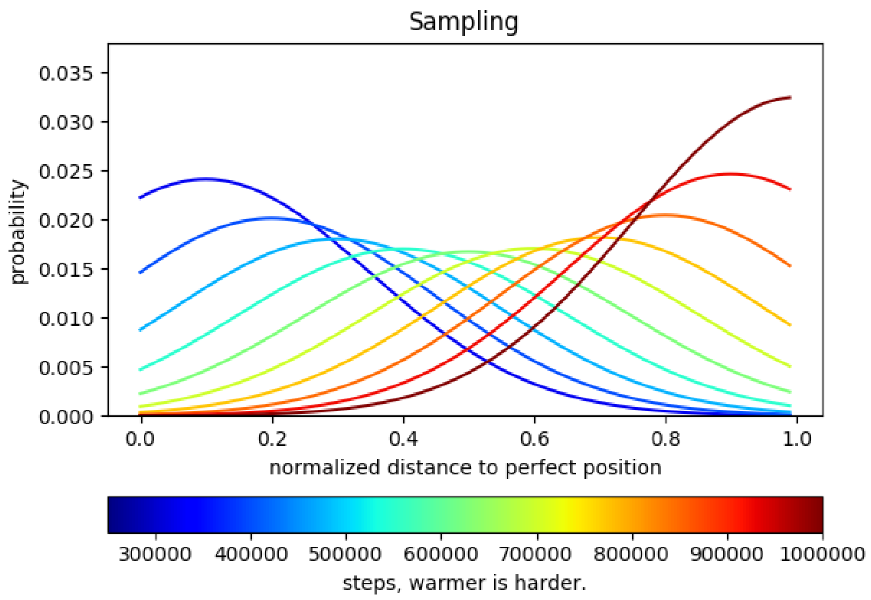


Figure 5.11: Probability density function depending on the number of steps. Each color corresponds to a step.

To spawn the robot, we pick a random position $(p_{\text{ideal\_x}}, p_{\text{ideal\_y}}$ on the ideal trajectory.

The heading $p_{\text{ideal\_}\theta}$ associated with this position is set to be tangent to this trajectory. Please note that the ideal distance is known beforehand, as the environment was generated using a cad model of known geometry. Using the probability density function f, we draw d and ε, numbers between $[0,1]$, and then apply the equations given in Equation 5.7 with, max_distance $= \pm 4$, max_angle $= \pm\frac{3\pi}{4}$.

$$\begin{aligned} p_{\text{spawn\_x}} &= p_{\text{ideal\_x}} + cos(p_{\text{ideal\_}\theta} + \pi/2) \times d \times \text{max\_distance} \\ p_{\text{spawn\_y}} &= p_{\text{ideal\_y}} + sin(p_{\text{ideal\_}\theta} + \pi/2) \times d \times \text{max\_distance} \\ p_{\text{spawn\_}\theta} &= p_{\text{ideal\_}\theta} + \text{arctan2}(-d, 5) \pm \varepsilon \times \text{max\_angle} \end{aligned} \tag{5.7}$$

Equation 5.7 moves the spawning position along the normal to the ideal trajectory. Ideally, the boat would spawn on the perfect trajectory, and its angular position would be computed using the tangent to that trajectory. However, as the distance to the ideal trajectory becomes large, this angle is no longer ideal. This is why we introduce the term arctan2$(-d,5)$, which sets the boat heading such that it reaches the ideal trajectory after five meters.

Examples of spawn positions generated by the curriculum can be seen in Figure 5.12

Our curriculum setup is designed such that the agent learns to drive around the lake in a single direction. To do so, we start by a warm-up phase of 250,000 steps, where the agent always spawns in the ideal position with the shore of the lake on its right. This creates a lot of samples on which the shore of the lake is on the right side of the agent. Naturally, this leads to the agent being more comfortable with the shore on its right side. We observed during field experiments that an agent trained this way would always fall back to this configuration. Counterintuitively, this behavior is highly desirable, as an agent trained this way will never do U-turns and retrace its steps. During the rest of the curriculum, between the step 250,000 and 1,000,000, we gradually increase the value of *mu*. During this phase, the boat faces increasingly difficult situations. And finally, for the remaining steps, we leave the curriculum

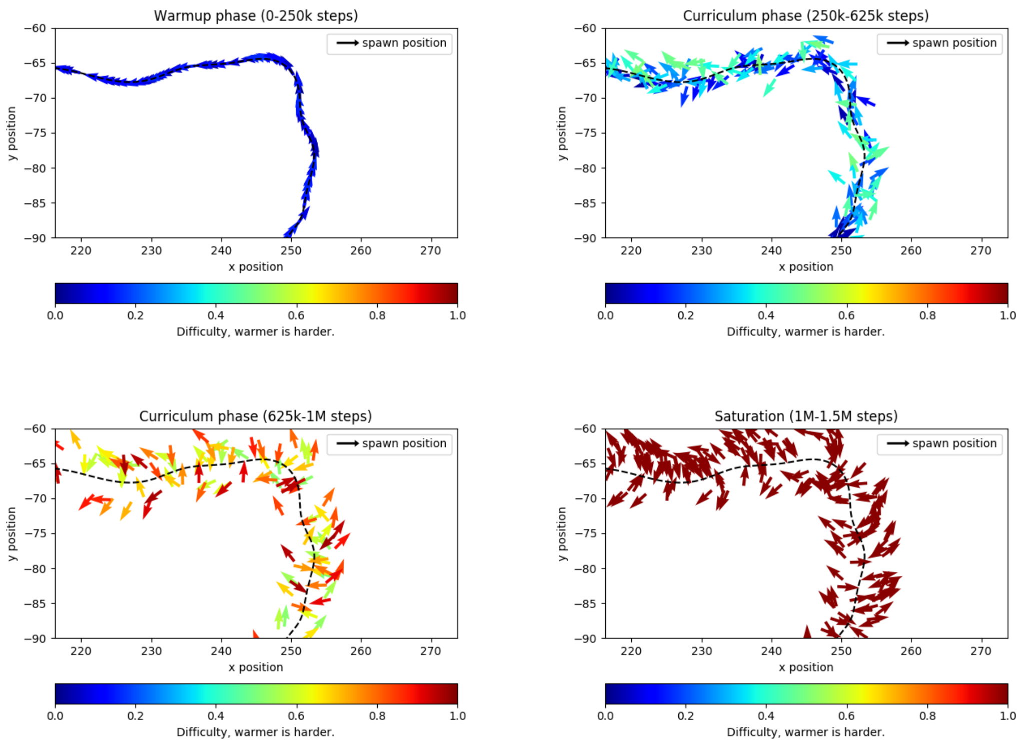


Figure 5.12: Possible spawn positions generated by the curriculum-based spawner. The training was cut into 4 phases for better readability. The warmer the color of the arrow, the higher the difficulty setting.

at the maximum difficulty setting.

ROS & Learning interactions

The way the RL agent interacts with ROS and gazebo can be described as follows:

Algorithm 7: Training Interaction

```
Start the Gazebo simulation $SIM$.
Start the ROS synchronization node $RS$.
Start the ROS agent node $RA$.
Start the training code $TR$.
while not converged do
    $TR$.request_new_episode($RS$).
    $RS$.reset($SIM$).
    $RS$.refresh_agent($RA$).
    $RA$.fetch_new_weight($TR$).
    $RA$.start_interaction().
    $RA$.done($RS$).
    $RS$.done($TR$).
    $TR$.fetch_last_episode($RA$).
    $TR$.train().
```

5.5 Model Based Reinforcement Learning for mobile robots

5.5.1 Motivation

Autonomous navigation in natural environments is critical in areas such as agriculture, inspection or environment monitoring. In these tasks, robots have to perform actions in complex and unstructured scenes, constantly changing. This requires them to have an in-depth understanding of their surroundings and own dynamics. To solve such tasks, a robot needs either a behavior model or a local planner. When using a local planner, the controller needs a dynamic model to follow the path and optionally an interaction model to know how changes in the environment impact the system.

Nowadays, most of the controllers use this kind of design [118]. However, as they rely on an accurate depiction of the robot dynamics, they require measuring the state of the system precisely. This means that the robot is carrying an expensive suite of sensors to acquire its state. Even though these approaches perform well, their cost, and the size of the sensors, make them hard to apply on small embedded systems. Furthermore, some sensors such as RTK-GPSs only work reliably in open-sky areas, making them impractical to use in occluded spaces, in forests, or in adverse meteorological conditions. Based on these observations, controllers based only on a small number of sensors such as cameras or laser scanners have gained interest over the recent years. The rise of these approaches was facilitated by the fast growth of the RL field [131], with its numerous benchmarks and the ever-increasing computational resources available on low-power devices. Building on this, an increasing amount of studies lean on RL-based controllers for their robot; among the popular methods, model-free techniques such as A3C [132] or SAC [133] have been used extensively [134, 135]. However, these approaches are rarely deployed in the field. This could be due to the cumbersomeness of training RL agents, and the issues inherent to field robotics.

In this section, we teach an under-actuated USV that exhibits thrust non-linearities to

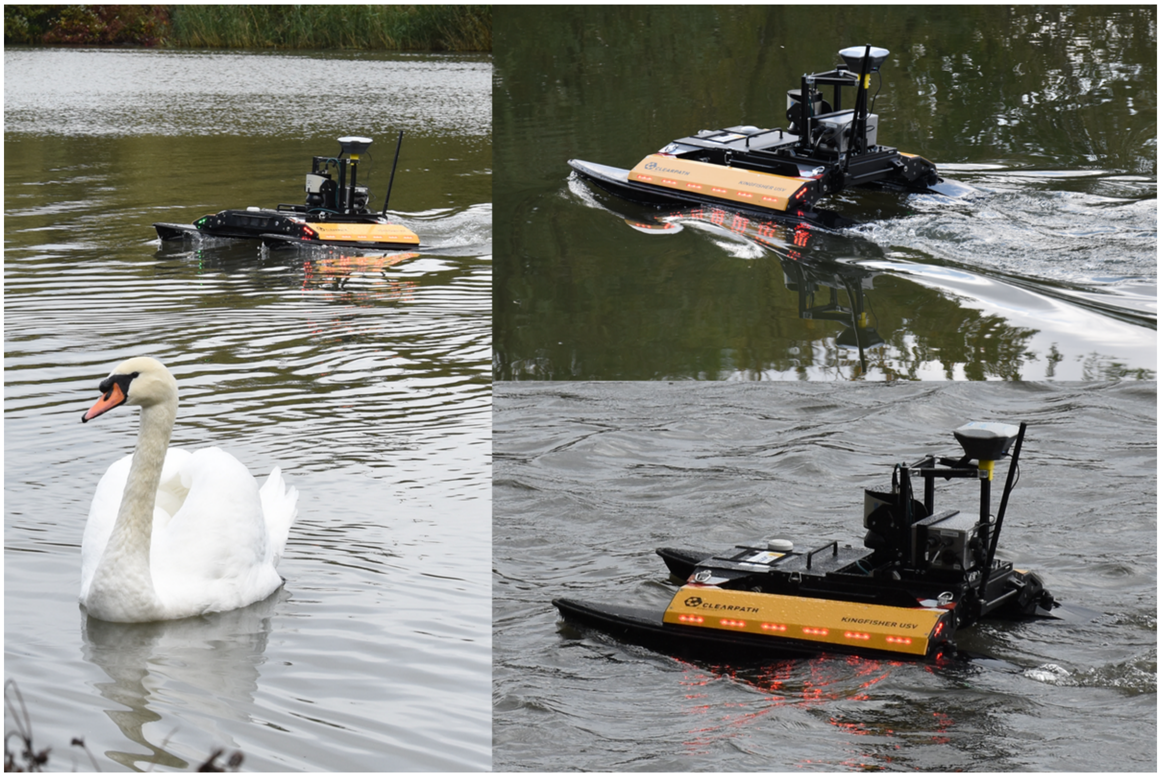


Figure 5.13: The Heron dashing around the Symphonie lake.

follow lake and river shores using Dreamer [122], a MBRL technique. MBRL is particularly interesting in robotics as it is more efficient than MBRL, reducing the amount of interaction with the environment needed to train the model. Another interesting point is that Dreamer, which uses latent imagination, natively learns the dynamics of the system to build its world model. To do so, our agent is exclusively provided with measurements from a laser scanner and must navigate at a fixed distance from the shoreline. Please notice, this task *is not* a path following task, as we rely on the environment to reactively find a path to follow. To the best of our knowledge, this is the first time Dreamer is being applied onto robots.

Yet, such RL agents come with disadvantages: they still require a lot of interactions to reach satisfying performances; as they learn by trial and error, they need to be constantly monitored to prevent accidents. A solution to this is either to use a very accurate simulator, or spend time training the model in the real environment, which is incredibly time-consuming and risky. To alleviate these issues, efforts have been made to create methods that allow to train agents in simulation and deploy them in real-life [136, 135]. Most of them rely either on Domain Randomization (Domain Randomization (DR)) [137] or Domain Adaptation (DA) [138]. While DA uses some real world data to adjust its parameters, DR does not. Here, although we apply DR, our study is more focused on what type of data-representation is used by the agent to minimize the use of these methods and deploy the agent without ever training it on the real system.

Our contribution are: **1)** training an RL agent solely in simulation to perform a shore-following task and applying it successfully on the real system in different weather conditions including moderate rain, and wind; **2)** evaluating different data representations and how they fare in a zero-shot deployment; **3)** reporting on lessons learned and giving some advice to practitioners.

5.5.2 Related Work

Designing algorithms to tackle reactive navigation problems is an active field of research in robotics since the 80s, as detailed in [139]. Many of these methods apply learned (RL) policies [140]. Actually, RL for policy search in robotics has been studied for decades [121, 141], and was applied to solve state-aware problems way before Deep RL came to be. However, those approaches often failed to process directly high dimensional inputs such as images. This is where Deep RL steps in, as NNs allow them to process complex inputs and act upon them.

Deep RL for Perception-Based Control

The emergence of deep learning has led to breakthroughs in many fields, and RL is no exception. The resulting field, Deep RL, is presented in [131]. Perception-based control happens to be the very task tackled by one of the most iconic Deep RL agent, namely DQN [142], an agent that achieves human-performance on a suite of classic video games. As of today, RL agents can be separated into two main families: model-free and model-based agents.

As their name suggests it, model-free agents do not try to learn transition or reward models. They are some of the most successful RL algorithms to date [143, 144, 133], including for robotics. For example, [135] uses A3C [132] to perform a visual servoing task where the agent follows a yellow marker in the image. Yet, one big disadvantage of MBRL is that it usually requires a huge amount of interactions.

On the other hand, model-based approaches are much more data-efficient but they often lagged behind the best model-free agents. As model-based agents learn the transition model and the reward model, they can interact with the environment virtually, mitigating the need for interaction with the real system. However, they then learn an imperfect model, which can deteriorate their performances. In [128, 122], the model is learned from simulation and used to train the policy. This allows the agents to train the policy for hundreds of billions of time-steps, without interacting with the real system.

Zero Shot Learning & Domain-Randomization

Even though data efficiency is getting more and more attention from the RL community, training an agent in the real world for millions of steps remains cumbersome. To avoid this, a set of methods grouped under the name DR have been developed. Robotics has seen an extensive use of DR applied to visual servoing [145, 146, 137]. For example, [145, 146] apply DR by making changes in the 3D scenes by randomly changing illumination or textures. This allows training object-detectors and collision detectors that work in the real world without ever being trained in it.

DR has also been used to change the dynamics of the system for learning robust policies [137, 147, 148]. DR does not require real-world data, but training a model under DR takes more time than what it would have if the true system was known. Finally, [135] trains an A3C agent to follow yellow semantic labels in simulation and shows that the agent transfers seamlessly between the simulation and the real world. This task is akin to a visual servoing task, where the robot is steered solely based on the label it has to follow.

In this section, we apply Dreamer [122], a MBRL agent, and use DR to learn a policy that is robust to the perturbation that can occur in the real world.

5.5.3 Method

This subsection details how the RL agent is trained, in simulation, to perform a shore-following task, both in simulation and on the lake. To learn more about how Dreamer works, please refer to section 5.3. To follow the shore, our agent relies only on a SICK laser scanner. Its goals are to remain at a fixed distance from the shore, to maintain a constant velocity, and to turn around the lake in a single direction. Figure 5.10 shows the robot and the task it has to accomplish.

Dreamer's modularity allows us to feed it with an observation and reconstruct something completely different. We study two types of observations and reconstructions.

Laser Measurements

The most naive way of using Dreamer with a 2D laser-scan is to change its encoders and decoders to process the laser-scans directly. To do so, we replaced the encoder/decoder based on 2D CNNs by 1D CNNs. In addition to this modification of Dreamer, we also project the laser-scans to a continuous representation. By default, laser-scans are a discontinuous representation of a 2D world: the laser points that are not reflected are set to zero or infinity. To alleviate this issue, we set all the zeros in the laser-scan to an arbitrary large value and compute its inverse. This creates a continuous representation where points close from the USV have a large value and points far from it have a small value. Finally, to make the laser more robust to the brutal changes that happen in a natural environment, we use an operation that would be similar to a *min_pool* with a stride of 2. In the end, we remove 7 points on each side such that the final laser-scan has a shape of 1x256x1 making it easy to manipulate using convolution and pooling operations.

Laser Projection

While using the laser-scan directly can sound appealing, the transfer between simulation and real environment is difficult. Indeed, in our natural environment, the movement of the leaves, or small branches, and the partial reflection of laser points due to the semi-transparent character of the vegetation make real and simulated laser-scans very different. Thus, we choose to transform the laser-scans into a robust representation. To do so, we create an image representing a local map with a width of 20 meters and a height of 12 meters, the map has a resolution of 1 pixel per 10cm. The origin of the map, the position of the robot, is set 2 meters from the top and 10 meters from the left and its background is blue. Then we convert laser-scans into points in the map and trace 4 meters red circles on top of each point. Finally, we trace a black one-meter wide curve 10 meters from the shore to represent the track and resize the map to a 64x64 image. This representation is robust to changes and behave similarly in simulation and in the real world, as shown later.

5.5.4 Environment & Domain Randomization

In the real world, the system's dynamics can be impacted by at least two environmental factors: wind and water current. To compensate for those, we add water current to the simulation. More precisely, at the beginning of each episode we draw a water current velocity from a uniform distribution in the range [0, 0.4] m/s and set its direction randomly. Additionally, as our real system has gone under significant modifications its characteristics are different from the one used in simulation. To account for that and the approximations of the simulator, we also change the water density at the beginning of every episode. Before the agent starts playing we draw a density from a uniform distribution in the range [1000, 2500] kg/m^3. Finally, we also make the agent spawn close and far from the shore to make sure it learns how to recover from such events.

5.5.5 Experiments

The system used in these experiments is the clearpath kingfisher. More about this system in subsubsection 5.4.1.

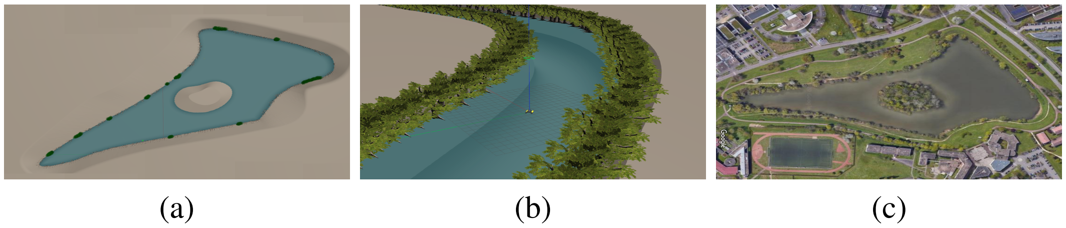


Figure 5.14: Simulation and Real environments: (a) and (c) are the simulated and real lake (Lac Symphonie, 57000 Metz, France, Google Maps, 2020), (b) is a simulated channel

Simulation Environments

We trained our models in simulation using Gazebo[2], a simulator that supports advanced physics modeling and allows to create custom robots. The simulation of the USV is done using the ROS packages heron-simulator[3] (simulated version of Clearpath's Heron) and uuv-simulator[4] (water buoyancy simulation and thrust non-linearities mimicking the real USV behavior). To train our agent, we have replicated our real test environment: a lake around which we have planted approximately 1000 trees to create a feature rich and intricate shoreline. We also test our agent in other simulated environments to ensure that it has learned something that can be applied in a wide variety of situations, as can be seen in Figure 5.14 (a, b).

Real Environment

Our real test environment is a small artificial lake with approximately 1.7 km of shoreline. As can be seen in Figure 5.14 (c) it presents two main complexities, a hairpin bend on the far right of the lake, and a small island that creates a narrow passage. Additionally, we deployed

[2]gazebosim.org
[3]github.com/heron/heron_simulator
[4]github.com/uuvsimulator/uuv_simulator

our system in various weather conditions, such as light rain, and wind with average speed of 40 kmph, and gust of wind reaching up to 60 kmph. Furthermore, our lake also features a very active and curious wild-life including swans and ducks that act as moving obstacles.

Training parameters

All our agents are trained for 2000 episodes of 500 steps, the agent plays 12 times per second. This makes for a total of 1.000.000 simulation steps, or about 24 hours of play time. Although we could have played at a higher frequency, we noticed no improvement from 50Hz to 12Hz: we decided to stay at 12Hz. All training parameters are provided on our git repository.

Baseline

Initially, our USV was controlled using a PID controller and a linear model of the system. This controller was not only hand-engineered for the environment but failed to reach velocities higher than 0.4 m/s. Hence, we compare the RL agent to a strong MPPI controller [115], using a cost function similar to the reward of our agent. The MPPI is a monte-carlo-based controller that requires a model of the system. Here this model is learned by a NN using the real USV, but unlike our method which is relying solely on a 2D laser scanner, the MPPI uses the on-board RTK GPS and IMU to infer its state: linear velocity, and angular velocity. The Neural-Network used to model the system was pre-trained in simulation on 1.000.000 samples and fine-tuned on the real-system with 100.000 samples. Both trainings were done using the PER scheme and a grid-search was used to find the optimal parameters. We collected samples on the real USV as the MPPI would not converge on the real system when using the model acquired in simulation.

Table 5.1: Results (simulated and real system). Target velocity 1m/s, target distance from the shore 10meters.

	Simulated system		Real system				
Representation	Ideal P2P	Ideal L2P	Ideal P2P	Ideal L2P	Extra Drag P2P	Wind + Rain P2P	MPPI
Collisions	0.0	**0.0**	**0.0**	0.5	0.0	0.0	0.0
Interventions	0.0	**0.0**	**0.0**	1.2	0.0	0.0	0.7
Velocity (m/s)	1.1 ± 0.22	$\mathbf{1.04 \pm 0.18}$	$\mathbf{0.99 \pm 0.17}$	0.98 ± 0.18	0.73 ± 0.12	0.97 ± 0.18	0.83 ± 0.19
Distance (m)	9.8 ± 0.77	$\mathbf{9.9 \pm 0.23}$	$\mathbf{9.9 \pm 1.67}$	11.1 ± 2.9	10.32 ± 1.51	9.9 ± 1.77	9.5 ± 2.5

Evaluation

To evaluate the performances of the different agents and the different representations, we record the following metrics: **1)** the number of collisions the agent has with the environment per 10 minutes; **2)** the number of times the agent fails to perform its shore-following task per 10 minutes; **3)** the USV average linear speed $\pm$ its standard deviation; **4)** the USV average distance from the shore $\pm$ its standard deviation.

To further assess the robustness of the system, we also study how the agent reacts when facing a hard situation, such as the hairpin bend, or when the agent starts too close or far from the shore. Furthermore, we deploy the agent in challenging weather conditions and also assess its behavior when its dynamic change.

5.5.6 Result

Data Representation

We evaluate how the different data representations compare, both in simulation, and in real life. In these experiments, we trained 2 types of agents: **"Projection To Projection": P2P** (reconstructs laser projections from observed laser projections) and **"Laser To Projection": L2P** (reconstructs laser projections from observed laser measurements). An **L2L** agent was also experimented, but not included for the sake of page limits. Its results are less good than the other agents', this could be tied to the difficulties of generating the values of a laser-scan in unstructured environment where most of the points are not reflected.

As can be seen in Table 5.1, the *L2P* is the best performing approach in simulation. The laser-scans offer a high resolution input and provide farther information than the projection of the laser-scans it reconstructs. This allows the world model of the agent to be more accurate both instantaneously and over multiple time steps. However, the convergence of this model is more delicate than the *P2P* one, as the decoder of the *L2P* agent must learn a more complex task. To alleviate this issue, we found it ideal to first train the *P2P* agent and then use it to collect around 100 episodes for the *L2P* agent. This allows to optimize the world model of the *L2P* agent before letting it explore its environment. It has two benefits: it prevents an explosion of the value loss, and it reduces the total time required to train the agent.

On the real lake, the results are different: *P2P* performs best. Overall, the *P2P* agents have always performed their task perfectly: with over 8.2 kilometers traveled during our experiments with this agent, the USV never required human assistance to fulfill its task. Regarding how well it kept its distance from the shore, with a mean of 9.9 meters and a standard deviation of 1.67 meters, the results are not drastically different from those in simulation. Concerning *L2P*, the agent performs globally well, but its performances are far from what it reached in simulation. This is due to the significant difference between real and simulated laser-scans. Nonetheless, both the *P2P* and the *L2P* agents are capable of maintaining a velocity of $0.99 m/s$ even if they do not directly measure it. This demonstrates that our agents, despite that they only have access to the laser-scans, are capable of estimating both the velocity of the system and their distance from the shore.

Finally, we observe that compared to other previously tried approaches for this task: a PID controller with a linear model, and the MPPI; the RL is the only one that is capable of following the shore at high speed. Furthermore, the RL is capable of following the shore in a single direction even-though its reward does not explicitly enforce it.

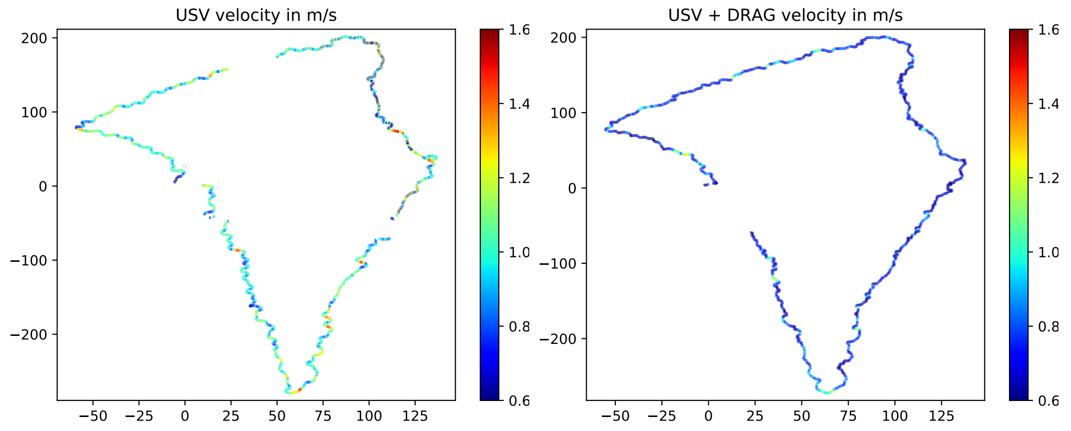


Figure 5.15: The velocity of the USV around our lake. Left the USV without the swimming board, right the USV with the swimming board.

Robustness

To assess the robustness of our agent, we attach to our USV a swimming board with a fin perpendicular to the forward direction. This device greatly increases the drag of the robot. We want to see how this contraption affects the behavior of the agent and if it is still capable of fulfilling its task. Because the *P2P* agent was the most reliable in the previous field tests, we only show its results here, on 2.2 kilometers traveled during our experiments. As can be seen in Figure 5.15 the RL agent is capable of fulfilling its task despite the radical change in the system dynamics. However, its velocity is seriously impacted, as the average speed of the USV with the extra drag is much lower.

If we look at the commands sent to the system, we can see that without the extra drag, the agent uses 90% of the available power to steer the boat, whereas with the drag it uses 95% of the power. This shows that it is trying to compensate for its lower speed, but is probably limited by the power of the engines. While the agent does not reach the target velocity, it can be seen from Table 5.1 that its behavior remains very close to the system without the swimming board. Most notably, the agent maintained a similar distance to the shore, and never required assistance. In addition to the swimming board test, we also tested our approach under a strong wind and moderate rain. As can be seen in Table 5.1, the metrics of the model are not that different from the one acquired under ideal weather conditions. Furthermore, all the RL agents were capable to swiftly avoid the swans, and kept following the shore.

Domain Randomization

When we first trained the agents with DR their performances decreased. Initially, we had a very aggressive random spawn policy that would make the USV spawn close from the shore. The idea behind it is that if the boat spawns close from the shore, then the agent learns how to quickly reach the correct distance and will know how to deal with complex situations. The main problem with this method is that with our settings, the episodes are not long enough, and the agents are not spending enough time navigating at the optimal distance from the shore, which decreases their performances. To make matter worse, agents trained without DR are as good as agents trained with DR to get out of hard situations. Which led us to reconsider the initial aggressive spawning strategy. Once this done, DR did provide some improvements on the L2P agent. Without DR, the agent could lose track of the shore after large bushes, requiring human intervention to get back close to the shore. With DR, we did not notice this behavior any longer, which indicates that DR improved the robustness of the system. All in all, DR is delicate to tune, but with software like gazebo some settings can be changed very easily through ROS services to mimic how the behavior of the system can change: for instance changing the water density is an easy way to simulate battery voltage drop or weight changes.

Baseline Comparison

In Table 5.1 we also compare the different agents to MPPI, a state-aware model predictive controller. Across all metrics, the *P2P* agent performed better than the MPPI. This is interesting because the MPPI is not only more computationally expensive, but was also trained on the real system. Before being able to perform its task, we train a Multi-layer-perceptron to learn the dynamics of the system using its entire state (linear and angular velocities). While it is surely possible to tune the MPPI to improve its results, these same results show that our model can be trained in simulation, dropped on the lake and perform better than an algorithm that was tuned, trained, and optimized on the real task. Furthermore, our agent is capable of recovering from cases where its heading is facing the opposite direction we want it to navigate, whereas the MPPI does not. It means that during a survey the MPPI can, from time to time, change its forward direction, following the shore on its right side, although it should follow it on its left side, or the other way around. Our agent does not.

5.5.7 Conclusion

In this section, we taught a robot to solve a navigation task only based on a 2D laser scanner in an unstructured and natural environment by training it only in simulation. We demonstrated that the behavior learned by the robot is robust to changes in the dynamics, but also in the environment. Our agents performed well when facing moving obstacles, swans, or when we changed drastically their dynamics by increasing their drag. Moreover, despite that they are solely relying on a laser scanner the agents are capable of estimating their speed and their distance from the shore, making this solution applicable in many other environments.

5.6 Dynamic model separation for modular MBRL

5.6.1 Motivation

Early forms of controllers, such as classic optimal control or dynamic programming, make use of a supposedly known model of the environment. On the learning side, RL can also learn a model and use it to train an agent.

It is only during the last decade that MBRL algorithms capable of controlling systems using high dimension inputs, such as raw images, were developed. These advances allowed their use on robots for solving complex tasks, on which traditional methods failed [149, 150]. In robotics, it is of paramount importance to learn quickly, as acquiring samples is not only ludicrously time-consuming, but can also be dangerous for both the robot and the operator overseeing the agent. Hence, due to their high sample efficiency and their potential resort to offline learning [151, 152], world models make for a compelling choice in robotics applications.

Yet, because most of the agents are meant to be used on classic benchmarks such as Atari [153], OpenAI Gym [154] or the DeepMind control suite [155], they process either images or low proprioceptive variables (for example, position or speed), but rarely both. In robotics, most of the works that used MBRL for solving high-dimensional tasks discard the proprioceptive variables [149, 150, 156], even though they could have been acquired fairly easily. Other works, such as [157, 127], make use of both, but embed them as a single model. Unfortunately, with neural networks there is no way of accessing the dynamics of the system directly integrated inside the environment because they make for a single black-box model. All in all, this means that the world model learned by an agent will be unique to this system.

By contrast, we argue that a world-model should be made of two components: an environment, which encompasses everything that the agent can sense using laser-scanners or cameras, and the dynamics, i.e. the set of physical equations and variables that rule the movement of the mobile robot in the world. To do so, within the paradigm of MDPs, we decompose the total state of the system in two parts, one related to the environment and one to the dynamics of the robot, $S^{\text{tot}} = (S_t^{\text{env}}, S_t^{\text{dyn}})$, and we assume that the transition from state to state can be decomposed as a change of the dynamics in response to the applied action, and a change of the environment in response to the new dynamics, $P(S_{t+1}^{\text{tot}} | S_t^{\text{tot}}, a_t) = P(S_{t+1}^{\text{dyn}} | S_t^{\text{dyn}}, a_t) P(S_{t+1}^{\text{env}} | S_t^{\text{env}}, S_t^{\text{dyn}})$.

A concrete example of why such representations are powerful is to think of car manufacturers. All manufacturers make different cars with different physical properties, yet those cars all share the same environment: the road. Hence, a method like ours could allow the cars to share their environment while having their own dynamics, speeding up the training process and potentially allowing the distributed training of heterogeneous agents.

Furthermore, it makes sense in robotics applications to use proprioceptive information, as most real robotic tasks involve low level physical constraints coupled to high level perception constraints. In the end, MBRL is a very promising solution to bring more autonomy to real-robots, yet we think it would be better suited to mobile robotics application if it was to natively account for the robots' dynamic.

In this section, we modify one of the strongest latent-model agent, Dreamer [122], and integrate the dynamics as an independent part of the world model. As such, our world model will feature two states: an environment state that uses low dimensions proprioceptive variables to transition from one state to another ($P(S_{t+1}^{\text{env}} | S_t^{\text{env}}, S_{t+1}^{\text{dyn}})$), and a physical state, that uses the action sent by the agent to transition from one state to another ($P(S_{t+1}^{\text{dyn}} | S_t^{\text{dyn}}, a_t)$). While it might seem fairly similar to the original world model with an observation of the proprioceptive input in addition to the images, it is not. Decoupling the dynamics from the environment allows for interesting manipulations within the imagination process used to learn the actor. Because the dynamics is no longer attached to the environment, we can now swap the dynamic model of a robot for the dynamic model of another robot, as long as their proprioceptive states are of the same size. Moreover, our method ensures that the imagination roll-out have environment-states that are consistent with the physical-states.

Eventually, we can use an analytic approximation of the robot dynamic model to alleviate the need to rely on a learned model.

Our contributions can be summarized as follows: **(1)** we propose a latent-state representation that decouples the environment from the dynamics to better fit mobile robots; **(2)** we test this method in simulation and on a real robot and show that it robustifies the agents compared to the baseline; and **(3)** we demonstrate that this method can also be used to transfer an environment from one robot to another in a zero-shot setup.

5.6.2 Related Work

Model-Based Reinforcement Learning.

The earliest form ofMBRL is probably optimal control. In optimal control, the model of the system is typically known, and an optimal controller is computed using methods such as LQR. Until recently MBRL algorithms such as Pilco [158], among others [121], were designed to process and model low dimensional inputs. These methods were applied to robotic applications [141, 159] with impressive results, but they were limited and struggled to control systems using high dimension sensory inputs such as images. However, with the advent of deep-neural-network, MBRL was applied successfully to high dimensions inputs, as shown in the comprehensive survey of [160]. As of today, MBRL performances are nearing if not exceeding the performances of model-free agents [122, 129]. The main advantage of MBRL relies on its higher sample efficiency when compared to the model-free agents. Methods based on Variational VAE [21], such as latent state methods [128, 122, 129], can learn robust policies thanks to the VAE stochastic states. Unfortunately, as these methods started processing higher dimension inputs, they also started to disregard proprioceptive information as it made solving the task easier.

Dreaming applied to real-world.

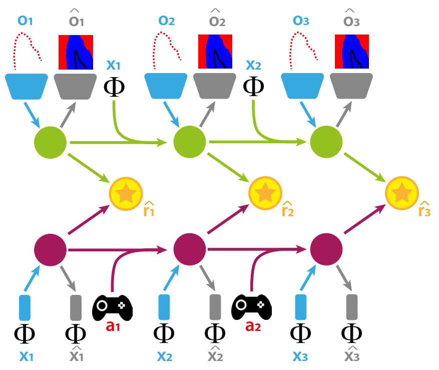
Latent-state methods have shown impressive results when applied on real robots. [156] applies Dreamer on a 1/10th racecar with a LIDAR and compares it to other model-free methods in simulation. They show that, on their task, Dreamer performs better than other MBRL algorithms, such as DDPG [161] or PPO [162]. They also demonstrate that Dreamer transfers well from simulation to real robots, in well controlled environments. [150] also applies Dreamer on an USV with a LIDAR and trains it in simulation on a shore following task. They then deploy it on the real robot, without retraining it, and evaluate the robustness of the approach by both changing the dynamics of the real systems and evaluating it in different weather conditions. Both of these works also use Dreamer as their backbone, but unlike ours, they completely disregard the physical state of the system. [149] shows an application of latent models to a real world system using cameras. In [157], they rely on latent state model and integrate both the physical state of the system and high level distance sensors to move their robot around. However, they still consider the systems dynamics to be part of the whole model. A work that is closer to ours is [163], in which they learn from images the Lagrangian dynamic of a system, and then use it to reconstruct the observed images. In its current state, [163] is not applicable to real robots: the examples are simplistic, and make strong approximations about the dynamic model. [164] also shares similarities with the work presented here: they use Graph Neural Networks to perform system identification as in [165], and then apply a MPC controller and RL on the predicted physical states. The main difference with our work, is that we don't only have a dynamic module but also a perception module. In our method, the outputs of the dynamic module are used to predict the next perception state, and both of their latent states are then used by the actor to decide on which action to apply next.

5.6.3 Method

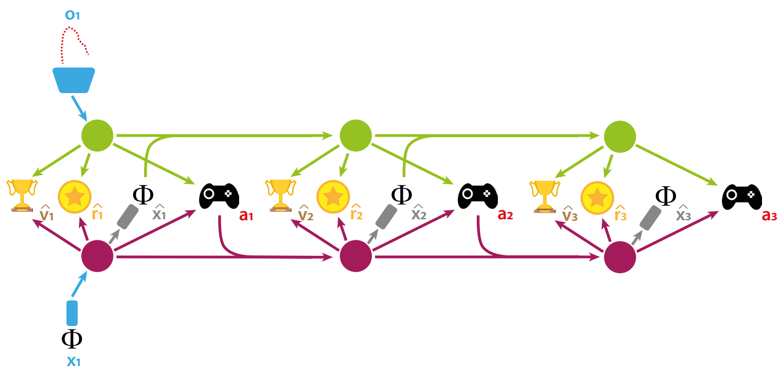
We now show how our model differs from the original implementation of Dreamer[122], which we started from. As in Dreamer, we solve a Partially Observable Markov Decision Process (POMDP), with discrete time steps t, for which we have continuous actions a_t, high dimension sensory inputs o_t, low dimension proprioceptive inputs x_t, and rewards r_t generated by the environment.

Similarly to Dreamer, our version is articulated around three main concepts: the learning of the world model from previous experiences (Figure 5.16a), the learning of the behavior inside the imagination process (Figure 5.16b), and finally the application of the policy using observations from the environment to collect new samples (Figure 5.16c).

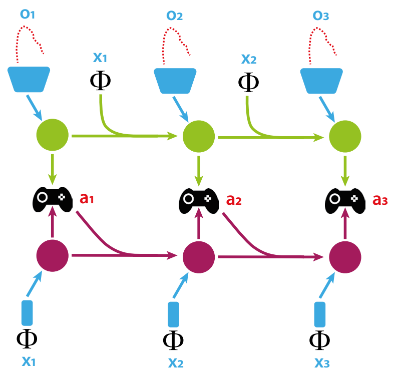
Unlike the original Dreamer, our agent does not rely on a single latent state. Instead, we consider that this single state can be decomposed into two states $S_t^{\text{tot}} = (S_t^{\text{env}}, S_t^{\text{dyn}})$, where $P(S_{t+1}^{\text{tot}}|S_t^{\text{tot}}, a_t) = P(S_{t+1}^{\text{dyn}}|S_t^{\text{dyn}}, a_t)P(S_{t+1}^{\text{env}}|S_t^{\text{env}}, S_t^{\text{dyn}})$. In practice, we consider that we can learn a function $q_\phi(x_t|S_t^{\text{dyn}})$, this means that we can further decompose $P(S_{t+1}^{\text{tot}}|S_t^{\text{tot}}, a_t) = P(S_{t+1}^{\text{dyn}}|S_t^{\text{dyn}}, a_t)P(S_{t+1}^{\text{env}}|S_t^{\text{env}}, x_t)$ as we will be able to estimate x_t from S_t^{dyn}within the imagination. In [122, 129], the latent dynamic model is composed of three modules: a representation module $p_\eta(s_t \mid s_{t\text{-}1}, a_{t\text{-}1}, o_t)$, a transition module $q_\eta(s_t \mid s_{t\text{-}1}, a_{t\text{-}1})$, and a reward module $q_\eta(r_t \mid s_t)$. With this setup, both the representation and transition modules embed the dynamics of the system and the understanding of the environment. This limits the capacities of the world model. Hence, we propose to separate the dynamics of the system from the environment. To do so, we change the latent dynamic model of Dreamer to include two states: S_t^{env} the environmental state, and S_t^{dyn} the physical state. We then add two extra modules and slightly modify the inputs of the original latent dynamic model. The new latent



(a) Learn dynamics



(b) Learn behavior in imagination



(c) Play

Figure 5.16: The three concepts of the proposed extension of Dreamer: (a) Using its replay buffer, the agent learns to encode environment high-dimension sensory observation and proprioceptive input into a compact latent environment state (🟢). Additionally, using a similar process, the agent learns to encode proprioceptive inputs and actions into a latent physical state (●). Both of these states are then used to estimate the reward (⭐). (b) Using both learned latent spaces, the agent predicts state values (🏆) and actions (🎮) as in the original Dreamer. (c) The agent observes its environment and predicts the best action. More details about the algorithms used here can be found in algorithm 8.

dynamic model is

$$\begin{aligned}
&\text{Dynamics representation module:} && p_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1}, x_t) \\
&\text{Dynamics transition module:} && q_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1}) \\
&\text{Environment representation module:} && p_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, x_{t\text{-}1}, o_t) \\
&\text{Environment transition module:} && q_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, x_{t\text{-}1}) \\
&\text{Reward module:} && q_\eta(r_t \mid S_t^{env}, S_t^{dyn}).
\end{aligned} \tag{5.8}$$

Learning dynamics & Reconstructing Observations

To learn both the dynamics and the environment, we rely on Recurrent State Space Models (RSSM) [128]. RSSMs can be seen as non-linear Bayesian filter [166], with an observation step and a prediction step. During the observation step, the RSSM is given its previous state and an observation, and outputs a compact latent state. During the prediction step, the RSSM is given its previous state and command, and outputs a compact latent state. This prediction step of the RSSM is the transition function, and the combination of a prediction and an observation is the representation module.

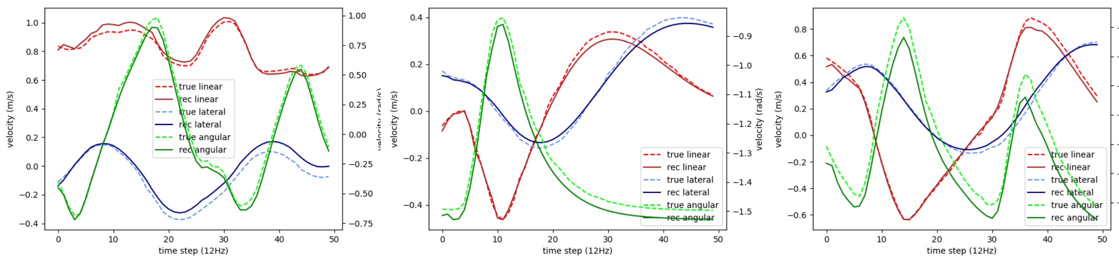


Figure 5.17: RSSM reconstruction of proprioceptive variables, 5 observations followed by 45 predictions.

Algorithm 8: Physics Driven Dreamer

```
Fill dataset $\mathcal{D}$ with $N$ random actions episodes.
Initialize neural network parameters $\theta, \eta, \psi$ randomly.
if load dynamics then
    Load network parameters $\phi$.
else
    Initialize neural network parameters $\phi$ randomly.
while not converged do
    if refine dynamics then
        for update step $i = 1..I$ do
            // Dynamics learning
            Draw $B$ data sequences $\{(a_t, x_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
            Compute dynamic states
            $S_t^{dyn} \sim p_\phi(S_t^{dyn} \mid S_{t-1}^{dyn}, a_{t-1}, x_t)$.
            Update $\phi$ using representation learning.
    for update step $c = 1..C$ do
        // Environment learning
        Draw $B$ data sequences $\{(x_t, o_t, r_t, a_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
        Compute dyn states
        $S_t^{dyn} \sim p_\phi(S_t^{dyn} \mid S_{t-1}^{dyn}, a_{t-1}, x_t)$.
        Compute env states
        $S_t^{env} \sim p_\eta(S_t^{env} \mid S_{t-1}^{env}, x_{t-1}, o_t)$.
        Update $\eta$ using representation learning.

        // Behavior learning
        Imagine trajectories
        $\{(S_\tau^{env}, S_\tau^{dyn}, a_\tau, x_\tau)\}_{\tau=t}^{t+H}$ from each $(S_t^{env}, S_t^{dyn})$.
        Predict rewards and values
        $\mathrm{E}\Big(q_\eta(r_\tau \mid S_\tau^{env}, S_\tau^{dyn})\Big)$, $v_\psi(S_\tau^{env}, S_\tau^{dyn})$.
        Update $\theta$ and $\psi$ using behavior learning.
    // Environment interaction
    $o_1 \leftarrow$ env.reset()
    for time step $t = 1..T$ do
        Compute $S_t^{dyn} \sim p_\phi(S_t^{dyn} \mid S_{t-1}^{dyn}, a_{t-1}, o_t)$ from history.
        Compute $S_t^{env} \sim p_\theta(S_t^{env} \mid S_{t-1}^{env}, x_{t-1}, o_t)$ from history.

        Compute $a_t \sim q_\phi(a_t \mid S_t^{env}, S_t^{dyn})$ with the action model.
        Add exploration noise to action.
        $r_t, o_{t+1} \leftarrow$ env.step($a_t$).
    Add experience to dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^T\}$.
```

Model components

Dynamics	$p_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1}, x_t)$
D-Transition	$q_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1})$
Environment	$p_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, x_{t\text{-}1}, o_t)$
E-Transition	$q_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, a_{t\text{-}1})$
Reward	$q_\eta(r_t \mid S_t^{env}, S_t^{dyn})$
Action	$q_\theta(a_t \mid S_t^{env}, S_t^{dyn})$
Value	$v_\psi(S_t^{env}, S_t^{dyn})$

Hyperparameters

Number of random episodes	N
Collect interval	C
Physics train step	I
Batch size	B
Sequence length	L
Imagination horizon	H
Learning rate	α

We consider two RSSMs: one learns the dynamics of the system and the other learns the environment.

(●) The RSSM that learns the dynamics takes as input both the raw proprioceptive variables and the actions. To teach the network to embed within its latent state relevant information, we also train a decoder to reconstruct the proprioceptive variables associated with that state. Figure 5.17 shows examples of reconstructed states.

$$\begin{aligned} &\text{Dynamics observation module:} && q_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, x_t) \\ &\text{Dynamics transition module:} && q_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1}) \\ &\text{Dynamic reconstruction module:} && q_\phi(\hat{x}_t \mid S_t^{dyn}). \end{aligned} \tag{5.9}$$

(●) The RSSM that learns the environment takes as input a high dimension sensory observation and the latent dynamic state. We then train another network to reconstruct from its latent state the inputs or a projection of it.

$$\begin{aligned} &\text{Environment observation module:} && q_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, o_t) \\ &\text{Environment transition module:} && q_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, x_{t\text{-}1}) \\ &\text{Environment reconstruction module:} && q_\eta(\hat{o}_t \mid S_t^{env}). \end{aligned} \tag{5.10}$$

Using these two latent states, S^{env} and S^{dyn}, we also learn to predict the reward that will be given by the environment $q_\eta(r_t \mid S_t^{env}, S_t^{dyn})$. Regarding the optimization of these networks, we use the same method as Dreamer [122, Sec. 4] with a caveat: the dynamics is not trained jointly with the environment and the reward. To train the dynamics, we propose two options. In the first option, we do not learn the dynamic model. For instance, one could use an already learned dynamic model and keep it fixed for the whole of the training. This is interesting as the physical model remains constant throughout the training. It helps learning the value and the reward model faster than if we were to learn it from scratch. Also, it offers the possibility to completely remove the learning and neural network components from the dynamic estimation, by using an analytical model, for instance. In robotics, this makes sense, as good analytical/learned models are often already known. If we don't know the dynamics, the second option is to learn it as we discover our environment. However, to ease the optimization of the actor, the reward, and the value, we do not train the dynamics at the same rate as the rest of our agent's neural networks. We train the dynamics less often, but for more steps than the other components. While this technique does not offer as much stability as the first method, it still increases the overall stability of the learning process.

Learning behaviors

With the learning of the system dynamics and environment covered, we can now use them in the imagination process. The imagination generates imagined trajectories from which the actor is learned. In the original Dreamer, imagined trajectories start from latent states s_t obtained from actual observations, and follow the predictions of the transition model $s_\tau \sim q(\cdot|s_{\tau\text{-}1}, a_{\tau\text{-}1})$ and the policy $a_\tau \sim q(\cdot|s_\tau)$. In our case, we use the dynamic transition model as a physic engine for the imagination, and the environment transition model as the rendering engine. This means that to imagine a trajectory, we start from a pair of latent states $(S_\tau^{env}, S_\tau^{dyn})$ obtained from actual observations, and iteratively apply the transitions models given in Eqs. Equation 5.9 and Equation 5.10, the policy being given by $a_\tau \sim q(\cdot|S_\tau^{env}, S_\tau^{dyn})$. As this process goes along, the proprioceptive variables x_τ are reconstructed from the physical state S_τ^{dyn} using $q_\phi(\hat{x}_t \mid S_t^{dyn})$. They are then used inside the environment transition model. An important note is that, since RSSMs are VAEs, if we sample from the stochastic state to reconstruct the physics, then the imagined proprioceptive variables will be noisy. To prevent that, we take the mode of the distribution which generates smooth physical predictions. Once we have a set of trajectories, we can get each pair of states $(S_\tau^{env}$,$S_\tau^{dyn})$ and use them to estimate the reward and value associated with them. Using these, we train the policy and the value estimator using the same technique as the original Dreamer [122, Sec. 3].

Environment Transfer

Unlike Dreamer, our method allows transferring environments from one robot to another. In this context, we consider that both robots share the same sensors and same state-space. We also consider that the environments in which the robots are going to evolve are similar. Let us consider robot A, with already known dynamics modules $p^A_\phi(S^{dyn}_t \mid S^{dyn}_{t\text{-}1}, a_{t\text{-}1}, x_t)$, $q^A_\phi(S^{dyn}_t \mid S^{dyn}_{t\text{-}1}, a_{t\text{-}1})$. Let us consider robot B, with already known environment modules $p^B_\eta(S^{env}_t \mid S^{env}_{t\text{-}1}, x_{t\text{-}1}, o_t)$, $p^B_\eta(S^{env}_t \mid S^{env}_{t\text{-}1}, x_{t\text{-}1})$, $q^B_\eta(r_t \mid S^{env}_t, S^{dyn}_t)$, and a pre-collected set of samples of robot B interacting with its environment. To learn the actor of robot A, using the environment of robot B, we learn entirely offline the actor, the reward, and the value, using the set of experiences collected on robot B. It should be noted that we would not need to retrain the reward if it was rewritten as $q^B_\eta(r_t \mid S^{env}_t, x_t)$. The main disadvantage of this method is that because we are using experiences collected on robot B, the observation of the dynamics will match the dynamics of robot B and not the one of robot A. However, as the imagination progresses the dynamics will match the one of robot A. Another issue can arise if the robots are radically different. In this case, the samples collected on robot B will most likely not explore the state-space of robot A enough to control it perfectly. Nonetheless, as we will demonstrate later, it allows learning good preliminary policies to further refine the agent from. The exact algorithm used to learn the agent policy is given in algorithm 9. Besides, the state-space of both robots must be the same, which makes this method mostly aimed towards mobile robots. Examples of the state-space of our robots are given in subsubsection 5.4.1.

Algorithm 9: Offline Learning, and Environment Transfer

```
Fill dataset $\mathcal{D}$ with all the episodes collected on Robot B.
Initialize neural network parameters $\theta, \psi$ randomly.
Load neural network parameters $\eta$ from robot B.
Load neural network parameters $\phi$ from robot A.
for training step $c = 1..C$ do
    // Reward Finetuning
    Draw $B$ data sequences $\{(x_t, o_t, r_t, a_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
    Compute dyn states
    $S_t^{dyn} \sim p_\phi(S_t^{dyn} \mid S_{t-1}^{dyn}, a_{t-1}, x_t)$.
    Compute env states
    $S_t^{env} \sim p_\eta(S_t^{env} \mid S_{t-1}^{env}, x_{t-1}, o_t)$.
    $r_t \sim q_\eta(r_t \mid S_t^{env}, S_t^{dyn})$
    Update reward $q_\eta$ using representation learning.
    // Behavior learning
    Imagine trajectories
    $\{(S_\tau^{env}, S_\tau^{dyn}, a_\tau, x_\tau)\}_{\tau=t}^{t+H}$ from each $(S_t^{env}, S_t^{dyn})$.
    Predict rewards and values
    $\mathrm{E}\Big(q_\eta(r_\tau \mid se_\tau, S_\tau^{dyn})\Big)$, $v_\psi(S_\tau^{env}, S_\tau^{dyn})$.
    Update $\theta$ and $\psi$ using behavior learning.
```

Model components

Dynamics	$p_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1}, x_t)$
D-Transition	$q_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1})$
Environment	$p_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, x_{t\text{-}1}, o_t)$
E-Transition	$q_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, a_{t\text{-}1})$
Reward	$q_\eta(r_t \mid S_t^{env}, S_t^{dyn})$
Action	$q_\theta(a_t \mid S_t^{env}, S_t^{dyn})$
Value	$v_\psi(S_t^{env}, S_t^{dyn})$

Hyperparameters

training steps	C
Batch size	B
Sequence length	L
Imagination horizon	H

5.6.4 Robots, Task & Evaluation

Robots

To evaluate our approach we use two robots (subsubsection 5.4.1,subsubsection 5.4.1): an USV and an UGV. The USV is a Clearpath Heron, a small catamaran with a turbine in each hull. There are two action dimensions: the first element controls the left turbine, the second element controls the right one. This means that to drive in a straight line, the agent must send the same value to both turbines. The UGV is a Clearpath Husky, a skid-steer ground robot controlled with a twist input. The size of the action of the Husky is 2, but unlike the Heron, the first element controls the forward velocity, while the second controls the angular velocity. Both robots share similar sensors: a 2D laser-scanner, an RTK GPS and a 9DoF IMU. To prevent damaging our robots' actuators, we saturate their acceleration before applying the output of the policy. More information regarding the exact specifications of the USV and UGV can be found in subsubsection 5.4.1 and subsubsection 5.4.1. We teach our agent to solve a sensor-based navigation task: the robots must follow the shore of a lake at a fixed distance while maintaining a set forward velocity. Hence, the reward generated by the environment is made of two components: a low-level constraint, the velocity, and a high level constraint, the distance from the shore. The exact description of the task and reward is available in subsection 5.4.2.

To fulfil their tasks, the robots rely on a 2D laser-scanner and on their own velocity, estimated from the fusion of the RTK GPS and the IMU through an Extended Kalman Filter (EKF). Using the EKF, we provide to the agent three proprioceptive variables: its forward-velocity, its lateral-velocity, and its angular-velocity. The action space of both robots is $[-1,1]^2$.

Evaluation

We test our method both in simulation and on a real USV. In simulation, to evaluate the performance of our method, we used the different components of the reward as metrics. Each model ran for two hours and at the end of each run we collected the standard deviation, the mean, the 10% quantile and 90% quantile of the rewards. In the real world, we cannot compute the distance reward accurately: our estimation of the distance to the shore is uncertain because of the high level of noise of the laser data in a natural environment. This is particularly true because the experiments were carried out in late spring, where the luxurious vegetation makes laser-scans highly unreliable. For this reason, we will not be evaluating the agent distance from the shore, but we will visually inspect the followed trajectories.

5.6.5 Simulation Results

We compare our method to the original version of Dreamer, and illustrate the benefits of having a separated state for the dynamics. All models trained with our approach have been trained with a fixed dynamic model. This means that the dynamics was learned ahead of time and could not be refined during the training process. This was done to show that we could have used analytical models instead.

Unmanned Surface Vehicle

To show the benefits of using a separated state for the dynamics, we ran a benchmark using the USV. To verify that the agents perform optimally, we first run a test where they are deployed in their original training environment. Then, to evaluate how the agents behave when the dynamics change, we multiply the simulated robots damping factor by two. Increasing this factor makes the robots' dynamics softer: for the same inputs, the acceleration will be slower. To test the robustness to perturbations in the dynamics, we also run a test with the original dynamics and a constant water current with a speed of $0.4m/s$. Finally, to further stress the agents, we multiply the damping factor by two and add $0.4m/s$ of current. The results of this benchmark can be found in Figure 5.18.

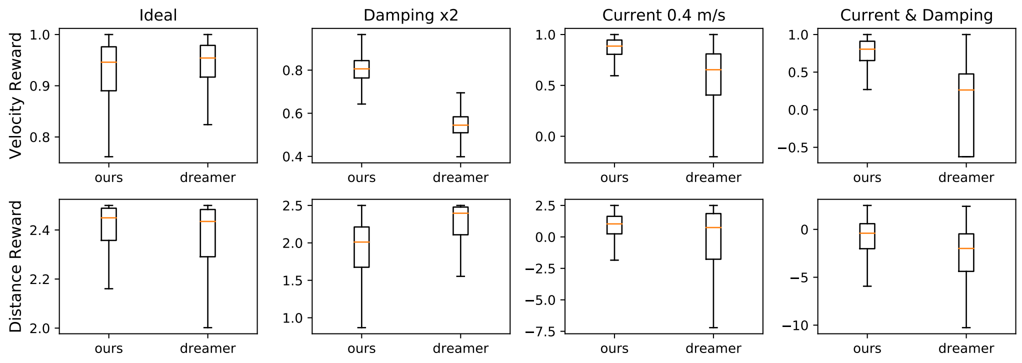


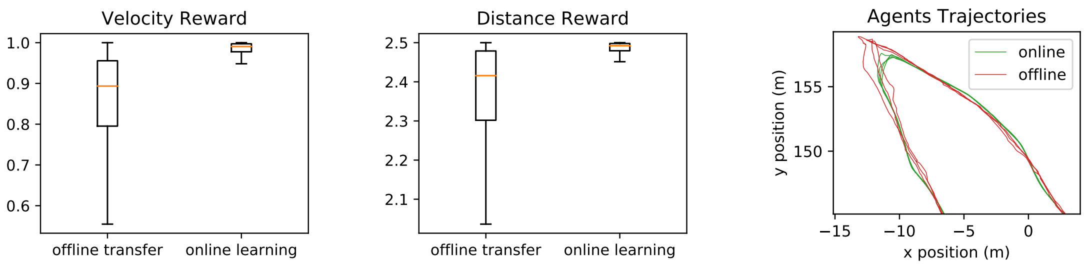
Figure 5.18: Box plots illustrating the benefits of using a separated state for the dynamics (ours) vs not using one (Dreamer). High value indicates better performances. Small spread indicates better consistency. Results obtained in simulation.

Our method is more robust than Dreamer to both dynamics change and perturbations. Most notably, it maintains a correct velocity reward across the whole benchmark, when Dreamer fails as soon as the dynamics changes. This suggests that images are not sufficient to approximate the velocity correctly, and that by using our separated physical state, the actor is able to adapt its behavior to match the desired velocity, despite having never seen such events in the training. Moreover, having access to a physical state seems to help to solve the high-level constraints, as our method outperforms the vanilla Dreamer on this part of the task too. The only exception is when the damping factor is multiplied by two. In this case, the lower velocity of the robot allows Dreamer to follow the shore better. Overall, these experiments show that splitting the single latent state into two, one for the physics and one for the environment, is not only a viable concept but also increases the robustness of the agent.

USV to UGV transfer

To showcase the transfer capacities of our method, we learn an actor completely offline, using the environment of the USV and the dynamics of the UGV. This is possible as we separate the latent-state into those two parts, dynamics and environment. As a reminder, the USV and the UGV do not react to actions in the same way. The USV is driven by sending commands to the left and right turbines, when the UGV is driven by sending forward velocity and angular velocity commands. Furthermore, the dynamics of both systems are radically different: the USV glides, has a huge inertia and has a significant amount of lateral velocity whereas the UGV has almost no slippage, no lateral velocity and no inertia. To train our actor, we first trained our USV to fulfill the shore following task, and we then took its environment model (including the reward) along with the samples of its interaction with the simulation. We secondly learned the dynamics of the UGV from a set of past interactions with the simulation. Finally, using the environment of the USV and the dynamics of the UGV, we trained the actor entirely offline. We then deployed it in simulation and compared it to an agent trained on the UGV and on the environment for an equivalent amount of training steps (300k).

Figure 5.19a and Figure 5.19b show the performance of the agent trained offline with environment transfer (offline transfer, ours) compared to the agent trained online with its



(a) Velocity reward (b) Distance reward (c) Trajectories

Figure 5.19: Environment transfer results. Higher values indicate better performances, smaller spread indicates better consistency. Results obtained in simulation.

own environment (Dreamer). It is worth noting that we did not include the results of the RL actor applied directly on the UGV: it completely failed to solve the task because of the complete difference in the commands mapping. From these results, we can see that the agent trained online achieves almost perfect results, with an average distance reward of 2.48. On the other hand, our agent achieves an average reward of 2.4, which amounts to about ±30cm of error on the 10m distance it must keep from the shore. Figure 5.19c shows the trajectory of the agents in the hardest spot of the simulation: a narrow hairpin-turn. We can see that the trajectories of the agent learned online are closely packed together. On the contrary, the trajectories of the agent learned offline are more loosely grouped, indicating that this policy is less reliable and most likely less overfitted to the environment. These approximate trajectories can be explained by the fact that the imagination is inaccurate at its beginning. Since the samples used to generate the starting states of the imagination are taken from the USV, these starting states contain dynamics that are not feasible on the UGV. Yet, despite these shortcomings, we demonstrated that we can easily transfer environments in a fully offline fashion even on robots with fundamentally different dynamics and action mappings.

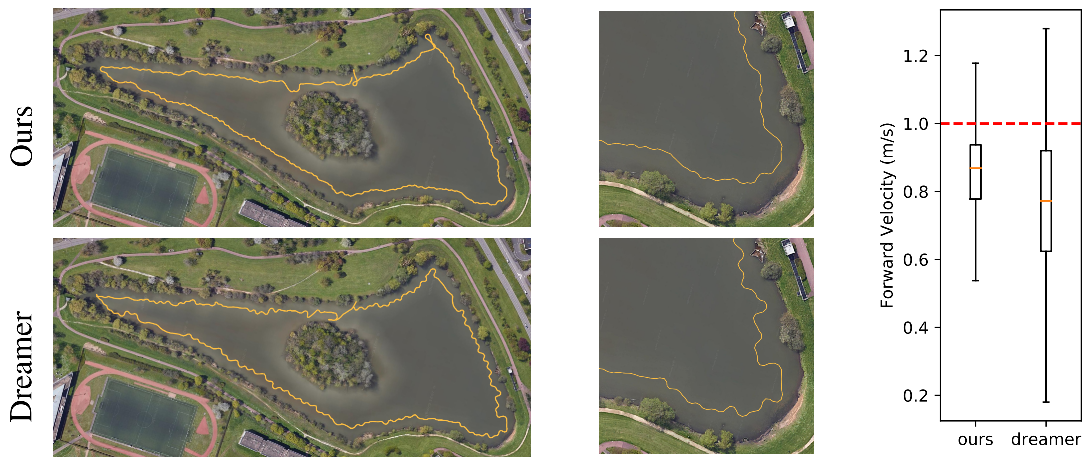
5.6.6 Real World Results

Finally, we evaluate our method on a real USV. To do so, we train two agents in simulation, one with Dreamer and one with our method, and deploy them directly on the real robot in a zero-shot setup. Thus, the dynamics learned by our agents is only approximately matching the one of the system. We evaluate the performances of both agents on a whole lap around the lake. These laps are shown in Figure 5.20a, a lap is about 1.6 km long and takes approximately 20 minutes. During their lap, neither of the agents collided with their environment. Regarding the trajectories, Figure 5.20b shows that the trajectory followed by our method is much less winding than the one followed by Dreamer. The reason for that seems to be that the agent using Dreamer tends to overshoot more than the agent trained using our method. This overshooting behavior can easily be spotted on the cropped image of the lake. On the middle right part of the image, we can clearly see the Dreamer agent overcompensating and navigating toward trees. This behavior cannot be seen on the agent trained using our method, which follows the shoreline at a much more constant distance. It further demonstrates that accessing the dynamics makes the policy naturally robust to changes in the dynamics.

As for the velocity, the right most picture of Figure 5.20c shows that the agent with a physical state (ours) is the closest from the desired forward velocity. Additionally, our method shows a smaller spread when compared to the original Dreamer. It is interesting to note that this model can fulfil its task despite having noise on the physical inputs provided by the EKF, which is something it had never encountered before. Furthermore, during the experiments, we observed that our agent was making less brutal accelerations and seemed to have a more fluid way of steering the boat. All in all, this confirms the simulation results shown in Figure 5.18: our method is more robust and better matches the forward velocity target.

5.6.7 Conclusion

In this section, we showed that splitting the latent-state of MBRL into two sub-latent-states, one for the environment and one for the dynamics, is a viable concept in both the simulated and real world. In all our experiments, our method was trained with a fixed dynamic model



(a) Trajectories (Full)

(b) Trajectories (Zoom)

(c) Velocities

Figure 5.20: Real world experiment. First row, overhead imagery of the deployment site, with the full trajectory of the agents in yellow. Center row: zoom on the bottom right corner of the lake, the trajectory of the agent can be seen in yellow. Last row, comparison of the forward velocities reached by the two agents. Overhead imagery from Google Earth, 2021, trajectories plotted using Google Earth KML API.

which seems to confirm that similar results could be achieved with analytical models, or any other system identification method. Through our experiments, we showed that explicitly adding information about the dynamics of the system make the agent more robust to both perturbations and changes of the dynamics. Finally, we showed that our method could achieve environment swaps from one robot to another, allowing to train robots fully offline. The learned policy could then be deployed on the target system and be refined. Despite showing strong results, some limitations remain. Most notably, the transfer between systems will only be possible if they share the same state-space. Additionally, with the current setup, the dynamics does not explicitly depend on the environment. This could be addressed by providing either exteroceptive inputs or access to the environment state to the physics module.

5.7 Improving the reusability, robustness and transferability

5.7.1 Motivation

To accomplish sensori-motor tasks robots often must combine perception constraints and physical constraints. For example, when a mobile robot drives in an complex environment, it has to balance its trajectory with the velocity it is moving at. Furthermore, we might want this robot to have the ability to move at different velocities in this environment. Unfortunately, classical control techniques struggle to solve these tasks because they have a hard time handling high-dimension inputs, such as LIDARs or images.

Several sub-fields of RL also tried to address this particular type of tasks. However, these methods have thus far produced either only partial solutions or have been expensive to implement (e.g., data collection time, compute power). In our section, we demonstrate that using physics-driven latent imagination [3] (derived from [167]) robots can efficiently learn how to reach multiple physical goals while following perception constraints.

In RL, every time the specifications of a task changes, the policy must be learned anew. In comparison, in control theory, to redefine a task with a MPC based controller, the only thing that needs to be done is to change the cost-function. In robotics, the main problem is not learning the policy, but interacting with the physical system to collect the data required to learn. Indeed, collecting data on real systems is often a very long and costly process; even simulators are relatively expensive to run. In RL, some methods address this issue by learning models of the environment. These models, also known as world models, can in turn be used to learn a policy. These methods are gathered under the umbrella of MBRL. They are particularly well suited to robotics, as once the world model is known, they can be reused to learn a policy, instead of learning on a simulator, or the physical system. This significantly reduces the cost of learning a policy. In this study, we teach policies to chase imagined goals. To do so, we use a world model based on physics-driven latent imagination [3]. This model, which is designed for mobile robotics, features a modular world model with two

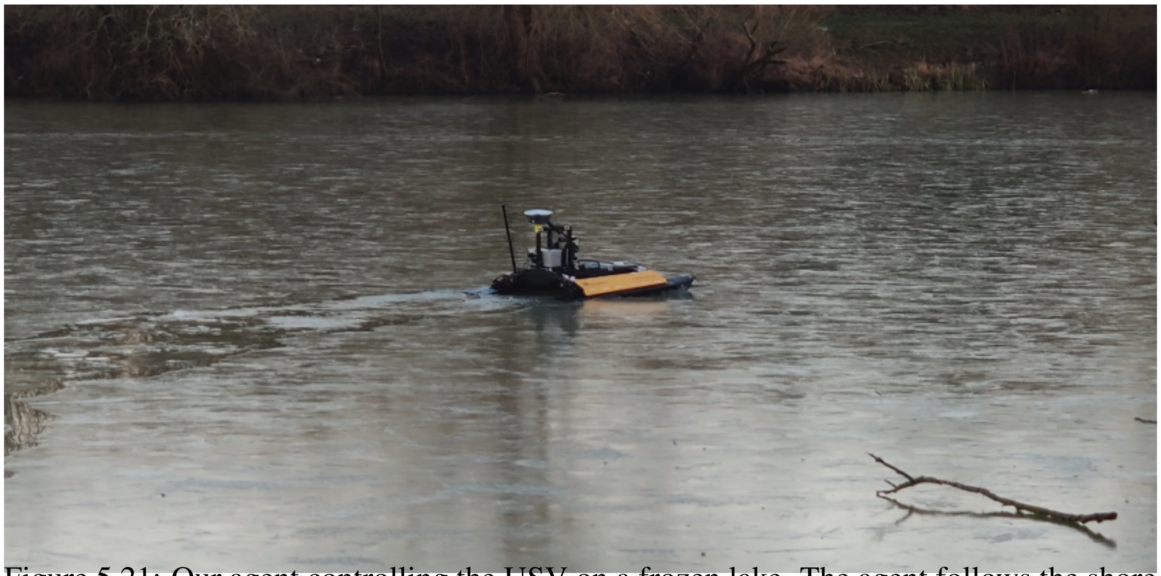


Figure 5.21: Our agent controlling the USV on a frozen lake. The agent follows the shore and maintains a target velocity while breaking 4mm thick ice.

components, a component that learns the dynamics of the robotic system and another that learns the dynamics of the environment. In all the experiments presented here, we reuse the robot's dynamics module of the world model.

Another, orthogonal, approach to this problem consists in learning a policy that can solve many tasks, alleviating the need to train a new policy for every task. The methods that allow this behavior are often referred to as Goal Conditioned Reinforcement Learning (GCRL) [168]. In GCRL, the policy learns to reach different goals in a given environment. A goal can be the position of a robotic arm's end-effector, the velocity of a car, or a more abstract objective, such as making a coffee. Goals are often given under the form of a state, and to know how well the policy is doing, distance functions [169, 168] are commonly used as metrics. In this work, we focus on goals that define physical constraints. More precisely, we apply our policies to robots who must navigate at different velocities.

Overall, all these methods are applied in simulation or to controlled laboratory robotics setup, and often lack in-depth performance study. Unlike these previous studies, we apply our method to a system deployed in a natural environment, in adverse weather conditions, such as a frozen lake, as shown in Figure 5.21.

In this section, we propose a novel approach to teach policies how to reach multiple physical goals while respecting a higher-level perception constraint. This method is designed with mobile robots in mind. It aims at providing highly reusable components, and learn a well-behaved controller that generalizes well to out-of-distribution goals. To achieve this, we use imagination learning [128] to generate long trajectories. In the imagination, the policy is tasked with reaching random goals that are directly related to the physical state of the robotic system. This goal is sampled randomly at the beginning of the generation process. This allows our method to teach policies to reach goals that are far from their current state.

Our contributions can be summarized as follows: **(1)** we propose a novel method to learn how to reach multiple physics-related goals, while imposing high level perception constraints; **(2)** we thoroughly evaluate our work in simulation, and analyze the effects of various hyperparameters on performance; **(3)** we demonstrate the robustness of our method by deploying simulation-learned models on a real robot, in a zero-shot setup. Through this deployment, we verify that the conclusions drawn in simulation still hold on the real system.

5.7.2 Related Work

Model Based Reinforcement Learning has gotten increasing traction over the last years [160, 170]. Until the last decade, MBRL used to focus on lower dimension models [121]. The earliest form of Model Based Reinforcement Learning can be seen as model-predictive, and optimal control, which branched to Dyna [171], and more recently into methods such as Pilco [158]. In recent years, the increase in processing power, and the advent of ConvNet allowed to learn high-performance visuo-motor policies on an expansive set of tasks, ranging from video games [172, 128, 167, 129, 130] to robotics [173, 114, 127, 149, 2, 3], with lower dimension tasks getting less attention. The main advantage of MBRL lies in its high sample efficiency relatively to MBRL. Also, while recent model free methods such as DRQ [174, 175] can achieve high sample efficiency, they focus data-augmentation on the image space, and not on the dynamics, which are usually a critical component in robotics.

Imagination Learning, often referred to as dreaming, is a branch of MBRL where the policy is learned entirely by interacting with the world model. One of the most well-known algorithms is Dreamer [167], in which the world model is a VAE [176] based deep Bayesian filter, or RSSM [128], with an observe and an update function. It is learned through the observation and reconstruction of sequences of images. This world model acts as a fast and highly parallel simulator, that is used to learn the policy. Many studies build on similar mechanisms, some use reconstruction [167, 128, 3, 157], some do not [177]. Our work builds on physics-driven latent imagination [3]. In physics-driven latent imagination, the state is decomposed into two sub-states, an environment state, and a physical state. The physics' transition function uses the past physical state and a new action to get the new physical state, while the environment's transition function uses the past environment state and the new physical state to get the environment state. The advantage of this method is that both states are decoupled and the transition functions can easily be replaced.

Real-world robots are among the most complex applications of RL. They require from the RL algorithms to be data-efficient as collecting samples on real robots and simulators [107, 178] is expensive, in particular on field robots. These algorithms also need to be computationally-efficient, as robots are often limited by the size of their battery, and the computers they can carry. Eventually, they need to be robust and to generalize well to previously unseen environments. This makes MBRL the go-to choice for most real-world applications with limited data. When compared to MBRL, MBRL, and in particular imagination learning [167, 129, 156], has shown superior performance to methods like PPO [162], D4PG [143] or SAC [133]. Imagination learning has also shown surprisingly good transferability from simulation to the real-world [3, 2, 156, 157]. Yet, one of the key limitations of most MBRL methods, and RL in general, is that they only learn how to solve a single task. To learn a new task, the policy must be learned anew, or distilled into a general policy [179]. In this work, we focus on learning a policy that can achieve multiple goals, with very high efficiency, through imagination learning.

Goal Conditioned Reinforcement Learning is an area of increasing interest in RL. These approaches often rely on model-free techniques [180, 181, 168, 182, 183]. In [180, 181, 168], a goal corresponds to a variation of a task in a given environment, the idea being to teach agents to master a wide range of skills. The main issue with these works is their lack of sample efficiency. To learn how to reach a goal, they have to simulate a trajectory where the agents try to reach the said goal. [168, Sec. 4.5] introduce goal relabeling with HER. They show that reaching goals that are close to the current state is what works best, and trying to reach random goals does not work. This is due to the fact that this kind of approach does not learn a transition model and hence cannot simulate novel-trajectories at training time. Thus, exploring and learning how to solve the different goals require a lot of interaction with the environment. Building on goal-conditioned value functions [181, 168], [183] introduce them to Temporal Difference Models (TDM). Instead of estimating the value from the discounted sum of rewards, they estimate the value by using a recursive value function. The learned value function is then used in an MPC controller to pick the best action. This method has shown improved performance over HER, but because it is MPC-based, it is computationally heavier. Most of these approaches focus on settings where the agent has direct access to the low-dimensional environment state. In comparison, we do not need for the goal to be the same as the state, and we can augment the state with high dimensions observations. Overall, these methods' main issue is the goal-relabeling, as they have to restrict themselves to samples that are close from the current state. This is where MBRL shines, and in particular imagination learning, as it can simulate transition functions over long time horizons, and thus, can reach virtually any goal from any starting state.

Many methods use MBRL to build agents capable of reaching multiple goals [127, 184, 185, 186, 187, 188, 169]. In [127, 184], they propose a visual MPC that learns transition functions from couples (action, image sequences). With this learned model, they sample multiple action-sequences and select the first action of the trajectory that leads to the best outcome. The best outcome is decided using different methods such as the probability of success, a classifier, pixel-distance, or registration. The main drawback of these methods is their cost at inference time. For each time step, a set of action trajectories are sampled, a set of video frames are then generated, a score is assigned to each video, and finally the first action of the video with the best score is selected. While this can be done in real-time with desktop hardware using short planning-horizons, it is not meant to be used on embedded systems.

The work of [186] is probably the most similar to ours. They use a VAE and latent space representation to relabel goals. Similarly to the method presented here, they propose to recompute the reward on the fly, at training time, based on sampled goals and the current state. The main difference between our work and theirs [186] is that we have a transition model, and we can learn offline without the need to sample transitions from the replay buffer. This allows us to simulate long trajectories through the imagination process.

In [169], they use the imagination process developed in [167] to train a world model. Then, instead of training a single policy, they train a policy that will explore previously unseen states, the explorer, and a policy that learns how to reach these states: the achiever. As demonstrated in their work, this allows the achiever policy to solve for a wide range of visuo-control tasks, most of which are applied to simulated robotic-arms. By contrast, our learning strategy is tailored for autonomous navigation, and enforcing physical constraints on the system. Our work builds on [3], where they separate the dynamic model of the system from the dynamic model of the environment. Leveraging this separation, we specify multiple dynamics-related goals and their rewards directly in the imagination process. In practice, this means that we need to learn only a single policy, as our goals are already well-defined. Furthermore, we inject the analytic expression of the reward on the dynamics directly in the imagination, which makes the learning process simpler.

In summary, most methods focus on MBRL, or MPC-like controllers, while we propose to use model-based policy learning, and more precisely physics-driven latent imagination to reach physics-related goals. The learning is done purely through imagination, which allows us to set thousands of different goals for each optimization step. The physics separation also allow us to reuse previously learned blocks, such as the dynamic model. We think that our work is orthogonal to that of [169] and could be used jointly.

5.7.3 Method

Our goal is to build a modular world model that can be used to learn to a robust controller. This means that we first have to learn a reusable world model and then use that model to learn policies capable of solving a set of tasks. The main challenge is to build a structure that allows for efficient goal relabeling. This need for an efficient way of generating new goals over long horizons is the reason why we chose to use an imagination learning framework to design our method.

World Model

MBRL can be seen as a fast and highly-parallel simulator. To model our system and its environment, we use a physics-driven latent imagination world model [3]. This model splits the state into two sub-latent-states $S_t^{\text{tot}} = (S_t^{\text{env}}, S_t^{\text{dyn}})$, one for the environment S_t^{env}, one for the dynamics of the robotic system S_t^{dyn}). Each of these sub-latent-states is acquired using an RSSM [128]. This allows to learn about the dynamics of the system and to reuse it in another environment, and vice-versa.

The RSSMs provide us with two key functions, an observe function that allows us to update the current latent states using observations, and an update function. To observe, the model learns a function that takes sensor measurements as an input and projects it to an embedding. This embedding is then processed by the observed function which outputs a compact latent-state. This state can then be used to reconstruct the observed input, or other variables related to it. The update function, is the transition function of our world model. It predicts the next latent states, $P(S_{t+1}^{\text{tot}}|S_t^{\text{tot}}, a_t) = P(S_{t+1}^{\text{dyn}}|S_t^{\text{dyn}}, a_t)P(S_{t+1}^{\text{env}}|S_t^{\text{env}}, S_t^{\text{dyn}})$. In this model, the dynamics is stepped using the action sent to the system, while the environment is stepped by the low-level dynamics reconstructed from the dynamic latent state, or the dynamic's latent state directly. Using the latent states of the world model, the reward of the agent can be learned. To summarize, our world model learns the components given in Table 5.2. o_t denotes a high-dimension observation, such as an image or a LIDAR scan, x_t is the low dimension input, such as the physical state of the system, r_t is the reward, a_t is the action.

Table 5.2: The modules that make our world model.

Modules	Environment Model	Dynamic Model
Decoder	$q_\eta(\hat{o}_t \mid S_t^{env})$	$q_\phi(\hat{x}_t \mid S_t^{dyn})$
Observation	$p_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, x_{t\text{-}1}, o_t)$	$p_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1}, x_t)$
Transition	$q_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, x_{t\text{-}1})$	$q_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1})$
Reward	$q_\eta(r_t \mid S_t^{env})$	Explicit

We chose this model as it allows us to leverage parts from previously learned models. In all the experiments presented in this section, the dynamic model was fixed and not learned with the rest of the system. Also, because the dynamic model is well known, the exploration does not matter as much as in other works [168, 169]. This means that the policies converge faster, and uses less resources to train. More information about how the model works can be found in [3]. In Table 5.2, one can notice that we do not learn a reward that uses the dynamic state. Indeed, in our method, the analytical expression of the dynamic reward is embedded directly inside the imagination process. In the following section, we will discuss how the goals are being learned by leveraging explicit rewards.

Learning to Reach Multiple Goals

In classical control theory, a controller is almost systematically defined to have a set of performance within a range of values. This part of the design of the controller requires having an analytical, well-formed, model. Here we propose an approach where we learn a controller over a continuous range of values, this controller will then be tested inside and outside this range of values. In doing so, we evaluate its robustness, and its applicability to real systems. To allow our agent to chase goals, we augment the state by concatenating the goal to it. S_t^{tot} becomes $S_t^{\text{tot}} = (S_t^{\text{env}}, S_t^{\text{dyn}}, g_t)$, where g_t is the goal given at time t. With this new state, we then consider the following policy function: $a_t \sim q(\cdot|S_t^{env}, S_t^{dyn}, g_t)$. In this kind of situation, the limiting point is the amount of interaction we can have with the system. Let us take the example of a velocity controller in a car. A classical use case would be for the car to drive anywhere between -15 and 130kmph. With RL, it would take a lot of system interactions to learn a well-behaved controller that can operate optimally on any values in

Algorithm 10: Physics-Driven GC Dreamer

```
Fill dataset $\mathcal{D}$ with $N$ random actions episodes.
Initialize neural network parameters $\theta, \eta, \psi$ randomly.
Load dynamic model parametrized by $\phi$.
while not converged do
    for update step $c = 1..C$ do
        // Environment learning
        Draw $B$ data sequences $\{(x_t, o_t, r_t, a_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
        Compute dynamics states
        $S_t^{dyn} \sim p_\phi(S_t^{dyn} \mid S_{t-1}^{dyn}, a_{t-1}, x_t)$.
        Compute environment states
        $S_t^{env} \sim p_\eta(S_t^{env} \mid S_{t-1}^{env}, x_{t-1}, o_t)$.
        Update $\eta$ using representation learning.
        // Behavior learning
        Select random goals $g_\tau$.
        Imagine trajectories
        $\{(S_\tau^{env}, S_\tau^{dyn}, a_\tau, x_\tau, g_\tau)\}_{\tau=t}^{t+H}$ from each $(S_t^{env}, S_t^{dyn})$.
        Predict rewards and values
        $\mathrm{E}\Big(q_\eta(r_\tau \mid S_\tau^{env})\Big) + f(\hat{x_\tau}, g_\tau)$, $v_\psi(S_\tau^{env}, S_\tau^{dyn}, g_\tau)$.
        Update $\theta$ and $\psi$ using behavior learning.
    // Environment interaction
    env.reset()
    env.set_goal()
    Play in environment for N steps.
    // Add experience to dataset
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^{T}\}$.
```

between these bounds. This problem is why goal relabeling exists. It allows assigning a new goal to a given sample at training time. However, in most algorithms, it is almost impossible to set goals that are too far from the current state, as in these cases, computing the result of the action taken is not possible. In our method, we use our RSSMs to predict forward trajectories for which we set random goals. The RSSMs allows us to optimize for any goal as they provide us with transition functions, allowing us to predict rewards, and the next states over long time horizons. Furthermore, because the RSSMs are highly parallel, we can optimize for thousands of goals simultaneously, for every optimization step. In our experiments, we collect 2000 episodes per training, and train for 100 steps, in between each episode collection. For each step, we generate 2500 trajectories in which we set a random goal. This means that each of our policies were trained on 500 millions imagined goals, allowing us to sample uniformly over the desired control range.

With the ability to predict future states, the limitation now becomes to have a well-defined reward to learn a controller from. In our case, we chose to separate the reward into two sub-rewards. The first one, which relates exclusively to the physical state or the actions, has its equation explicitly written inside the imagination process. As such, it does not need to be learned. The other one, which is learned, includes the rest of the reward that is needed to solve the task, i.e. higher dimension inputs. The value function then learns to predict the n-step returns of the sum of the two rewards. To accommodate for this, we use the following reward function : $R = \mathrm{E}\Big(q_\eta(r_\tau \mid S_\tau^{env})\Big) + f(\hat{x}_\tau, g_\tau)$, where f is the analytical expression of the dynamics sub-reward, and $\hat{x}_\tau$ is the reconstructed physical state from S^{dyn}. Finally, we also change the value function to include the goal, $v_\psi(S_\tau^{env}, S_\tau^{dyn}, g_\tau)$. The full algorithm used to train our agents is provided algorithm 10.

5.7.4 Experiments

In the following experiments, we taught an USV to follow lake shores at different velocities. The velocity being the goals sent to the system. These experiments have been designed to answer the following questions:

1. How important is the imagination horizon when learning to reach multiple goals?
2. Do the targets set in the simulation matter?
3. Can the policy learned in simulation be applied without fine-tuning on the real robot? And if so, how well does it perform?

To answer these questions, we will first describe the system we will be using, then we will present the task we want our agent to solve. Next, we will discuss the different training configurations we chose and evaluated, and finally we will see how we evaluated each model.

System

All our experiments were carried out using an USV. More details regarding the system can be found in subsubsection 5.4.1. To evaluate our method, we test it on a shore following task, for which we can set different linear velocities. In this section, we use the same task as before (subsection 5.4.2). For clarity's sake, we repeat the reward formulation. The reward associated with the task is given in Equation 5.11, where v denotes the linear velocity of the USV, v_{goal} the velocity to track, d the distance between the USV and the shore, and finally d_{target} the desired distance from the shore. Equation 5.12 corresponds to the perception part of the reward, and is learned with the world model by $q_\eta(r_t \mid S_t^{env})$. Equation 5.13 corresponds to the dynamic part of the reward, and is embedded inside the imagination process by $f(\hat{x}_t, g_t)$.

$$\begin{aligned} R &= 2.5 \times R_{\text{dist}} + R_{\text{vel}} && (5.11) \\ R_{\text{dist}} &= \max(-20, 1 - 0.5 \times (d_{\text{target}} - d)^2) && (5.12) \\ R_{\text{vel}} &= 1 - \left| \frac{v_{\text{goal}} - v}{v_{\text{goal}}} \right| && (5.13) \end{aligned}$$

Training Configurations & Environment

To evaluate and analyze our method, we trained 18 models with different imagination horizons, target-velocity ranges in simulation, and target-velocities ranges in the imagination. Table 5.3 shows the different training configurations tested. The first line of the table shows the least favorable setting. In simulation, the agent is asked to go at a low speed of 0.5 m/s, and the imagination velocities are also limited to the [0.5-1.0] range. This means that the policy will not explore a lot of its state space, in particular with a short horizon. The last line of the table is the most comfortable setting, with the simulation matching the imagination distribution of velocities. Each of these configurations are then trained using different imagination horizons. Our models are all deployed on the real lake in a zero-shot

Table 5.3: Each line is a configuration. All the configurations are tested for three different imagination horizons: 15, 30 and 45, making for a total of 18 different training configurations. Configurations with a $\star$ have been trained five times with different seeds and an horizon of 30.

Simulation velocity range	Imagination velocity range
0.5	0.5-1.0 $\star$
0.6-1.0	0.5-1.0
0.6-1.0	0.3-1.3
1.0	0.5-1.0 $\star$
1.0	0.3-1.3 $\star$
0.3-1.3	0.3-1.3 $\star$

setup.

Evaluation

Goal Generation: To compare the different training scenarios, we devised a benchmark heavily inspired by control theory metrics. In this benchmark we evaluate three behaviors: 1) fixed velocity regime, 2) step response, 3) velocity tracking using sinus and saw-tooth. The benchmark is done in simulation only, as running it on the real system would take too much time.

1. To evaluate the quality of the policy in fixed regime, we deploy it in the evaluation

environments and set goals within the range [-0.7,1.7]. Each goal is maintained for one minute. This process is repeated five times. The goal with this test is to see how well the agents perform on seen velocities, but also on out-of-distribution velocities. A well-learned-controller should be robust to out-of-distribution targets. Furthermore, this test will allow us to see if the policy has a static error.

2. Analyzing the system's response in transient regimes is done by setting velocity goals that follow square-shaped signals of different periods and amplitudes. This aims at evaluating if the policy overshoots, as well as the rise time, and fall time. Since our agents use GRUs to embed their states, there could be a delay between the time the command is received and the time the command is actually applied.

3. Finally, we assess the velocity-tracking capacities of the policy by sending velocity goals that follow sinusoidal and sawtooth-shaped signals. With this test, we want to measure the tracking error, and make sure the policies can adjust for small velocity increments. This task uses the same combination of periods and amplitudes as the previous one.

Environments: To evaluate the models, we generated 3 types of simulation environments. These environments are used to run the benchmark described earlier.

- The simplest environment is a straight line of 400 meters, formed by a set of poles of 20cm radius, separated by a distance of one meter from each other. This environment was created to see how the models would perform in ideal conditions, where there are no challenges imposed by the perception task. An image of this environment can be seen on Figure 5.22a.

- The second type of environment is a succession of half-circles forming a sinusoidal curve. This curve is made by the poles, with the same separation. An image of this environment can be seen on Figure 5.22b

- The third and last type of environment is a sawtooth like curve, also made with the same poles and the same separation. An image of this environment can be seen on Figure 5.22c.

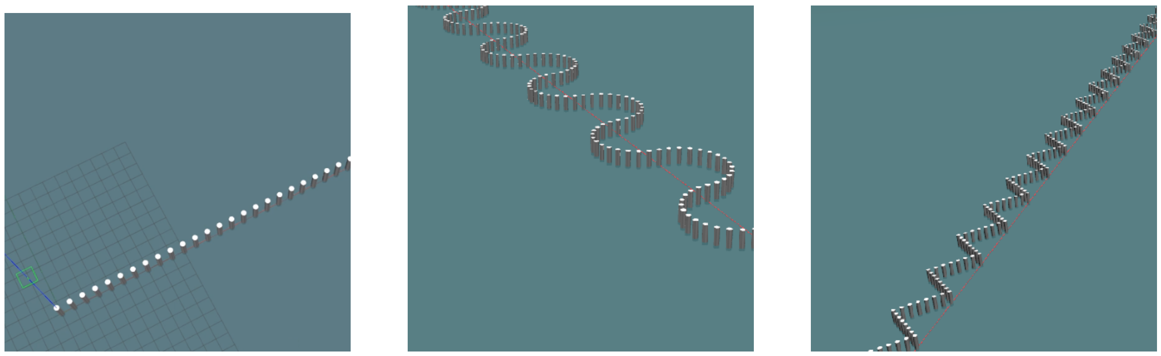

(a) The Line environment. (b) half-circle environment. (c) A sawtooth environment.

Figure 5.22: The different type of environments used to evaluate the models.

Using these environments and the previously defined benchmark, evaluating a model on one of our servers takes about 1 server-days. This is mostly due to the simulator we use, Gazebo, which is barely running real-time.

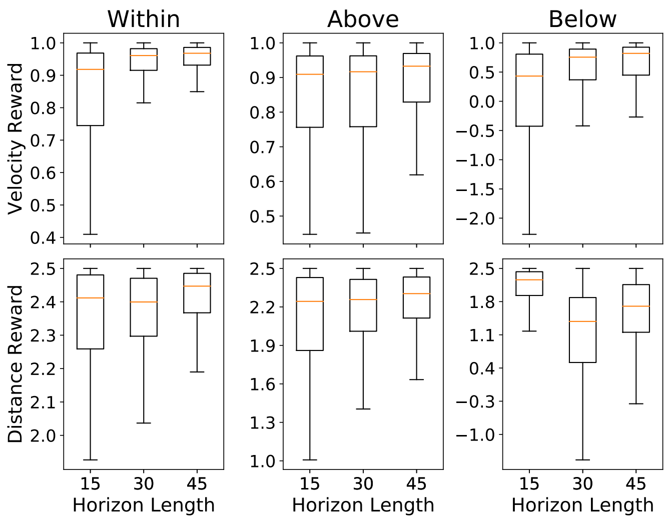
On top of these environments, we also deploy our algorithms in zero-shot transfer on a real lake. **Metrics:** To measure the performance of each model, we measure two properties. The first one evaluates how well the goals are reached. This is done by using the velocity reward on aggregated data, or the average velocity. The reward is akin to an accuracy. This makes it particularly well suited to study a large batch of different velocities. On the other hand it is not really tangible. Hence, when we want to compare un-aggregated data, we use the average error which is simpler to relate to. The second evaluation criteria is the distance to the shore. Similarly to the velocity goals, it uses the perception reward, and the absolute error. In addition, of these metrics, we also measure specific metrics when the policies follow square-shaped velocity goals. In these instances, we measure the rising and falling time, as well as how much the policy overshoots.

5.7.5 Results

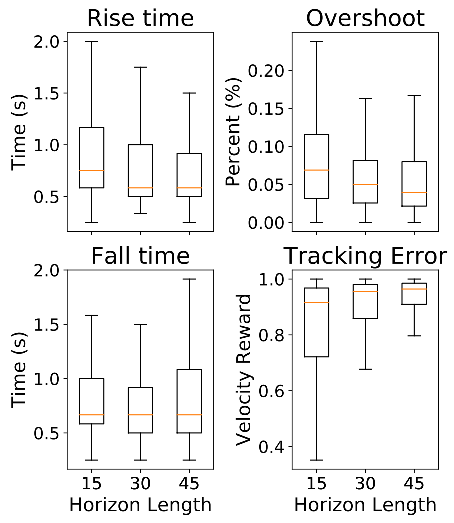
We answer the questions raised in the experiment part. We start by studying the impact of the imagination horizon, then we explore the importance of the simulation targets, and finally we demonstrate the transferability of the simulation-learned policies to a real system in a zero-shot setup.

On the Importance of the Imagination Horizon

We explore the importance of the imagination horizon in our learning process. To measure the impact of the imagination horizon, we aggregate the results of all our training configurations by the length of their imagination horizon.



(a) Fixed regime.



(b) Transient regime.

Figure 5.23: Aggregated performance of all the models, on all the tracks, based on their horizons.

Figure 5.23a shows the fixed regime performance. On the leftmost column, which shows the performance on the goals that are within the range given in the imagination, we can see that the longer the horizon, the better the performance. Similar results can be seen on the goals that are above the training distribution. At first, the results on goals below the training distribution can seem surprising, with the shorter horizons achieving better performance on the distance part of the task than the longer horizons. However, the velocity constraint is not respected at all, which means that the shortest horizon models fail to reach their goals, and hence can more easily maintain their distance to the shore. It is also worth noting that the models can drive backward, even though they had never been trained to do such a thing. Even more surprisingly, they manage to do so despite the fact that they cannot see behind themselves. These experiments show that not only the controller work on the training distribution, but also generalizes well to out-of-distribution goals. Figure 5.23b shows the performance of the models on the tracking of square signals. The left column shows the rise time and fall time based on the horizon length. We can see that longer horizons have shorter rise time, meaning that the system reaches its target quicker. Similarly, when looking at how much the models overshoot, the longer the horizon, the less they overshoot. We can also see that longer horizons improve the tracking performance of the models (bottom right) with short horizon models struggling to reach the velocity goals. The fall time (bottom left) are more homogeneous with all horizons having the same median, and longer horizons having a slightly larger spread. Overall, this confirms the intuition that longer horizons do provide better performances. However, there is a trade-off. As the horizon get longer, the accuracy of the world model decreases. While we do not see this behavior here, if the horizon gets too long, the performance will decrease as shown in [156]. Before moving on to the model performance comparison, let us analyze how a controller behaves.

Figure 5.24 shows the performance of one of the top controllers on the evaluation goals. The evaluation environment is the straight line. From this graph, we can see that the controller is capable of reaching and maintaining the goal velocity well, whether it is inside or outside the training goal distribution. Inside the training goal distribution the agent maintains an almost constant performance, except when the goal reach and exceeds 1 m/s. One can observe that the faster the agent goes, the harder it is for it to follow the shore. While this can be expected, as this kind of system has a lot of sliding, there are other reasons that could explain this behavior, as we will see later. Another interesting fact is the backward performance. The controller struggles to maintain the distance to the shore while going backward, yet on a straight line it should be fairly simple. To understand and explain this behavior we need to take a look at the commands sent by the agent to the system. Figure 5.25 shows these commands on some evaluation tasks.

On Figure 5.25, the first two columns show a constant velocity tracking task. In this case, we can see that the agent sends commands with an oscillatory pattern, first more left, then more right, and then compensate with left, etc... This behavior is particularly noticeable on the first column, the 1 m/s goal. There are two reasons for this behavior. The first is due to our training environment: never in the training environment the agent is trained to follow a straight line, it always follows a winding shore line. The second is linked to the fact that the boat does not exactly spawn at the right distance from the shore, and it has to compensate for it, which creates oscillations. The faster the boat goes, the stronger the oscillations, and thus the larger the distance error. This kind of system is particularly hard to control, as the oscillations on the commands translate in lateral motion that can only be countered by compensating and introducing further oscillations. When driving on the real lake, we measured that the lateral velocity was approximately half the goal velocity. At a

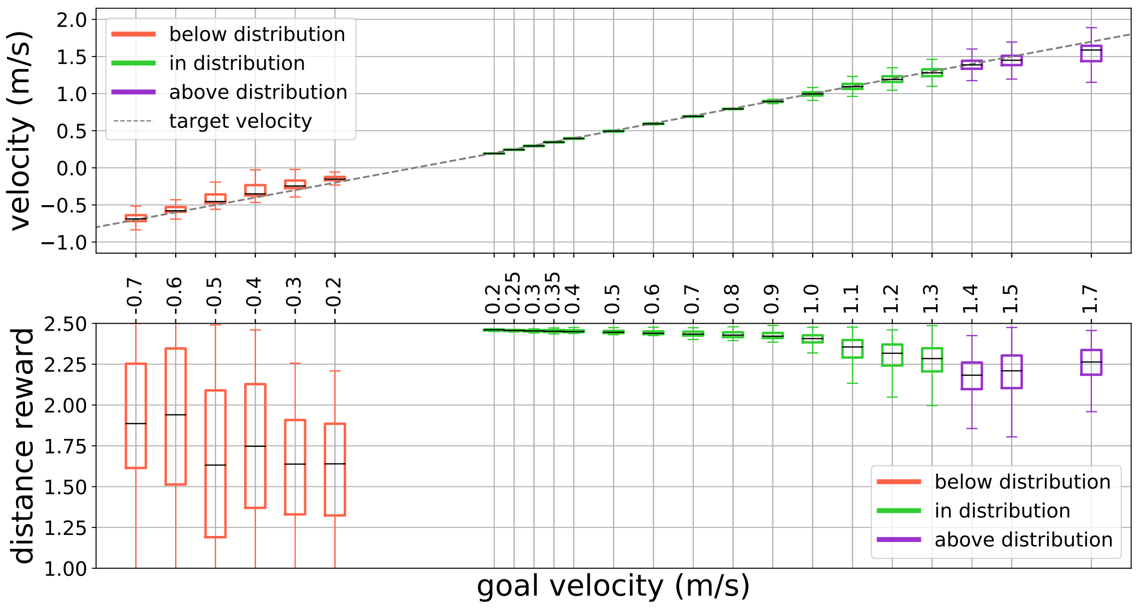


Figure 5.24: The velocities and distance rewards reached by one of the top controller on a fixed velocity tracking task.

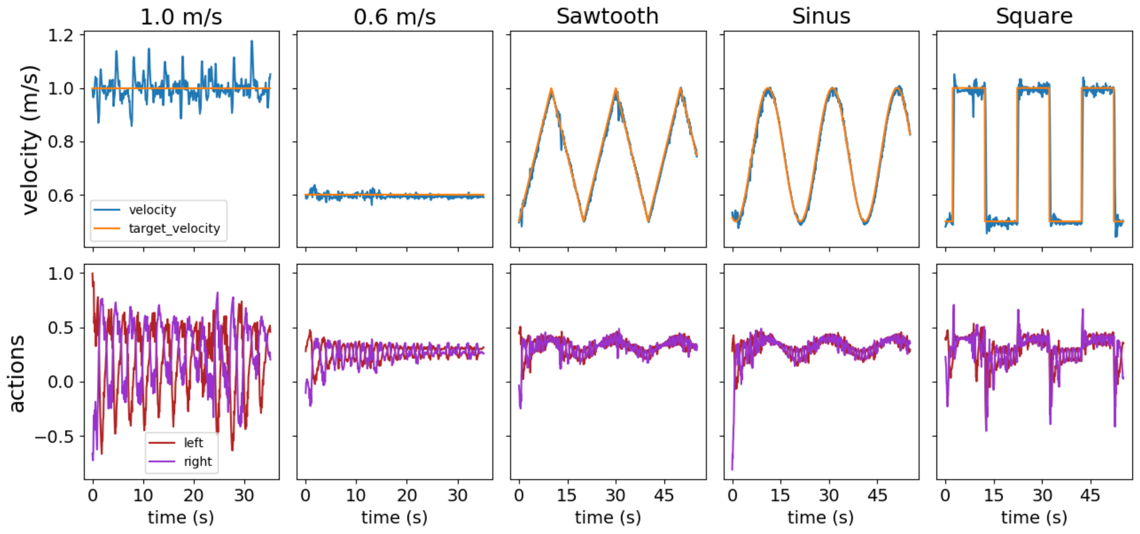


Figure 5.25: The velocities reached by the system, as well as the commands sent by the agent.

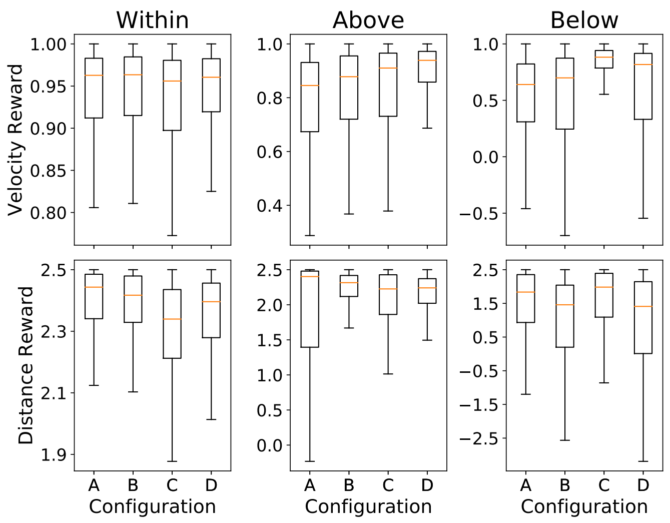
lower velocity, such as on column two, we can see that these oscillations slowly fade as time goes. On the remaining three columns, we can see that the policy can follow sawtooth, sinusoidal, and square goal signals, even though it was never explicitly trained to follow this kind of signal, always fixed velocity goals. Something interesting to notice on the fifth column is the sharp accelerations and breaks when the system must slow down or speed up to catch on with the signals. We can also notice a limited overshooting, and little delay when matching the goal signal.

In summary, our policies are capable of reaching multiple velocity goals, while maintaining their distance to the shore. They generalize well to previously unseen goals, and having a long imagination horizon improves the performance. In our case, increasing the horizon from 15 to 30 steps provides a major performance uplift, whereas going from 30 to 45 only yields minor improvements.

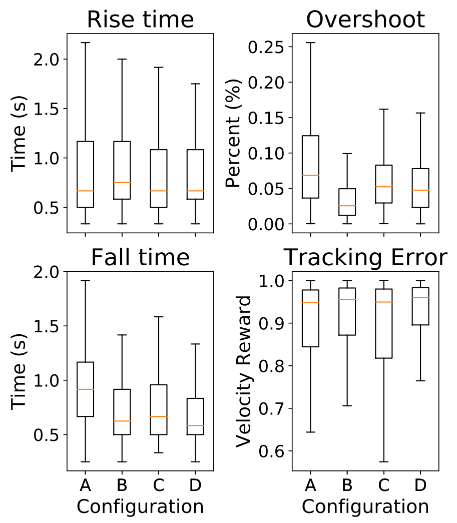
On the Importance of the Simulation Targets

One key question, when learning to reach multiple goals with a world model, is what impact does the simulation have on the learning of the world model. Here, all the models share the same pre-learned dynamics. Hence, we will not study their impact on the learning process. However, the reconstruction of the environment may depend on the velocity range explored. Ideally, if our model does the dynamics separation well enough, it should not make much of a difference. We recall that the RSSM, or world model, that learns the environment is stepped using the physical state and not the action. Hence, extrapolating outside of the seen velocity range should be simpler that when using actions transitions, as the dynamics of the robot can be non-linear.

With Figure 5.26, we show that overall, all the configurations provide fairly similar results. They are organized based on the expected difficulty to learn from them (Table 5.3). On each plot, the leftmost boxes are the hardest configuration (A), and the rightmost boxes are the easiest (D). On Figure 5.26a, when comparing the performance on the training



(a) Fixed regime.



(b) Transient regime.

Figure 5.26: Aggregated performance based on the training scenario. A has a simulation goal of 0.5m/s. B and C have a simulation goal of 1.0 m/s. D has simulation goals between 0.3 and 1.3 m/s. A and B are trained on imagination velocities between 0.5 and 1.0 m/s; C and D are trained on velocities between 0.3 and 1.3 m/s.

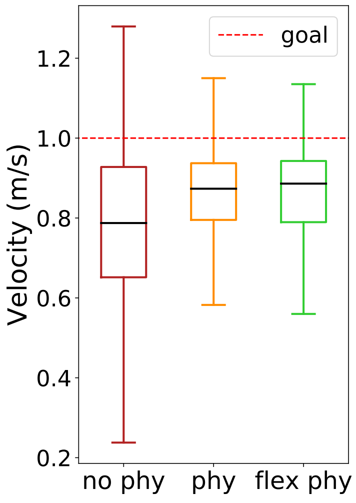
distributions, we can see that all the models perform more or less the same. Except for C. Because of the slowness of gazebo, we only trained our policies on 5 unique seeds. This is not large and could explain why the models trained on C are not as good as the others. Still on Figure 5.26a, for the velocities above the training distribution, the performance of the policies are better as the training scenario gets easier. Configuration A is the worst on this scenario, in particular on the distance to the shore. This could indicate that policies that learned on higher simulation and imagination speeds can better match high velocity goals. Finally, the results on the velocities are somewhat equivalent for all models with large variances. This would indicate that the different configurations do not provide improved performance in this scenario. The results presented in Figure 5.26b are similar to the one in Figure 5.26a. In the end, it seems that the velocity range in simulation does not matter that much. This is interesting, as it would mean that we can leverage most of the data to learn new behavior. These results are also consistent with the great generalization capacities observed earlier.

On the Transferability of the Policy

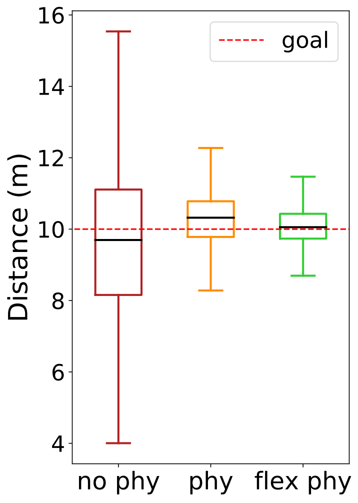
When training models in simulation, a natural question is ”how well will they do in the real world?” In this section, we deploy our policies in a zero-shot setup and compare their performances. Figure 5.27 shows the performance of our agents on the real lake. We compare ourselves, “flex phy”, to a regular version of Dreamer “no phy” [2], as well as to a physics-driven Dreamer “phy” [3]. Both of these policies were trained to reach a single velocity goal of 1 m/s. The comparison is done with “flex phy” being one of our best controller, in different scenarios.

On Figure 5.27a, we can see that the method proposed here does not increase the performance drastically over the physics-driven version. On the one hand, the velocities are within the same bounds. On the other hand, the distance to the shore, seen on Figure 5.27b is better maintained by our controller. This could be due to numerous factors specific to this experimental run. We would like to stress that, while our policy does not significantly improve the performance on a single target, it knows how to reach multiple goals. The "phy" policy does not, and has to be retrained for each new goal. Interestingly, if we look at Figure 5.27c we can see that our method consumes significantly less power than both the "phy" and "no phy" policies. Our method median power consumption is 160 Watts around the lake when "phy" consumes 320 watts. This amounts to a 50% power decrease, while maintaining the same performances. It is likely that, since our agent can reach a wide range of velocities, it understands its action space better, which allows it to be more efficient.

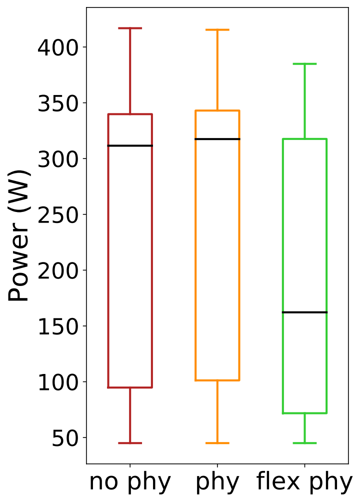
Moving on to Figure 5.28a, where the boat was deployed on a lake with a 4mm thick ice crust, we can see that the policy struggles to maintain the target velocity. However, the distance to the shore is well-matched (Figure 5.28b). Figure 5.28c shows the power drawn when navigating on the frozen lake. We can notice that the motors are almost always working at full power (the median power consumption is 277 Watts). This means that this controller truly understood how to move. It does not associate a velocity with a range of commands, but compensate for the error in velocity by increasing or decreasing the commands sent to the system. Ideally, we would have wanted to replicate this experiment



(a) Velocity of the boat.

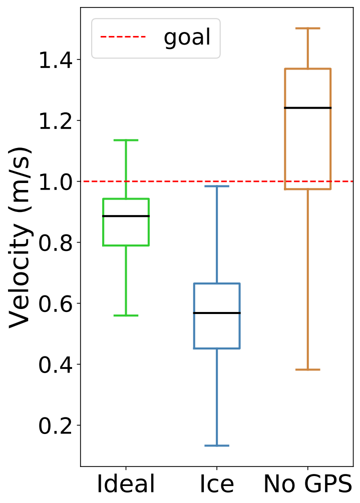


(b) Distance of the boat from the shore.

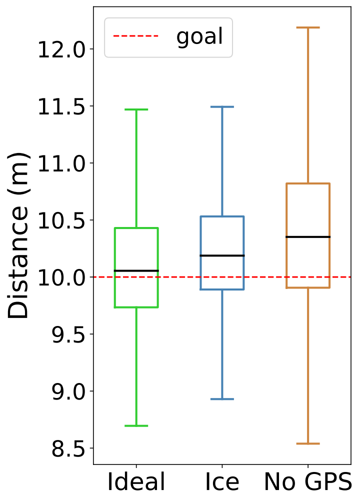


(c) Instantaneous power consumption.

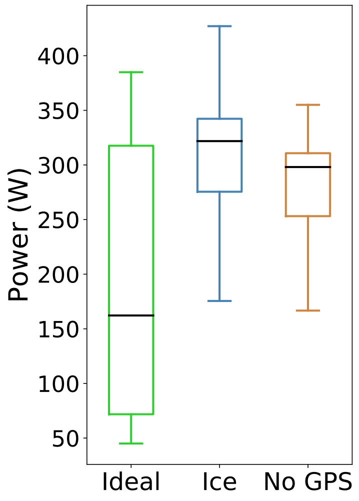
Figure 5.27: The performance of different policies around the lake.



(a) Velocity of the boat.



(b) Distance of the boat from the shore.



(c) Instantaneous power consumption.

Figure 5.28: The performance of our policies around a frozen lake, and in a scenario where the GPS signal is lost. The same policy but in ideal conditions was added for reference.

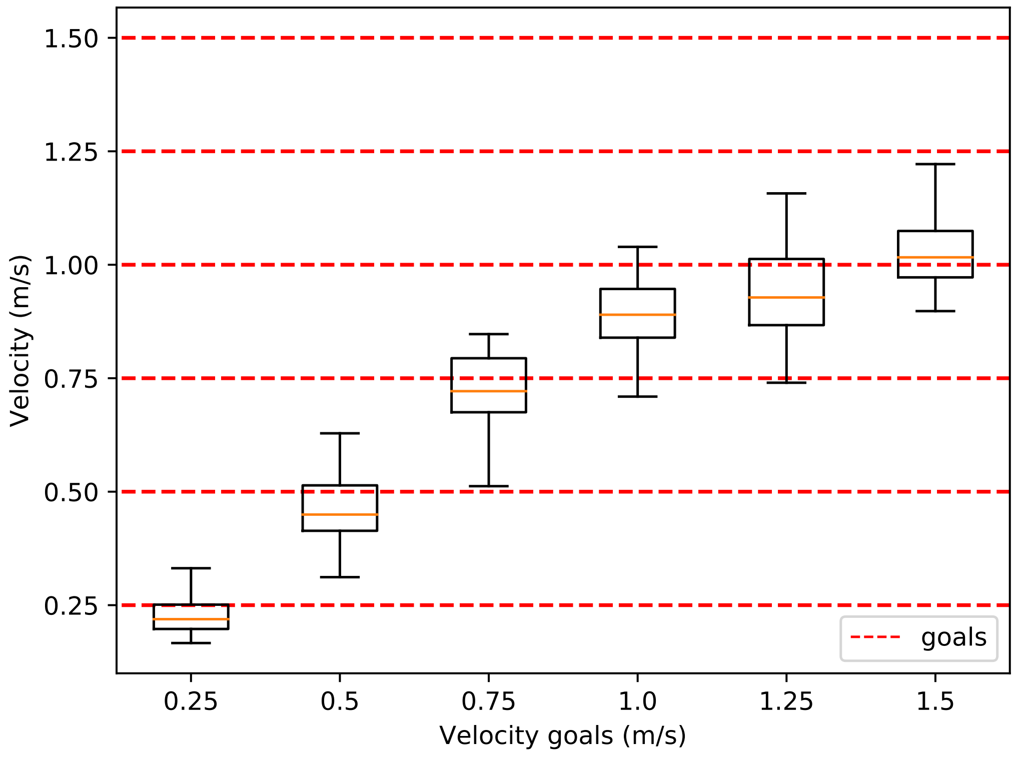


Figure 5.29: Different goals reached by our policy on the real USV.

with the “phy” policy, but getting a frozen lake with a thin enough crust to break is rare and we could not record this policy on a frozen lake yet. This behavior can also be seen when we cut the GPS signal from the USV. In this case, the velocity is null. Hence, the policy increases the commands as much as it can to match the target, which it never will. This translates to the boat moving near its top speed while still following the shore fairly precisely.

To test that our controller could reach different velocity goals, we tasked our agent to follow a given goal on a 150 meters long portion of the shore. The results of this test can be seen on Figure 5.29. We can see that on velocities from 0.25 to 0.75m/s the policy matches the target fairly well. However, for the velocities between 1.0 and 1.5m/s, the policy fails to reach its goal. We believe this is due to the perception constraints, and that the USV cannot drive fast enough without sacrificing its navigation accuracy. It is probable that on a straight line the USV could have reached higher velocities.

Overall, the policies trained with this method transfer well. However, in comparison to the simulation, the performances of the agents are degraded. There are multiple reasons for that, but the main one is probably the fact that the dynamics of the real system is different from the one the policies learned on, hence it is harder for the policies to control the real USV. Another key factor is that, when testing on the lake, the shore is extremely different from the simulation. In a natural environment, and particularly in winter, the laser is much noisier, and there are many ghost points. If a ghost point appears and then disappears close to the robot, then it forces the agents to quickly move away from the obstacle.

5.7.6 Discussion & Conclusion

In this section, we presented a method that uses physics-driven latent imagination to teach a policy to reach different velocity goals, while performing a sensorimotor task, namely following the border of the environment. In our method, the policies are learned exclusively through the imagination process. Through thorough simulation experiments, we show that the policies learned with our method can reach the goals they were trained on, but also goals that they had never seen. Moreover, we study the behavior of the controller when tracking different goals. From this, we observed that while the agents learned how to reach different goals, they tend to send commands that result in oscillations. We then showed that longer horizons improve the overall performance of the learned policy. Building on these results, we evaluated different training scenarios and concluded that they had little impact on the quality of the learned policy. This aligns well with the fact that agents learned by our method have strong generalization capacities. Finally, we deployed one of our policies in a zero-shot setup on a real lake. We demonstrated that our method was capable of matching policies trained to reach a single velocity while using less power. We also demonstrated the robustness of the learned policy by deploying the robot on a frozen lake. In this scenario, the policy failed to reach the target velocity but was able to maintain the desired distance to the shore. More importantly, the agent tried to compensate for dynamics change created by the ice, by using almost all the available power.

5.8 Conclusion on Model Based RL

In this chapter, we have shown that MBRL is a viable choice when deploying autonomous navigation agents in natural environments. In our experiments, we have made ample demonstrations of zero-shot transfer from the simulation to the real system. Furthermore, we proved the robustness of our agent by deploying in a variety of weather conditions, ranging from calm silk waters to a frozen lake. When compared to the MPPI, our agent not only worked better, but was also more reliable. This is interesting because the MPPI was using a dynamic model trained on the real boat data. Overall, this shows that the RL is a very promising alternative to MPC controllers in navigation tasks. Future work should focus on further robustifying the policies, integrating safety layers, or integrating domain adaptation within the imagination. Another interesting point would be to apply our methods to more robots and different tasks. While we do not talk extensively about it in this chapter, the simulation environment plays a critical role in the learning of the agent. As of today, our training environments are hand-crafted, which means that they are full of human bias about what is a hard or a simple environment. Ideally, we would want our environments to be bias free and their complexity related to the current capacities of the agent. This is currently explored by researchers like Jeff Clune[189, 190], or Natasha Jacques[191] and it is our belief that this is the next frontier for reinforcement learning. After all, a model is only as good as the environment it has been trained on. Thus, we would highly encourage future work to investigate adversarial environment generation. While this field is very time-consuming, we are confident that this would be highly rewarding.

CHAPTER 6
CONCLUSION

6.1 Summary of Contributions

This thesis studied state-of-the-art modeling techniques based on NNs. We first applied our methods to the modeling of crop fields. On this problem, we showed that NNs could be used to estimate the daily irrigation recommendation for a crop field. The methods developed to solve this problem have been tested on real crop fields and achieved expert-level irrigation recommendations while observing fewer variables. We then designed a novel technique based on transformers to fill gaps inside measurement streams. We showed that our method achieved state-of-the-art results when filling gaps in EC data.

Then we explored different importance sampling schemes to improve the quality of learned robots' dynamic models. We started by showing the benefits of using these schemes by modeling systems with an uneven exploration of their state/action spaces. In these scenarios, the importance sampling provided a significant accuracy uplift. We then applied the models learned with the prioritization inside an MPPI controller. In simulation, we learned the dynamic model of a small electric boat, and used it to make the boat follow lakeshores. We showed that the improvement in modeling accuracy provided by the sampling scheme also translated into increased control performance. Lastly, we applied these techniques to a real robot and showed that it performed well.

Building on the results of the MPPI, we trained an RL agent to control the boat, solving the same task as the MPPI. We modified Dreamer, a strong MBRL agent to control the boat using the laser inputs and performed zero-shot deployment from the simulation to the real system. We then modified this algorithm to better fit mobile robots. We created a system with two sub-states, one for the robot dynamics, one for the environment. We showed that our modified version of dreamer performed better than the vanilla version and could transfer knowledge between different systems. Finally, we further modified this algorithm, allowing the agent to reach imagined goals. Efficiently teaching the agent to reach multiple goals.

6.2 Perspective and Future Work

As suggested in the conclusions of the different chapters, estimating the uncertainty on the neural networks' predictions could be a very interesting research direction. Indeed, for NNs to be applied to real-world problems, we need to know when they are failing. However, as of today, few methods allow for accurate measurement of the networks' uncertainty. Furthermore, when using sampling-based MPC controllers, such as the MPPI, this uncertainty could be leveraged to regulate the variance of the sampling process.

Regarding the RL, there are many research opportunities. We believe that the most exciting is adversarial environment generation. When training an agent, the environment is often designed by a human and contains many biases regarding the difficulty of the task. However, what a human considers hard, is not necessarily hard for an RL agent. Moreover, an agent provided with a curriculum may be able to solve tasks that it could not have solved without said curriculum. Automatizing the generation process of the environments to create such curriculums could be of tremendous benefit to the robotics learning community.

6.3 acknowledgement

The research presented in this thesis was supported by two grants. One from the Ministry of Science and Technology (MOST), Israel, under the France-Israel Maimonide Program, & Ministry of Europe and Foreign Affairs (MEAE), and the Ministry of Higher Education, Research and Innovation (MESRI) of France. Another from the French Agence Nationale de la Recherche (ANR) under the reference number ANR-19-CE10-0011.

REFERENCES

[1] A. Richard, L. Fine, O. Rozenstein, J. Tanny, M. Geist, and C. Pradalier, “Filling gaps in micro-meteorological data,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2020, pp. 101–117.

[2] A. Richard, S. Aravecchia, T. Schillaci, M. Geist, and C. Pradalier, “How to train your heron,” *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5247–5252, 2021.

[3] A. Richard, S. Aravecchia, M. Geist, and C. Pradalier, “Learning behaviors through physics-driven latent imagination,” in *5th Annual Conference on Robot Learning*, 2021.

[4] O. Nelles, *Nonlinear System Identification: From Classical Approaches to Neural Networks, Fuzzy Models, and Gaussian Processes*. Springer Nature, 2020.

[5] R. Moradi, R. Berangi, and B. Minaei, “A survey of regularization strategies for deep models,” *Artificial Intelligence Review*, vol. 53, no. 6, pp. 3947–3986, 2020.

[6] S. Ghosh-Dastidar and H. Adeli, “Spiking neural networks,” *International journal of neural systems*, vol. 19, no. 04, pp. 295–308, 2009.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[9] K. Cho *et al.*, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.

[10] D. Misra, “Mish: A self regularized non-monotonic activation function,” *arXiv preprint arXiv:1908.08681*, 2019.

[11] P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for activation functions,” *arXiv preprint arXiv:1710.05941*, 2017.

[12] D. Hendrycks and K. Gimpel, “Gaussian error linear units (gelus),” *arXiv preprint arXiv:1606.08415*, 2016.

[13] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[14] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.

[15] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, 2013, p. 3.

[16] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," in *Advances in neural information processing systems*, 2017, pp. 971–980.

[17] X. Jin, C. Xu, J. Feng, Y. Wei, J. Xiong, and S. Yan, "Deep learning with s-shaped rectified linear activation units," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[18] B. Carlile, G. Delamarter, P. Kinney, A. Marti, and B. Whitney, "Improving deep learning by inverse square root linear units (isrlus)," *arXiv preprint arXiv:1710.09967*, 2017.

[19] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.

[20] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," *arXiv preprint arXiv:1906.02691*, 2019.

[21] ——, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[23] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[24] 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.

[25] G. Williams *et al.*, "Information theoretic mpc for model-based reinforcement learning,"

[26] R. Isermann, S. Ernst, and O. Nelles, "Identification with dynamic neural networks-architectures, comparisons, applications," *IFAC Proceedings Volumes*, vol. 30, no. 11, pp. 947–972, 1997.

[27] Y. LeCun, Y. Bengio, *et al.*, “Convolutional networks for images, speech, and time series,” *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[28] A. v. d. Oord *et al.*, “Wavenet: A generative model for raw audio,” *arXiv preprint arXiv:1609.03499*, 2016.

[29] S. Genc, “Parametric system identification using deep convolutional neural networks,” in *2017 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2017, pp. 2112–2119.

[30] Z. C. Lipton, J. Berkowitz, and C. Elkan, “A critical review of recurrent neural networks for sequence learning,” *arXiv preprint arXiv:1506.00019*, 2015.

[31] J. Gonzalez and W. Yu, “Non-linear system modeling using lstm neural networks,” *IFAC-PapersOnLine*, vol. 51, no. 13, pp. 485–489, 2018.

[32] A. Brusaferri, M. Matteucci, P. Portolani, and S. Spinelli, “Nonlinear system identification using a recurrent network in a bayesian framework,” in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, IEEE, vol. 1, 2019, pp. 319–324.

[33] Y. Wang, “A new concept using lstm neural networks for dynamic system identification,” in *2017 American Control Conference (ACC)*, IEEE, 2017, pp. 5324–5329.

[34] A. Rehmer and A. Kroll, “On using gated recurrent units for nonlinear system identification,” in *2019 18th European Control Conference (ECC)*, IEEE, 2019, pp. 2504–2509.

[35] X. Xie, B. Wang, T. Wan, and W. Tang, “Multivariate abnormal detection for industrial control systems using 1d cnn and gru,” *IEEE Access*, vol. 8, pp. 88 348–88 359, 2020.

[36] F. Yu and V. Koltun, “Multi-scale context aggregation by dilated convolutions,” *ICLR*, 2016.

[37] A. Vaswani *et al.*, “Attention is all you need,” in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[39] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, and S. Ho, "Lagrangian neural networks," in *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*, 2020.

[40] S. Greydanus, M. Dzamba, and J. Yosinski, "Hamiltonian neural networks," 2019.

[41] M. Lutter, C. Ritter, and J. Peters, "Deep lagrangian networks: Using physics as model prior for deep learning," in *International Conference on Learning Representations*, 2018.

[42] J. Qiao, G. Wang, W. Li, and X. Li, "A deep belief network with plsr for nonlinear system modeling," *Neural Networks*, vol. 104, pp. 68–79, 2018.

[43] J. Qiao, G. Wang, X. Li, and W. Li, "A self-organizing deep belief network for nonlinear system modeling," *Applied Soft Computing*, vol. 65, pp. 170–183, 2018.

[44] A. Amini, W. Schwarting, A. Soleimany, and D. Rus, "Deep evidential regression," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[45] D. Hafner, D. Tran, T. Lillicrap, A. Irpan, and J. Davidson, "Noise contrastive priors for functional uncertainty," in *Uncertainty in Artificial Intelligence*, PMLR, 2020, pp. 905–914.

[46] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *international conference on machine learning*, PMLR, 2016, pp. 1050–1059.

[47] S. Srinivasan, I. Sa, A. Zyner, V. Reijgwart, M. I. Valls, and R. Siegwart, "End-to-end velocity estimation for autonomous racing," *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 6869–6875, 2020.

[48] Y. Gal, J. Hron, and A. Kendall, "Concrete dropout," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 3584–3593.

[49] A. Kendall and Y. Gal, "What uncertainties do we need in bayesian deep learning for computer vision?" In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 5580–5590.

[50] A. Amini, A. Soleimany, S. Karaman, and D. Rus, "Spatial uncertainty sampling for end-to-end control," *arXiv preprint arXiv:1805.04829*, 2018.

[51] M. Sensoy, L. Kaplan, and M. Kandemir, "Evidential deep learning to quantify classification uncertainty," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018, pp. 3183–3193.

[52] T. Joo, U. Chung, and M.-G. Seo, "Being bayesian about categorical probability," in *International Conference on Machine Learning*, PMLR, 2020, pp. 4950–4961.

[53] M. Hein, M. Andriushchenko, and J. Bitterwolf, "Why relu networks yield high-confidence predictions far away from the training data and how to mitigate the problem," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 41–50.

[54] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, "Revisiting unreasonable effectiveness of data in deep learning era," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 843–852.

[55] S. Schaal, C. G. Atkeson, and S. Vijayakumar, "Real-time robot learning with locally weighted statistical learning," in *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, IEEE, vol. 1, 2000, pp. 288–293.

[56] A. Mahé, C. Pradalier, and M. Geist, "Trajectory-control using deep system identication and model predictive control for drone control under uncertain load.," in *2018 22nd International Conference on System Theory, Control and Computing (ICSTCC)*, Oct. 2018, pp. 753–758.

[57] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[58] A. Katharopoulos and F. Fleuret, "Biased importance sampling for deep neural network training," *CoRR*, vol. abs/1706.00043, 2017. arXiv: 1706.00043.

[59] ——, "Not all samples are created equal: Deep learning with importance sampling," *CoRR*, vol. abs/1803.00942, 2018. arXiv: 1803.00942.

[60] K. Kitamori, T. Manders, R. Dellink, and A. Tabeau, "Oecd environmental outlook to 2050: The consequences of inaction," OECD, Tech. Rep., 2012.

[61] FAO, "Faostat: Food and agriculture organization of the united nations-statistics division," 2019.

[62] M. Peet, "Physiological disorders in tomato fruit development," in *International Symposium on Tomato in the Tropics 821*, 2008, pp. 151–160.

[63] R. G. Allen, L. S. Pereira, D. Raes, M. Smith, *et al.*, "Crop evapotranspiration-guidelines for computing crop water requirements-fao irrigation and drainage paper 56," 1998.

[64] O. Rozenstein, N. Haymann, G. Kaplan, and J. Tanny, "Estimating cotton water consumption using a time series of sentinel-2 imagery," *Agricultural water management*, vol. 207, pp. 44–52, 2018.

[65] ——, "Validation of the cotton crop coefficient estimation model based on sentinel-2 imagery and eddy covariance measurements," *Agricultural Water Management*, vol. 223, p. 105 715, 2019.

[66] G. Kaplan *et al.*, "Estimating processing tomato water consumption, leaf area index, and height using sentinel-2 and venμs imagery," *Remote Sensing*, vol. 13, no. 6, p. 1046, 2021.

[67] V. Manivasagam, G. Kaplan, and O. Rozenstein, "Developing transformation functions for venμs and sentinel-2 surface reflectance over israel," *Remote Sensing*, vol. 11, no. 14, p. 1710, 2019.

[68] G. Tmušić *et al.*, "Current practices in uas-based environmental monitoring," *Remote Sensing*, vol. 12, no. 6, p. 1001, 2020.

[69] H. Aasen, E. Honkavaara, A. Lucieer, and P. J. Zarco-Tejada, "Quantitative remote sensing at ultra-high resolution with uav spectroscopy: A review of sensor technology, measurement procedures, and data correction workflows," *Remote Sensing*, vol. 10, no. 7, p. 1091, 2018.

[70] N. Ohana-Levi *et al.*, "A weighted multivariate spatial clustering model to determine irrigation management zones," *Computers and Electronics in Agriculture*, vol. 162, pp. 719–731, 2019.

[71] M. Romero, Y. Luo, B. Su, and S. Fuentes, "Vineyard water status estimation using multispectral imagery from an uav platform and machine learning algorithms for irrigation scheduling management," *Computers and electronics in agriculture*, vol. 147, pp. 109–117, 2018.

[72] M. Reichstein, G. Camps-Valls, B. Stevens, M. Jung, J. Denzler, N. Carvalhais, *et al.*, "Deep learning and process understanding for data-driven earth system science," *Nature*, vol. 566, no. 7743, pp. 195–204, 2019.

[73] M. Kumar, N. Raghuwanshi, and R. Singh, "Artificial neural networks approach in evapotranspiration modeling: A review," *Irrigation science*, vol. 29, no. 1, pp. 11–25, 2011.

[74] D. Papale and R. Valentini, "A new assessment of european forests carbon exchanges by eddy fluxes and artificial neural network spatialization," *Global Change Biology*, vol. 9, no. 4, pp. 525–535, 2003.

[75] E. R. Coutinho, R. M. d. Silva, J. G. F. Madeira, P. R. d. O. d. Coutinho, R. A. M. Boloy, A. R. S. Delgado, *et al.*, "Application of artificial neural networks (anns) in the gap filling of meteorological time series," *Revista Brasileira de Meteorologia*, vol. 33, no. 2, pp. 317–328, 2018.

[76] S. Weksler, O. Rozenstein, N. Haish, M. Moshelion, R. Walach, and E. Ben-Dor, "A hyperspectral-physiological phenomics system: Measuring diurnal transpiration rates and diurnal reflectance," *Remote Sensing*, vol. 12, no. 9, p. 1493, 2020.

[77] M. Aubinet, T. Vesala, and D. Papale, *Eddy covariance: a practical guide to measurement and data analysis*. Springer Science & Business Media, 2012.

[78] A. M. Moffat *et al.*, "Comprehensive comparison of gap-filling techniques for eddy covariance net carbon fluxes," *Agricultural and Forest Meteorology*, vol. 147, no. 3-4, pp. 209–232, 2007.

[79] C. Moureaux *et al.*, "Eddy covariance measurements over crops," in *Eddy Covariance*, Springer, 2012, pp. 319–331.

[80] R. Rosa and J. Tanny, "Surface renewal and eddy covariance measurements of sensible and latent heat fluxes of cotton during two growing seasons," *Biosystems Engineering*, vol. 136, pp. 149–161, 2015.

[81] A. Graves, S. Fernández, and J. Schmidhuber, "Bidirectional lstm networks for improved phoneme classification and recognition," in *International Conference on Artificial Neural Networks*, Springer, 2005, pp. 799–804.

[82] E. Falge *et al.*, "Gap filling strategies for long term energy flux data sets," *Agricultural and Forest Meteorology*, vol. 107, no. 1, pp. 71–77, 2001.

[83] M. Reichstein *et al.*, "On the separation of net ecosystem exchange into assimilation and ecosystem respiration: Review and improved algorithm," *Global change biology*, vol. 11, no. 9, pp. 1424–1439, 2005.

[84] J. Lloyd and J. Taylor, "On the temperature dependence of soil respiration," *Functional ecology*, pp. 315–323, 1994.

[85] B. H. Braswell, W. J. Sacks, E. Linder, and D. S. Schimel, "Estimating diurnal to annual ecosystem parameters by synthesis of a carbon flux model with eddy covariance net ecosystem exchange observations," *Global Change Biology*, vol. 11, no. 2, pp. 335–355, 2005.

[86] A. M. Moffat, "A new methodology to interpret high resolution measurements of net carbon fluxes between terrestrial ecosystems and the atmosphere," Ph.D. dissertation, 2012.

[87] Y. Kim *et al.*, “Gap-filling approaches for eddy covariance methane fluxes: A comparison of three machine learning algorithms and a traditional method with principal component analysis,” *Global change biology*, vol. 26, no. 3, pp. 1499–1518, 2020.

[88] M. Längkvist, L. Karlsson, and A. Loutfi, “A review of unsupervised feature learning and deep learning for time-series modeling,” *Pattern Recognition Letters*, vol. 42, pp. 11–24, 2014.

[89] T. Xu *et al.*, “Evaluating different machine learning methods for upscaling evapotranspiration from flux towers to the regional scale,” *Journal of Geophysical Research: Atmospheres*, vol. 123, no. 16, pp. 8674–8690, 2018.

[90] N. Alavi, J. S. Warland, and A. A. Berg, “Filling gaps in evapotranspiration measurements for water budget studies: Evaluation of a kalman filtering approach,” *Agricultural and Forest Meteorology*, vol. 141, no. 1, pp. 57–66, 2006.

[91] N. Boudhina *et al.*, “Evaluating four gap-filling methods for eddy covariance measurements of evapotranspiration over hilly crop fields,” *Geoscientific Instrumentation, Methods and Data Systems*, vol. 7, no. 2, pp. 151–167, 2018.

[92] L. Foltnová, M. Fischer, and R. P. McGloin, “Recommendations for gap-filling eddy covariance latent heat flux measurements using marginal distribution sampling,” *Theoretical and Applied Climatology*, vol. 139, no. 1, pp. 677–688, 2020.

[93] A. Dosovitskiy *et al.*, “An image is worth 16x16 words: Transformers for image recognition at scale,” in *International Conference on Learning Representations*, 2020.

[94] Z. Liu *et al.*, “Swin transformer: Hierarchical vision transformer using shifted windows,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 10 012–10 022.

[95] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, “Convolutional sequence to sequence learning,” in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, 2017, pp. 1243–1252.

[96] A. Richard, A. Mahé, C. Pradalier, O. Rozenstein, and M. Geist, “A comprehensive benchmark of neural networks for system identification,” 2019.

[97] M. Lange, B. Dechant, C. Rebmann, M. Vohland, M. Cuntz, and D. Doktor, “Validating modis and sentinel-2 ndvi products at a temperate deciduous forest site using two independent ground-based sensors,” *Sensors*, vol. 17, no. 8, p. 1855, 2017.

[98] T. Wutzler *et al.*, “Basic and extensible post-processing of eddy covariance flux data with reddyproc,” *Biogeosciences*, vol. 15, no. 16, pp. 5015–5030, 2018.

[99] V. T. Ambas, E. Baltas, *et al.*, “Sensitivity analysis of different evapotranspiration methods using a new sensitivity coefficient,” *Global NEST Journal*, vol. 14, no. 3, pp. 335–343, 2012.

[100] M. Reichstein, A. Moffat, T. Wutzler, and K. Sickel, “Reddyproc: Data processing and plotting utilities of (half-) hourly eddy-covariance measurements,” *R package version 0.6–0/r9*, vol. 755, 2014.

[101] G. Alain, A. Lamb, C. Sankar, A. Courville, and Y. Bengio, “Variance reduction in sgd by distributed importance sampling,” *arXiv preprint arXiv:1511.06481*, 2015.

[102] L. Ljung, “System identification,” in *Signal analysis and prediction*, Springer, 1998, pp. 163–173.

[103] B. De Moor, P. De Gersem, B. De Schutter, and W. Favoreel, “Daisy: A database for identification of systems,” *JOURNAL A*, vol. 38, pp. 4–5, 1997.

[104] V. A. Akpan and G. D. Hassapis, “Nonlinear model identification and adaptive model predictive control using neural networks,” *ISA transactions*, vol. 50, no. 2, pp. 177–194, 2011.

[105] M. Abadi *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[106] I. Loshchilov and F. Hutter, “Online batch selection for faster training of neural networks,” *arXiv preprint arXiv:1511.06343*, 2015.

[107] N. Koenig and A. Howard, “Design and use paradigms for gazebo, an open-source multi-robot simulator,” in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, IEEE, vol. 3, pp. 2149–2154.

[108] M. Quigley *et al.*, “Ros: An open-source robot operating system,” in *ICRA Workshop on Open Source Software*, 2009.

[109] S. Yaghoubi, N. A. Akbarzadeh, S. S. Bazargani, S. S. Bazargani, M. Bamizan, and M. I. Asl, “Autonomous robots for agricultural tasks and farm assignment and future trends in agro robots,” *International Journal of Mechanical and Mechatronics Engineering*, vol. 13, no. 3, pp. 1–6, 2013.

[110] D. Lattanzi and G. Miller, "Review of robotic infrastructure inspection systems," *Journal of Infrastructure Systems*, vol. 23, no. 3, p. 04 017 004, 2017.

[111] G. Pannocchia, "Offset-free tracking mpc: A tutorial review and comparison of different formulations," in *Control Conference (ECC), 2015 European*, IEEE, 2015, pp. 527–532.

[112] T. Zhang, G. Kahn, S. Levine, and P. Abbeel, "Learning Deep Control Policies for Autonomous Aerial Vehicles with MPC-Guided Policy Search," *ArXiv e-prints*, Sep. 2015. arXiv: 1509.06791 `[cs.LG]`.

[113] S. J. Qin and T. A. Badgwell, "A survey of industrial model predictive control technology," *Control engineering practice*, vol. 11, no. 7, pp. 733–764, 2003.

[114] G. Williams *et al.*, "Information theoretic mpc for model-based reinforcement learning," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 1714–1721.

[115] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Aggressive driving with model predictive path integral control," in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, IEEE, 2016, pp. 1433–1440.

[116] T. Naegeli, J. Alonso-Mora, A. Domahidi, D. Rus, and O. Hilliges, "Real-time motion planning for aerial videography with dynamic obstacle avoidance and viewpoint optimization," *IEEE Robotics and Automation Letters*, vol. 2, no. 3, pp. 1696–1703, Jul. 2017.

[117] J. Dentler, S. Kannan, M. A. O. Mendez, and H. Voos, "A tracking error control approach for model predictive position control of a quadrotor with time varying reference," in *Robotics and Biomimetics (ROBIO), 2016 IEEE International Conference on*, IEEE, 2016, pp. 2051–2056.

[118] R. Lenain, B. Thuilot, C. Cariou, and P. Martinet, "High accuracy path tracking for vehicles in presence of sliding: Application to farm vehicle automatic guidance for agricultural tasks," *Autonomous robots*, vol. 21, no. 1, pp. 79–97, 2006.

[119] E. Lucet, R. Lenain, and C. Grand, "Dynamic path tracking control of a vehicle on slippery terrain," *Control engineering practice*, vol. 42, pp. 60–73, 2015.

[120] A. Malinin and M. Gales, "Predictive uncertainty estimation via prior networks," in *Advances in Neural Information Processing Systems*, 2018, pp. 7047–7058.

[121] M. P. Deisenroth, G. Neumann, and J. Peters, *A survey on policy search for robotics*. now publishers, 2013.

[122] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi, "Dream to control: Learning behaviors by latent imagination," in *ICLR*, 2020.

[123] R. S. Sutton, A. G. Barto, *et al.*, "Introduction to reinforcement learning," 1998.

[124] D. P. Bertsekas and J. N. Tsitsiklis, "Neuro-dynamic programming: An overview," in *Proceedings of 1995 34th IEEE conference on decision and control*, IEEE, vol. 1, 1995, pp. 560–564.

[125] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 4, no. 1, pp. 1–103, 2010.

[126] O. Sigaud and O. Buffet, *Markov decision processes in artificial intelligence*. John Wiley & Sons, 2013.

[127] C. Finn and S. Levine, "Deep visual foresight for planning robot motion," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 2786–2793.

[128] D. Hafner *et al.*, "Learning latent dynamics for planning from pixels," in *ICML*, 2019.

[129] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, "Mastering atari with discrete world models," *arXiv preprint arXiv:2010.02193*, 2020.

[130] J. Schrittwieser *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[131] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, Nov. 2017.

[132] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," in *ICML*, 2016.

[133] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy MaxEnt deep RL with a stochastic actor," in *ICML*, 2018.

[134] P. Cai, X. Mei, L. Tai, Y. Sun, and M. Liu, "High-speed autonomous drifting with deep reinforcement learning," *IEEE RA-L*, 2020.

[135] Z.-W. Hong *et al.*, "Virtual-to-real: Learning to control in visual semantic segmentation," in *IJCAI*, 2018.

[136] M. Andrychowicz *et al.*, “Learning dexterous in-hand manipulation,” *IJRR*, vol. 39, no. 1, pp. 3–20, 2020.

[137] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, “Sim-to-real transfer of robotic control with dynamics randomization,” in *ICRA*, 2018.

[138] Y. Chebotar *et al.*, “Closing the sim-to-real loop: Adapting simulation randomization with real world experience,” in *ICRA*, 2019.

[139] R. C. Arkin, *Behavior-Based Robotics*. MIT Press, 1998.

[140] A. Ram, R. C. Arkin, R. J. Clark, and K. Moorman, “Case-based reactive navigation: A case based method for on-line selection and adaptation of reactive control parameters in autonomous robotics systems,” *IEEE Transactions on SMC*, 1992.

[141] J. Kober, J. A. Bagnell, and J. Peters, “Reinforcement learning in robotics: A survey,” *International Journal of Robotics Research*, 2013.

[142] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[143] G. Barth-Maron *et al.*, “Distributed distributional deterministic policy gradients,” in *ICLR*, 2018.

[144] A. X. Lee, A. Nagabandi, P. Abbeel, and S. Levine, “Stochastic latent actor-critic: Deep reinforcement learning with a latent variable model,” *arXiv*, 2019.

[145] F. Sadeghi and S. Levine, “Cad2rl: Real single-image flight without a single real image,” in *RSS*, 2017.

[146] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, “Domain randomization for transferring deep neural networks from simulation to the real world,” in *IROS*, 2017.

[147] J. Tan *et al.*, “Sim-to-real: Learning agile locomotion for quadruped robots,” in *RSS*, 2018.

[148] J. Hwangbo *et al.*, “Learning agile and dynamic motor skills for legged robots,” *Science Robotics*, vol. 4, no. 26, 2019.

[149] A. Piergiovanni, A. Wu, and M. S. Ryoo, “Learning real-world robot policies by dreaming,” in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2019, pp. 7680–7687.

[150] A. Richard, S. Aravecchia, T. Schillaci, M. Geist, and C. Pradalier, "How to train your heron," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5247–5252, 2021.

[151] R. Kidambi, A. Rajeswaran, P. Netrapalli, and T. Joachims, "Morel: Model-based offline reinforcement learning," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 21 810–21 823.

[152] T. Yu *et al.*, "Mopo: Model-based offline policy optimization," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 14 129–14 142.

[153] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[154] G. Brockman *et al.*, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[155] Y. Tassa *et al.*, "Deepmind control suite," *arXiv preprint arXiv:1801.00690*, 2018.

[156] A. Brunnbauer *et al.*, "Model-based versus model-free deep reinforcement learning for autonomous racing cars," *arXiv preprint arXiv:2103.04909*, 2021.

[157] P. Becker-Ehmck, M. Karl, J. Peters, and P. van der Smagt, "Learning to fly via deep model-based reinforcement learning," *arXiv preprint arXiv:2003.08876*, 2020.

[158] M. Deisenroth and C. E. Rasmussen, "Pilco: A model-based and data-efficient approach to policy search," in *Proceedings of the 28th International Conference on machine learning (ICML-11)*, Citeseer, 2011, pp. 465–472.

[159] P. Abbeel, A. Coates, M. Quigley, and A. Ng, "An application of reinforcement learning to aerobatic helicopter flight," in *Advances in Neural Information Processing Systems*, B. Schölkopf, J. Platt, and T. Hoffman, Eds., vol. 19, MIT Press, 2007.

[160] A. Plaat, W. Kosters, and M. Preuss, "Model-based deep reinforcement learning for high-dimensional problems, a survey," *arXiv preprint arXiv:2008.05598*, 2020.

[161] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning.," in *ICLR (Poster)*, 2016.

[162] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[163] Y. D. Zhong and N. Leonard, "Unsupervised learning of lagrangian dynamics from images for prediction and control," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[164] A. Sanchez-Gonzalez *et al.*, "Graph networks as learnable physics engines for inference and control," in *International Conference on Machine Learning*, PMLR, 2018, pp. 4470–4479.

[165] P. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, *et al.*, "Interaction networks for learning about objects, relations and physics," in *Advances in Neural Information Processing Systems*, 2016, pp. 4502–4510.

[166] D. Fox, S. Thrun, and W. Burgard, *Probabilistic Robotics*. Kybernetes, 2006, ISBN: 9780262201629.

[167] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi, "Dream to control: Learning behaviors by latent imagination," in *International Conference on Learning Representations*, 2020.

[168] M. Andrychowicz *et al.*, "Hindsight experience replay," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 5055–5065.

[169] R. Mendonca, O. Rybkin, K. Daniilidis, D. Hafner, and D. Pathak, "Discovering and achieving goals via world models," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[170] T. M. Moerland, J. Broekens, and C. M. Jonker, "Model-based reinforcement learning: A survey," *arXiv preprint arXiv:2006.16712*, 2020.

[171] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM Sigart Bulletin*, vol. 2, no. 4, pp. 160–163, 1991.

[172] D. Silver *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[173] A. S. Polydoros and L. Nalpantidis, "Survey of model-based reinforcement learning: Applications on robotics," *Journal of Intelligent & Robotic Systems*, vol. 86, no. 2, pp. 153–173, 2017.

[174] D. Yarats, I. Kostrikov, and R. Fergus, "Image augmentation is all you need: Regularizing deep reinforcement learning from pixels," in *International Conference on Learning Representations*, 2021.

[175] D. Yarats, R. Fergus, A. Lazaric, and L. Pinto, “Mastering visual continuous control: Improved data-augmented reinforcement learning,” *arXiv preprint arXiv:2107.09645*, 2021.

[176] D. P. Kingma, S. Mohamed, D. J. Rezende, and M. Welling, “Semi-supervised learning with deep generative models,” in *Advances in neural information processing systems*, 2014, pp. 3581–3589.

[177] M. Okada and T. Taniguchi, “Dreaming: Model-based reinforcement learning by latent imagination without reconstruction,” *arXiv preprint arXiv:2007.14535*, 2020.

[178] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “CARLA: An open urban driving simulator,” in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[179] Y. Teh *et al.*, “Distral: Robust multitask reinforcement learning,” in *Advances in Neural Information Processing Systems*, I. Guyon *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017.

[180] L. P. Kaelbling, “Learning to achieve goals,” in *IJCAI*, Citeseer, 1993, pp. 1094–1099.

[181] T. Schaul, D. Horgan, K. Gregor, and D. Silver, “Universal value function approximators,” in *International conference on machine learning*, PMLR, 2015, pp. 1312–1320.

[182] M. Plappert *et al.*, “Multi-goal reinforcement learning: Challenging robotics environments and request for research,” *arXiv preprint arXiv:1802.09464*, 2018.

[183] V. Pong, S. Gu, M. Dalal, and S. Levine, “Temporal difference models: Model-free deep RL for model-based control,” in *International Conference on Learning Representations*, 2018.

[184] F. Ebert, C. Finn, S. Dasari, A. Xie, A. Lee, and S. Levine, “Visual foresight: Model-based deep reinforcement learning for vision-based robotic control,” *arXiv preprint arXiv:1812.00568*, 2018.

[185] D. Pathak *et al.*, “Zero-shot visual imitation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2018, pp. 2050–2053.

[186] A. V. Nair, V. Pong, M. Dalal, S. Bahl, S. Lin, and S. Levine, “Visual reinforcement learning with imagined goals,” *Advances in Neural Information Processing Systems*, vol. 31, pp. 9191–9200, 2018.

[187] V. H. Pong, M. Dalal, S. Lin, A. Nair, S. Bahl, and S. Levine, “Skew-fit: State-covering self-supervised reinforcement learning,” *arXiv preprint arXiv:1903.03698*, 2019.

[188] E. Chane-Sane, C. Schmid, and I. Laptev, “Goal-conditioned reinforcement learning with imagined subgoals,” in *International Conference on Machine Learning*, PMLR, 2021, pp. 1430–1440.

[189] R. Wang, J. Lehman, J. Clune, and K. O. Stanley, “Poet: Open-ended coevolution of environments and their optimized solutions,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO ’19, Prague, Czech Republic: Association for Computing Machinery, 2019, pp. 142–151, ISBN: 9781450361118.

[190] R. Wang *et al.*, “Enhanced poet: Open-ended reinforcement learning through unbounded invention of learning challenges and their solutions,” in *International Conference on Machine Learning*, PMLR, 2020, pp. 9940–9951.

[191] M. Dennis *et al.*, “Emergent complexity and zero-shot transfer via unsupervised environment design,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 13 049–13 061, 2020.