Evaluation of prioritized deep system identification on a path following task

Antoine Mahé, Antoine Richard, Stéphanie Aravecchia, Matthieu Geist, Cedric Pradalier

JINT-D-20-00540R1

Evaluation of prioritized deep system identification on a path following task

Antoine Mahé* · **Antoine Richard*** · **Stéphanie Aravecchia** · **Matthieu Geist** · **Cédric Pradalier**

Received: 02/07/2020 / Accepted: 05/02/2021

Abstract This paper revisits system identification and shows how new paradigms from machine learning can be used to improve it in the case of non-linear systems modeling from noisy and unbalanced dataset. We show that using importance sampling schemes in system identification can provide a significant performance boost in modeling, which is helpful to a predictive controller. The performance of the approach is first evaluated on simulated data of a Unmanned Surface Vehicle (USV). Our approach consistently outperforms baseline approaches on this dataset. Moreover we demonstrate the benefits of this identification methodology in a control setting. We use the model of the USV in a Model Predictive Path Integral (MPPI) controller to perform a track following task. We discuss the influence of the controller parameters and show that the prioritized model outperform standard methods. Finally, we apply the MPPI on a real system using the know-how developed here.

This work is done under the Grande Region rObotique aerienNE (GRoNe) project, funded by a European Union Grant thought the FEDER INTERREG VA initiative and the french "Grand Est" Région.

A.Mahé*
CentraleSupélec, Université de Lorraine, CNRS, LORIA, France
E-mail: antoine-robin.mahe@centralesupelec.fr

A. Richard*
Georgia Institute of Technology, Atlanta, Georgia 30332–0250
E-mail: arichard@georgiatech-metz.fr

S. Aravecchia
UMI2958 GT-CNRS, France
E-mail: stephanie.aravecchia@georgiatech-metz.fr

M. Geist
Google Research, Brain team
E-mail: mfgeist@google.com

C. Pradalier
UMI2958 GT-CNRS, France
E-mail: cedric.pradalier@georgiatech-metz.fr

*These authors contributed equally

Keywords System identification · Machine learning · Importance sampling

1 Introduction

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 [32, 14]. 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 Model Predictive Control (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 [25, 33], and is now widely adopted in the industry [26].

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 Auto Regressive Moving Average (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 Neural-Networks (NNs) to learn over time [30] offers interesting possibilities, increasing the relevance of deep learn-

ing 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 [27]. 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 results 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 Reinforcement Learning (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 [28] suggests to focus the NNs training on the samples that are most useful to their convergence. [28,11,3] showed that using prioritization schemes, NNs are capable of learning from highly unbalanced datasets on both RL tasks and image classification tasks. In this paper, 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 study, we use the MPPI [31] controller. This controller can work with any dynamic model and has been shown to work well [29,30] coupled to NNs on robotic systems.

This paper is an extension of work originally presented in the 19th International Conference on Advanced Robotics [22]. As in the original version, this paper explores two different sampling strategy: the Prioritized Experience Replay (PER) [28] and the upper-bound gradient prioritization [12]. In this extended version, a more thorough study of the importance sampling scheme is presented, along with their application on an MPC task that was not present in [22]. Furthermore, 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.

2 Related work

For a long time ARMA [17] has been the way to go for most black-box system identification. This computationally inexpensive modeling method allows to perform efficient grid-searches over its parameters (orders of the numerator and denominator of the transfer function). Hence, it can easily be tuned to yield high quality results. However, its application is limited to linear systems. This limitation is troublesome as most robotic problems rely on non-linear systems. Yet, an increasingly popular method to fit non-linear functions is deep NNs. Over the last decade, Deep Learning has had tremendous success in numerous fields, ranging from computer vision to Natural Language Processing. Naturally, it also caught the interest of researchers in control and system identification.

Recent attempts at black-box system identification using deep learning showed great results, on both linear and non-linear systems [10,34,7]. These works rely on architectures such as Multi Layer Perceptron (MLP) and Long Short-Term Memory (LSTM) networks [8]. Unfortunately, training these networks requires a massive amount of data. While this is usually acceptable in many applications, it is particularly problematic in the field of control and robotics where acquiring a large balanced[1] dataset is difficult [27]. If the dataset is unbalanced, then the training is saturated with frequently occurring samples while the most interesting data points are less represented and have less impact on the learning process. All in all, properly learning hard cases is hindered by the overwhelming representation of the simplest cases in the data.

Despite this drawback, the NNs capacity to keep improving over new data by training continuously has been used to alleviate this problem [30]. 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.

To address these problems, prior works performed importance sampling in NN-based model identification [21]. This work relies on the PER [28] algorithm, which demonstrated great results in RL. This method is used in the Double DQN algorithm [16] and has recently been extended to support distributed architectures [9].

The PER method uses the error made by the model on examples to estimate how important those example are. Then it samples the original dataset to emphasis on the examples where the model makes mistakes. The model is then trained again on that new prioritized dataset.

When using PER, one needs to tune two parameters: α, and β. They allow for a fine control over the prioritization, which is particularly interesting as it offers to put a low amount of prioritization at the beginning of the training to grasp the general dynamic of the model, and then to increase the prioritization as the training reaches its end to maximize the learning of hard cases. However the tuning of these parameters is not easy and requires expensive grid searches.

[1] In this case, balance refers to a uniform exploration of the state-action space.

While this method shows great results, it is costly as it relies on the error of the network on each sample to compute the sampling distribution. If this distribution is not updated frequently enough, then the PER will loose in efficiency as it may over-sample already known samples. If it is updated too frequently it becomes prohibitively expensive.

More recently, a similar study has been conducted on a classification task. In particular, [11] shows that the loss of the network on a given sample could be used as an indicator of the sample's importance. However, [12] outlines that using the loss as an indicator of a sample's importance can result in degraded learning performances in some cases. 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 samples importance based on the most recent network parameters. 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. This method has been tested and compared with the PER in a prior work [22]. The main drawback of this method, is that unlike the PER it does not offer a mean to control the prioritization precisely. Instead, it relies on a single parameter: the super-batch-size. In the end, it is a question of trade-off, the PER is heavier, and sensitive to outliers. Yet, it also offers increased flexibility over how the samples are drawn and how much this unbalance in the sampling is being compensated for.

To show the interest 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 [24,5] and rovers [15,19] 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 here is a NN. Hence, in our study we use the MPPI, an MPC controller first defined in [29] and then refined in [31]. 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 work, 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. Building on the promising simulated result of [22], 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 on the different prioritization schemes. We conclude our study by applying our findings on a real system, and provide insights on the main challenges that arose when we deployed the MPPI on the field.

3 Method

In this section, we first detail the different approaches to system identification that we compare in our experiments. From the standard linear system identification and methods using Multi-Layer-Perceptron (MLP), we show how prioritized sampling can be adapted to the system identification task. Then, we detail how we use those models to achieve the control of a USV with the MPPI.

3.1 Importance sampling for deep system 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 input and output at time t, ARMA considers a model of the system given by the following discrete-time linear difference equation:

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

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

$$x(t) = -\sum_{k=1}^{p} a_k x(t-k) + \sum_{k=1}^{q} b_k u(t-k). \tag{2}$$

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

More recently, the ARMA methods have been challenged by NNs because they expand the range of systems that can be modeled from data. In particular, modeling non-linear systems and functions using NNs have been quite successful [2] even though the use of dataset collected from real robot as been challenging [27]. In this study, we rely on the well known MLP networks. Despite their apparent simplicity, these networks are capable of modeling complex non-linear functions, and, when well designed, are perfectly suitable for real time operations on embedded systems, as shown in [30].

When training a NN, two main problems related to the data occur: quantity and quality. Although the two are intertwined, the most naive approach consists in increasing the amount of data available as much as possible. Unfortunately, this may not be very helpful when trying to learn dynamic-features poorly represented in the dataset.

This is particularly true on datasets that have not been acquired for the sole purpose of model identification. Indeed, when operators use a system, they tend to stay in conditions they are comfortable with. Thus, the interesting data where the network should learn the most is the rarest. This leads to traditional learning approaches failing to train on those seldom seen events.

3.1.1 Prioritized experience replay

To address this problem, we propose to adapt the prioritization scheme introduced in [28] for reinforcement learning to the context of system identification. In fact, prioritization forces the training on harder samples even if they have poor representations in the dataset.

The adaptation of this sampling strategy to system identification yielded encouraging results illustrated in [22]. 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 ones 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 algorithm 1.

Algorithm 1 Prioritized Experience Replay

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

One of the limitations of this approach is its focus 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 because it is harder to make the distinction between complex cases and outliers. To mitigate those problems and make the method practical, two hyper-parameters are introduced: α and β. These parameters allow to choose how much the training should focus on hard cases.

The probability of choosing the sample i during the sampling is given in eq. (3) where δ_i is the score of sample i, in our case the error between the NN prediction and the actual observation.

The α hyper-parameter allows to soften the prioritization. Choosing $\alpha = 0$ gives the uniform distribution, in that case there is no prioritization at all. While a higher α value encourages learning on edge cases

$$P(i) = \frac{\delta_i^\alpha}{\sum_k \delta_k^\alpha}. \tag{3}$$

Nonetheless, the sensibility to the hyper-parameters makes the approach difficult to apply and makes a systematic tedious grid search mandatory to find optimal values for these parameters.

Additionally, the difference between the original distribution and the re-sampled distribution introduces a bias. To correct this bias, the importance sampling weights are computed during training using (4).

$$w_i = \left(\frac{1}{N} \frac{1}{P(i)} \right)^\beta. \tag{4}$$

This is where β, the second hyper-parameter for training, is used. With $\beta = 1$ the prioritized sampling bias is completely corrected but it also slows down the learning. The α parameter increases the aggressiveness of the prioritization while the β parameter increases the correction. Therefore, there is a trade-off between these parameters.

3.1.2 Gradient upper-bound

Another way to prioritize samples is to use the gradient upper-bound as explained in [12]. As suggested by its name, the gradient upper-bound method relies on an approximation of the norm of the network's gradient. [3,18] shows 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 replay 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, [12] introduces an accurate and computationally inexpensive estimation of it, which is the gradient upper-bound. It 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 hyper-parameters than prioritized experience replay. Instead of updating the weights at some arbitrary training step, or epoch, this technique samples *super-batches*, that are n-time larger 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. Since the *super-batches* are sampled uniformly, this reduces both the risk of over-fitting and the risk of focusing on outliers. The *super-batches* size is the hyper-parameter that replaces α, β and the number of replay in the previous algorithm. Depending on the size of the *super-batches*, the training time can be significantly extended. Algorithm 2 shows our implementation of the gradient prioritization scheme.

Algorithm 2 Gradient Prioritization

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

3.2 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 eq. (5), 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.

$$\begin{cases} v_{x_t} = -\sin(\theta_t)v_{\text{lin}_t} + \cos(\theta)v_{\text{lat}_t} \\ v_{y_t} = \cos(\theta_t)v_{\text{lin}_t} + \sin(\theta)v_{\text{lat}_t} \\ x_{t+1} = x_t + v_{\text{lin}_t}dt \\ y_{t+1} = y_t + v_{\text{lat}_t}dt \\ \theta_{t+1} = \theta_t + \omega_t dt \end{cases} . \tag{5}$$

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 [29], 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 [31, alg. 2]. A pseudo-code of our implementation can be seen in algorithm 3. 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 eq. (6), where v_{target} is the desired speed and v the actual speed of the USV.

$$\text{vel\_cost} = \frac{\left|v_{\text{target}} - v\right|}{0.0001 + v}. \tag{6}$$

- 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 Sec. 4.1).

In the end, the total cost is given by eq. (7), with α_1, α_2 and α_3 regulating the weights of the different components of the cost function.

$$cost = \alpha_1 \text{pos\_cost} + \alpha_2 \text{vel\_cost} + \alpha_3 \text{head\_cost}. \tag{7}$$

Algorithm 3 Model Predictive Path Integral [31]

Require: F : Dynamic model
T : number of timesteps
K : number of sampled trajectories
ϕ : 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**
    $u_t = \boldsymbol{u}_t + \varepsilon_k^t$
    $s_{t+dt} \leftarrow F(s_t, u_t)$
  end for
  $S_k = \{s_t \text{ for } t \text{ in } [0, T]\}$
  $C_k \leftarrow \phi(S_k)$
end for
$\beta \leftarrow min_k[C_k]$

$\eta \leftarrow \sum_{k=0}^{K-1} exp(-(C_k - \beta))$

for $k = 0$ to $K-1$ **do**
  $w_k \leftarrow \frac{1}{\eta} exp(-(C_k - \beta))$
end for
for $t = 1$ to T **do**
  $\boldsymbol{u}_t = \boldsymbol{u}_t + \sum_{k=1}^{K} w_k \varepsilon_k^t$
end for
return U

4 Experiments

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

4.1 Simulation Setup

We tested our approach in simulation using the Gazebo software[2], a simulator that allows creating complex simulations with custom robots and hardware. The simulation of the USV itself is done using the heron package[3] along with the uuv-simulator[4]. The first one provides a simulated version of Clearpath Robotics's Heron an USV, while the later provides advanced water buoyancy simulation, and realistic thrust non-linearities that imitates the real USV behavior. All the experiments were carried out using Robotic Operating System (ROS)[5], 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 slow changes, while the second one has abrupt orientation changes. The two tracks can be seen in figure 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 1. This is different from the implementation of the cost-map in the original MPPI [30,31] 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.

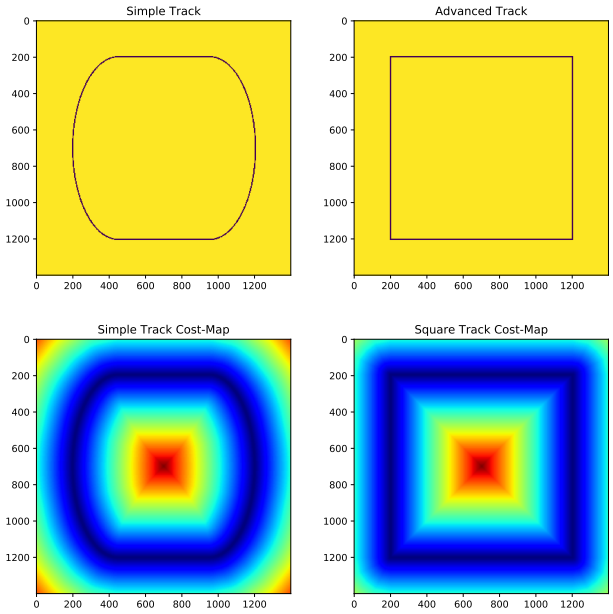

Fig. 1: The two tracks used to evaluate the different learning paradigm and parameters of the MPPI (first row), associated with their respective cost maps (second row).

4.2 Neural Networks

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

4.2.1 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

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

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.

4.2.2 Training parameters

The NNs used in this paper 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 here are Leaky-ReLUs [20], 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 [13] with a learning rate of 0.001. All the networks are trained using Tensorflow [1] version 1.15.

4.3 Evaluation

4.3.1 Networks 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 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 comprised 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 Sec. 4.2.1. 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 combinations 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 Sec. 4.2.1. 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.

4.3.2 MPPI evaluation

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 3, 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 sub-optimal 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 term 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.

5 Results

5.1 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.

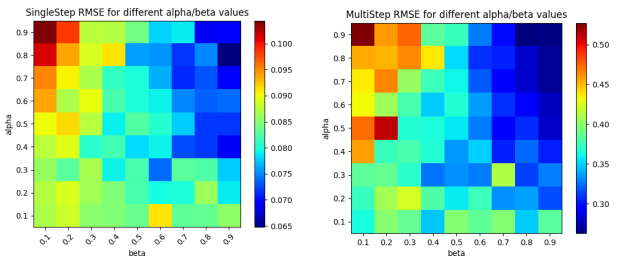

Fig. 2: The PER results on the unbalanced dataset. Left: single-step accuracy. Right: multi-step accuracy. The colder the color the lower the RMSE. The lower the RMSE the better.

Figures 2 and 3 show the grid-search results of the gradient upper-bound and the PER prioritization schemes on the

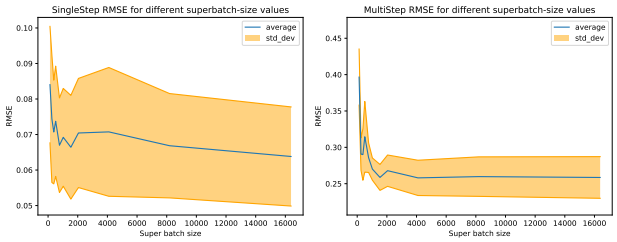

Fig. 3: The gradient upper-bound results on the unbalanced dataset. Left: single-step accuracy. Right: multi-step accuracy. The lower the RMSE the better. The narrower the orange area the better.

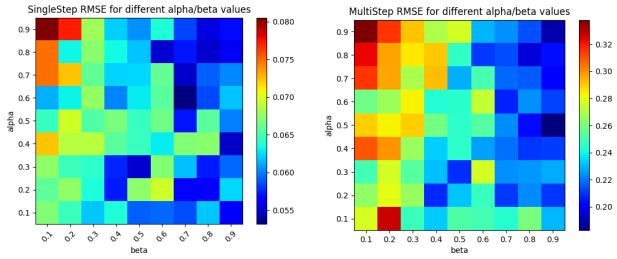

Fig. 4: The PER results on the fully random dataset. Left: single-step accuracy. Right: multi-step accuracy. The colder the color the lower the RMSE. The lower the RMSE the better.

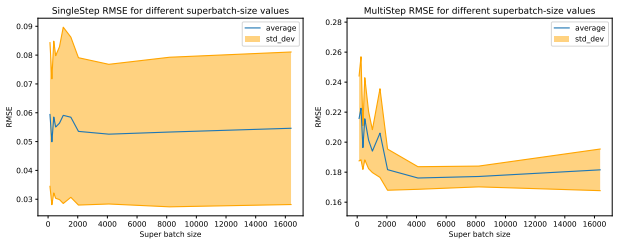

Fig. 5: The gradient upper-bound results on the fully random dataset. Left: single-step accuracy. Right: multi-step accuracy. The lower the RMSE the better. The narrower the orange area the better.

unbalanced dataset. Figure 4 and 5 show the grid-search results of the gradient upper-bound and the PER prioritization schemes on the fully random dataset.

Let us first have a look at figure 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, we can see that on the fully random dataset, the single step prediction is more homogeneous in performance, with the notable exception of select-

	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

Table 1: 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.

ing 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 full random dataset (figure 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 1.

We can now move on to figure 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 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 1 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 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.

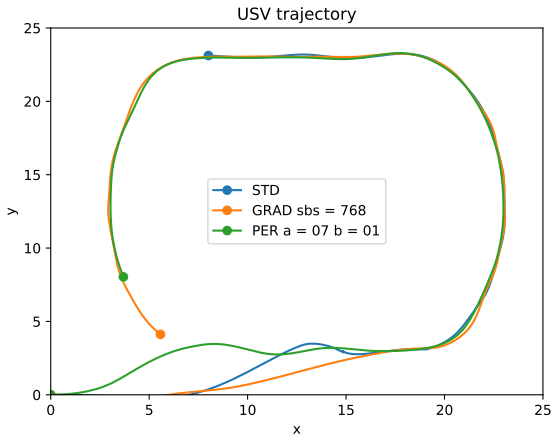

Fig. 6: Comparison of standard neural network (STD) and prioritized (PER/GRAD) version on a composite track

5.2 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 Sec. 4.3.2, the results reported in the figures are averaged over 15 runs.

5.2.1 Comparing the different schemes on the simple track

First, the controller is tested in 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 6 shows how the different learning schemes performed after 670 time-steps of 0.2 seconds. The models shown are 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.

5.2.2 Comparing the different models on the advanced track

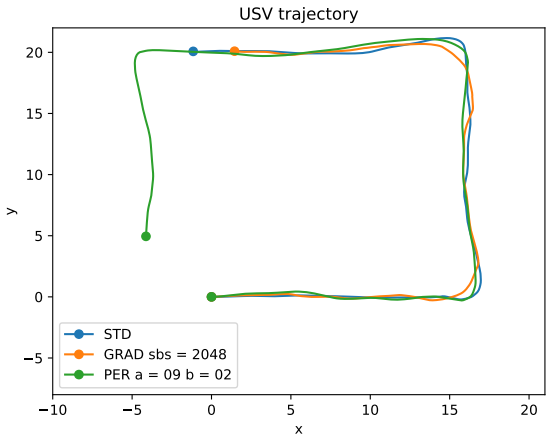

Fig. 7: Comparison of standard neural network (STD) and prioritized (PER/GRAD) version on a square track

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 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 with the lowest map-cost, showing that it respects the track better. The detail of the costs is as follow : 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.

5.2.3 Study of the number of samples impact on the performance

Figure 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, and

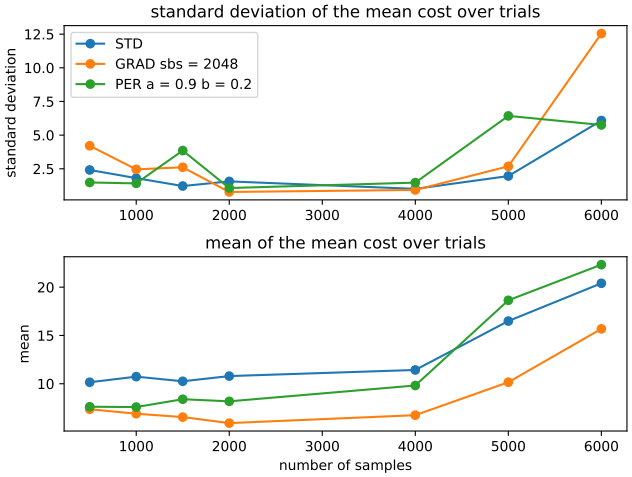

Fig. 8: Average map-cost and variance over several trials for different number of sample trajectories. Lower is better.

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 7, we see that the PER is faster than the gradient. However, figure 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 7 is on the best run only while figure 8 show the results over 15 trials.

5.2.4 Study of the number of time-steps impact on the performance

Figure 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 costs 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 term 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 Sec. 5.2.

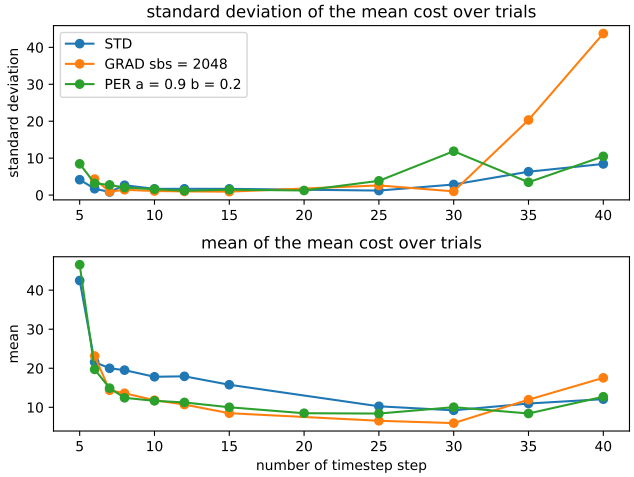

Fig. 9: Average map-cost and variance over several trials for different number of time-steps per trajectory

5.2.5 *Study of the number of the sampling variance*

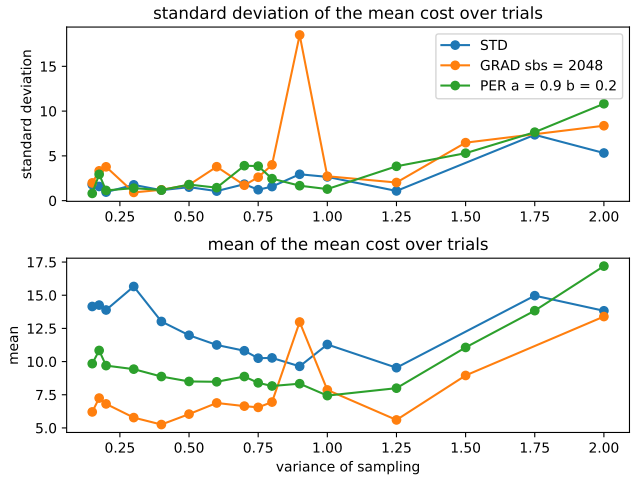

Fig. 10: Average map-cost and variance over several trials for different sampling variance

In figure 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 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 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

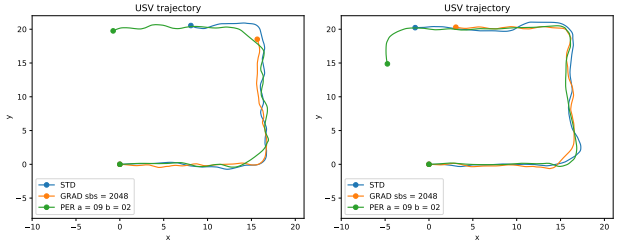

Fig. 11: Snapshot of the robot position (dot) and their trajectory (line). Command sampling variance at 0.15: left; and variance at 2.0: right.

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 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 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 term 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.

5.3 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.

Let us start by analyzing the impact of the number of times-steps on the the mean velocity around the track. From figure 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 of the track. This behavior can be seen in figures 9, 8, 10 where the standard model almost always has a higher map-cost than the prioritized models. Similar behavior can be seen on figures 13, 14 where the standard model is constantly faster than the prioritized approach. In the end, while the standard model goes faster on average,

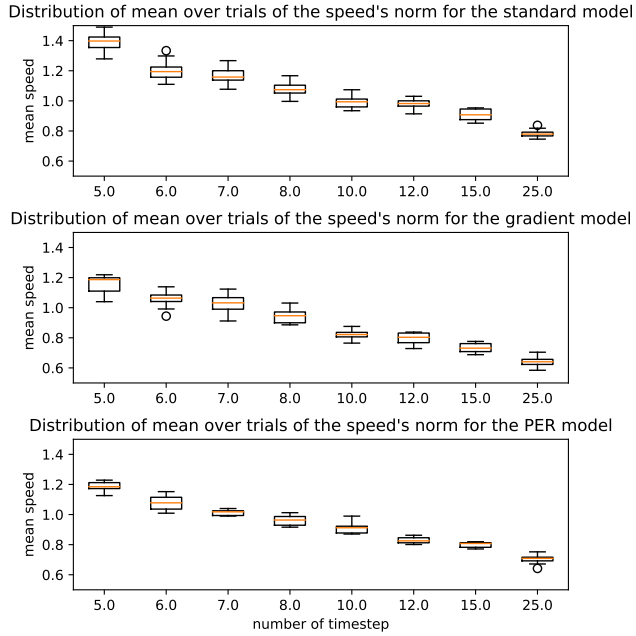

Fig. 12: Average velocity over several trials for different number of time-steps. Higher is better.

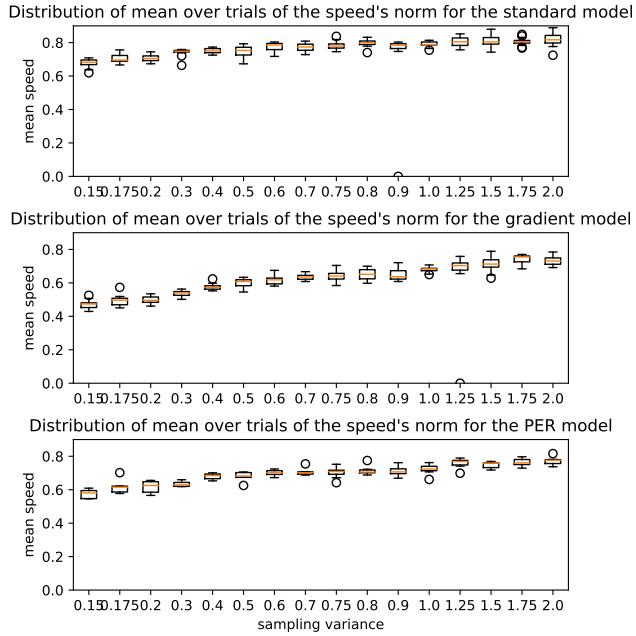

Fig. 13: Average velocity over several trials for different values of variance. Higher is better.

this extra velocity is misused and leads to worst performance on the track-following task.

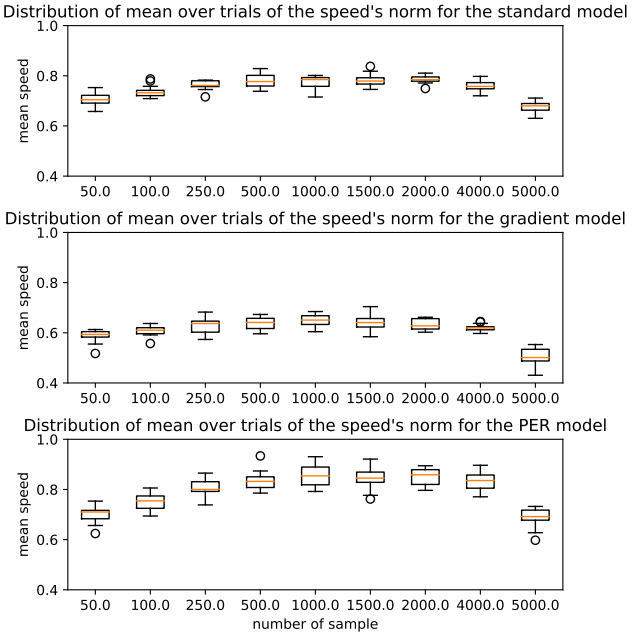

Fig. 14: Average velocity over several trials for different number of samples. Higher is better.

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.

6.1 Problem definition and setup

In this subsection we used the know-how presented previously to make a USV autonomously follow a lake shore. 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 15, shows our real system and the test environment.

Unlike the previous section 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. 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.

Fig. 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)

An illustration of the system and its task can be found in figure 16.

In the end, the cost function that we used to solve this task is similar to the one shown in (7), 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 6.2. Finally we trained our NN using a PER scheme on a dataset collected in the real environment with the method described in 4.2.1. For this dataset, we also used a grid-search to find the optimal PER parameters.

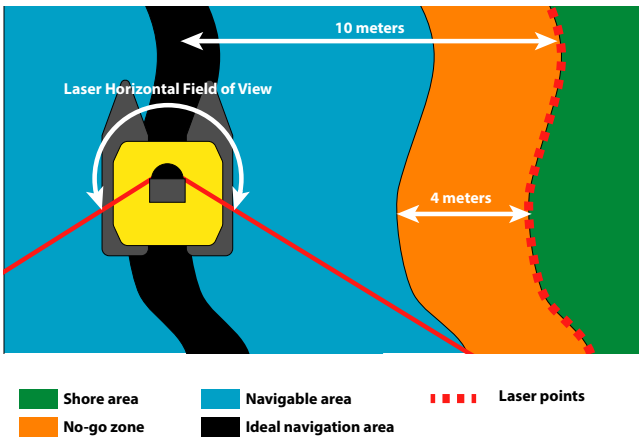

Fig. 16: The USV and its shore-following task. Black is a zero cost area, blue has a gradually increasing cost as we move away from the track, orange has a positive cost of 500, green is collision and has a positive cost of 10000.

6.2 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 of the parameters. Figure 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.

7 Conclusion

Following the work done in [22], this paper looks at the interest of prioritization in training for deep model identification. After showing the advantage of our method in the modeling of the USV used for our experiment, we show that it translates into good performance on a track-following task for two different tracks. Then, we study the reliability of the controller for the different models depending on the algorithm parameters. We show that the prioritization improves the result, in particular, the gradient-based method outperforms other methods in most cases. Even though these methods bring significant performance boost, both the PER

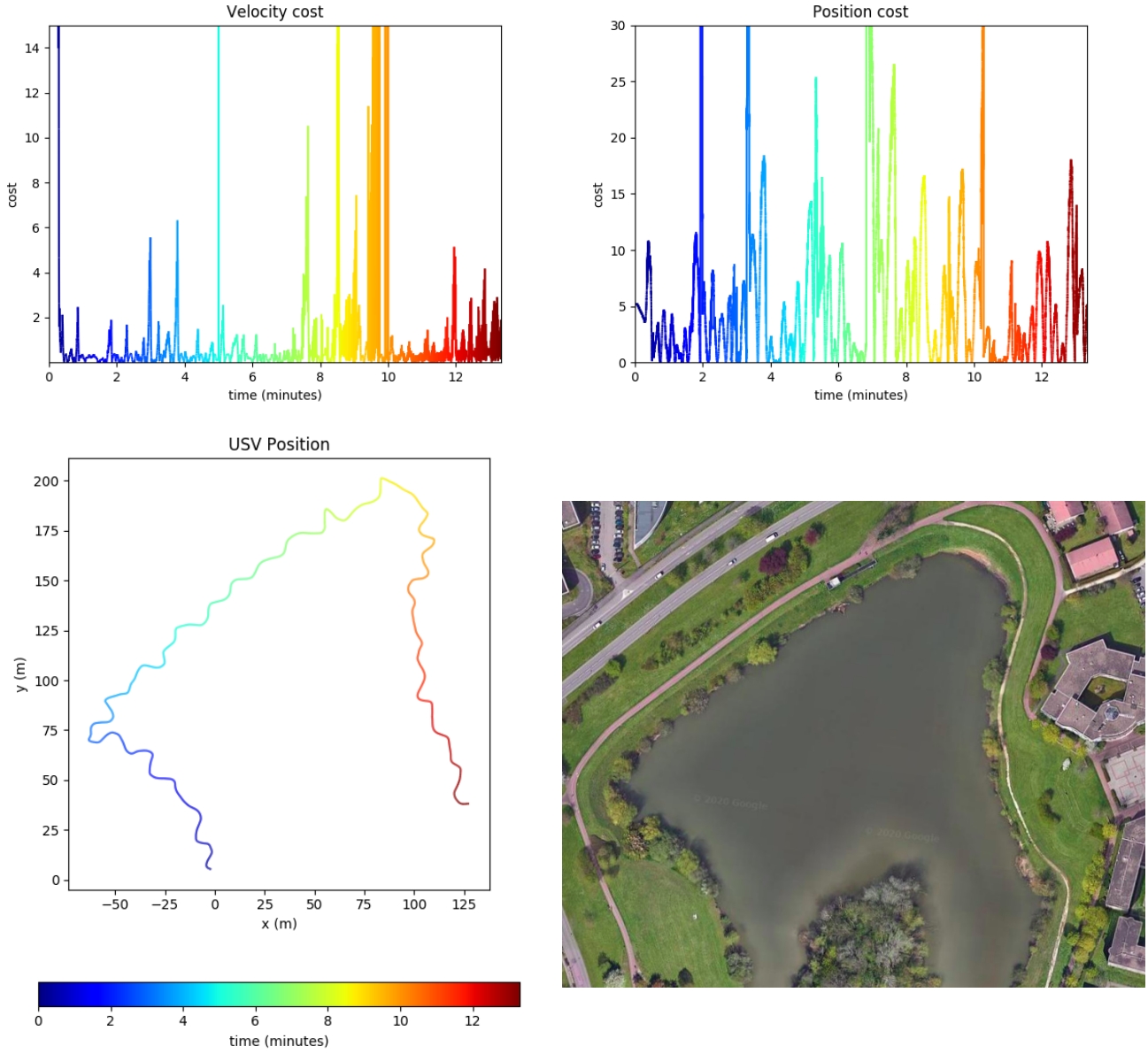

Fig. 17: Top: cost of the USV along the trajectory shown on the bottom left. For the cost. A lower cost indicates a better performance. The colors on the three plots match so the costs can be associated easily to the position. On the bottom right, a satellite image of the lake is given.

and gradient upper-bound require grid-searches to work optimally. This is particularly true on datasets where there are outliers to which the PER is particularly 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 [4,6,23] show promising results. In future research, we are planning to adapt the MPPI by replacing the current NN inside it by Evidential Networks[4]. This should improve the robustness of the MPPI by discarding the most unreliable particles.

Ethical Approval

Not applicable

Consent to Participate

Not applicable

Consent to Publish

Not applicable

Authors Contributions

- Antoine Mahé: Simulation experiments, writing, coding
- Antoine Richard: Simulation/Field experiments, writing, coding
- Stéhanie Aravecchia: Field experiments, writing, coding
- Matthieu Geist: Supervision, writing, review
- Cédric Pradalier: Supervision, writing, review

Funding

This work is done under the Grande Region rObotique aerienNE(GRoNe) project, funded by a European Union Grant thought theFEDER INTERREG VAinitiative and the french "Grand Est" Région.

Competing Interests

None

Availability of data and materials

None

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
2. Akpan, V.A., Hassapis, G.D.: Nonlinear model identification and adaptive model predictive control using neural networks. ISA transactions **50**(2), 177–194 (2011)
3. Alain, G., Lamb, A., Sankar, C., Courville, A., Bengio, Y.: Variance reduction in sgd by distributed importance sampling. arXiv preprint arXiv:1511.06481 (2015)
4. Amini, A., Schwarting, W., Soleimany, A., Rus, D.: Deep evidential regression. Advances in Neural Information Processing Systems **33** (2020)
5. Dentler, J., Kannan, S., Mendez, M.A.O., Voos, H.: 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, pp. 2051–2056. IEEE (2016)
6. Gal, Y., Ghahramani, Z.: Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In: international conference on machine learning, pp. 1050–1059 (2016)
7. Gonzalez, J., Yu, W.: Non-linear system modeling using lstm neural networks. IFAC-PapersOnLine **51**(13), 485–489 (2018)
8. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
9. Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., Silver, D.: Distributed prioritized experience replay. arXiv preprint arXiv:1803.00933 (2018)
10. Hwangbo, J., Sa, I., Siegwart, R., Hutter, M.: Control of a quadrotor with reinforcement learning. IEEE Robotics and Automation Letters **2**(4), 2096–2103 (2017)
11. Katharopoulos, A., Fleuret, F.: Biased importance sampling for deep neural network training. CoRR **abs/1706.00043** (2017). URL `http://arxiv.org/abs/1706.00043`
12. Katharopoulos, A., Fleuret, F.: Not all samples are created equal: Deep learning with importance sampling. CoRR **abs/1803.00942** (2018). URL `http://arxiv.org/abs/1803.00942`
13. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
14. Lattanzi, D., Miller, G.: Review of robotic infrastructure inspection systems. Journal of Infrastructure Systems **23**(3), 04017004 (2017)
15. Lenain, R., Thuilot, B., Cariou, C., Martinet, P.: High accuracy path tracking for vehicles in presence of sliding: Application to farm vehicle automatic guidance for agricultural tasks. Autonomous robots **21**(1), 79–97 (2006)
16. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. CoRR **abs/1509.02971** (2015). URL `http://arxiv.org/abs/1509.02971`
17. Ljung, L.: System identification. In: Signal analysis and prediction, pp. 163–173. Springer (1998)
18. Loshchilov, I., Hutter, F.: Online batch selection for faster training of neural networks. arXiv preprint arXiv:1511.06343 (2015)
19. Lucet, E., Lenain, R., Grand, C.: Dynamic path tracking control of a vehicle on slippery terrain. Control engineering practice **42**, 60–73 (2015)
20. Maas, A.L., Hannun, A.Y., Ng, A.Y.: Rectifier nonlinearities improve neural network acoustic models. In: Proc. icml, vol. 30, p. 3 (2013)
21. Mahé, A., Pradalier, C., Geist, M.: 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), pp. 753–758 (2018). DOI 10.1109/ICSTCC.2018.8540719

22. Mahé, A., Richard, A., Mouscadet, B., Pradalier, C., Geist, M.: Importance sampling for deep system identification. In: 2019 19th International Conference on Advanced Robotics (ICAR), pp. 43–48. IEEE (2019)
23. Malinin, A., Gales, M.: Predictive uncertainty estimation via prior networks. In: Advances in Neural Information Processing Systems, pp. 7047–7058 (2018)
24. Naegeli, T., Alonso-Mora, J., Domahidi, A., Rus, D., Hilliges, O.: Real-time motion planning for aerial videography with dynamic obstacle avoidance and viewpoint optimization. IEEE Robotics and Automation Letters **2**(3), 1696–1703 (2017). DOI 10.1109/LRA.2017.2665693
25. Pannocchia, G.: Offset-free tracking mpc: A tutorial review and comparison of different formulations. In: Control Conference (ECC), 2015 European, pp. 527–532. IEEE (2015)
26. Qin, S.J., Badgwell, T.A.: A survey of industrial model predictive control technology. Control engineering practice **11**(7), 733–764 (2003)
27. Schaal, S., Atkeson, C.G., Vijayakumar, S.: Real-time robot learning with locally weighted statistical learning. In: Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on, vol. 1, pp. 288–293. IEEE (2000)
28. Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized experience replay. arXiv preprint arXiv:1511.05952 (2015)
29. Williams, G., Drews, P., Goldfain, B., Rehg, J.M., Theodorou, E.A.: Aggressive driving with model predictive path integral control. In: Robotics and Automation (ICRA), 2016 IEEE International Conference on, pp. 1433–1440. IEEE (2016)
30. Williams, G., Wagener, N., Goldfain, B., Drews, P., Rehg, J.M., Boots, B., Theodorou, E.A.: Information theoretic mpc for model-based reinforcement learning
31. Williams, G., Wagener, N., Goldfain, B., Drews, P., Rehg, J.M., Boots, B., Theodorou, E.A.: Information theoretic mpc for model-based reinforcement learning. In: 2017 IEEE International Conference on Robotics and Automation (ICRA), pp. 1714–1721. IEEE (2017)
32. Yaghoubi, S., Akbarzadeh, N.A., Bazargani, S.S., Bazargani, S.S., Bamizan, M., Asl, M.I.: Autonomous robots for agricultural tasks and farm assignment and future trends in agro robots. International Journal of Mechanical and Mechatronics Engineering **13**(3), 1–6 (2013)
33. Zhang, T., Kahn, G., Levine, S., Abbeel, P.: Learning Deep Control Policies for Autonomous Aerial Vehicles with MPC-Guided Policy Search. ArXiv e-prints (2015)
34. Zhang, T., Kahn, G., Levine, S., Abbeel, P.: Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. In: 2016 IEEE international conference on robotics and automation (ICRA), pp. 528–535. IEEE (2016)