12-2019

Domain Adaptation in Unmanned Aerial Vehicles Landing using Reinforcement Learning

Pedro Lucas Franca Albuquerque
pedro.albuquerque@huskers.unl.edu

DOMAIN ADAPTATION IN UNMANNED AERIAL VEHICLES LANDING USING REINFORCEMENT LEARNING

by

Pedro Lucas Franca Albuquerque

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Carrick Detweiler

Lincoln, Nebraska

December, 2019

DOMAIN ADAPTATION IN UNMANNED AERIAL VEHICLES LANDING USING REINFORCEMENT LEARNING

Pedro Lucas Franca Albuquerque, M.S.

University of Nebraska, 2019

Adviser: Carrick Detweiler

Landing an unmanned aerial vehicle (UAV) on a moving platform is a challenging task that often requires exact models of the UAV dynamics, platform characteristics, and environmental conditions. In this thesis, we present and investigate three different machine learning approaches with varying levels of domain knowledge: dynamics randomization, universal policy with system identification, and reinforcement learning with no parameter variation. We first train the policies in simulation, then perform experiments both in simulation, making variations of the system dynamics with wind and friction coefficient, then perform experiments in a real robot system with wind variation. We initially expected that providing more information on environmental characteristics with system identification would improve the outcomes, however, we found that transferring a policy learned in simulation with domain randomization to the real robot system achieves the best result in the real robot and simulation. Although in simulation the universal policy with system identification is faster in some cases. In this thesis, we compare the results of multiple deep reinforcement learning approaches trained in simulation and transferred in robot experiments with the presence of external disturbances. We were able to create a policy to control an UAV completely trained in simulation and transfer to a real system with the presence of external disturbances. In doing so, we evaluate the performance of dynamics randomization and universal policy with system identification.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor Dr. Carrick Detweiler for his support and encouragement throughout my time at the NIMBUS lab. His spot on observations and willingness to dive in new fields were essential to complete this thesis. I really appreciate his encouragement and guidance during this time. This work would also not been possible without the technical help and discussions of my fellow NIMBUSers, especially Ajay Shankar, who has conceded a good portion of his time to assist me during my time in the lab. I had a lot of fun learning machine learning and robotics, including the hundreds of UAV crashes, both in automatic as in manual control! I will carry fond memories working in the huskers' home. I want to thank my friends close and afar, who have been essential to make life great, and Gisela Sayeras, for her love and companionship during this time. I would like to thank my family for their love and support. They are my cornerstone and have been present despite the thousands of miles between us.

Grant Information

This work was partially supported by the United States Department of Agriculture - National Institute of Food and Agriculture (USDA-NIFA) 2017-67021-25924, and the National Science Foundation, NSF IIS-1638099, and NSF-IIS 1925368.

Chapter 1

Introduction

Unmanned Aerial Vehicles (UAV) have been increasingly popular due to their versatility, portability, and low cost. They are increasingly more used in a wide range of applications where exact modeling of dynamics can be challenging. Landing a UAV on a moving platform has been a classical problem in robotics, and several techniques have been used to tackle it, such as [4, 5, 6]. Most of these approaches require accurate models of the system dynamics, such as platform characteristics and environmental conditions, machine learning creates an opportunity to learn policies that do not require an explicit formulation of the dynamics to produce a controller. Rodriguez *et al.* presented the results of Reinforcement Learning to land a UAV on a moving platform [7] successfully producing a controller for a real vehicle by training solely in simulation. In this paper, we investigate how different machine learning approaches work with varying levels of domain knowledge when trained in simulation and executed both in simulation and on a real robot system with dynamics variation. Domain knowledge in our case means the inclusion of the dynamic variable in the policy to calculate the actions.

We can identify two fields to attempt to solve this problem: optimal control theory and reinforcement learning. Optimal control theory uses previous knowledge of the system and the environment to formulate the best actions that can be taken in any

situation that the agent is inserted. With this, advantages of optimal control theory are the guarantees of stability, robustness, but it comes with the price of meticulous investigation of the system and the environment in which the system interacts with [8]. These interactions can be highly complex, due to reasons such as the number of actions that the system can make, the different variables that it can observe, or the number of interactions that the system presents. Reinforcement learning, on the other hand, provides a solution that, instead of relying on formulations of the problem, a data collections on the problem [3]. It leverages the access to the agent and the environment to collect information on how to achieve the task in hand. A significant downside of this is that the guarantee of optimality for this approach is dependent on the agent exploring all the different scenarios of interaction between the agent and the environment [9]. In some cases, such as an agent that uses images to make decisions, this would require showing every different possible image that the robot can see, which is virtually impossible with a reasonable image resolution. In this work we focus on the resolution of tasks with reinforcement learning. We view this work as a stepping stone to future work that tackle even more complex tasks that are unsuitable for modelling in optimal control theory.

Reinforcement learning is a machine learning approach that has been incorporated into robotics development and research. It aims to train an agent that takes optimal actions in an environment that maximize a reward function. Reinforcement learning training in real UAVs can be unsafe and expensive in time and resources, as it requires extensive interaction with the environment to optimize a policy function in relation to an engineered reward function [10]. Leveraging simulations mitigates these problems as the systems are able to explore disastrous scenarios in simulation that would otherwise cause physical damage in real experiments. However, policies trained in simulation are not always trivially transferred to the real-world, as simulation may

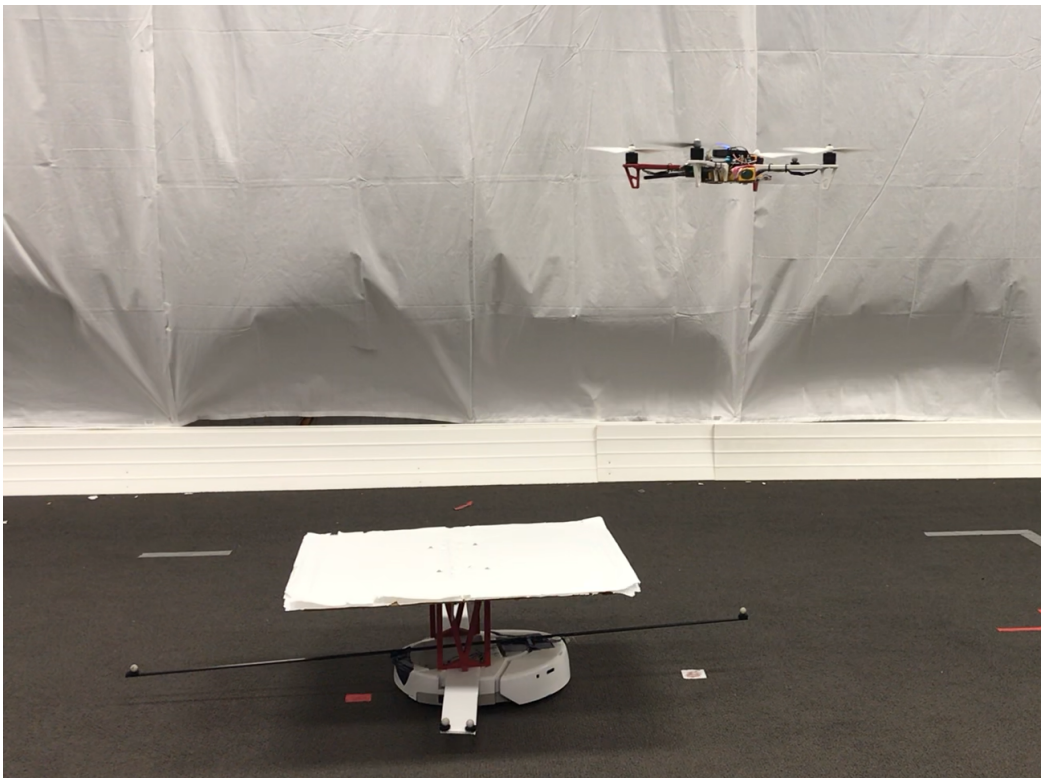

Figure 1.1: UAV landing on the moving platform.

not represent the dynamics found in real scenarios, causing a mismatch between the simulation and real-world generally called reality gap. In the same way, using reinforcement learning in simulation to learn a policy to land on a moving platform prevents training in real vehicles that could result in crashes and unsafe situations, but might generate inefficient policies due to the reality gap.

In this work, we train a policy in simulation to land a UAV on a moving platform (as shown in Figure 1.1) and compare their ability to transfer to the real-world and their robustness with the presence of external disturbances. We investigate three different approaches in simulation to land a UAV on a moving platform (shown in Figure 1.1) investigating their robustness with disturbances both in simulation as well as real-world experiments. The first is a System Identification and Universal Policy

(UP) approach that can identify disturbances and optimize the actions accordingly [11]. System identification (SysID) is a data-based method to infer variations of dynamics in an environment. Depending on the disturbance that influences the system, it is arguably complex to model analytical solutions estimating the disturbance and account for its influence in the control loop. For this reason, Universal Policy and system identification aim to offer a data-driven solution that trains a reinforcement learning policy that is parametrized by inferred parameters of the system dynamics. The main advantage of this technique is the ability of the policy to specialize in the set of parameters. That is, the system will always take the best actions, anticipating the influence that the parameters will cause in the state/action transitions.

Another technique to manage the reality gap included in several works is named domain randomization. Domain randomization is a technique employed to generate policies robust to variation both as observations as well as dynamics of the environment. In this work, we focus on the gap between the dynamics from simulations and the real-world. Its main idea is that by exposing the policy during training to the range of parameters, the policy is trained to perform under all the different circumstances. This approach uses a distinct strategy compared to UP+SysID, which instead of creating a policy that is able to handle different parameters, creates policies that are parametrized by the dynamics parameter. A downside of this technique is that, as the range of parameters increase, the policy becomes less specialized in order to generalize for all the parameters. However, this comes with the benefit of avoiding mistakes on parameter estimation, in which UP + SysID policies are prone to [12]. In this work, we perform a pragmatic analysis of employing UP+SysID in contrast to domain randomization and identify the advantages and disadvantages of the techniques. This is performed by running simulated and real experiments with variation of dynamic variables.

In this work, we propose a reinforcement learning framework to train policies to land a UAV on a moving platform and investigate the ability of machine learning techniques to produce policies robust to dynamic variations. We evaluate and contrast two approaches and a baseline, which is a policy trained with fixed parameters. The policies aim to perform this task successfully even with the presence of external disturbances.

We contribute in three main ways in this thesis.

- We extend the work of [7] developing a reinforcement learning system that is able to control a target velocity in all three axes and adds complexity by including changes in the system dynamics in the experiments.

- The analysis and contrast of domain randomization and universal policy with system identification to render the system more robust to changes in dynamics. This analysis provides insight of the techniques' trade-off that can be used in novel environments tackling domain adaptation. For this, we ran simulated and real-world experiments and observed their performances.

- Develop a fast Pybullet simulated environment able to transfer the policy successfully to the real-world by small adaptations in the reward function and adopting an LQR controller in the real UAV. Using a lightweight simulation allows the development of more complex systems with decreased requirements of computational resources.

The rest of the document is organized as follows: first we provide an overview of the background knowledge to understand the approach and the design choices that we made throughout this project, then talk about our approach, the methodology to setup the experiments, a discussion of the results, and a conclusion.

Chapter 2

Related Work and Background

This work uses several technologies and in this chapter we present some of the essential background information for a better understanding of our proposal and experiments. This background chapter is separated in four parts. The first gives an overview of the theory and practice in training neural networks, which is a machine learning artifice for function approximation used in several elements of this work, such as reinforcement learning and system identification. Second, we present reinforcement learning, some of its techniques and methods that the scientific community has adopted. Third, we present a brief overview on previous literature on landing a vehicle on a moving target. Fourth and last, we describe some of the literature in domain adaptation, which is the niche in which this work is focused on. This last section builds up on the previous two and describes how systems can transfer information between domains.

2.1 Deep Neural Networks

Deep neural networks are structures that have been essential to the development of recent technologies. This is so as it led Geoffrey Hinton, Yoshua Bengio, and Yann LeCunn to win the 2018 Turing Award for their contribution in foundational work

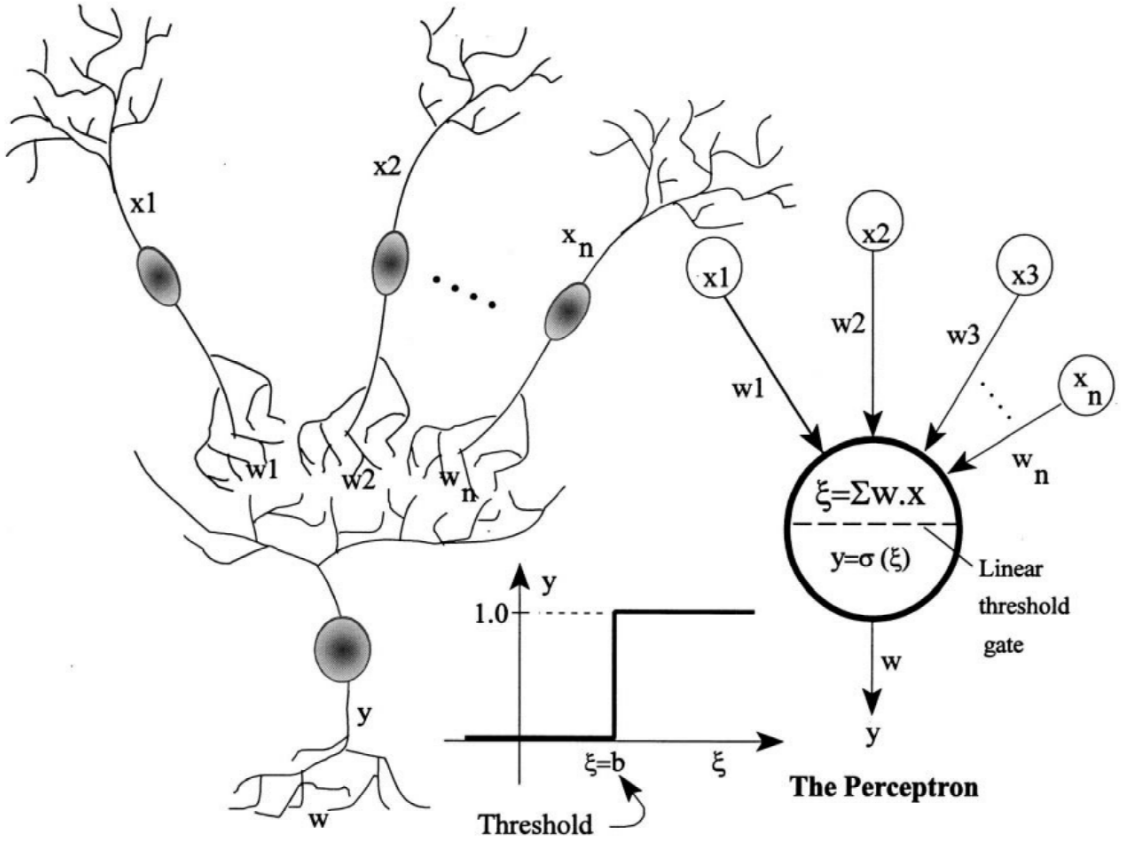

Figure 2.1: Comparison of a neuron (left) with an artificial neuron (right) used in neural networks. Image from [1]

in Deep Neural Networks[1]. Neural networks are structures created with biological inspiration: the neural connections. Similar to the brain connections, the neural networks have dense connections among neurons. The input information is carried along a network that activates according to non-linear functions and finishes in a layer that outputs the network's decision.

The artificial neuron is biologically inspired in the nervous systems. The neurons are elemental cells of the nervous systems and are responsible for the complex cognitive and motor activities. The brain has about 86 billion neurons, in which there

[1]https://amturing.acm.org/

are around 1 quadrillion connections among them [13]. These extremely abundant connections are responsible to transmit information to neighbor neurons in a process called sinapse. Artificial neural networks are virtual elements that are not meant to be a complete model of its biological counterpart, but has many elements inspired by it. Figure 2.1 shows a comparison between biological and artificial neurons. The values x represent how strong the signal in the neuron is, while w represents the synaptic strength between the neurons. The signals travel the neurons and arrive the bottom neuron that outputs y. Similarly, a neuron performs a weighted sum with the neighbor neurons, where w is the set of weights and x is the signal in each neuron. If the stimulating signals are over a certain threshold, the neuron is triggered; this process in neurophysiology is known as spatial summation [14]. In the artificial neuron, the sum is processed in an activation function, which in the case of the image, is a step function. When the sum is over the threshold, the neuron outputs 1, otherwise 0. We will later discuss some of other activation functions and their role in learning.

The neurons are powerful cells when grouped. The complexity of neurons arise when observed in a collection. Research has shown that memory is recorded with a circuitry of neurons, modulating the strength of the connections between neurons through synapses [15]. Similarly, the neural networks can encode information by modulating the connections between the neurons, so it is capable of approximating functions that are presented to the network during training.

In this section, we discuss some of the intricacies of training and using neural networks. In the next subsections we talk about feed forward networks, which is the type of networks used in our experiments; discuss about training neural networks, how to get a model that works well with unseen instances by using regularizers, the importance of activation functions and lastly, how to find the best set of hyper-parameters during the training of a neural network. Hyper-parameters are parameters

that are not learned during the training process and therefore have to be stipulated beforehand.

2.1.1 Feedforward Networks

Feedforward networks is the type of neural networks that we use throughout this thesis. They are structures in which there is flow of data from the input of the network, through the layers of neurons to the output. That is, there is no feedback in any part of the architecture. This network is divided in layers of neurons, the first layer that receives the input is called input layer; the layer that outputs the network decision is called output layer. All the layers between the input and the output layers are called hidden layers. Additionally, we use fully connected layers: these layers each neuron is connected to every other neuron from the next and the previous layers. This forms a dense network that can be used for the detection of patterns and representation of the trained function. Figure 2.2 shows a feedforward neural network with two hidden layers, note that each neurons is fully connected with the previous and next layers.

2.1.2 Training

Training neural networks is a data driven process that involves attempting to approximate a function and adjust tiny steps towards the correct values. The training is performed by providing the system a large number of data instances in the format of input and expected output. The neural network is initialized with random initial weights. In every instance of the dataset, the input is given to the network and the output is compared with the expected output. The difference of the outputs, the error, is then used as a signal to perform updates in the neural network. After finding the error, it is calculated how each element of the neural network has contributed to

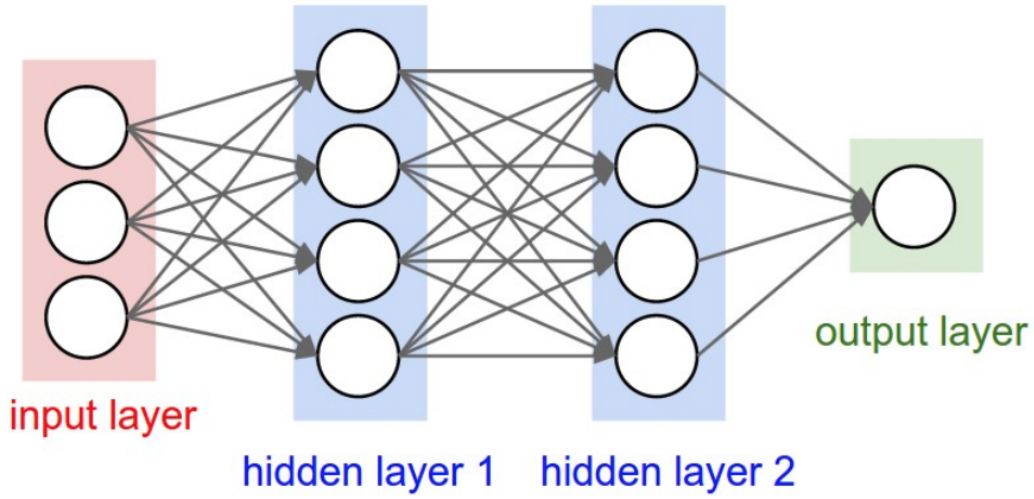

Figure 2.2: Example of a neural network with fully connected layers. Image from http://cs231n.github.io/neural-networks-1/

the erroneous calculation; these values are then used in different strategies to update the network. This is a very brief overview of the training that has many variations and has evolved in years of research. We expands these steps in the paragraphs below.

Calculating the loss of the prediction varies with the task. Neural networks are intensively used for activities such as numerical regression or classification. In a regression task, the loss is generally defined as the squared error between prediction and the true value. In classification, the most common approach is to use the last layer of the neural network as a softmax layer, which transforms the output of the previous layer as a probability distribution of the classes in the problem [2]. The loss is then calculated as the cross-entropy between the softmax output and a one-hot probability distribution, which is defined as 100% probability in the true class, and 0% in all the other classes. The cross-entropy is defined in Equation 2.1; it calculates the distance between two probability distributions p and q, assigned to the one-hot vector and the softmax output, respectively, over the interval x. When the

distribution of the softmax is very different to the one-hot, the value of cross-entropy is high and appoints the system to perform a bigger change in the network weights.

$$H(p,q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x) \tag{2.1}$$

Calculating the contribution of each element for the final error consists of finding a gradient vector. The gradient vector is the set of partial derivatives of each of the weights of the network in relation to the output. Some weights might have a higher contribution to the output, so their partial derivative is higher compared to other weights. The most common method of finding these gradients is called backpropagation [16]. Backpropagation takes advantage of the structure of neural networks to calculate the partial derivatives of each weight in relation to the output using the chain rule. By using the solution of one neuron to calculate further neurons, this technique has been viewed as an example of dynamic programming.

$$\nabla F(x,y,z) = \left(\frac{dF}{dx}, \frac{dF}{dy}, \frac{dF}{dz} \right) \tag{2.2}$$

There are many techniques to efficiently use the gradients to optimize the function, we name optimizers the techniques that perform such task. In charge with the information that we have dealt so far, an intuitive optimizer would, in every instance of the dataset, calculate the loss and update each weight of the network according to its gradient. This technique is called stochastic gradient descent. Stochastic gradient descend is a first order optimizer, which means that it optimizes a function following the gradient (first derivative) of the objective function. The update of each weight w indexed in i is given in Equation 2.3. The value η is the learning rate: it scales how much the weight will be updated with the gradient value. This allows the system to follow the trajectory of parameters set that leads to a global minima. Observe the

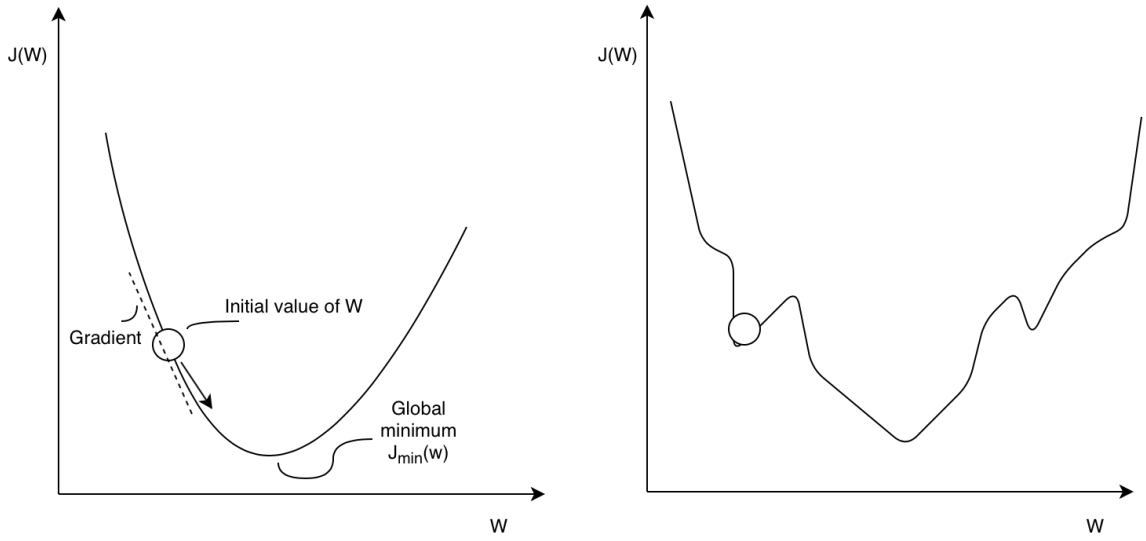

Figure 2.3: Convex function gradient estimation and path to optimization.

first image of Figure 2.3, it depicts the loss of a network training $J(W)$, where W is the parameter set of the neural network.

$$w_i' = w_i - \eta \frac{\partial L}{\partial w_i} \tag{2.3}$$

A problem with stochastic gradient descent is that the updates from each instance of training can be noisy. The updates from the noisy data can lead the training into an undesirable parameter set, constraining succesful trainings with a very small learning rate [17]. A method that mitigates this problem is called batch gradient descent. This method calculates the gradient for each instance of the dataset, averages them out, then perform an update with the result. This mitigates the noisy values of the dataset. A major drawback of this technique is that memory usage increases linearly with the size of the dataset, since all the gradients must be stored in memory. This is impractical for very large datasets.

A technique that seats between stochastic gradient descent and batch gradient

descent is called mini-batch gradient descent. In this technique, instead of calculating the gradient for every image in the dataset, the optimizer calculates the dataset for a subset of the dataset. This allows the training to use smooth gradients as in batch gradient descent, but with memory efficiency of the stochastic gradient descent. The algorithm that we adopt in our experiments builds upon mini-batch gradient descent to optimize the loss function.

We adopted Adam optimizer for our experiments. Beyond using minibatches, Adam optimizer [18] is a very popular optimizer that combines two important strategies. The first is momentum. Momentum is a technique that helps avoid getting the system stuck in global minima. This is done by including an element in the equation of weight update that is equivalent of the momentum of previous updates. That is, if in previous iterations the weights were being updated in a certain direction of the function, it is likely that the global minima is in that direction. The weights update according to Equation 2.4, where $\mathbf{m}$ is modified in each learning step according to Equation 2.5. The value β is a hyper-parameter that weights the value of the momentum, while η is the learning rate. For better understanding, observe Figure 2.3: the first image depicts a scenario of a function that has no global minimas, while the second image is a more realistic scenario in which the changes in the weights (W) might lead to local minimas in which the optimizer could get stuck. If the initial set of weights has value close to zero, the optimizer will increase the value of w, decreasing the loss and acquiring momentum since there is no local minima in the initial phase of the graph; in this same phase, the value of m is being updated with the direction of the gradients (leading to a more positive value). When the update reaches the point that is marked in the image, the system may have enough momentum (depending on β) to increase W so the system escapes the region of the local minima.

$$w_i' = w_i - m_i \tag{2.4}$$

$$m_i = \beta m_{i-1} + \eta \frac{\partial L}{\partial w_i} \tag{2.5}$$

The second resource incorporated in adam is RMSProp [19]. RMSProp is a technique to dynamically change the learning rate. In regular gradient descent, the update of the weights is proportional to the gradient; this means that when the slope of the loss function (the gradient) is large, the updates will have a large update, while small slopes will cause small updates. This can be problematic in very large gradients since the optimizer might miss important data between the updates. RMSProp proposes a solution by decreasing the update according to the recent history of the gradients size. Equation 2.6 shows that in RMSProp, the update of the weight is divided by the square root of G indexed by the weight i. The value of G is started as zero and updated according to Equation 2.7: it is the sum of its previous value with the squared gradient of the current iteration. This represents then how large the recent gradients have been. The variable β is a hyper-parameter between 0 and 1 that weights historic sum of gradients and the current gradient. Without the scaling, the value of G could reach an extremely large value and overtake the value of G, causing an aggressive downscaling in the updates. The value ϵ simply avoids that the division of the learning rate by zero.

$$\Delta w_i = -\frac{\eta}{\sqrt{G_i(t)} + \epsilon} \frac{\partial L}{\partial w_i} \tag{2.6}$$

$$G_i' = \beta G_i + (1 - \beta) \left(\frac{\partial L}{\partial w_i} \right)^2 \tag{2.7}$$

2.1.3 Regularization

Regularization is a group of techniques that aim to generalize the network by decreasing the complexity of the represented function to avoid overfitting. Observe Figure 2.4 where the black dots represent the data instances with attribute x_0 and label y. The first image exemplifies underfitting. Underfitting arises when the trained function has a much lower complexity than the function in which the data was sampled from. In the figure, the data is approximated with a linear function, which probably will not generalize well for unseen instances. A detrimental effect is also shown in the last figure, overfitting. This happens when the function is overly trained and has a very high accuracy in the training data, but is unlikely to perform well in unseen instances. The right balance is pictured in the second image, in which the approximated function is likely to fit the function that sampled the dataset, therefore would have a high performance in further data. Underfitting is generally tracked by the accuracy of the model in the dataset, while overfitting can be tackled with regularizers, which are discussed next. We include these models here since they were either employed in our models, or considered during hyper-parameter search, which is detailed later in this thesis.

2.1.3.1 L1-Norm and L2-Norm

L1 and L2 regularization norms include the neural networks weights in the loss function [20]. This strategy encourages the optimizer to choose a set of weights that are compact, but also effective. By having a large sum of weights, the network has the chance of being an over-engineered function that works well in the dataset, but does not generalize to unseen cases. This helps ensuring that only the weights in the network relevant to decreasing the loss are included. Moreover, some feature are also

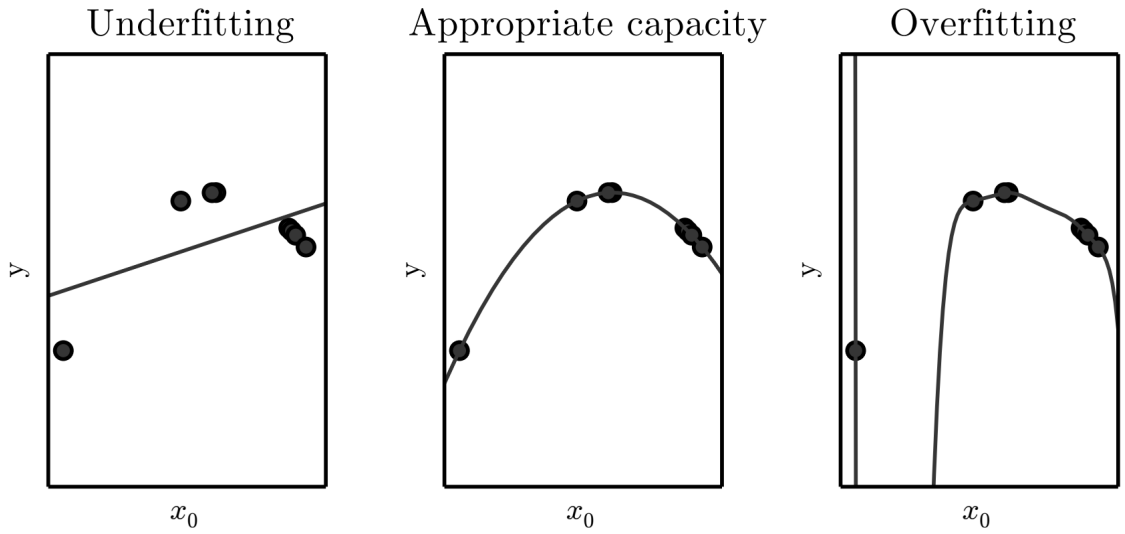

Figure 2.4: Example of the three cases in fitting a function from a dataset. Image from [2]

highly correlated with each other. In this case, it is unnecessary to keep both; the regression also encourages the optimizer to remove these redundant features. Thus, the optimizer will find a set of weights that is dependent both on the task in hand (e.g. classification, numerical regression) as well as minimizing the set of weights in the network.

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2 \tag{2.8}$$

L2 norm includes the second norm of the weights into the cost function. The additional term of the cost function is shown in Equation 2.8.

2.1.3.2 L1 Norm

The L1 norm, as the name suggests includes the first norm of the weights in the loss provided to the optimizer. Although the similarity of L2 norm, using L1 produces different effects in the optimization. The first notable difference is that L2 norm

penalizes larger values, since it squares the weights, while L1 only increases linearly. This generates sparsivity in the parameter set, meaning that while in L2 many weights will have small values, in L1 many values might be zeroed-out [17]. This is also known as feature selection, since it selects the weights that are most informative for the network.

2.1.3.3 Early Stopping

Early stop also decreases the complexity of the system, but in a different fashion [21]. Early stops is generally executed when the dataset is divided in three sections: training, test, and validation. Early stop consists in training the system with the "training" portion of the dataset, and verifying the performance of the system in the "validation" portion of the dataset. Whenever it is observed that the performance in validation is decreasing, the training halts.

Early stop's strategy caps the increase of complexity by stopping training. The validation set is a group of data instances that are representative of the data found in the real-world. It is important to note that there is no training on these images. The performance is calculated in these instances to verify whether the model has been overfited to the training data. The increase of performance in the validation set indicates that the model is also improving in accuracy in other unseen instances. However, the decrease in performance of the validation set might indicate that the model has been overfitted to the training data, so the system halts the training and restores the last best set of weights for the validation set. This is a powerful strategy, specially when the dataset is large enough such that part of it can be assigned for the validation set without losing significant information in training. Early stop produces great results with low parametrization such that Geoffrey Hinton has described Early

stopping as "free lunch"[2].

2.1.4 Activation Function

Activation functions receive the weighted sum from the connected nodes from input and calculates the output. In deep learning, this function are always non-linear, since the combination of the linear outputs will also be linear. In order to achieve a non-linear approximation, it is required to adopt non-linear activation functions [22], as it is described in the Universal approximation theorem. In fact, this theorem states that a neural network with only one hidden layer can approximate any continuous functions [23], but observing the non-linear nature of the activation function.

Early research in neural networks adopted sigmoid functions. This non-linear activation function, also known as logistic function, was highly adopted in early research of neural networks. They work specially well in shallow networks [22], which was the feasible type of architecture with the available computational power at the time. The sigmoid function is defined in Equation 2.9, and as previously shown through Figure 2.1, x is the weighted sum of the output of connected neurons. The main issue with sigmoid is the saturation of gradients: observe that the function does not change linearly with the sum of the input weights. When the sum is close to one, the addition of more neurons does not affect the result of the activation function with sigmoid, as it reaches a plateau. Therefore, even if the value of a previous neuron is very high, it is not taken as much into account due to the saturation and as a result receives a smaller gradient. This is also known as gradient saturation, and it increases the training time due to the small weight updates.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.9}$$

[2]https://media.nips.cc/Conferences/2015/tutorialslides/DL-Tutorial-NIPS2015.pdf, slide 63

Hyperbolic tangent ($tanh$) is considered a smoother option compared to sigmoid. A good feature of this activation function is that it is zero-centered. This allows centered data from layer-to-layer. In sigmoid, the activation function always returns positive numbers, so the gradient will always be positive or negative. When this happens, there will be a high update in one direction of the parameter space, overshooting the update. Subsequently, the system might correct the parameters, overshooting to the other direction. This creates a zig-zag like behavior that slows down the training. Hyperbolic tangent avoids this effect as it is a function whose center is zero, and can return either negative or positive values, producing a smoother update.

$$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} \tag{2.10}$$

The next activation function, Rectified linear unit($ReLU$) [24], is highly adopted [17, 25]. It is interesting how this simple function came to be the most used activation function. The linearity on positive instances avoids the saturation of gradients from distant layers, as it was with sigmoid. Another great feature of this activation function is the low computational cost: the absence of exponentials that are included in the previous methods makes $ReLU$ a much faster alternative.

$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} \tag{2.11}$$

Leaky $ReLU$ is another option similar to $ReLU$ that makes use of a larger portion of the network. Leaky $ReLU$ attempts to remedy a frequent issue with $ReLU$: dying neurons. As the negative regime of the $ReLU$ function is zero, some neurons do not activate since they present negative sums. In many cases, the neurons do not activate throughout the training, and do not contribute in any means in the networks. With

leaky $ReLU$, instead of having a negative sum as zero, a linear activation with a smaller scope than its positive counterpart takes place. This encourages the optimizer to train the dead neurons. However, this does not guarantee more efficient models than $ReLU$.

$$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{2.12}$$

2.1.5 Batch Normalization

Batch normalization [26] is a technique commonly used for the normalization of the layer output. Batch normalization is a powerful resource that produces two effects in the learning system. First, it decreases the training time by reducing the variance of the output from layer to layer; second, it works as a regularization technique when the minibatch in training is sufficiently low [27]. We applied batch normalization in our experiments, and it indeed increased the performance of the final model.

The batch normalization layer is implemented following the Equations 2.13 to 2.16. We call $\boldsymbol{Z}$ the result of the multiplication of $\boldsymbol{W}$, which is the set of the weights in the layer, and $\boldsymbol{X}$, the input values of the layer. In Equation 2.13 it is displayed the weighted sum that is calculated as the input to the activation layer for each neuron in the layer. Batch normalization calculates the mean across the mini-batch per feature and subtract the mean, such that the mean (again, across the minibatch) is zero (Equation 2.14). Subsequently, in Equation 2.15 a similar operation is performed with the standard deviation, normalizing the output. Lastly, in Equation 2.16, the output is transformed to have mean β and standard deviation γ, that are parameters learned with $\boldsymbol{W}$. The final output $\boldsymbol{Y}$ is then passed for the activation functions in each neuron.

$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{W} \tag{2.13}$$

$$\tilde{\boldsymbol{Z}} = \boldsymbol{Z} - \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{Z}_i : \tag{2.14}$$

$$\hat{\boldsymbol{Z}} = \frac{\tilde{\boldsymbol{Z}}}{\sqrt{\epsilon + \frac{1}{m}\sum_{i=1}^{m} \tilde{\boldsymbol{Z}}_i^2}} \tag{2.15}$$

$$Y = \boldsymbol{\gamma}\hat{Z} + \boldsymbol{\beta} \tag{2.16}$$

2.2 Reinforcement Learning

Reinforcement learning algorithms aim to generate a policy function that optimally performs decisions in a Markov decision process (MDP)[3]. This is generally started by taking random actions in the environment and observing the outcomes (Figure 2.5). An efficient reinforcement learning algorithm would then take increasingly better actions as time progresses, since it learned with the previous interactions. Formally, the objective of the Reinforcement learning model is to find a function π that, given the states of the agent, selects action that maximizes an accumulated reward $R_t = \sum_{i=0}^{\infty} \gamma^{(i-t)} r(s_i, a_i)$, where $\gamma^{(i-t)}$ is the discount factor. The discount factor decreases the reward, as the timestep of prediction is increased. We go into more details of this process throughout this chapter.

In this section, we expand on the building blocks of reinforcement learning: we start by defining how the environments for reinforcement learning are defined, talk about policy and value functions, that are important elements of reinforcement learning algorithms, and exploration-exploitation trade-off, then we explain some reinforcement learning techniques with intuitive approaches and explain their limitations until arriving in the technique the we adopted for this work, Deep Deterministic

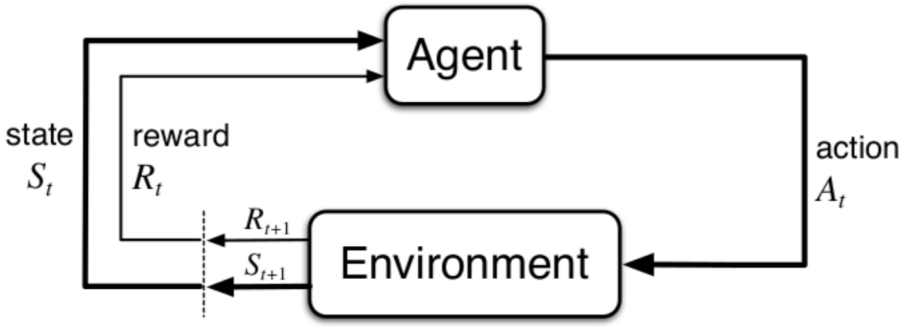

Figure 2.5: Overview structure of reinforcement learning techniques. The agent interacts with the environment by performing actions and receiving the next state as a result. Image from [3].

Policy Gradient.

2.2.1 MDP problems

MDP problem is described by a set of state-space, S, and action-space, A, in which the agent takes actions that may result in transitions between states. The transitions are probabilistic in nature. That is, an action a, taken in a timestep t and state s_t, may successfully transition to a new state, s_{t+1}, according to the probability distribution $p(s_{t+1}|s_t, a_t)$. The objective of the agent is to maximize the reward signal, provided as a feedback to achieve the task. Additionally, the transition from the state s to the state s_{t+1} accompanies a reward signal r_{t+1}.

An important characteristic of the MDP problems is that they must fulfill the Markov property. This property states that the current state of the agent does not depend on previous states. A good example to understand this property is the task of releasing an object into a target. Suppose the agent is flying in fixed height and varied speed towards the target. The action space would be a binary variable describing whether to retain or release the object. If the state space for this problem consists of only the relative position of the agent in relation to the target, this setup does not

fulfill the Markov property. This is so as the current position would depend on the previous state to estimate the velocity. However, if we include the relative velocity in the state space, it is not necessary to check historical transition of the nodes to fully describe the current state of the system; therefore, the system would fulfill the Markov property.

Formally, a Markov decision process is composed of:

- Set of actions A
- Set of states S
- Transition probabilities $p(s'|s, a)$, that describes the probability of transitioning from state s to s' taking the action a
- Reward function $R(s, s')$ that describes the reward that an agent receives when transitioning from s to s'. The reward function will define the goals of the agent.

2.2.2 Discounted Accumulated Reward

Another variable generally employed with the MDP definition is the discount factor γ. This variable scales down the accumulated sum of future state transitions. Therefore, the variable γ ensures that the rewards collected in earlier are more valuable than later rewards. If the rewards are equally valued, then the agent would not have preference of transitioning to the states that achieve high rewards more quickly. The use of a discounted rewards affects the reward exponentially in function of gamma. Observe the equation 2.17 describing the discounted accumulated reward G from the timestep t onward. This principle is applied in a wide variety of reinforcement learning algorithms, including the one adopted in our experiments.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.17}$$

2.2.3 Exploration vs exploitation

Reinforcement learning tackles the exploration-exploitation trade-off. Reinforcement learning systems start by picking random actions in the environment, but as the time progresses, start to follow the actions that are believed to reach better states. By following this strategy, the systems are able to improve upon the knowledge that has been acquired so far. However, a question arises from this process. When and in which cases will the agent stop taking random actions and take the actions that are believed to be good? This is a very famous problem, which is named the exploration-exploitation tradeoff. Essentially, if the agent does not spend sufficient time exploring, its estimation of what is the best action to be taken next will be faulty and suboptimal; on the other hand, if the agent spends a long time exploring, then it will take an infeasible amount of time to learn the whole task during training. This problem is so hard, that according to [28], during the second world war, the Allies strategically sent this problem to Nazi Germany so they could be distracted, since they believed that it was impossible to solve this problem.

Reinforcement learning techniques tackle this problem according to the type of algorithm in hand. We will explorer later in the chapter how our adopted RL technique can deal with exploration, and how its predecessor also approach it. Meanwhile, it is worth to mention that exploration vs exploitation is an important and very hard problem, and our concern on dealing with it was crucial for the success of these experiments.

2.2.4 Value and Q-function

The value function is an estimation of how good it is for the agent is to be in that state, also known as utility. The higher the value function, the better chances are that the agent will be collecting a good sum of rewards very soon or in the near feature; this depends of the discount value γ that we discussed about. The "wellness" of being in a certain state is the best discounted sum of rewards that can be achieved by starting from that state. This is formally defined in Equation 2.18. The value function of the state s using the policy π is the expected sum of rewards starting from the timestep t (G_t), given that the initial state S_t is s.

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t | S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \text{ for all } s \in \mathcal{S} \tag{2.18}$$

The value function helps indicating the potential that the agent has by being in a certain state, but by itself does not assist the selection of the next action. That is the role of the q-value (Equation 2.19), in which it is estimated what is the expected discounted reward when the agent follows the policy π after taking the action a in the state s. This then allows the calculation of the best action as selecting the action in the current state in which the q-value is maximum. This is the base of several reinforcement learning techniques, including the method that we use in our experiments.

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \tag{2.19}$$

2.2.5 Model-free vs Model-based learning

The agent can take two strategies regarding its knowledge of the environment: model-free or model-based. Model-based learning consists of techniques that use or create a model to predict the state transitions according to the actions that the agent can take; model-free, on the other hand, does not use an explicit representation of the environment, but only calculates the best actions to be taken.

Model-based algorithms are efficient to train multiple tasks, but limited to its ability to learn the model of the environment. This class of algorithms make use of an estimation of the MDP of the environment. This estimation can be in the form Neural Networks, tabular data, or any other means that could inform the agent what would be the consequences of its actions. The agent then follows the actions that maximize its reward. Observe that if the agent already has the full information about the environment, it is not necessary to perform reinforcement learning. The agent would only need a planning technique to follow the best actions since it does not need to interact with the environment to learn how to achieve optimality. An advantage of model-based learning algorithms is that when the agent has a good model trained, it can be used for other tasks since it would be able to leverage its knowledge of state transitions to achieve the new task. Additionally, every interaction with the environment is used to update its model, so it is generally more data efficient [3]. However, model-based approaches are limited if the model of the environment is inaccurate [3]. Depending on the complexity of the environment, learning a model by sampling can be unsuitable for training a policy [29].

Model-free algorithms are data hungry, but can achieve near optimal results. On contrary to model-based techniques, model-free algorithms has no explicit knowledge about the MDP of the environment. This has pros and cons in the quality of the

algorithms. A detrimental feature is that training can be data intensive: since the agent is attempting to include the dynamics into the policy, the agent has to sample several times from the environment and increase the confidence of its estimation rather than plan on the information learned from the model-estimation. However, since both tasks are included in one estimation, it is less likely of being stuck at a point of failure in the model.

In this work, we focus on model-free techniques, since we aim on having efficient policies, even if that means that it is necessary to execute training for a longer time in a simulated environment.

2.2.6 Monte Carlo Methods

Monte Carlo is an intuitive technique to solve the reinforcement learning problem with a model-free setting. In order to estimate the value function, we simply run the policy in the environment and verify what is the accumulated reward when the starting state is s. When enough samples are collected, all these measurements are averaged out to obtain the value function. The Monte Carlo method is guaranteed to converge in the states that are visited by the policy, as the estimation follows the principle of the central limit theorem. The central limit theorem states that the sum of random variables follows a normal distribution, so the average of these values sampled in the rollouts will indeed be the expectation.

The training of Monte Carlo techniques consists of loops of estimation and improvement steps (Figure 2.6). In estimation, the policy π is used in interaction with the environment until the episode reaches a terminating state. The estimations of the states from the starting state until the final one are updated according to accumulated rewards (from the state onward) observed during the episode. The improvement step updates the policy with the new estimates of the value function; the policy will select

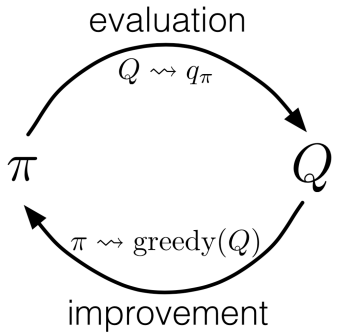

Figure 2.6: Updates used in Monte-Carlo. The evaluation step is sampled with the policy in the environment, and the improvement step maximizes the actions that lead to the best states. [3]

in every timestep the action that leads to the states with the highest value function estimations. This loop is repeated until there is no variation in the value function.

Monte Carlo's update limits its number of feasible applications. A limiting factor of Monte Carlo is the necessity of reaching the end of the episodes to update the value function. This limits the scope of applications with very long episodes or environments in which reaching the end can be intractable. This is solved in the next group of algorithms: the temporal difference methods. However, Monte Carlo can still be employed due to its unbiased estimations, as it only incorporates information that has been collected directly from the environment.

2.2.6.1 Temporal Difference Methods and Q-Learning

Temporal difference (TD) methods is another class of algorithms to solve reinforcement learning. Similarly to Monte Carlo, TD methods do not have the previous knowledge of the dynamics of the environment, so it is also necessary to explore the environment to estimate the expected reward. It differs from Monte Carlo in the sense that the updates of the value function estimation do not require the agent to collect experiences until episode termination. Instead, the updates are performed using the

observation of the rewards in the rollout of experiences and the estimation of the value function from that transitioned state on. Updating the estimation of the value function with another estimation is called bootstrapping, and is abundantly present in reinforcement learning. Here we analyze a very popular TD method: Q-learning.

Q-Learning [30] is a TD method that updates the estimation in an off-policy manner. Being off-policy means that the update of the estimation computes the value function assuming a certain action in the next timestep, which might not be the same action taken by the current policy, hence the name off-policy. The update of the Q-values is shown in Equation 2.19. In each timestep, Q-learning observes the q-value of a state-action pair, and executes an action a; a reward is then collected from the environment when the agent transitions from the state s to the new state s'. The q-value of the state-pair is updated with the difference between the predicted Q-value and the collected reward summed with an estimation of the Q-value (this difference is also known as TD-error) following a greedy policy. The current policy might not be greedy, but the q-value is always updated estimating the best return that the agent might have if it follows the most profitable actions.

$$q(s,a) = q(s,a) + \alpha \left[r + \max_a \gamma q\left(s', a\right) - q(s,a) \right] \tag{2.20}$$

Q-learning starts off exploring the environment with a random set of Q-values. As was discussed at the beginning of the section, one of the fundamentals problems in reinforcement learning is the exploration vs exploitation tradeoff. Q-learning generally tackles this problem with ϵ-greedy technique. It uses a variable that defines the chance of taking a random action instead of following a greedy choice, which is the action whose Q-value in the current state is maximal (Equation 2.21). The coefficient ϵ defines how fast the variable increases the probability of using the greedy choice. In

this way, as training proceeds, the system takes progressively less random actions as the estimations of value function is more robust.

$$a^* = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ Q(s, a) \tag{2.21}$$

2.2.6.2 Deep Reinforcement Learning

The reinforcement learning methods presented so far work well in small tasks, in which the value function can be represented in a tabular form, a matrix. However, when there is a high-dimensional state space, which is the case of our application, the problem is rendered infeasible using a tabular representation. Moreover, visiting all the combinations of state-action pairs might be impossible in certain cases. This problem has been tackled by adopting neural networks to approximate the value function and/or the policy. Such substitution provides a powerful generalization of these functions in more complex systems since it approximates the value function for states that were not yet visited and it does not require that all the state-action combinations be stored in memory. A drawback of this approach is that using neural networks removes the certainty of optimality, although empirical results demonstrate good results.

2.2.7 Deep Q-Networks

An iconic successful use of non-linear function approximation is the algorithm Deep-Q-Networks (DQNs). It is a well-known algorithm since the publication of the pioneer Deep reinforcement learning Nature paper of Atari game playing [31]. The paper describes the implementation of a system with a deep neural network that approximates the Q-values of the action/state pairs. Resources such as experience replay, target network, and batch normalization enabled the system to stabilize learning and achieve

convergence. With such infrastructure, the authors were able to implement a system capable of attaining human-level performance in Atari games without providing prior knowledge for the system, except the position of the game score and the action buttons of the joystick.

The Q-function is approximated with a deep neural network with parameters θ. In order to train the network, the system bootstraps the Q-values and calculates the loss according to Equation 2.22. The loss of the network is the TD-error, similar to the right side of the addition in Equation 2.20. The inner expectation is the expected value of the accumulated sum of rewards of the new state following the greedy choice.

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim p(\cdot)}\left[\left(\mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) \,|s, a\right] - Q(s, a; \theta_i)\right)^2\right] \quad (2.22)$$

DQN is able to achieve such performance by including two important resources: experience replay buffer and target networks. These two resources were essential to stabilize the training.

The first trick, prioritized experience replay buffer, allows the reduction of variance during training by keeping recent historical information. This means that every interaction with the environment in the form $e_t = (s_t, a_t, r_t, s_{t+1})$ is recorded in a replay buffer $D_t = \{e_1, \ldots, e_t\}$. When the value function is trained, instead of only training the most recent interaction with the environment, all the values in the buffer are computed and updated using minibatches. An important detail is that the batches are sampled randomly from the buffer. This increases the variance of the information that trains the value function since experiences from very close timesteps will more likely be correlated.

The second trick, target networks, reduce the instability in learning. The insta-

bility comes from the fact that updating the value function affects the neighboring states of the updated state. States which might be in the region that the agent is exploring at the moment. Updating and using the value immediately might cause biased predictions that cause oscillations in the learning system. The solution for this is to create a copy of the network and perform updates in this copy while estimating the values in the original network. The original network then updates its weights with the copied network every C steps (a hyper-parameter)

Deep Q-networks have been used in many applications in robotics. Some include keeping a vehicle in a desired street lane [32], autonomous vehicle driving [33], coordinate robot with gripping mechanism to follow visual cues [34], robotic object grasping through images [35], playing first-person shooting games [36], robotic exploration with object avoidance [37]. An important limitation of DQN in robotics, however, is that it evaluates Q-values of all the action-pairs to define the best action. That is, in continuous tasks, the action space has to be discretized. This raises questions in the development of the system, such as the accuracy necessary for the actions being sliced, and the time to train this action space. In precise robotic tasks, DQN might not be a feasible algorithm, as the discretization of certain environments might result in an exploded action space.

2.2.7.1 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is an off-policy, model-free reinforcement learning algorithm [38]. It takes advantage of continuous state spaces to backpropagate the Q-function. The Q-function is a numerical estimation of the discounted accumulated reward that the agent could potentially collect starting with a certain action.

DDPG makes use of actor-critic architectures, which trains the policy using two

components: the actor and the critic. The actor is responsible to select the next best action available given its current state. On the other hand, the critic approximates the expected discounted accumulated reward through the Bellman Equation [38].

Critic

The actor is responsible to identify the utility of the state-space. Similarly to Q-learning, keeps track of the Q-value for each state and action by bootstrapping. This is done by calculating the bellman function (2.23) for a given state s and action a.

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim P}\Big[r(s,a) + \gamma \max_{a'} Q^*\left(s', a'\right)\Big] \tag{2.23}$$

The Bellman equation defines the best action as a greedy recursive function that maximizes the sum of rewards in each step. The critic uses the Bellman equation to bootstrap the Q-values until it converges. The actor approximates a function that returns the best action a, given the current state s.

The critic performs gradient descend to approximate the estimation of the Q-values with observations of the state transitions. The update is executed when performing a transition in the environment: starting from the state s, taking the action a, then observing the new state s'. The update is then calculated comparing the estimation of the q-value with the estimation of the new state and the observed reward. Observe the Equation 2.24, it shows the mean square error (loss) of the estimated Q-value and the expected Q-value. Using an estimation to update the estimation of Q-values is called bootstrapping.

$$L(\phi, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d) \sim \mathcal{D}}\left[\Big(Q_\phi(s,a) - \Big(r + \gamma(1-d)\max_{a'} Q_\phi\left(s', a'\right)\Big)\Big)^2\right] \tag{2.24}$$

In this setting, given an action, the Q-Value is differentiable, as the actions are continuous. Deep deterministic policy gradient uses this property to calculate the gradients and updates the actions that maximize the Q-values. In order to increase the stability during learning, DDPG also includes target networks. These are copies of the original networks whose weights are updated gradually to avoid destabilization from sharp gradient updates.

Actor

The actor is responsible to find the best action when informed with the current state. Observe the Equation 2.25, when training the actor, it performs a gradient ascend on the parameters of the network θ such that it maximizes the Q-value of the states S sampled from the distribution $\mathcal{D}$, which is the training data.

$$\max_{\theta} \mathop{\mathbb{E}}_{S\sim\mathcal{D}} \left[Q_{\phi}\left(s, \mu_{\theta}(s)\right)\right] \tag{2.25}$$

2.2.8 Exploration techniques

Since DDPG outputs a deterministic action given a state, the exploration generally is done by varying the actions. There are three main exploration techniques in light of this: Ornstein-Uhlenbeck process, adaptive parameter noise, and the normal action noise.

2.2.8.1 Ornstein-Uhlenbeck process

Ornstein-Uhlenbeck (OU) process [39] leverages the idea of correlating the noise outputs to have a higher impact on the final result. The noise generated in each reading of the noise is correlated with its previous outputs. The authors of the original DDPG paper [38] adopt this noise with the argument that it is a reasonable strategy when

dealing with problems with inertia. The work published in [40] analyses the use of correlated noise for continuous tasks. The intuition behind this can be imagined with an environment whose action space is consisted of forces that drives a robot. By having forces in the action-space, the output actions will directly affect the acceleration, which will be mitigated when the system behavior is evaluated by observing the position. This is so as the actions will affect the velocity, which in turn will affect the final position. When the noise is uncorrelated, the final integration could be insufficient for exploration since it might cancel itself out.

Ornstein-Uhlenbeck is formally defined as a stochastic process following the differential equation in Equation 2.26. The value θ describes a an element of the next noise sample that makes it closer to the mean (μ), while the variable σ is a coefficient of how much "randomness" the noise will have; finally, W_t is the Weiner process, also known as Brownian motion, which is a random process that in implementation will be a Gaussian distribution with zero mean and unit covariance.

$$dx_t = \theta \left(\mu - x_t\right) dt + \sigma dW_t \tag{2.26}$$

2.2.8.2 Normal Action Noise

Normal action noise is precisely what the name suggests: a noise sampled from a normal distribution. This action noise is uncorrelated; that is, each reading is independent of the previous samples. Despite the argument provided in 2.2.8.1, there is work that suggests that Normal Action Noise works just as well as OU noise [41].

The noise is modeled with a Gaussian distribution, as shown in Equation 2.27. The variable μ is the mean and σ the standard deviation of the normal distribution.

$$n = \mathcal{N}\left(\mu, \sigma^2\right) \tag{2.27}$$

2.2.8.3 Adaptive Noise

Adaptive noise [42] uses a very different strategy to the previous types of noise. Instead of affecting directly the action taken by the network, adaptive noise performs slight variations in the network parameters. The changes in the network parameters will directly affect the final actions of the network. This approach takes into account the magnitude of the changes to the action caused by the network. In fact, it is unreasonable to change all the network weights with the same noise and expect the actions to vary linearly. For this reason, adaptive noise uses layer normalization, which normalizes the mean and variance of the neurons within the same layer before making updates. This helps to make sure that there is a similar distribution of the layers sequentially positioned. Additionally, the authors also verify how each layer affects the final action and limit their magnitudes in such a way that there is a controlled variance.

The network generally becomes increasingly more sensible throughout the training. This is so as there might be a more stable set of weights to achieve complex tasks. As discussed, adaptive noise checks the magnitude of the changes in the networks; depending on changes, the noise can be scaled up or down to have a smoother exploration. This is shown in the Equation 2.28. The variable σ_k is the noise applied in the timestep k; $d(\pi, \widetilde{\pi}) < \delta$ is the distance between the policies with $(\widetilde{\pi})$ and without (π) the noise; α is a scaling factor; δ is a threshold.

$$\sigma_{k+1} = \begin{cases} \alpha\sigma_k, & \text{if } d(\pi, \widetilde{\pi}) < \delta \\ \frac{1}{\alpha}\sigma_k, & \text{otherwise} \end{cases} \tag{2.28}$$

2.3 Landing on a Moving Platform

The task of landing on a moving platform has many facets that render it a very interesting problem. First, the interaction of the vehicle and the platform requires a sophisticated model to take into account elements such as the ground-effect when the vehicle is close to the platform [4, 5]. Moreover, external factors such as environmental conditions, cargo, or even the friction between the vehicle and the platform increase the complexity of this task [43].

In this work, we focus on evaluating domain randomization applied in robotics. We have applied the system to perform a classic robotics problem: learning to land on a moving platform. A significant amount of this work is based on [44] to deploy the learning system. As far as we could research, their work was the first scientific report describing reinforcement learning to learn a UAV on a moving platform from a policy learned purely in simulation and deployed in the real-world. Our work differs in some points, such as the action-space of the policies: the reinforcement learning policy is trained to generate three-dimensional action space, while they fixed a descending rate, which reduced the action space to two dimensions. The formulation of the reinforcement learning training adopts different strategies, both in the reward function as the action-space exploration function. A sequence of their work [7] includes visual perception in the control loop. Additional work includes [45], that used least squares to perform the landing only in a simulated environment, and [46] using DQN to output directional actions (up, down, left, right, descend) for landing with visual information. Other papers do not employ reinforcement learning but use neural networks to perform the visual servoing for landing [47].

There has been work on developing models for landing without machine learning. Some of these solutions include the use of optical flow fused with the vehicle's IMU to

perform autonomous landing and hovering over the platform with close to zero relative velocity [48]. In their work, a VTOL vehicle lands on a platform in rough seas, whose movement is modelled as a sinusoidal wave. The control system is designed in a similar fashion as [5]. Similarly, the work in [6] describe a system that performs landings using visual servoing with a quadcopter. The work in [49] tackles helicopter landing on a ship using tether that is attached between both vehicles, providing information about the motion of the ship and an accurate state estimation of the relative position of the helicopter.

2.4 Domain Adaptation

Domain Adaptation is the set of techniques that aim to succeed in transferring knowledge acquired in a domain to another. The domain can be different due to different reasons: perception, dynamics, classes. When a system is trained in a certain environment, or dataset, it specializes in tackling problems in that distribution of data, and when the system is tested in a different distribution, the algorithm might fail. For instance, a neural-network might be trained to classify images with a "white dog", "not white dog" classes. If this model is tested on a situation of "dog", "not dog" it might be fail in most of the cases in which the dog is not white, even though there is a high similarity between the two tasks. Domain adaptation techniques aim to train models such that this gap between the performance of the two environments is decreased.

Domain adaptation has been tackled in many ways. Perhaps the most intuitive manner is to simply perform training in the target domain; however, this is not always possible due to system restrictions on safety, financial costs of operating the real-system, or fragility. An interesting technique in domain adaptation is named

progressive nets [50]. This technique implements several neural networks for different tasks, but in each training, instead of starting the weights from scratch, it sets a connections in layer level with the previously trained networks to leverage the knowledge previously acquired in other tasks. The work in [51] leveraged interactions with the real world to make the simulation behavior more similar to the real counterparts, decreasing the reality gap. Meanwhile, [52] uses deep inverse dynamics to train a model that bridges the gap between the policy's intention and the execution. They train a deep inverse dynamics that models the transition probabilities in the real-world scenario. When the policy is given a state, it outputs the actions to be taken. Instead of taking the action, the system replicates the scenario in simulation and observe what is the transitioned state. The deep inverse dynamics then estimates what is the action that has to be taken in the real-world that provides the maximum likelihood to transition to the same state observed in simulation. This helps mitigating the gap between the consequences of the actions that the policy chooses in simulation into the real-world.

Although very promising, these techniques still require training in the real world instances to be optimized. Next, we describe the two techniques that we analyze throughout the experiments: domain randomization and universal policy with system identification.

2.4.1 Domain Randomization

Domain randomization is an increasingly adopted technique to adapt reinforcement learning policies acquired in simulations to perform successfully in the real world. The main principle of the technique it to augment the training conditions of reinforcement learning systems with a diversity of simulation parameters; thus, the system is able to decrease the reality gap that real-systems might face when compared to the simulation

[53]. Domain adaptation borrows the principles of data augmentation in computer vision applications, that is a commonly used alternative to increasing the classification robustness [54]. In computer vision, the dataset is increased by performing copies of the images with slight variations such as rotation, noise or scale. The increased data then encourages the neural network to identify patterns for classification that are invariant to the mentioned operations. In the same way, data augmentation on reinforcement learning stimulates the system to find the optimal behavior with variations of the parameters.

Domain randomization is generally implemented in two manners: perception and dynamics. In perception, the issue arises when a vision based system learns visual representations in simulation that are divergent to the real-world. This can be seen in a regularization perspective: when a system maps visual clues to action, the features that are learned during training might be presented only in simulation; some of the features include colors, shapes, textures. When the system is deployed in real-world, these features might dissimilar to the point of impeding the policy to behave as expected. Examples of this include [55] which randomizes lighting, textures and positions of objects in a low-fidelity simulator to learn a policy that is able to identify the objects shape. The authors in [56] trained a neural-network in a simulation with a high variety of visual information such as textures, colors, and lightning to perform collision avoidance with an unmanned aerial vehicle. The final setup was able to succeed under challenging conditions such as flying through a staircase, narrow corridors and by moving objects. Using the same principles, [57] were able to perform complex object manipulation, such as rolling a cube to a goal position, using a robotic hand with a policy learned from a simulated environment. The manipulation of such objects is extremely difficult and require extensive control experience to design controllers using tactile sensors; being able to incorporate high dimensional

sensory information from cameras enables the system to take advantage of an efficient feedback loop.

In dynamics, the simulation in which the reinforcement learning system is trained might not be representative, or might not model of the idiosyncrasies of the real-world scenario. That is, there might be physical phenomena that is not represented in simulation, but that occur in the real robot. Moreover, the physical interaction might be modelled, but with erroneous magnitudes. The work described in [58] utilizes a neural-network structured with long short-term layers, a deep learning architecture capable of recording information that is important for future analysis, to identify a variety of parameters such as delay, noise, mass, friction for the task of using a robotic arm to push a puck towards the desired position and successfully identify the parameters of the real-world scenario. Moreover, [59] trained a policy for quadruped robot gait using a variety of parameters such as motor friction, strength, and latency. The result was similar in speed performance of state-of-art handicraft approaches, but with the improvement in power consumption. The authors also conduct an ablation study to prove that domain randomization is the deciding factor to fill the reality gap between the simulation and the real world. Similarly, the work in [60] also randomizes friction coefficient, but to train a robotic gripper to pivot objects into a desired configuration by pushing it. Our work focuses more on the dynamics side, but also considers extensions with perception in the future.

2.4.2 Universal policy with System Identification

Training a system identification module assisting a universal policy was described in [11], in which a the universal policy (UP) is trained with a variation of parameters in effect on simulation, and the varied parameter is included in the state-space; later they describe the training of the universal policy, which in test time infers the

parameters and includes the estimation in the state-space, similar to training. We adopt this strategy for our experiments and compare the results with pure domain randomization. In a similar fashion, but with different architectures, the authors in [61] instead of training the system identification with dense layers, use recurrent neural networks to implicitly extract temporal context from the state/action rollout capable of inferring the dynamic parameters. This assists the critic to identify more meaningful assessment of the actors performance. However, due to the stability of learning with dense layers, we kept the original work in [11].

System identification (sysID) allows the system to incorporate information about the dynamics of the environment into its decision-making process. Adding wind estimation, for instance, could result in an offset of the velocity to compensate for the force that the wind exerts on the vehicle. As previously discussed, the variation of the dynamic variables allows the generalization of the system in the range of values used in training. However, simply generalizing to an arbitrary range may degrade the performance as the range increases. With system identification, this scenario is ameliorated with an intermediate step that is included: the parameter's inference. This setting enables the system to select the actions that are better suited for an agent interacting with the inferred dynamics, rendering the actions more specialized to the parameter set.

The intuition of this inference is that depending on the parameters used in simulation, there will be different outcomes on actions taken even if the initial states are identical. Therefore, the system identification module receives a history of sequential states and actions to find patterns of the dynamics parameters into the rollout of experiences. Once the system is trained, it would be fed with real-time state-action pairs to identify the parameters in play on the system.

Chapter 3

Approach

In this work, we investigate the performance of DDPG with domain randomization, a Universal Policy with System Identification, and a baseline trained with no parameter variation. In this section, we give the details of our approach in implementing and using these methods. We first trained a policy using DDPG [38] and domain randomization by only exposing the parameters to the policy, but omitting the parameter value in the state-space. In addition, we trained a Universal Policy (UP) according to the work described in [11], which not only exposes the policy to the variations in the environment but also includes the parameter estimation in the state-space. This approach also involves the implementation of system identification for the parameter inference. We detail the design choices for each technique below.

3.1 DDPG

We adopted DDPG [38] due to its ability to handle continuous observations, output continuous actions, and because DDPG uses model-free learning. In model-free learning, the agent makes no assumptions about the environment with which it is interacting. The dynamics of the environment are learned as part of the policy. This is advantageous as the user does not need to determine the exact dynamic model

of the robot, nor dedicate an inference network for this task. Moreover, DDPG is able to handle continuous spaces, as discussed in Section 2.2.7.1. This is a significant advantage since it allows the policy to take precise actions compared to a discretized action-space.

3.2 Domain Adaptation

We adopt system identification to estimate the model parameters used in the Universal Policy. The system identification module is trained in a supervised manner, so we adopted the use of fully connected neural networks as it is described in [11] due to its stability in learning and robustness. The objective of this module is to estimate the parameters, given an array containing a recent history of actions and states.

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(H_i, \boldsymbol{\mu}_i) \subseteq B} \left\| \phi_\theta \left(H_i \right) - \boldsymbol{\mu}_i \right\|^2 \tag{3.1}$$

The objective of the system identification module is defined as the selection of the set of weights θ that minimizes the error in the parameter prediction (Equation 3.1). The optimal weights for the neural network ϕ minimizes the squared difference between the true parameter, μ, and the predicted parameters. The prediction is performed by the neural network with the rollout of recent experiences, H_i. The parameters and the rollout of experiences are stored in a memory buffer, B, that is shuffled to improve stability during training. Shuffling the data, specially when the buffer is large, allows the training to be unbiased to the neighboring regions and provide a more robust inference.

The system identification module followed specific strategies depending on the parameters in hand. The use of dense layers, although being efficient in inferring the parameters, does not keep a historical context of previous inferences. This means that

the information given to the network has to be better selected as useful information for inference. For this reason, we adapt the system identification module to collect experiences rollouts only in relevant state transitions. To train the wind magnitude inference system, every state transition is included in the memory buffer. The wind may influence the motion of the vehicle throughout the episode, therefore the inference of this parameter is active at every timestep. The experiments with a variation of friction coefficient, on the other hand, only offer relevant information when the vehicle and the platform bodies are colliding. Therefore, we train the system identification module for friction coefficient inference by feeding only the experiences rollout after the first contact between the two bodies; in contrast, the wind magnitude inference is performed with experience rollout throughout the episode.

3.3 System Identification

With the simulation setup that is described in Section 4.1, we trained the system identification module. We trained the system identification module with two hidden layers of 128 neurons with $ReLU$ activation function [24] between them and *tanh* in the last layer due to its ability to handle negative outputs, which might be used in some dynamic parameters. The $ReLU$ activation functions were defined in the hyperparameter search, and the *tanh* was adopted to normalize the output from parameters that have negative values, such as wind magnitude.

Some design choices attempt to decrease the complexity of the model. The architecture that we adopted for this work is simpler compared to the one used in [11], which uses three hidden layers to approximate the system identification. This design choice attempts to generalize the model to the real-world instance. Similar to the regularizers, using a simpler architecture forces the system to use less complex func-

tions to approximate the system identification function. Additionally, we dedicated 10% for the training data for validation. This would increase the chances that the training data is generalizable to other simulated instances.

The sysID modules to predict wind and friction coefficient were trained in a different fashion. The estimation of parameters is performed by finding patterns in the data that indicate how the parameter affects the interaction with the environment. The inference is then is performed, as explained in the previous chapter, by collecting rollouts of experiences with the environment. However, not all the interactions with the environment may be influenced by the dynamics parameter, and including these experiences would not contribute to the estimation of the parameter. Therefore, we trained wind and friction in two different ways: experiences with wind would always affect the state transitions since from the initial state of the UAV to the landing, wind can affect the trajectory of the vehicle. For this reason, the rollout of every transition is taken into account. On the other hand, the friction coefficient only alters the experiences in which the vehicle is colliding with the moving platform. Since the vehicle already has in its state-space the boolean indicating contact with the moving platform, the experiences are only concatenated for friction prediction when the boolean is true.

Experiences gathered by the UP were used to train the sysID module. We collected the data to train the sysID from episodes of the trained UP interacting with the environment. This ensures that the system identification module is specialized in the states and actions that the policy generally takes. The first approach for this was to create a simplified environment in which we could replicate the experiences that are relevant to the parameter. In the friction coefficient module, it was randomly started near the platform and also with random velocity. In this way, the system would be able to predict the friction coefficient with the vehicle coming from different scenarios.

However, we realized that training the system using the universal policy would result in an estimator that is trained in the scenario that it has to tackle, which could result in a more robust sysID.

3.4 Reinforcement Learning

The reinforcement learning approach in our work has a state space with two key elements. The first is the relative position and velocity, both in $\mathbb{R}^3$. This gives the policy real-time feedback of the progress towards the landing. We use relative positions so that the controller is agnostic about the absolute position of the UAV and the platform. Second, we add a binary variable that indicates if there is physical contact between the platform and the vehicle. Adding this state allows the reinforcement learning system to correlate a positive reward with landing on the platform. The action space is defined as the three-dimensional target velocity. Finally, the reward-function has three components:

$$reward = -\alpha|a_t - a_{t-1}| \tag{3.2}$$

$$-\gamma(ref_x^2 + ref_y^2 + ref_z^2) \tag{3.3}$$

$$+\beta(\eta \cdot (ref_x^2 + ref_y^2) + 1) + \phi \tag{3.4}$$

The first element of the reward function (Equation 3.2) is designed to avoid the policy to create periodic movements. It is the magnitude of the difference between the current action and the previous one, in which a_t is the action taken in timestep t. This encourages the policy to take a smoother sequence of actions, which is more likely that the real controller will be able to execute. The second element (Equation 3.3)

is a quadratic penalty of the relative position between the vehicle and the platform, where ref_x, ref_y, ref_z are the variables containing the relative positions in the three axes. This creates an incentive for fast conversion for landing. The third element (Equation 3.4), lastly, creates an incentive for the policy to land the vehicle in the center of the platform. It is a quadratic function that returns the maximum reward when the vehicle lands exactly on the center of the platform. As the simulation does not terminate upon landing, the policy will continuously receive a positive reward inversely proportional to the squared distance between the vehicle and the center of the platform. The variables α, β, γ, η and ϕ are scalars that were defined heuristically; the adopted values are shown in Table 4.3.

The learning parameters were defined by performing hyper-parameter search on the simulation described in Section 4.1. The policies were trained using DDPG, in which the actor and the critic were defined as multilayer perceptrons with two fully connected layers of 200 and 100 neurons each. The action space exploration was defined with normal action noise. Additionally, the policies were trained with a batch-size of size 128 trained for $3 \cdot 10^6$ timesteps.

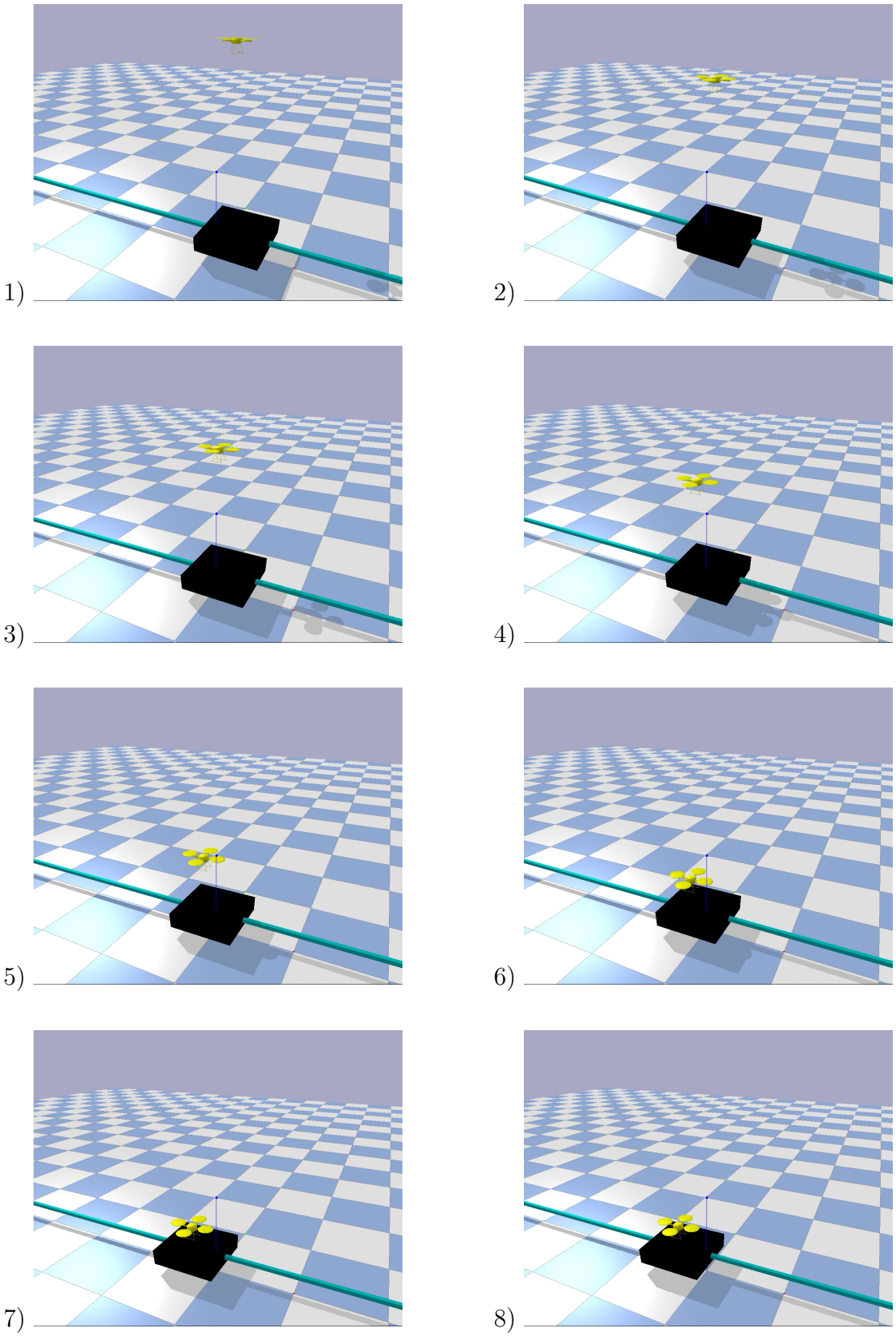

Figure 3.1: Sequence of eight frames showing the UAV landing on the moving platform in the simulated environment.

Chapter 4

Simulation and Experimental Setup

This chapter explains how we structured the experiments to verify the efficiency of the policies both in simulation as well as their ability to adapt to real world experiments. We start by discussing the results of the hyper-parameter search and our final design choices, then we proceed by providing details of how the training was performed; we talk about the experiments that were conducted in simulation; subsequently, we discuss how the real experiments were set up and structured. Lastly, there is a short discussion of previous simulation setups that were discarded and evolved into the current setting.

4.1 Training and Simulation Setup

In this section, we provide finer details of how the simulation was structured in order to both train and perform the experiments of the reinforcement learning policies.

4.1.1 Hyper-parameter Search

The hyper-parameter search is the process of selecting the best parameters of the learning system. It consists of the experimentation of different values to heuristically define the best set of parameters for the system. Producing robust deep neural net-

Table 4.1: Hyper-parameter search in DDPG

Parameter	Values
Batch-size	[32, 64, 128]
Actor LR	[1e-5, 1e-4, 1e-3]
Critic LR	[1e-2, 1e-3, 1e-4]
Activation function	[$ReLU$, $tanh$, Leaky $ReLU$]
Batch normalization	[True, False]
Exploration	[normal, decaying, OU, adaptive]
σ_1	[0.1, 0.2, 0.3, 0.5]
σ_2	[0.3, 0.5, 0.8]
ϵ	[0.99, 0.999, 0.9999]

work systems is highly dependent on the hyper-parameter set. Moreover, these parameters are highly dependent on the architecture of the network; the hyper-parameters used in a small network does not transfer well for a more complex network, even if both are trained on the same dataset [62]. Therefore, we produced a systematic hyper-parameter search to make sure that we have good parameters to compare the techniques and to perform experiments successfully. The hyper-parameter search was performed by analyzing two main design choices for the networks: the architecture for the system identification module and the actor-critic architecture.

The first hyper-parameter search was dedicated to finding an efficient parameter set for the UP and the domain randomization policy. We decided that each combination of parameters would be executed multiple times. Due to the stochastic nature of reinforcement learning, some runs may have a better result than others. Performing multiple experiments was essential to ensure that the performance of the policies were caused by the parameters and not by chance. We present Table 4.1 with some of the combinations that were used in the search for an effective policy. The row σ_1 is only varied when the exploration technique being executed is the normal noise; similarly, σ_2 executed with decaying noise, and ϵ with OU noise. A crucial factor for achieving the results was the exploration method. As previously explained, DDPG

takes deterministic actions when inserted with a state. In this way, the noise is generally applied in the final actions taken from the architecture. Therefore, as discussed in the second chapter, we perform the hyper-parameter search in three exploration methods: Ornstein Uhlenbeck process, normal action noise, and adaptive noise. We did not change the original settings for adaptive parameter noise since we adopted a small sigma value (0.1) according to the original paper description for a dense environment (frequent rewards) [42]. These combinations were not all trained with the same number of repetitions. The number of repetitions would vary according to the tests of reward functions; when evaluating a new element in the reward function, a small number of experiment would be run with to find an efficient set of coefficients (around 10 repetitions per configuration). As the reward function becomes more stable, we increased the number of experiments with less combinations (around 50 per configuration).

In system identification, we perform the search of hyper-parameters, as well as the method of data capture. When developing the system identification module, there were many choices to make to maximize the quality of the estimations in each parameter. First, as it was discussed in Section 3, the two dynamics parameters were dealt differently in the learning systems. In experiments with wind, the experiences were being constantly used for the wind estimation, while in friction coefficient, only when the vehicle is in contact with the moving platform. Part of the search was to evaluate what was the reasonable length of the buffer with the experience rollout that would feed the neural network. We trained the system identification module with the hyper-parameters shown in Table 4.1.1 for an experience buffer of 10, 20, and 30 experiences. The buffer contains sequential experiences in the form $e_t = (s_t, a_t)$ recording the states and actions in each timestep t. The batch-normalization row indicates the addition of a batch-normalization layer that could potentially stabilize the training

Table 4.2: Hyper-parameter search for system identification

Parameter	Values
Batch-size	[64, 128]
Hidden layers	[2, 3, 4]
Batch-normalization	[True, False]
Neurons per layer	[64, 128]
Activation function	[relu, tanh]

and cause faster convergence. The other parameters were trained in order to find the best set with minimal complexity so that the sysID is more generalizable and faster to run in real-time since this architecture is also used in real world experiences.

4.1.2 Simulated Environment

We implemented the simulated environment employing Pybullet, a Python interface for Bullet physics simulator [63]. The collision model of the platform and the UAV are simplified as prismatic objects instead of creating mesh molding the UAV's shape in order to speed up the simulation while still keeping realistic results in the interaction of the objects. The collision model of the platform and the UAV are defined as prismatic objects in order to speed up the simulation while keeping realistic results in the interaction of the objects. The visual representation of the UAV and the platform is shown in Figure 4.1. With this setting, we can process an average of 1225 steps per second using a Macbook equipped with an Intel Core i7-3635QM CPU, 2.40GHz. The UAV uses a velocity controller with bounded accelerations, according to Table 4.3.

4.1.3 Experimental Setup

In order to verify how the policies manage the dynamic variations, two dynamic variables are varied throughout the episodes: 1) the coefficient of friction between the

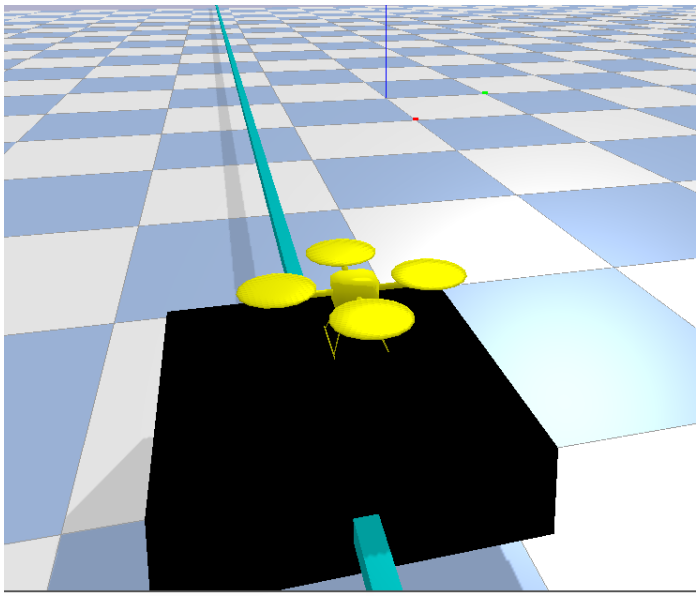

Figure 4.1: UAV and moving platform visual representation in Pybullet physics simulator. The platform has one degree of freedom to move along the y-axis through a prismatic joint.

UAV and the platform and 2) the average wind magnitude. Changes in these parameters require different actions from the policies to succeed. The episodes with wind variation can act reactively and adjust according to the disturbance; however, the friction coefficient requires previous planning from the policy, as there is no error recovery once the vehicle is colliding with the platform. The simulation of the friction coefficient is set by changing the dynamics of the simulation, as it is provided as a parameter for the simulation. The wind was modeled as an external force whose magnitude is sampled at every timestep from a Gaussian distribution to include additional noise. The mean is provided in the experiments, and is the value that the SysID module predicts. Modeling wind as a Gaussian distribution allows the insertion of noise in the data, which turns the system more likely to train a policy that is generalizable to real world scenarios. The system is integrated using OpenAI Gym, a framework that interfaces the simulation and the reinforcement learning agents,

Table 4.3: Simulation Parameters used in the experiments. The majority of the parameters were empirically defined.

Parameter	Value
Max RL steps	300
initial UAV height	$3m$
Simulation frequency	$100hz$
Policy frequency	$10hz$
Platform dimensions	$[0.8, 0.8, 0.2]$ m
UAV dimensions	$[0.4, 0.4, 0.3]$ m
Maximum velocity	$1m/s$
Max MP velocity	$0.2m/s$
Maximum acceleration	2.5 m/s^2
Friction Coefficient range	$[0.025, 0.2]$
Wind Magnitude range	$[0, 1]$
Motors off height	0.45 m
α	2
β	0.22
γ	7.0
ϕ	0.25
η	-3.07

enabling the use of reinforcement learning libraries such as OpenAI Baselines.

In the simulation, we adopted a similar setting as the training of the policies. Each episode starts with the UAV positioned in random x and y coordinates, and a fixed height, according to Table 4.3. This helps to decrease possible bias from the policies in performing under specific portions of the state-space. The velocity of the platform is kept constant and switches directions whenever it reaches a position threshold. The policies have a maximum time of 30 seconds to attempt the landing before a timeout. An example of successful landing in simulation is depicted in 8 frames in Figure 3.1. Table 4.3 also provides additional parameters used in the simulation that were defined empirically or selected to replicate the expected conditions of the real world experiments.

4.2 Simulation Structure

We compare the performance of domain randomization, UP+SysID, and the baseline. The baseline consists of a model that is trained with no variation of parameters, in which all the episodes have no wind, and the friction coefficient is 0.2. To investigate whether the methods are capable of generalizing to other variations of dynamics, we analyze two experiments with two parameters, one randomizing the coefficient of friction between the UAV and the platform and another randomizing the magnitude of wind. These parameters display changes in dynamics that are representative of other applications, such as action delay, air density, or coefficient of restitution. As it was previously stated, the friction coefficient requires more anticipated planning from the policy, while the variation of wind allows a more reactive strategy. Many other parameters affect the dynamics in a similar fashion; air density, for instance, can be instantaneously dealt as a damping force [64], while the coefficient of restitution requires previous planning since the energy after a collision might be irrecoverable.

4.2.1 First Experiment

In the first experiment, we evaluate how the system is able to perform under periodic variations of parameters. The parameters are randomly sampled on intervals of five episodes. This range allows a fair comparison among the policies, as the UP+SysID policy has no prior knowledge of the parameters in the first episode after the parameter is modified. This experiment verifies how the policies would perform in case the parameters are unknown at first interaction. The metrics for this experiment consist of the number of successful landings, the number of times that the policy attempts landing and fails, the number of timeouts, and the average number of seconds before the first physical contact between the vehicle and the platform. The timeout is the

number of times that the maximum number of simulation steps was reached and there was no contact between the UAV and the platform. This measure indicates how frequently the policy is hesitant to land or fails to approximate the platform. The average time to land indicates whether the policy is leveraging the information of the dynamic variables to perform the landing faster.

4.2.2 Second Experiment

The second experiment verifies the biases of the parameter values on the policies' performance. Instead of randomly sampling the variables, it iterates fifty intervals in the parameter range and evaluates the performance within 25 episodes. The range used for friction coefficient was from 0.025 to 0.2, and for wind from 0 to 1; both values are within the range that the policies were trained on. The time to land and the success rate is evaluated in the second experiment. The UP+SysID policy is not employed in this experiment as it is upper bounded by the UP with real parameters, and its results on the experiment can be confounding with the results of the SysID estimation.

Certain behaviors are expected in the experiments with wind and friction coefficient. In wind variation, a successful setting is to accelerate or decelerate depending on the position of the platform and the direction of the wind. In order to land safely, the policy has to leverage or fight against the wind disturbance, as the wind can be in favor or against the movement towards the platform. In experiments with friction coefficient, when the coefficient is small, the policy has to approach the platform either near to the edge, so it has enough space to slide on the surface to reduce kinetic energy or land with small relative velocity on the x and y axes. The latter is a safer manner to achieve the task, while the former is faster since it does not require the vehicle to decelerate to complete the task.

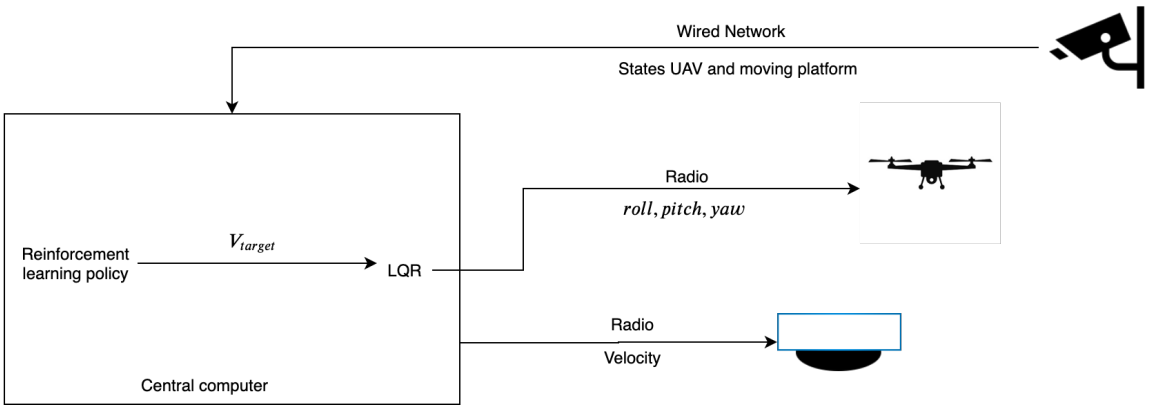

Figure 4.2: Diagram depicting the approach of the real-worl experiments. The vicon motion capture system collects the state information of both the UAV as the moving platform, which is used as input fot the reinforcement learning policy. The ouput of the policy is a target velocity wich is input to the LQR controller, along with the state information from vicon. The LQR controller, in turn, sends the values of roll, pitch and yaw to the Pixhawk autopilot that controls the vehicle. The moving platform, in similar fashion, receives commands from the central computer according to its state that is estimated by the motion capture system.

4.3 Real World Experiment Setup

After the simulation analysis, we performed experiments to evaluate the performance of the policies when transferred to the real world domain.

We chose real systems that behave similarly to the simulated environment. The UAV utilized for the experiments was a DJI Flame Wheel F450 with a Pixhawk autopilot (Figure 4.3). The moving platform is built with an adapted Roomba vacuum equipped with a landing platform on its top (Figure 1.1). The state of the system was provided by a Vicon motion capture system, which is transformed into the necessary relative positions and velocities. The target velocities provided by the policies are provided to an LQR controller [65] that translates it into attitude commands for the autopilot. Figure 4.2 shows a diagram depicting the setup employed in the real-world experiments. The external disturbance, the wind, is produced by a fan positioned at

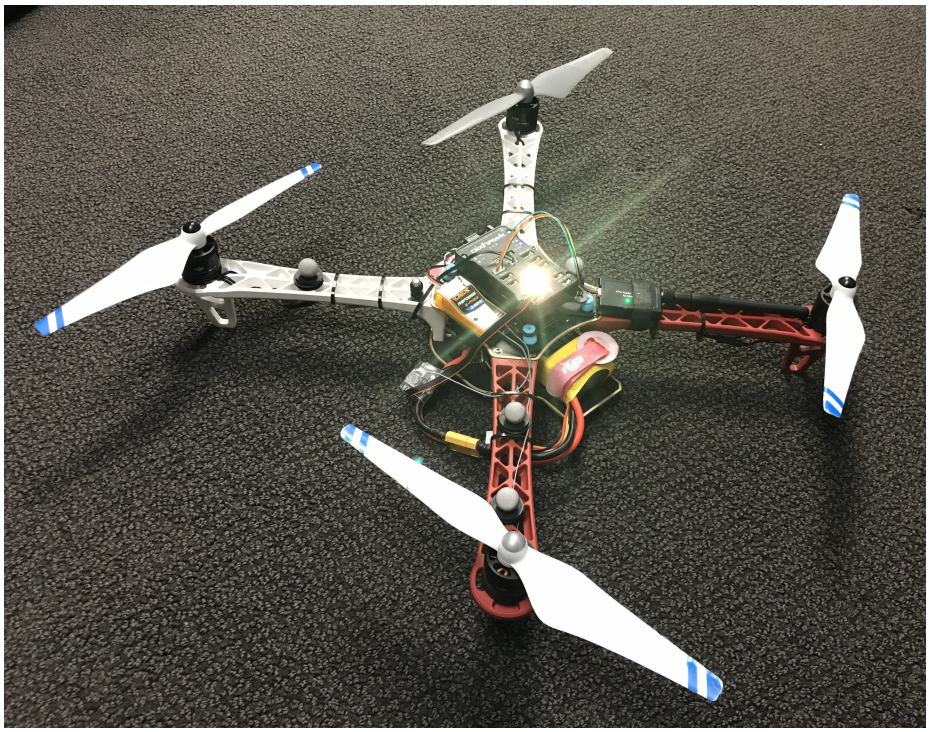

Figure 4.3: UAV employed in the experiments.

one extremity of the space, producing winds as strong as 4 m/s. The fan, displayed in the left part of Figure 5.1, was static and positioned along the x-axis.

4.4 Experiment Structure

The experiment was divided into four categories: each is a different task combination of the elements in the environment. The first setting consisted of the platform positioned centered and still. This evaluates whether the policy is able to transfer from the simulation to the real world and adapted to an environment in which the vehicle is not in constant motion. The second setting consists of having the platform in movement, but no external disturbance. This is the equivalent environment in which the baseline is trained in the simulation. In the third setting, the platform is still, but there is a presence of wind; lastly, the fourth setting, the platform is in motion, and

there is wind. In this way, it is possible to isolate the elements that could possibly cause the system to fail.

These design choices avoid ambiguity in the results. The experiment in each combination is executed twice, in which the vehicle is initialized in six positions, according to Figure 5.1. These positions are chosen such that they test whether there is a poorly trained state that causes unexpected behavior and how the wind affects the experiment. An example is that one of the points is directly in front of the fan, and another one on the opposite side, forcing the vehicle to go for and against the wind. For the sake of avoiding confounding variables in the experiment results, a fully charged battery is used at the beginning of each of the runs.

4.5 Alternative Simulation Setups

We adopt Pybullet as the platform for the simulated environment. Pybullet is a Python wrapper for Bullet Physics SDK, written in C++. Bullet is the physics engine for renowned software such as Maia, Blender, and Cinema 4D. Pybullet is also very efficient and has been used in the simulation of many complex robotic systems to perform reinforcement learning tasks, such as Robotic grippers [66], quadruped robots [59], humanoids [67] and even robotic hands playing ping-pong [68]. A realistic physics simulator increases the chances of transferring the policy to the real-word [69, 70]. The closer the simulation dynamics is to the real world, the more prepared the policy will be in the dynamic interactions with the real world. However, the models are generally not identical and still require additional strategies to bridge the gap [69].

Before developing the Pybullet environment, we attempted other approaches. The first version of the simulation consisted of a Simulink simulation with an LQR controller. LQR controllers are highly adopted as they are guaranteed to operate opti-

mally. We initially attempted to use a Simulink implementation from NIMBUS lab that includes the model of a quadcopter with LQR controller. To be able to perform communication with the simulation we interconnected the Simulink simulation with the Python reinforcement learning modules using ROS and OpenAi Gym. Additionally, this system was trained in the university's cluster inside a Docker container since some of the modules in the stack were not natively supported in the cluster environment. The environment was successfully implemented, but it resulted in a slow system. This setting was not sufficient to train a reinforcement learning system in feasible time.

The second version of the simulation was implemented using a PID controller. PID controllers are frequently used in Unmanned Aerial Vehicles due to the simplicity of implementation. However, tunning the parameters to produce a good controller is not an easy task and is generally performed heuristically. This is so as PID controllers instead of modeling the system dynamics, generates the actions according to the error of the system state and the target state. There are many different several formal techniques to attempt shortening the tunning-process [71], but they are not guaranteed to succeed. We have tried tunning the PID controllers with a formal method described in [71], but with no success. Not only we tried the formal methods, but also followed instructions from an experienced Professor in the matter to no avail. After the unsuccessful attempt with PID controllers, we decided to adopt a more straightforward implementation in simulation and shape the reinforcement learning reward system to prepare the policy to transfer well to the controller used in the real robot.

Chapter 5

Results

In order to define the performance of different strategies in training the policy, we adopted two methods of evaluation: the simulations and real-world experiments. We first report the results of simulated environments with friction and wind variation, then discuss the results of the real-world experiments.

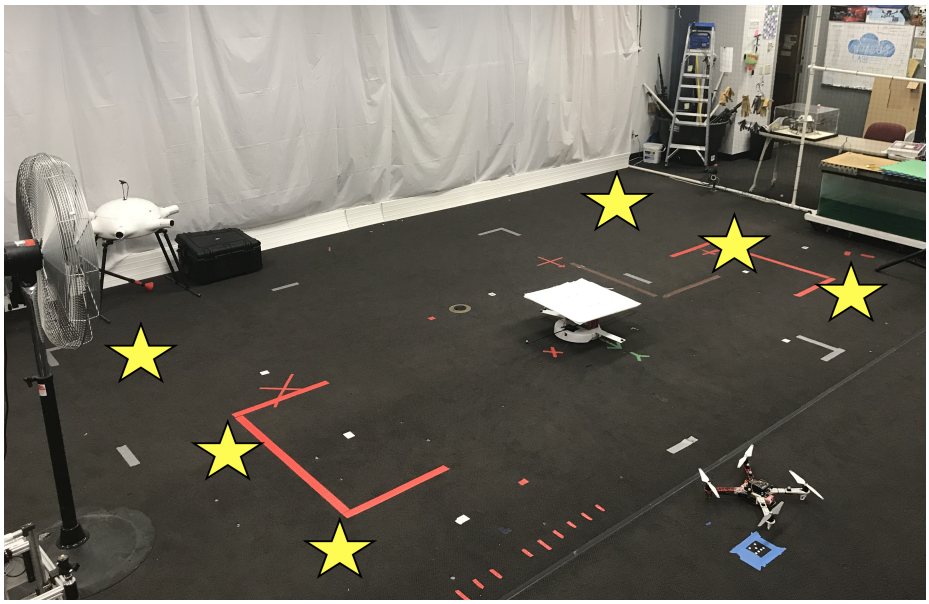

Figure 5.1: Experimental setup of the real-world experiments. The yellow stars mark the six initial positions that are initialized twice during an experiment. At the left of the image is the fan used to generate the disturbance; bottom-right shows the UAV used in the landings and, at the center, the Roomba robot with the moving platform.

5.1 Hyper-parameter Search

We performed the hyper-parameter search according to the description in the last chapter and defined the architectures accordingly. Here we provide insight into our findings and some of our intuition behind it. We start by talking about the reinforcement learning architecture, then discuss the system identification module.

5.1.1 Reinforcement Learning

Performing the search for reinforcement learning was a complex task. Having the wrong set of parameters could dampen or even impede the convergence of the system, as it happened in many configurations. As we were analyzing the results, some configuration showed better results, but with fewer policies reaching convergence. This could be a result of the stochasticity of the initial weights of the network and the exploration function, so we kept the best reward as the criteria for the parameter search. The hyper-parameter search involved a very high use of computational resources, so we could not increase the number of experiments to eliminate this question of convergence. Next, we talk about some of the interesting results from the search.

The best type of noise in our experiments was, surprisingly, the normal noise. This may be associated with the manner that the rewards are distributed in the system and the type of actuation. As previously discussed, OU noise is generally employed due to the fact that the error is correlated, and has a more significant influence in environments whose final behavior is two derivatives away (e.g., forces and position). In our case, the action space is the relative position and velocity, which are translated into different positions in only one derivative; therefore, the changes in the action space will be reflected more rapidly in the system's behavior. Moreover, we had the intuition that the adaptive parameter noise would generate better results,

but it was not the case. One possible reason is that the network is highly sensitive to perturbations in the parameter set, which causes the system to underperform. This is similar to what was reported for the inverted pendulum task in the original paper [42]. Another cause could be that we should have analyzed different parameters of sigma, that we avoided due to the already high number of parameter combinations. Decaying gaussian also performed well, but not as well as pure Gaussian noise, so we preferred to keep Gaussian noise due to its performance and algorithmic simplicity.

A significant part of this project was allocated for reward shaping. The first trial was using the values specified in Rodriguez et al. [7]. This did not produce satisfying results, maybe because we are attempting three-dimensional action space, and the paper uses a two-dimensional one. Different combinations of elements of their reward function also did not produce good results. This could be due to the addition of the third velocity in control, but could also be affected by the difference in simulations. From this point on, we started incorporating different elements according to the behavior that was observed in simulation.

The coefficients of the positive reward and the negative reward (λ and β in Equations 3.2 and 3.4, respectively) of the squared relative position also greatly affects the behavior of the system. We observe specific behaviors according to the magnitude of these coefficients. If the negative reward is too high, then the system might avoid performing additional strategies beyond approaching the platform, such as finding immediate episode termination by plummeting into the ground. The first attempt to solve this problem was to include a negative reward every time the vehicle reaches the floor. However, this caused another problem: it discourages the vehicle to land, since landings might fail, causing the vehicle to fall. This resulted in policies that approach the platform, and instead of landing, it hovers over it. The solution was to find a suitable value for λ that stabilizes the system.

A very high positive reward when the vehicle is in contact with the platform disregards the path of the vehicle until it reaches the platform. The agent attempts to interact in the environment, maximizing the reward function. This is started with random actions and adjusting them in a way that the expected reward increases. When the positive reward is too high, the negative reward from the path would be minor compared to the treasure that it is to arrive on the platform. This resulted in clumsy policies that would oscillate considerably and take non-optimal paths to the platform since it would not attempt to correct the path as it does not affect much the overall accumulated reward. A partial solution for this was simply to find a suitable value of β and λ that encourages landing in the center spot, but also encourages the system to optimize the path to the platform.

Even after the coefficients were defined, the vehicle would oscillate (shaking) when going towards the platform. To solve this, we decided to include another element in the rewards function: a negative reward proportional to the derivative of the actions. The benefit of this addition was twofold, the oscillation of the vehicle would be gone, but also the policy would be more likely to be transferable to a real vehicle. Previously, the vehicle could do more sharp changes in the velocity, which would only be limited by the acceleration cap in the simulation. Including this element in the reward function turned the selected actions much smoother, which are more likely to be executed by the LQR controller in the real vehicle.

The combination of finding the right set of hyper-parameters and the right reward function renders this problem quite challenging. This is associated with the fact that reinforcement learning systems have to interact with the environment as it learns, different to supervised learning techniques that collect the data only once, but also because the hyper-parameters in one reward function would not necessarily work well in another. On top of that, multiple runs with the same parameters had to executed

due to the previously mentioned stochasticity of reinforcement learning systems. A single run, which trains the policy over multiple episodes totaling 3×10^6 timesteps, would last for about eight hours, so using parallel executions on HCC clusters was essential for this work. Despite these challenges, we were able to find the right elements and coefficients in the reward function, after several days of experimentation, to perform the reinforcement learning training with success. It is worth noting we ran experiments according to the observed behavior and intuitive tradeoffs as discussed, so a future extension of this work could perform a methodical search for optimization of these parameters to maximize the performance of the policies.

5.1.2 System Identification

After analyzing the results of the hyper-parameter search, we decided on the architecture for the system identification module. In this search, we were prioritizing simpler networks in order to have better chances to use this module in the real-world. In our experimentation, $ReLU$ had superior performance than $tanh$ as an activation function. This is generally the case in many applications since $ReLU$ does not saturate the gradients, which can accelerate learning [72].

The results of the architectures were similar, so we adopted the simpler one. This held true both for wind and friction coefficient; moreover, we used the same architecture for both. The final architecture consists of two hidden layers with 64 neurons each and interleaved with batch normalization layers. The system is trained with an Adam optimizer minimizing the squared error of the estimations. An example of the training of the system identification training is shown in Figure 5.2; the optimizer quickly decreases the loss and stabilizes, when the training is halted to avoid overfitting.

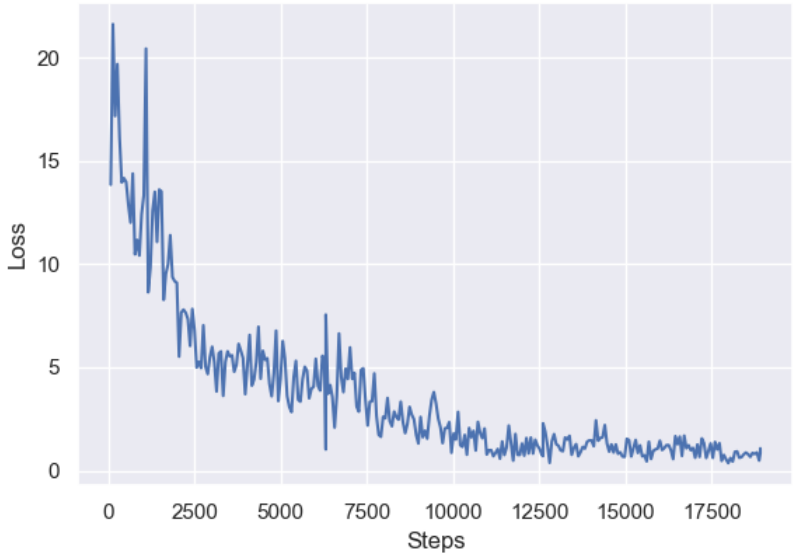

Figure 5.2: Loss of supervised learning for wind estimation

5.2 Simulation

5.2.1 Friction Coefficient

Table 5.1: Performance of the policies with varied friction coefficient. In an interval of five episodes, a new friction coefficient is sampled.

	Baseline	Domain randomization	UP + True Params	UP + SysID
Success	127	375	352	360
Falls	373	125	137	130
Timeouts	0	0	11	10
Avg. landing time	2.60	5.32	3.40	3.58

The first experiment examines how the policies adapt to successive episodes of friction coefficients. Table 5.1 shows how the policies compare on a high level. Falls is the number of times that the policy failed in landing and fell on the ground; Timeout is when the maximum number of steps was reached and the vehicle did

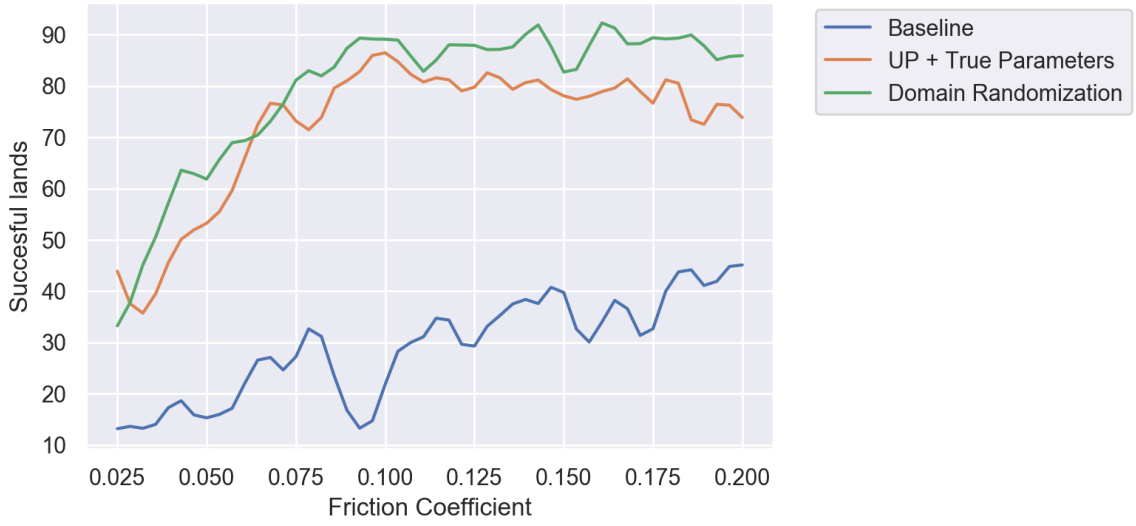

Figure 5.3: Percentage of successful landings given the friction coefficient between the vehicle and the platform. A smaller friction coefficient means that the vehicle will slide on the platform if it lands with a non-zero velocity vector along to the surface of the platform. The blue line represents the baseline, green is the domain randomization policy and orange represents the UP with true parameters.

not attempt landing; Landing time is the average time in seconds that the vehicle takes for the first physical contact with the platform. The domain randomization policy has the best success rate, but also the lowest average landing time. UP in both scenarios outperform the baseline. In this test, the Universal Policy with SysID slightly outperformed UP with true parameters, but the difference is not statistically significant.

Many observations that can be drawn from this result. First, domain randomization outperforms both UP policies but with a much higher average time. This may be an indication that the domain randomization learned a more careful policy that is conservative across all the friction coefficient values. On the other hand, the UP policies have a shorter landing time as they might approach the moving platform according to the observed friction coefficient. That is, landing faster when the friction

coefficient is high and takes a longer time when it is low since that landing would have to be approached in low x and y velocities to avoid sliding off the platform. Second, UP policies are the only techniques that have timeouts. This may be caused in very low friction coefficients, where the confidence of the policy to perform the landing is very low, and the system would have a better overall reward by just hovering on the top of the platform.

The second experiment investigates whether there is a range of parameters in which the policies could be biased. The results are shown in Figures 5.3 and 5.4. The policies perform poorly when the friction coefficient is closer to zero, as it is expected (Figure 5.3). Both the domain randomization as the UP policies have a steep increase in performance, reaching a plateau in 0.1. This would suggest that the strategies used in both policies are similar; however, Figure 5.4 shows how the time to land is decreasing with the increase of friction coefficient when UP is adopted. This would suggest that the policy is using the friction coefficient to measure how aggressively it is able to land on the platform while keeping a high success rate. This could be further verified by performing experiments observing the path that the vehicle executes to approach the platform; if the UP takes paths closer to a straight line as the friction coefficient increases, our analysis is further ratified. Although it is able to decrease significantly the average time to land, its success rate is almost always higher bounded by the domain randomization policy. Another interesting phenomena is the sudden increase of landing time close to friction coefficient 0.1. We believe that this could be caused by a bias during training, that favoured other coefficients. An approach to prevent this bias in future work could be consistent of training that is only halted when the performance of the agent is consistent across the parameter range. A first attempt can be consisted of simply verifying from time to time the performance across the spectrum of parameter and present more episodes

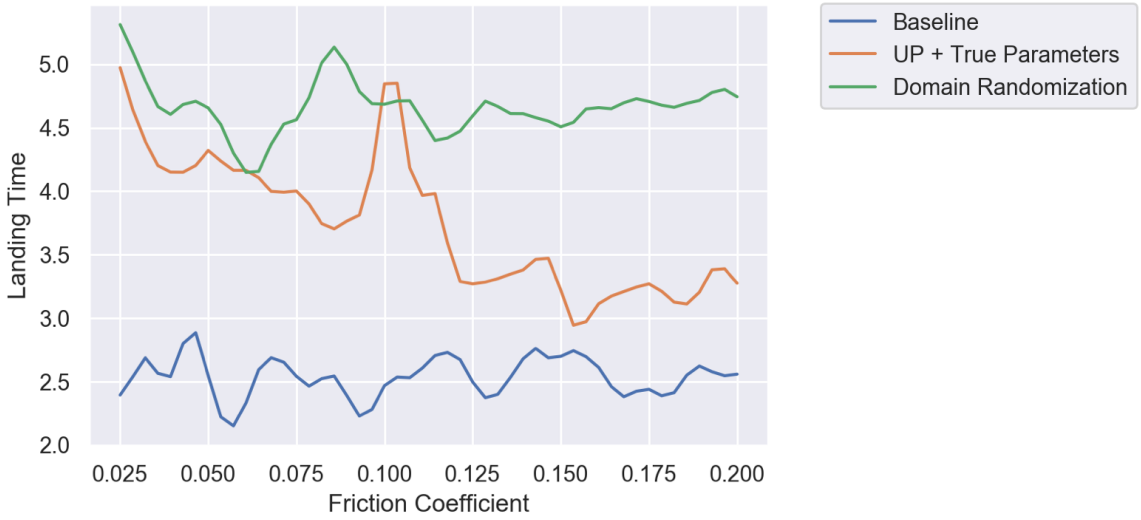

Figure 5.4: Landing time between the beginning of the policy and the first physical contact between the platform and the vehicle. The blue line represents the baseline, green is the domain randomization policy and orange represents the UP with true parameters.

of the parameters with poor performance.

5.2.2 Wind

Table 5.2: Performance of the policies with varied wind magnitude. In an interval of five episodes, a new wind magnitude is sampled.

	Baseline	Domain randomization	UP + True Params	UP + SysID
Success	319	465	426	399
Falls	181	34	56	101
Timeouts	0	1	18	0
Avg. landing time	2.24	3.6	4.6	10.9

The results of the first experiment are shown in Table 5.2. It shows that the pure domain randomization policy, as well as the Universal policy, are able to outperform the baseline. However, the policy with pure domain randomization is more accurate than the Universal Policy as it succeeds in 93% of the trials compared to 85% from

UP. This might be related again to the complexity of adding one more element in the state-space since all the policies were trained with the same number of timesteps, and spaces with higher dimensions may need significantly more sampling. Additionally, it might be necessary to perform a hyper-parameter search only for each of the policy type, instead of using a single hyper-parameter set. Observing the UP polices, the replacement of the True params to the SysID module does not affect the performance significantly on the simulation, only circa five percentage points. This means that, first, the policy is indeed using the information from the dynamic parameter to ponder the selection of the next action, and second, the sysID module is doing a good job of predicting the wind module.

The gap of the performance of the UP + real and UP + SysID indicates that the system identification module is not overfitted to this environment, as the model would be very close to the real values and result in extremely similar performances. A counter-argument would be that the final policy is very sensible to the variations of parameters, which could be the case, and we intend to analyze these possibilities in future work by performing ablation studies of the influence of the sysID perfomanc in the UP performance.

The second experiment allows a closer analysis of the parameter's influence in the policies' performances. Similar to the friction experiments, twenty-five experiments are run in an interval of normalized wind magnitudes. The results are shown in Figures 5.5 and 5.6. Figure 5.5 shows how the policies keep their performances relatively constant througout the experiment. However, Figure 5.6 shows an interesting phenomena with the domain randomization and the UP: the domain randomization policy starts off with better performance, but as the time progresses, the universal policy is able to perform the task faster; finally, at point 0.5 of wind magnitude, the UP is consistently faster than the domain randomization policy. We believe that at

Table 5.3: Results of the real-world experiments. Each cell correspond to the percentage of successful landings in a combination of policy and setting with 12 trials.

	Baseline	Domain randomization	Universal Policy + SysID estimation
Still	91.67%	100%	100%
Moving	100%	100%	83.33%
Still + Wind	75%	100%	100%
Moving + Wind	75%	100%	83.33%

this point having information of the wind magnitude becomes more relevant and gives an advantage for the policies that account for it to arrive to the platform. Therefore, the universal policy makes use of this information to complete the landing faster. We believe that the value in which the policies cross in the chart could shift to the right in case the UAV is more aggressive in acceleration. This is so as the knowledge of the wind becomes less important when the controller can change its velocity more rapidly. In the same manner, if the maximum acceleration is smaller, the point of crossing would shift to the left. This phenomena could be further verified with an experiment that verifies a third dimension of this chart: maximum acceleration. This can be performed by performing this same experiment in a range of maximum accelerations.

5.3 Real-world

The results of the real-world experiments are shown in Table 5.3. All the policies are able to perform landing on the still platform, successfully transferring the knowledge from the simulation to the real world. As the complexity progresses, only the policy with domain randomization is capable of finishing the task with 100% of success. The UP+SysID policy is able to perform well with the presence of wind, but has its performance diminished with the movement of the platform. The baseline, in accordance with the simulations, is able to perform well in cases without the external

disturbance, but has its performance decreased with the addition of wind. This indicates that the policies trained with wind simulation indeed generate policies that are robust in the real-world. The UP+SysID policy had decreased performance in the last scenario with wind; this could be caused by a mismatch on how the wind is modeled in the simulation and how it actually affects the controller in the real world. Such difference can result in low accuracy of the SysID, which in turn depletes the UP performance. Therefore, from this experiment suggests that domain randomization is a safer technique to improve robustness of the policy to disturbances as it does not depend on a model accuracy to infer the dynamics.

Visually, the UP policy seems to be more stable. Observing the path that the policies take until landing, the UP is visually the most stable one, since the domain randomization policy despite being the best one in landing, does not always have a smooth trajectory. Maybe the instability of UP policies are associated with the system identification imprecision in the real-world. The system identification could be failing in modelling phenomena such as the ground-effect when the vehicle is closer to the platform. This can be verified with a future experiment by performing wind readings in selected points of the environments and compare with the sysID results.

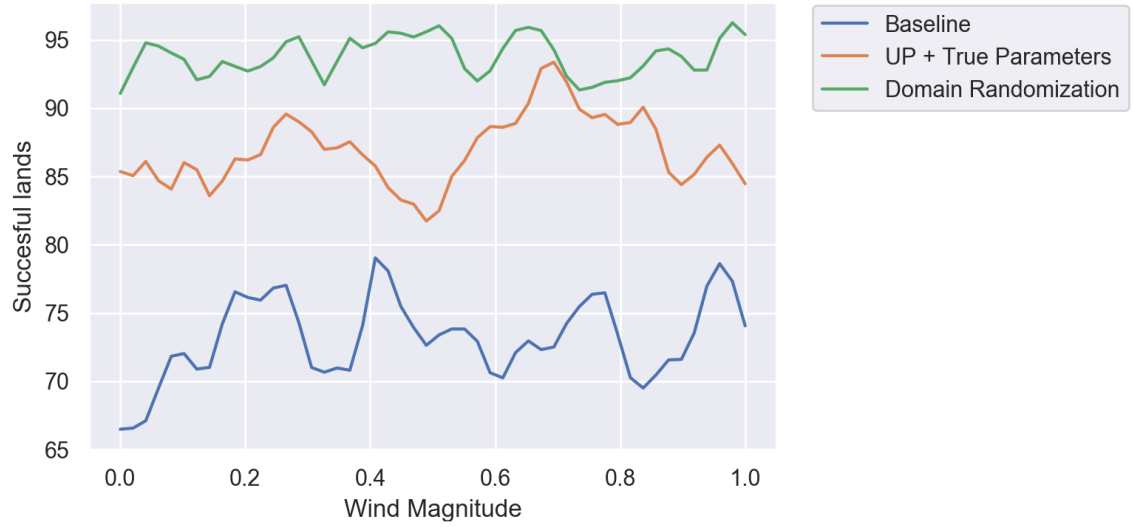

Figure 5.5: Percentage of successful landings given a average wind magnitude. The wind is sampled at every timestep from a Gaussian distribution whose mean is the value in the x-axis of the figure. The blue line represents the baseline, green is the domain randomization policy and orange represents the UP with true parameters.

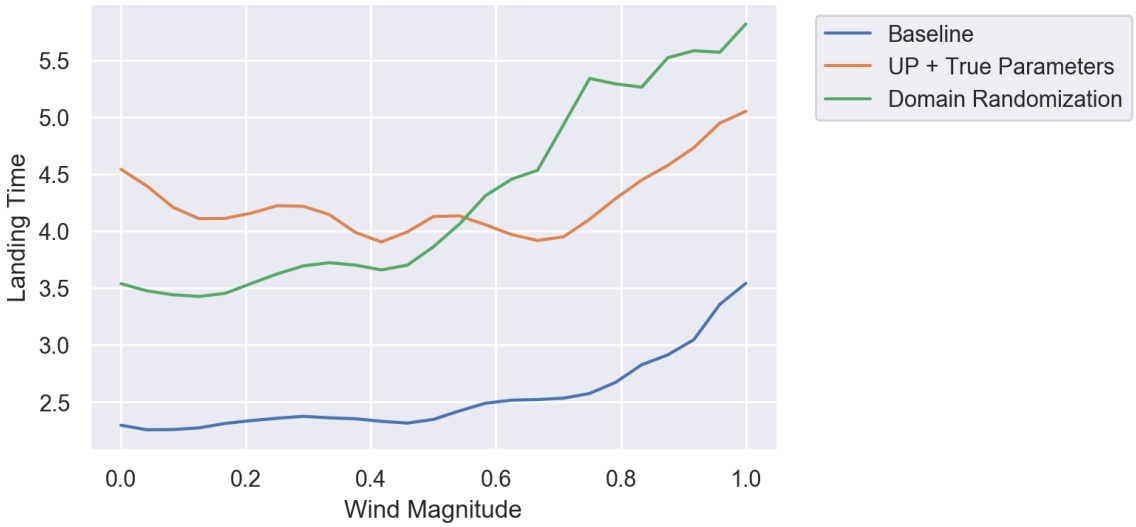

Figure 5.6: Landing time between the beginning of the policy and the first physical contact between the platform and the vehicle. The blue line represents the baseline, green is the domain randomization policy and orange represents the UP with true parameters.

Chapter 6

Conclusion

This work presents three contributions: first, a reinforcement learning system capable of control an Unmanned Aerial Vehicle to automatically land on a moving platform. We ran simulated and real-world experiments to confirm the efficiency of the learning set up. The system is able to perform the task with and without the presence of wind. In order to be able to make the policy robust, we investigate strategies of previous works.

Second, we train the policy in a fast simulated environment. We describe our training using Pybullet, a python wrapper of bullet physics simulator. We describe how we speed up our simulations by simplifying the simulated environment and shaping the reward function to increase the chances of transferring to the real-world.

Third, this thesis also investigates two strategies of training and transferring policies learned in simulation to disturbed environments: domain randomization and Universal Policy with System Identification (UP+SysID). We have conducted experiments both in a simulated environment as well as in a real-world scenario to compare these two techniques against a baseline. The experiments lead to the indication that domain randomization, which exposes the policies to the different varieties of environments in simulation, produces a more effective policy than including the parameters in the state-space, which increases the dimensionality of the observations. We discuss

how our UP+SysID implementation could be losing performance in estimating parameters in the real-world. On the other hand, using UP+SysID resulted in policies that make use of the information of the parameters to perform the tasks generally faster.

Future work includes the implementation and analysis of techniques of system identification to improve the quality of the estimation in the real and a systematic study of how the accuracy of SysID affects the performance of Universal Policies. There are many ways that this work can be extended. First, the system identification module can be extended to receive training with real-world data. It is unclear what was the performance of the SysID module in our real-world experiments, so if it is performing poorly, there is the chance that this policy could be highly efficacious with a proper estimation.

Another line of study would be to train more than one dynamics parameter at a time. When multiple parameters are adopted, there might be a relationship between them. Some papers, such as [73], incorporate the parameters in a latent space, which could define their internal dependencies and their relevance for the task. Another option would be to train the SysID with a methodology similar to physics informed neural networks [74, 75], which incorporates previously defined physics constraints in the prediction of the target value.

Bibliography

[1] Imad A Basheer and Maha Hajmeer. Artificial neural networks: fundamentals, computing, design, and application. *Journal of microbiological methods*, 43(1):3–31, 2000.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[4] Kevin Ling, Derek Chow, Arun Das, and Steven L Waslander. Autonomous maritime landings for low-cost vtol aerial vehicles. In *2014 Canadian Conference on Computer and Robot Vision*, pages 32–39. IEEE, 2014.

[5] Lorenzo Marconi, Alberto Isidori, and Andrea Serrani. Autonomous vertical landing on an oscillating platform: an internal-model based approach. *Automatica*, 38(1):21–32, 2002.

[6] Daewon Lee, Tyler Ryan, and H Jin Kim. Autonomous landing of a vtol uav on a moving platform using image-based visual servoing. In *2012 IEEE international conference on robotics and automation*, pages 971–976. IEEE, 2012.

[7] Alejandro Rodriguez-Ramos, Carlos Sampedro, Hriday Bavle, Paloma De La Puente, and Pascual Campoy. A deep reinforcement learning strategy

for uav autonomous landing on a moving platform. *Journal of Intelligent & Robotic Systems*, 93(1-2):351–366, 2019.

[8] Donald E Kirk. *Optimal control theory: an introduction*. Courier Corporation, 2004.

[9] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[10] Pierre-Yves Oudeyer and Frederic Kaplan. What is intrinsic motivation? a typology of computational approaches. *Frontiers in neurorobotics*, 1:6, 2009.

[11] Wenhao Yu, Jie Tan, C Karen Liu, and Greg Turk. Preparing for the unknown: Learning a universal policy with online system identification. *arXiv preprint arXiv:1702.02453*, 2017.

[12] Jingru Luo and Kris Hauser. Robust trajectory optimization under frictional contact with iterative learning. *Autonomous Robots*, 41(6):1447–1461, 2017.

[13] Frederico AC Azevedo, Ludmila RB Carvalho, Lea T Grinberg, José Marcelo Farfel, Renata EL Ferretti, Renata EP Leite, Wilson Jacob Filho, Roberto Lent, and Suzana Herculano-Houzel. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, 513(5):532–541, 2009.

[14] HB Barlow. Temporal and spatial summation in human vision at different background intensities. *The Journal of physiology*, 141(2):337–350, 1958.

[15] Marina A Lynch. Long-term potentiation and memory. *Physiological reviews*, 84(1):87–136, 2004.

[16] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[17] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[18] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14:8, 2012.

[20] Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004.

[21] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.

[22] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[23] Balázs Csanád Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48, 2001.

[24] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[25] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[26] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[27] Twan van Laarhoven. L2 regularization versus batch and weight normalization. *arXiv preprint arXiv:1706.05350*, 2017.

[28] John C Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society: Series B (Methodological)*, 41(2):148–164, 1979.

[29] Yevgen Chebotar, Karol Hausman, Marvin Zhang, Gaurav Sukhatme, Stefan Schaal, and Sergey Levine. Combining model-based and model-free updates for trajectory-centric reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 703–711. JMLR. org, 2017.

[30] JCH Christopher. Watkins and peter dayan. *Q-Learning. Machine Learning*, 8(3):279–292, 1992.

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[32] Zhenshan Bing, Claus Meschede, Guang Chen, Alois Knoll, and Kai Huang. Indirect and direct training of spiking neural networks for end-to-end control of a lane-keeping vehicle. *Neural Networks*, 121:21–36, 2020.

[33] Sahand Sharifzadeh, Ioannis Chiotellis, Rudolph Triebel, and Daniel Cremers. Learning to drive using inverse reinforcement learning and deep q-networks, 2016.

[34] Fangyi Zhang, Jrgen Leitner, Michael Milford, and Peter Corke. Modular deep q networks for sim-to-real transfer of visuo-motor policies, 2016.

[35] Stephen James and Edward Johns. 3d simulation for robot arm control with deep q-learning, 2016.

[36] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning, 2016.

[37] Lei Tai and Ming Liu. Towards cognitive exploration through deep reinforcement learning for mobile robots. *arXiv preprint arXiv:1610.01733*, 2016.

[38] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2015.

[39] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[40] Pawel Wawrzynski. Control policy with autocorrelated noise in reinforcement learning for robotics. *International Journal of Machine Learning and Computing*, 5(2):91, 2015.

[41] Gabriel Barth-Maron, Matthew W Hoffman, David Budden, Will Dabney, Dan Horgan, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed distributional deterministic policy gradients. *arXiv preprint arXiv:1804.08617*, 2018.

[42] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

[43] Sungsik Huh and David Hyunchul Shim. A vision-based automatic landing method for fixed-wing uavs. *Journal of Intelligent and Robotic Systems*, 57(1-4):217, 2010.

[44] Alejandro Rodriguez-Ramos, Carlos Sampedro, Hriday Bavle, Ignacio Gil Moreno, and Pascual Campoy. A deep reinforcement learning technique for vision-based autonomous multirotor landing on a moving platform. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1010–1017. IEEE, 2018.

[45] Marwan Shaker, Mark NR Smith, Shigang Yue, and Tom Duckett. Vision-based landing of a simulated unmanned aerial vehicle with fast reinforcement learning. In *2010 International Conference on Emerging Security Technologies*, pages 183–188. IEEE, 2010.

[46] Riccardo Polvara, Massimiliano Patacchiola, Sanjay Sharma, Jian Wan, Andrew Manning, Robert Sutton, and Angelo Cangelosi. Autonomous quadrotor landing using deep reinforcement learning. *arXiv preprint arXiv:1709.03339*, 2017.

[47] Yingcai Bi and Haibin Duan. Implementation of autonomous visual tracking and landing for a low-cost quadrotor. *Optik-International Journal for Light and Electron Optics*, 124(18):3296–3300, 2013.

[48] Bruno Herissé, Tarek Hamel, Robert Mahony, and François-Xavier Russotto. Landing a vtol unmanned aerial vehicle on a moving platform using optical flow. *IEEE Transactions on robotics*, 28(1):77–89, 2011.

[49] So-Ryeok Oh, Kaustubh Pathak, Sunil Kumar Agrawal, Hemanshu Roy Pota, and Matt Garrett. Autonomous helicopter landing on a moving platform using a tether. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 3960–3965. IEEE, 2005.

[50] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[51] Yevgen Chebotar, Ankur Handa, Viktor Makoviychuk, Miles Macklin, Jan Issac, Nathan Ratliff, and Dieter Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8973–8979. IEEE, 2019.

[52] Paul Christiano, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba. Transfer from simulation to real world through learning deep inverse dynamics model. *arXiv preprint arXiv:1610.03518*, 2016.

[53] Niko Sünderhauf, Oliver Brock, Walter Scheirer, Raia Hadsell, Dieter Fox, Jürgen Leitner, Ben Upcroft, Pieter Abbeel, Wolfram Burgard, Michael Milford, et al. The limits and potentials of deep learning for robotics. *The International Journal of Robotics Research*, 37(4-5):405–420, 2018.

[54] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

[55] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from

simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30. IEEE, 2017.

[56] Fereshteh Sadeghi and Sergey Levine. Cad2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*, 2016.

[57] OpenAI, :, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation, 2018.

[58] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. *CoRR*, abs/1710.06537, 2017.

[59] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

[60] Rika Antonova, Silvia Cruciani, Christian Smith, and Danica Kragic. Reinforcement learning for pivoting task. *arXiv preprint arXiv:1703.00472*, 2017.

[61] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2018.

[62] Thomas M. Breuel. The effects of hyperparameters on sgd training of neural networks, 2015.

[63] Erwin Coumans and Yunfei Bai. Bullet real-time physics simulation, 2014-2019.

[64] Murtaza Hazara and Ville Kyrki. Transferring generalizable motor primitives from simulation to real world. *IEEE Robotics and Automation Letters*, 4(2):2172–2179, 2019.

[65] Jeff Ferrin, Robert Leishman, Randy Beard, and Tim McLain. Differential flatness based control of a rotorcraft for aggressive maneuvers. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2688–2693. Ieee, 2011.

[66] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics, 2019.

[67] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic. *ACM Transactions on Graphics*, 37(4):114, Jul 2018.

[68] Reza Mahjourian, Risto Miikkulainen, Nevena Lazic, Sergey Levine, and Navdeep Jaitly. Hierarchical policy design for sample-efficient learning of robot table tennis through self-play. *arXiv preprint arXiv:1811.12927*, 2018.

[69] Pieter Abbeel, Morgan Quigley, and Andrew Y. Ng. Using inaccurate models in reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 1–8, New York, NY, USA, 2006. ACM.

[70] J Zico Kolter and Andrew Y Ng. Policy search via the signed derivative. In *Robotics: science and systems*, page 34, 2009.

[71] Andrew W Smith Jr. Digital computer process control with operational learning procedure, September 26 1972. US Patent 3,694,636.

[72] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[73] Wenhao Yu, Visak CV Kumar, Greg Turk, and C Karen Liu. Sim-to-real transfer for biped locomotion. *arXiv preprint arXiv:1903.01390*, 2019.

[74] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[75] Guofei Pang, Lu Lu, and George Em Karniadakis. fpinns: Fractional physics-informed neural networks. *SIAM Journal on Scientific Computing*, 41(4):A2603–A2626, 2019.

Appendix

```python
#!/bin/python

import subprocess
import itertools
import tensorflow as tf

hyp_search = {
    "sigma": [ 0.3, 0.5, 0.8],
    "batch-size": [32, 64, 128],
    "actor-lr": [1e-5, 1e-4, 1e-3],
    "critic-lr": [1e-2, 1e-3, 1e-4],
    "act-fun": ["tf.nn.relu", "tf.nn.tanh", "tf.nn.leaky_relu"],
    "normalize-returns": [True, False]
}

keys = list(hyp_search.keys())
values = list(hyp_search.values())
gen_param = lambda x,y: "--%s %s" % (x, str(y))
slurm_path = "/work/nimbus/pfrancaalb/moving_platform/hyp_search_slurm.sh"

for params in itertools.product(*values):
    param_list = " ".join(list(map(gen_param, keys, params)))
    com = "sbatch %s %s" % (slurm_path, param_list)
    out, error = subprocess.Popen(com.split(), stdout=subprocess.PIPE).communicate()
    print( "{} {}".format(param_list, out) )
```

Listing 1: Code to launch hyper-parameter search

```python
import numpy as np
from stable_baselines import DDPG
from stable_baselines.ddpg import policies
from stable_baselines.common.vec_env import DummyVecEnv
import gym
import pickle
from sys import argv

sysid_path = "/work/nimbus/pfrancaalb/moving_platform/src"

```

```
11
12 class CustomPolicy_DDPG(policies.FeedForwardPolicy):
13     def __init__(self, *args, **kwargs):
14         super(CustomPolicy_DDPG, self).__init__(*args, **kwargs,
15                                                 layers      = [200, 100],
16                                                 act_fun     = tf.nn.tanh,
17                                                 feature_extraction = "mlp")
18
19
20 def eval_params(friction, wind):
21     e = gym.make("MovingPlatform3d-v{}".format(argv[1]))
22
23     if len(argv) > 3:
24         e.enable_sysid(argv[3])  # Sysid model for training
25
26     env = DummyVecEnv([lambda: e])
27
28     rewards = []
29     ep_reward = 0
30     ep = 0
31     obs = env.reset()
32
33     e.wind_mu = wind
34     e.lateralFriction = friction
35
36     while ep < 100:
37         action, _states = model.predict(obs)
38         obs, reward, done, _ = env.step(action)
39         ep_reward += reward
40         if done:
41             ep += 1
42             rewards.append(ep_reward)
43             ep_reward = 0
44             obs = env.reset()
45
46             e.wind_mu = wind
47             e.lateralFriction = friction
48
49     performance.append([np.mean(rewards), np.std(rewards), e.falls, np.mean(e.
```

```
    times_collision)])


# Load the policy
model = DDPG.load(argv[2], policy=CustomPolicy_DDPG)

# Variations of the paramaters
performance = []
frictions = np.linspace(0, 0.2, num=50)
wind = np.linspace(0, 1, num=50)

for f in frictions:
    eval_params(f, 0)

pickle.dump(performance, open("performance_{}_friction.pickle".format(argv[1]), "wb")
    )
performance = []

for w in wind:
    eval_params(0.2, w)

pickle.dump(performance, open("performance_{}_wind.pickle".format(argv[1]), "wb"))
```

Listing 2: Simulation experiments with variation of both wind as friction coefficient

```
#!/bin/bash
for s in 0 1; do
  for j in {1..50}; do
    for t in {1..7}; do
        batchthis python3.5 -u test_rewards_3d.py --environment MovingPlatform3d-v2
    --rw-function ${t} --batch-size 128 --timesteps 3.5e6 --selected-param ${s}
    done
  done | egrep -o "[0-9]+" | tr "\n" " "  > ../data/runs/‘date ’+%Y_%m_%d__%H_%M_%S’‘
done
```

Listing 3: Script to launch jobs evaluating reward different reward functions

```
#!/bin/bash
envs=(2, 4)
args=("$@")
```

```
param=1
# Navigate through the text files
for (( i=0; i<=$(( ${#args[*]} -1 )); i++ )); do
  # For each job in the file
  for j in $(cat ${args[$i]}); do
    # If it is the first file, use the first env element of the array, so on...
    batchthis python3.5 -u evaluate_net.py /work/nimbus/pfrancaalb/output/${j}/
    network.pkl ${envs[$i]} ${param}
  done | egrep -o "[0-9]+" | tr "\n" " " > ${args[$i]}_evaluation
done
```

Listing 4: Script to start the evaluation of a sequence of jobs

```
    def sample_params(self):
        if self.domain_rand and self.episodes % self.domain_rand == 0:
            self.restitution = 0.5
            self.lateralFriction = np.random.uniform(0.0, 0.2)
            self.wind_mu = np.random.uniform(-0.10, 0.10)
            #self.wind_direct = np.random.dirichlet(np.ones(3),size=1)[0]  # Random
    direction
            self.wind_direct = [1, 0, 0]

            self.estimation = 0.01  # Be careful in the first episode

            # Fix parameter according to the experiments
            if self.selected_param == 0:
                self.laterlFrition = 0.35
            else:
                self.wind_mu = 0

        elif not self.domain_rand:
            self.restitution = 0.5
            self.lateralFriction = 0.35
            self.wind_mu = 0
            self.wind_direct = [1,0,0]
```

Listing 5: Code snippet from the environment sampling the dynamic parameters

```
    def control_drone(self, action):
```

```
        target_vel = np.array( action ) *  self.max_speed   # Get the velocity based on the normalized velocity
        current_vel = self.lin_vel
        vel = [0,0,0]
        max_vel_dot = self.max_acc * self.skip_iterations * self.Tsample_physics
        for i in range(3):
            vel[i] = min( max( target_vel[i], current_vel[i] - max_vel_dot), current_vel[i] + max_vel_dot)
            vel[i] += np.random.normal(self.wind_mu, 0.1) * self.wind_direct[i]

        ref_pos = np.array(self.pos_meas) + np.array(vel) * norm_value * self.Tsample_physics  # Next position based on the reference velocity
        p.changeConstraint(self.quad_constraint_id, ref_pos, 0, maxForce=50000)  # Force the object to go to the calculated position
```

Listing 6: Code snippet from the environment that controls the UAV

```
def dense_sysID(hidden_size, hidden_layers = 2, dropout = 0, act_fun = "tanh", bn = 0):
    '''
    Architecture of the sysid network
    '''
    model = Sequential()
    model.add(Dense(hidden_size, input_dim=(240) ))
    model.add(BatchNormalization())

    # Adds the hidden layers
    for _ in range(hidden_layers -1):
        model.add( Dense(hidden_size) )
        if bn: model.add(BatchNormalization())
        model.add( Activation(act_fun) )
    if dropout: model.add(Dropout(0.5))

    # Output layer
    model.add(Dense(1, activation='tanh'))

    model.compile(loss=custom_loss,
                  optimizer='adam')

    return model
```

```
23
24
25 def save_model(model):
26     '''
27     Saves the weights of the network
28     '''
29     open("./model.json", "w").write(model.to_json())  # Save model architecture
30     model.save_weights("./model.h5")  # Savel model or
31
32
33 def load_memories():
34     x_train = []
35     y_train = []
36     for i in argv[1:]:
37         mem = pickle.load(open(i, "rb"))
38         for x, y in mem:
39             x_train.append(x[0])
40             y_train.append(y[0])
41
42     return np.array(x_train), np.array(y_train)
43
44 x_train, y_train = load_memories()
45
46 tb = TensorBoard(log_dir='./logs', histogram_freq=1, batch_size=64, write_grads=True,
       write_images=True, update_freq='batch')
47
48 model = dense_sysID(128, hidden_layers = 2, act_fun="relu")
49 model.fit(x=x_train, y=y_train, batch_size=64, epochs=3, verbose=1, callbacks=[tb],
      validation_split=0.1)
50 save_model(model)
```

Listing 7: SysID training using supervised learning with previously stored experience buffer.

```
1
2 import gym
3 import argparse
4 from stable_baselines.ddpg.policies import FeedForwardPolicy
5 from stable_baselines.common.vec_env import DummyVecEnv
```

```
6  from stable_baselines import DDPG
7  from stable_baselines.ddpg.noise import ActionNoise
8  import tensorflow as tf
9  from pprint import pprint
10 import os
11 from stable_baselines.ddpg.noise import OrnsteinUhlenbeckActionNoise,
       NormalActionNoise, AdaptiveParamNoiseSpec
12 import keras
13 import numpy as np
14 import pdb
15 from inspect import getsource
16
17
18 class DecayingGaussian(ActionNoise):
19     """
20     A gaussian action noise
21
22     :param mean: (float) the mean value of the noise
23     :param sigma: (float) the scale of the noise (std here)
24     """
25     def __init__(self, mean, sigma, magnitude, epsilon=0.9999):
26         self._mu = mean
27         self._sigma = sigma
28         self._epsilon = epsilon
29         self._magnitude = magnitude
30
31     def __call__(self):
32         self._epsilon *= self._epsilon
33         return self._epsilon * self._magnitude * np.random.normal(self._mu, self.
       _sigma)
34
35     def __repr__(self):
36         return 'NormalActionNoise(mu={}, sigma={}, magnitude={}, epsilon={})'.format(
       self._mu, self._sigma, self._magnitude, self._epsilon)
37
38
39 def getCustomPolicy(act_fun, layers=[200, 100]):
40     class CustomPolicy(FeedForwardPolicy):
41         def __init__(self, *args, **kwargs):
```

```
42            super(CustomPolicy, self).__init__(*args, **kwargs,
43                                               layers= layers,
44                                               act_fun= act_fun,
45                                               feature_extraction="mlp")
46
47    return CustomPolicy
48
49
50
51 # REWARD FUNCTIONS
52
53 def reward1( ref_pos, ref_vel, vel, c, action, touched_floor, t):
54    return ( ( - np.linalg.norm(action - e.prev_action) * 2  + 0.15  ) +
55            0.80 * ( - (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)) / 15  +
56            4.0 * c )
57
58 def reward2( ref_pos, ref_vel, vel, c, action , touched_floor, t):
59    return ( ( - np.linalg.norm(action - e.prev_action) * 2  + 0.15  ) +
60            0.80 * ( - (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)) / 15  +
61            5.0 * c )
62
63 def reward3( ref_pos, ref_vel, vel, c, action, touched_floor, t):
64    return ( ( - np.linalg.norm(action - e.prev_action) * 2  + 0.15  ) +
65            0.80 * ( - 2 * (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)) / 15  +
66            6.0 * c )
67
68 def reward4( ref_pos, ref_vel, vel, c, action, touched_floor, t):
69    return ( ( - np.linalg.norm(action - e.prev_action) * 2  + 0.15  ) +
70            0.80 * ( - 2 * (ref_pos[0]**2 + ref_pos[1]**2 +ref_pos[2]**2)) / 15  +
71            7.0 * c )
72
73 def reward5( ref_pos, ref_vel, vel, c, action, touched_floor, t):
74    return ( ( - np.linalg.norm(action - e.prev_action) * 2  + 0.15  ) +
75            0.80* (15 - (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)) / 15  +
76            10.0 * c )
77
78
79 # ================================
80 # Squaring the derivative from the actions
```

```
# ================================


def reward6( ref_pos, ref_vel, vel, c, action , touched_floor, t):
    return ( ( - np.linalg.norm(action * 10 - e.prev_action * 10 ) ** 2  / 300  ) +
            0.80 * ( - (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)) / 15  +
            5.0 * c -
            25 * touched_floor)
#
def reward7( ref_pos, ref_vel, vel, c, action , touched_floor, t):
    return ( ( - np.linalg.norm(action * 10 - e.prev_action * 10 ) ** 2  / 400  ) +
            0.80 * ( - (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)) / 15  +
            5.0 * c -
            25 * touched_floor)

def reward8( ref_pos, ref_vel, vel, c, action , touched_floor, t):
    return ( ( - np.linalg.norm(action * 10 - e.prev_action * 10 ) ** 2  / 500  ) +
            0.80 * ( - (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)) / 15  +
            5.0 * c -
            25 * touched_floor)

def reward9( ref_pos, ref_vel, vel, c, action , touched_floor, t):
    return ( ( - np.linalg.norm(action * 10 - e.prev_action * 10 ) ** 2  / 600 ) +
            0.80 * ( - (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)) / 15  +
            5.0 * c -
            25 * touched_floor)

# ================================
# Including time in the state space
# ================================

# Time decreasing the positive reward of collision

def reward10( ref_pos, ref_vel, vel, c, action , touched_floor, t):
    return ( ( - np.linalg.norm(action * 10 - e.prev_action * 10 ) ** 2  / 500  ) +
            0.80 * ( - (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)) / 15  +
            10.0 * c - ( - t * 32 * c) -
            25 * touched_floor)

```

```
120 def reward11( ref_pos, ref_vel, vel, c, action , touched_floor, t):
121     return ( ( - np.linalg.norm(action * 10 - e.prev_action * 10 ) ** 2  / 500  ) +
122             0.80 * ( - (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)) / 15  +
123             10.0 * c - ( - t * 16 * c)  -
124             25 * touched_floor)
125
126 # Time decreasing the reward from distance
127
128 def reward12( ref_pos, ref_vel, vel, c, action , touched_floor, t):
129     distance = (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)
130     d_action = np.linalg.norm(action * 10 - e.prev_action * 10 ) ** 2
131
132     return ( ( - d_action / 1000  ) +
133             0.80 * ( -distance ) / 15  +
134             5.0 * c -
135             distance * t * 1/200 -
136             25 * touched_floor)
137
138 def reward13( ref_pos, ref_vel, vel, c, action , touched_floor, t):
139     distance = (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)
140     d_action = np.linalg.norm(action * 10 - e.prev_action * 10 ) ** 2
141
142     return ( ( - d_action / 1000  ) +
143             0.80 * ( -distance ) / 15  +
144             5.0 * c -
145             distance * t * 1/160 -
146             25 * touched_floor)
147
148 def reward14( ref_pos, ref_vel, vel, c, action , touched_floor, t):
149     distance = (ref_pos[0]**2 + ref_pos[1]**2 + ref_pos[2]**2)
150     d_action = np.linalg.norm(action * 10 - e.prev_action * 10 ) ** 2
151
152     return ( ( - d_action / 1000  ) +
153             0.80 * ( -distance ) / 15  +
154             5.0 * c -
155             distance * t * 1/50 -
156             25 * touched_floor)
157
158
```

```
159 def reward15( ref_pos, ref_vel, vel, c, action, touched_floor, t):
160     return ( ( - np.linalg.norm(action - e.prev_action) * 2  + 0.15  ) -
161             10**(max(np.linalg.norm(ref_vel) - 0.5, 0)) - 1 +
162             0.80 * ( - 2 * (ref_pos[0]**2 + ref_pos[1]**2 +ref_pos[2]**2)) / 15  +
163             7.0 * c )
164
165 # agent parameters
166 parser = argparse.ArgumentParser(description='provide arguments for DDPG agent')
167 parser.add_argument('--actor-lr', default=1e-4, type=float)
168 parser.add_argument('--critic-lr', default=1e-3, type=float)
169 parser.add_argument('--sigma', default=0.3, type=float)
170 parser.add_argument('--batch-size', default=128, type=int)
171 parser.add_argument('--act-fun', default="tf.nn.tanh", type=str)
172 parser.add_argument('--normalize-returns', default=False, type=bool)
173 parser.add_argument('--environment', default="MovingPlatform3d-v4", type=str)
174 parser.add_argument('--sysid', default="", type=str)
175 parser.add_argument('--rw-function', default=4, type=int)
176 parser.add_argument('--selected-param', default=0, type=int)
177 parser.add_argument('--timesteps', default=3e6, type=float)
178 parser.add_argument('--gamma', default=0.999, type=float)
179 parser.add_argument('--epsilon', default=0.9999, type=float)
180 parser.add_argument('--exploration', default='gau', type=str)
181
182
183 args = vars(parser.parse_args())
184 pprint(args)
185
186 # Save in the job id folder if running on crane
187 if os.environ.get("USER") == 'pfrancaalb':
188     id = os.environ.get("SLURM_JOB_ID")
189     save_path = "/work/nimbus/pfrancaalb/output/{}/".format(id)
190 else:
191     save_path = "./"
192
193 tim = int(args.pop("timesteps"))
194
195 rw = args.pop("rw_function")
196
197 # Change environment if timestep is necessary
```

```
198 if rw < 10:
199     print ("Including timestep on state space")
200     env = "MovingPlatform-noy-v1"
201 else:
202     print ("Including velocity on state space")
203     env = "MovingPlatform-noy-v2"
204 
205 env = args.pop("environment")
206 
207 # Environment based on the reward function
208 e = gym.make(env)
209 e.calc_reward = eval("reward{}".format(rw))
210 print(getsource(e.calc_reward))
211 
212 # In case there is system identificatoin, import it
213 sysid = args.pop("sysid")
214 e.selected_param = args.pop("selected_param")
215 if sysid:
216     e.enable_sysid(sysid)
217 
218 env = DummyVecEnv([lambda: e])
219 
220 sigma = args.pop("sigma")
221 
222 # Noise for exploration
223 n_actions = env.action_space.shape[-1]
224 noise_dict = {  'ou': OrnsteinUhlenbeckActionNoise(mean=np.zeros(n_actions), sigma=
    sigma * np.ones(n_actions)),
225                 'gau': NormalActionNoise(mean=np.zeros(n_actions), sigma=sigma * np.
    ones(n_actions)),
226                 'adap': AdaptiveParamNoiseSpec(),
227                 'decg': DecayingGaussian(mean=np.zeros(n_actions), sigma = sigma * np
    .ones(n_actions), magnitude = 2, epsilon = args.pop("epsilon") )
228             }
229 #action_noise = OrnsteinUhlenbeckActionNoise(mean=np.zeros(n_actions), sigma=args.pop
    ("sigma") * np.ones(n_actions))
230 noise = args.pop("exploration")
231 
232 # Execute the learning
```

```
233 CustomPolicy = getCustomPolicy(eval(args.pop("act_fun")))
234
235 if noise == 'adap':
236     model = DDPG('MlpPolicy', env, verbose=1, param_noise=noise_dict[noise],
    tensorboard_log=save_path, **args)
237 else:
238     model = DDPG(CustomPolicy, env, verbose=1, action_noise=noise_dict[noise],
    tensorboard_log=save_path, **args)
239
240 model.learn(total_timesteps=tim)
241 model.save(save_path + "network")
```

Listing 8: Training deep deterministic policy gradient with the option of selecting a variety of reward functions.