University of Genoa
Department of Computer Science, Bioengineering,
Robotics and System Engineering

Master's Degree in Robotics Engineering

Safe Learning for Robotics: Abstraction Techniques for Efficient Verification

Candidate
Andrea Pitto

Advisor
Prof. Armando Tacchella

Co-advisor
Prof. Dario Guidotti

March, 30th 2022

The best way to predict the future is to invent it.

Alan Kay (1971)

Abstract

Day after day, Neural Networks are becoming more popular in many different fields due to their incredible results and ductility. Among the others, they are also superseding previous state-of-the-art approaches in Robotics where, for example, the capability of a robot to perceive the world around itself is more and more based on Machine Learning rather than canonical edge-matching algorithms. One of the problems related to Neural Networks, though, is bound to their "black box"/opaque nature, in the sense that they do not formally provide any guarantee on their results. This is a crucial problem, especially in environments in which robots need to work alongside humans, that is still holding back the application of Neural Networks in even more cases.

To solve this, Verification techniques have been proposed, whose duty is to check whether the Nets are safe to use in any given scenario. As another issue related to Neural Nets is their sheer number of parameters, we focus on Verification solutions that feature Abstraction algorithms in order to reduce computational load and thus making Verification applicable in real-world scenarios. Indeed, we propose a case study that features sound Networks applied to the Robotics field, along with formal guarantees provided by our Verification engine. In addition to that, we also propose some improvements on the novel tool NeVer2, which aims at integrating Neural Network learning, editing, verification (among other capabilities) inside an easy-to-use GUI, hopefully featuring a first step towards making Deep Learning accessible even to non-experts of the field.

Acknowledgements

I would like to thank my family for supporting me during these academic years. I could not have made it this far alone. A special thanks goes to my grandfather Federico Flavio, who has been the greatest source of inspiration during my study days. I want to share all my gratitude with Veronica for all she has done, especially during greyer times. She has always been here for me, and I will never be able to thank her enough for that.

I wish to also thank Stefano Demarchi, who helped solving some of my doubts during the development of this Thesis. Many thanks also to my advisor Armando Tacchella, who of course partook in the process of supporting me while progressing on this work. I also wish to thank him for his precise and interesting lessons on *Artificial Intelligence*, which proved to be one of the most interesting courses of my entire study career, despite the remote classes restrictions imposed due to the COVID-19 pandemic. Thanks also to Dario Guidotti. Special thanks to Jes Grydholdt Jepsen and Carmine Tommaso Recchiuto. Carmine has been the kindest and most helpful professor I have ever encountered.

Contents

1 Introduction **1**

1.1 Context 1
1.2 Motivations 2
1.3 Objectives 2
1.4 Contributions 3
1.5 Structure of the Thesis 3

2 Background and Related Work **5**

2.1 Neural Networks 5
2.2 Deep Neural Networks (Learning) 11
2.3 Pruning of Neural Networks 12
2.4 Verification of Neural Networks 13
2.5 Abstraction/Refinement for Verification 14
2.6 pyNeVer 17
2.7 Behavior Trees 19
2.8 ROS and Gazebo 19
2.9 Reinforcement Learning 21

3 Improvements on NeVer2 **27**

3.1 Introduction 27
3.2 NeVer2 27
3.3 Adaptive Cruise Control 29
3.3.1 Context 29
3.3.2 Data Set 32
3.3.3 Neural Networks 33
3.3.4 Verification Properties 34
3.3.5 Results 37

4 Case Study **41**

4.1 Introduction 41
4.2 Turtlex's Behavior Tree 42

4.3 Scripted Skills 44
4.4 Learned Skills 46
4.5 Learning Techniques 47

5 Learning **51**
5.1 Introduction 51
5.2 Navigation 53
5.3 Manipulation 58
5.4 Test 61
5.5 Integration 63

6 Verification **65**
6.1 Introduction 65
6.2 Navigation Verification 65

7 Conclusions and Future Work **71**
7.1 Conclusions 71
7.2 Future Work 72

List of Figures

2.1 Schematic structure of the formal neuron (Perceptron) 6
2.2 A single-hidden-layer example of a multi-layer Neural Network 9
2.3 Venn diagram representing the relationship among Artificial Intelligence, Machine Learning, Neural Networks and Deep Learning 11
2.4 Schematic depiction of a generic Reinforcement Learning process 22

3.1 The GUI of NeVer2 showing a sample Neural Network layout 28
3.2 Box plot for a million samples of the Adaptive Cruise Control data set ($TIV = 1,5; D_0 = 5$) 33
3.3 NeVer2 representation of NET0's architecture ($TIV = 1; D_0 = 2.5$ data set) 34
3.4 NeVer2 representation of NET1's architecture ($TIV = 1; D_0 = 2.5$ data set) 35
3.5 NeVer2 representation of NET2's architecture ($TIV = 1; D_0 = 2.5$ data set) 36
3.6 Creation process of the OutBounds property with NeVer2. Detached property blocks are treated as input preconditions, while property blocks linked to the Neural Network are the output areas to Verify 37

4.1 The Turtlex robot 42
4.2 The domestic environment designed for our Turtlex case study. Some obstacles and colored spheres (junk elements) are visible, along with the grey dropbox 43
4.3 Behavior Tree for the Turtlex case study 43

5.1 Learning episode rewards and lengths for the navigation task, with sparse rewards and incremental goals 57

5.2 Learning episode rewards and lengths for the navigation task, with dense rewards, incremental goals, and a routine to prevent prolonged stillness 57
5.3 Learning episode rewards and lengths for the navigation task, with dense rewards and incremental goals for our final training session 58
5.4 Testing episode rewards and lengths for the navigation task, with sparse rewards and 100 random goals 62

List of Tables

3.1 Results for the ACC data set with TIV set to 1.5 seconds, D_0 to 5.0 meters and with 0 meters of tolerance (ϵ). Elapsed times [s] have been rounded to the third decimal place 39
3.2 Results for the ACC data set with TIV set to 1.5 seconds, D_0 to 5.0 meters and with 20 meters of tolerance (ϵ). Elapsed times [s] have been rounded to the third decimal place 40

6.1 Verification results for the TURTLEX navigation task. Dashes indicate missing entries. Elapsed times [s] have been rounded to the third decimal place 70

List of Algorithms

1 Pseudocode for single-epoch Network learning 8
2 Stellantis algorithm for the Adaptive Cruise Control 31
3 Turtlex dense reward scheme for the navigation learned skill 55
4 Turtlex dense reward scheme for the manipulation learned skill 60

Chapter 1

Introduction

This is the introductory chapter, which outlines the Thesis' Context (section 1.1), Motivations (1.2), Goals (1.3) and Contributions (1.4). A brief overview of the work is then reported in section 1.5.

1.1 Context

In the recent years, Artificial Intelligence has been consistently one of the greatest fields in terms of growth pace and new areas of application. This is mostly thanks to the constant increase of computation capabilities of modern computers, which allowed the diffusion of Machine Learning tools, such as Neural Networks, in many scenarios. Neural Networks themselves are the backbone of Deep Learning algorithms, which have proven to yield unprecedented results in many fields, often superseding state-of-the-art approaches. We are confident that Deep Learning may represent, for the next future, what electronics has been in the last few decades.

Albeit being extremely promising also for the near future, Deep Learning techniques still feature some weak spots. In this work, we try to tackle one of these, namely their lack of formal guarantees over their outputs. Indeed, even if the results of Deep Learning algorithms are usually unmatched, to date they do not provide any certainty over their behavior: this is why high-risk environments are still a taboo for Neural Networks. One of the most relevant examples of such environments is certainly Robotics, where Artificial Intelligence often comes in contact with humans. That said, Deep Learning algorithms should be deemed as safe even for robots themselves, i.e., regardless of human proximity, both for economic and safety reasons in general. We may say that most - if not all - Deep Learning applications involving moving components should provide formal safety guarantees.

A possible solution to this problem is given by Neural Network Verification [24], a technique that allows to test whether said networks fulfill certain properties. These properties consist in some preconditions over the Network's inputs and some safe regions for the outputs: if, given these preconditions, the outputs never break outside of the safe areas, the Neural Network is considered safe for that property. The biggest challenge of Verification is provided by the sheer amount of computation required to perform it, especially for larger Networks. Thus, there have been various approaches aimed at overcoming this problem, like Network pruning [11, 16], Network slimming [27], and abstraction techniques [32, 38, 13, 36, 30, 12].

1.2 Motivations

Given the context presented in section 1.1, we feel confident about the relevance of Deep Learning in the scientific world, and we also think that Robotics could immensely benefit from an improvement over current Network Verification techniques. Modern systems are still incapable of dealing with complete Verification in reasonable times, especially when considering the amount of neurons involved in real-world Neural Networks.

Thus, at the moment there is no way that Network safety can be confirmed/rejected by checking every possible input: for this reason, finding and improving efficient ways of performing Verification is pivotal for the spreading of Deep Learning algorithms in several fields, out of which Robotics is surely one of the most relevant. We want to prove the usefulness of our Abstraction techniques by Verifying Neural Networks applied to Robots, and by delivering a novel tool which can be easily operated by most of the scientific community, without them having to learn how to code the Nets beforehand.

1.3 Objectives

This work has the purpose of continuing the development of the NeVer2 tool, whose original version (NeVer) had been portrayed in [32, 33]. NeVer2 has been introduced to the scientific community by [17, 18].

Also, we want to provide practical applications of the NeVer2 tool in Robotics-related contexts, while creating an interface between the Deep Learning capabilities of NeVer2 and robot simulation. While pursuing this, we also aim to produce sound and possibly safe Neural Networks able

to handle typical tasks of the Robotics field, such as self-aware navigation and ambient manipulation.

1.4 Contributions

We provide the scientific community with a novel tool for Deep Learning, NeVer2. NeVer2 features creation, loading/saving, editing, conversion, training, and verification capabilities (soon, also pruning and repair), all accessible via an easy-to-use *Graphical User Interface* (*GUI*) which, to the best of our knowledge, represents an unprecedented piece of software at the time of writing. NeVer2 will allow users agnostic to the details of Deep Learning to be able to easily tinker with Neural Networks, facilitating the spread of Machine Learning solutions even more. This will hopefully result in the birth of new state-of-the-art paradigms in many research fields.

Leveraging NeVer2 features, we further provide trained and verified Neural Networks that we believe are both an additional proof of the tool's usefulness, and valid resources on their own, due to their formal guarantees and their performance. The code of this Work is publicly available[1]. The NeVer Tools organization[2] is free to access as well and, among other elements, also encompasses NeVer2, CoCoNet, and PyNeVer. As it will be detailed in the next Chapters, the recent development of these 3 tools has been supported by the work of this Thesis.

1.5 Structure of the Thesis

This section constitutes an overview of the Thesis' structure, and presents a brief summary of the all the next chapters. Chapter 2 provides a detailed introduction to all the necessary elements required to justify and to understand this work, along with the description of various state-of-the-art solutions present in literature. The following chapter (Chapter 3) describes NeVer2, an unique tool that allows even non-experts of the field to be able to easily tinker with Neural Networks. This Chapter also illustrates an educational case study we carried out, which introduces to the tool usage, while proving some of its capabilities in the meantime. Chapter 4 is dedicated to the presentation of the main case study we worked on, which involves the simulation of a Neural Network-driven robot inside a domestic

[1] https://github.com/andreabradpitto/turtlex
[2] https://github.com/NeVerTools

environment. Chapter 5 focuses over the detailed description of learning techniques we leveraged in our case study, and reports the results of our testing sessions. Then the next one, Chapter 6, does the same for the Network Verification step. Lastly, Chapter 7 draws the conclusions and proposes some future work, aimed at improving or enriching what we have developed.

Chapter 2

Background and Related Work

Chapter 2 is dedicated to the description of all the necessary concepts that will let the reader adequately understand the work of this Thesis. We will provide detailed information over the topics of interest, and mention the solutions available in literature as of the time of writing. Section 2.1 provides an introduction to Neural Networks, whose knowledge is necessary to acquire a comprehensive picture of Deep Learning approaches, which in turn are presented in section 2.2. After that, section 2.3 introduces Network pruning and slimming, two essential techniques that make Network Verification more accessible. Next, section 2.4 depicts the background of one the core subjects of this Thesis: Network Verification. Section 2.5 will be dedicated to Network Abstraction/refinement techniques, which provide and allow to balance efficiency and precision to Verification attempts. Section 2.6 presents pyNeVer, i.e. the Python back-end of the NeVer2 tool. Section 2.7 introduces the concept of behavior trees, which have been leveraged by our main case study. Next, 2.8 provides an introduction to ROS and Gazebo, i.e., the framework and the physics simulator that were used for the case study development and testing. Lastly, in section 2.9 we are going to describe the Reinforcement Learning Machine Learning technique, to which we resorted in order to train our robot.

2.1 Neural Networks

One of the main focuses of the Thesis is the application of Verification solutions for Neural Networks. In order to adequately understand Network Verification, it is important to first be aware of the essential notions about Neural Networks themselves.

Let us start with defining what “Neural Network” actually stands for.

A *Neural Network* (also called Artificial Neural Network, or Neural Net), is a computing model with a layered structure that resembles the one of neurons in human brain, with layers of connected nodes. A Neural Network can learn from data, it can be trained to recognize patterns, classify samples, or forecast future events. Nowadays, Neural Nets are among the best solutions in the state-of-the-art of many fields, such as aerospace [37], automotive [4], banking [31], electronics [1], natural language processing [21], and many more.

Artificial Neural Networks, or *ANN*s for short, are meshes of several interconnected units, each featuring a simple behavior capable of carrying out input-to-output transformations. Their knowledge is summarized in parameters called weights. They own features which are different (actually, mostly opposite) from the ones typically attributed to computers: ANNs are heavily parallel, built of simple and strongly connected processors or units, not necessarily fast, asynchronous, and they usually handle analog data. These characteristics allow them to excel in various tasks, such as association of complex patterns, recognition, classification, regression, storage and recall of temporal sequences and complex/uncertain/inconsistent/incomplete data, concept formation (or clustering), and so on. They are part of the *Artificial Intelligence* (*AI*) field, as they simulate the result of brain processing, and they are called this way since their myriad of simple units mimes the inner mechanics of human neurons. The formal (i.e., artificial) neuron we consider (namely, the *Perceptron* [35]) features the structure described in Figure 2.1:

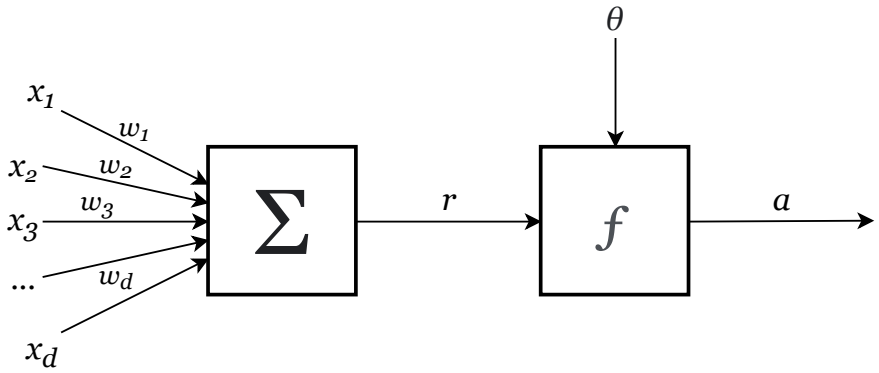

Figure 2.1: Schematic structure of the formal neuron (Perceptron)

Perceptron-based Nets are a special class of ANNs. Furthermore, the Perceptron is the foundation which also the NeVer project is based [32, 33] upon, and allows us to introduce most of the elements belonging to Neural Network neurons in a formal way. On the left side of Figure 2.1 are the input data components x_i, that are a total of d (i.e., $\bar{x}$ is a d-dimensional

vector of input values). All these components are matched with w_d synaptic weights, which alter the "relevance" of each of said inputs based on the Neural Net current beliefs. All these weighted elements are then collected together, and this sum becomes the Net input on the neuron's "membrane" $r = \mathbf{x} \cdot \mathbf{w}$ that gets passed to f, which is called *activation function*. It is a non-linear function (in general, activation functions are used to convert linear input signals into non-linear output ones), whose shape is chosen from a pool of single or multi-neuron activation policies. The most commonly used single-neuron activation functions are: *Heaviside step*, *Signum function*, *Sigmoid* (or *Logistic*), *Hyperbolic tangent*, *Ramp*, *Rectified Linear Unit* or *Rectifier* (usually abbreviated with *ReLU*), *Softplus*. For the multi-neuron case, *Max pooling* or *Softmax* are commonly applied. For example, the ReLU activation function is defined as such:

$$f(r) = ReLU(r) = max(0, r) = \begin{cases} 0 & r \leq 0 \\ r & r > 0 \end{cases} \tag{2.1}$$

which is defined on $(-\infty, +\infty) \rightarrow \{0, +\infty\}$. Figure 2.1 also features θ, which is the *threshold* (or *bias*): it is an additional weight, that gets applied to f itself alongside r.

Finally, a is the *activation value*: it indicates value of the membrane; that is, the output of the neuron:

$$a = f(r - \theta) \tag{2.2}$$

In biological neurons, learning occurs by the slow modification of synaptic strengths; similarly, learning in formal neurons occurs by slow weight variations, input sample after input sample. It is thus clear that iteration is a key aspect of learning, in which each step implies weight adjustments; as this is similar to the way babies learn, we can say that the ANN is able to acquire experience. The general learning formula is:

$$\mathbf{w}_{\tau+1} = \mathbf{w}_\tau + \boldsymbol{\Delta}\mathbf{w}_\tau \tag{2.3}$$

Equation 2.3, in which $\mathbf{w}_\tau$ stands for the weight vector at iteration τ, shows the intrinsic simplicity of the evolution mechanism of Neural Networks.

Now we will present an outline for the learning algorithm. But first, let us introduce a brief description of the necessary prerequisites:

- A training set of patterns x_l with targets t_l

- A real, positive value assigned to the step size η (also known as *Learning rate*). This parameter is a sort of "gain", which determines how fast you want the weights to adapt: setting it too high can lead to missing the optimal solution, while setting it too low slows the computation, but in return assures better results
- A policy to assign starting values to the elements of the weight vector w (e.g., random values in $[-1; 1]$). In more complex architectures, this also applies to the bias θ

The pseudocode for a single epoch of learning is described in Algorithm 1.

Algorithm 1 Pseudocode for single-epoch Network learning

procedure NETLEARNING(dataset, η, Net)
  load dataset
  compute $samples\_number$
  set η
  assign weights to the Net
  $l = 1$
  while $l \leq samples\_number$ **do**
    read $\mathbf{x_l}$ ▷ read l-th sample
    $r_l = \mathbf{w} \cdot \mathbf{x_l}$, $a_l = f(r_l)$ ▷ compute output
    $\delta_l = (t_l - a_l)$ ▷ compute output error
    $\Delta \mathbf{w_l} = \eta \delta_l \mathbf{x_l}$ ▷ compute correction
    $\mathbf{w_{l+1}} = \mathbf{w_l} + \Delta \mathbf{w_l}$ ▷ weights update
    $l = l + 1$
  end while
end procedure

If we put together more layers of neurons, we obtain more complex and powerful Neural Networks. The increase in terms of complexity with respect to single-layer Neural Networks pays off with much better performances, and also grants the ability of solving non-linearly separable problems. Indeed, today's Neural Networks are often composed by many layers, which can be of different kind. During this thesis, we only consider Perceptron-based *Feed-Forward Neural Networks* (*FFNN*) [43]: this class of Nets does not feature cycles, i.e., the data flow strictly follows a mono-directional path. In FFNNs, the first layer is called the *input* layer, and the last one is the *output layer*, while all the ones in between are called *hidden* layers. As already mentioned, there are several kinds of layers: *Affine*

(or *Fully Connected*/*Dense*/*Linear*) layer, ReLU, Sigmoid Logistic, *Batch Normalization* are among the most common, but a more complete list can be found in [6]. The VNNLIB standard, proposed in [6], is aimed at providing a clear reference for Neural Network development, with the utilmate objective of standardizing Network Verification. The standard is built on the Open Neural Network Exchange (ONNX) format for model description, and on the *Satisfiabily Modulo Theories Library* (*SMTLIB*) format for property specification.

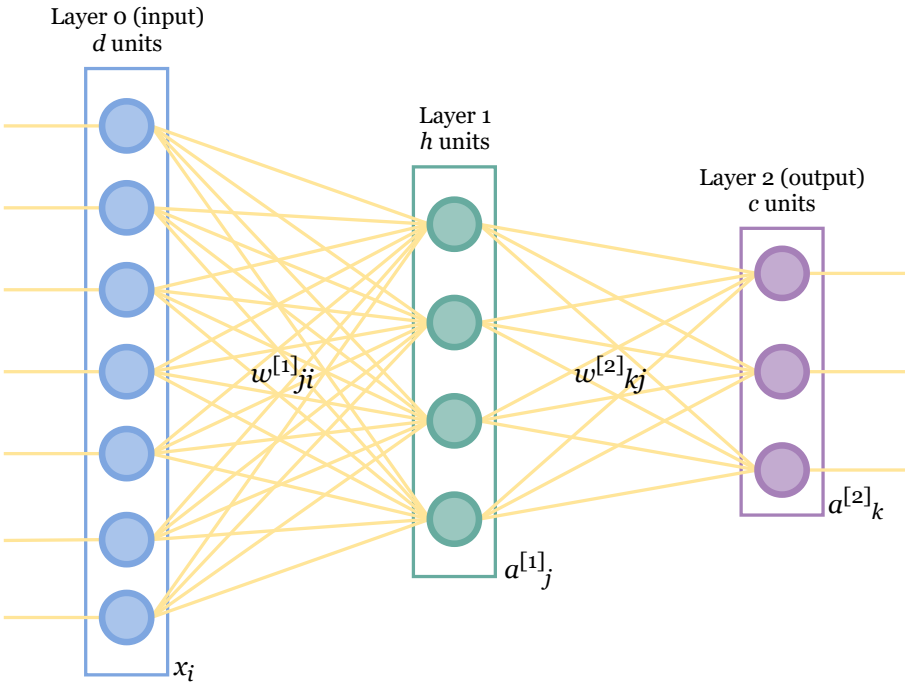

Figure 2.2: A single-hidden-layer example of a multi-layer Neural Network

Figure 2.2 shows an example of a multi-layer Net; here d stands for the number of input units, h for the number of hidden units, and c is the number of output units. The total number of weights n_w is $n_w = (d+1)h + (h+1)c$. The i-th component of input pattern x is denoted as x_i, while $a_j^{[1]}$ is the j-th hidden unit activation value. Next, $a_k^{[2]}$ represents the k-th component of output. Finally, $w_{ji}^{[1]}$ and $w_{kj}^{[2]}$ denote the weight matrices to be optimized by learning. $w_{ji}^{[1]}$ is the weight associated to the j-th hidden unit from the i-th input unit, and $w_{kj}^{[2]}$ is the weight of the k-th output unit from the j-th hidden unit.

In our context, we will restrict to *Multi-Layer Perceptrons* (or *MLP*s), which are a sub-class of multi-layered Networks. These are characterized featuring Affine layers coupled with non-linear activation functions. In general, multi-layered Networks with as few as a single hidden layer (like the one in Figure 2.2), settle the basis for Deep Learning, and represent

the simplest possible connectivity that holds the *universal approximation property*. This property states that, under some assumptions on the activation function, "*A feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of* $\mathbb{R}^d$" [9]. The universal approximation property guarantees that the Network can learn any mapping, even non-linearly-separable ones. This works because each layer can be seen as actively *learning features*, i.e., the Net learns an *internal representation of the data* to be fed to the following layer. Neural Network training may translate into a *Supervised Learning* problem, an *Unsupervised Learning* problem, or *Reinforcement Learning* task. Furthermore, there are several policies for multi-layer Nets to adjust weights (that is, to learn) after each processed input, based on the degree of error made by the Net on the previous input. In general, the connection weights get adjusted in order to compensate the previous error. In MLPs, this process happens via *backpropagation* algorithms [20]; the idea is to generalize the *mean square error* algorithm for the *cost function* exploited by single-layer Perceptrons:

$$E = E_{MS} = \frac{1}{n}\sum_{l=1}^{n}(t - \mathbf{w}\cdot\mathbf{x})^2 \tag{2.4}$$

where t is the target value, $\mathbf{w}\cdot\mathbf{x}$ is the prediction of the Perceptron, and n is the number of outgoing connections. As already hinted, E_{MS} represents the degree of error made by the Net. The more the Network prediction is far from the target value, the more the weights get adjusted. The backpropagation algorithm performs *gradient descent*, and applies the above illustrated weight correction with cascade effect, starting from the output layer up to the input one. With these algorithms, FFNNs/MLPs of any size can effectively learn from input data. Training can happen online or via *batches*. In the first case, weights and bias values are adjusted for every training item based on the difference between computed outputs and the training data target outputs. In the case of batch training instead, the adjustment delta values are accumulated over all training items of the batch, so that they generate an aggregate set of deltas, and then those aggregated deltas set are applied to each weight and bias.

2.2 Deep Neural Networks (Learning)

Now that we have introduced Neural Networks, we can move on and present the concept of *Deep Learning*. Deep Learning is a subset of Machine Learning, which is in turn part of Artificial Intelligence, that involves Networks potentially capable of correctly handling inputs that they have not seen before, i.e., they learn to make intelligent decision on their own. In Deep Learning, we let Networks perform autonomous decisions and find novel solutions to existing problems. This field is also called Deep Neural Learning, and Deep Learning Nets have several advantages with respect to shallow/traditional ones. Indeed, they are far less dependent on human intervention to learn, and are capable of learning from unsupervised (i.e., unstructured or unlabeled) data: this is an important benefit, as unlabeled data is more abundant and in general easier to obtain. Figure 2.3 shows the context in which Deep Learning is located:

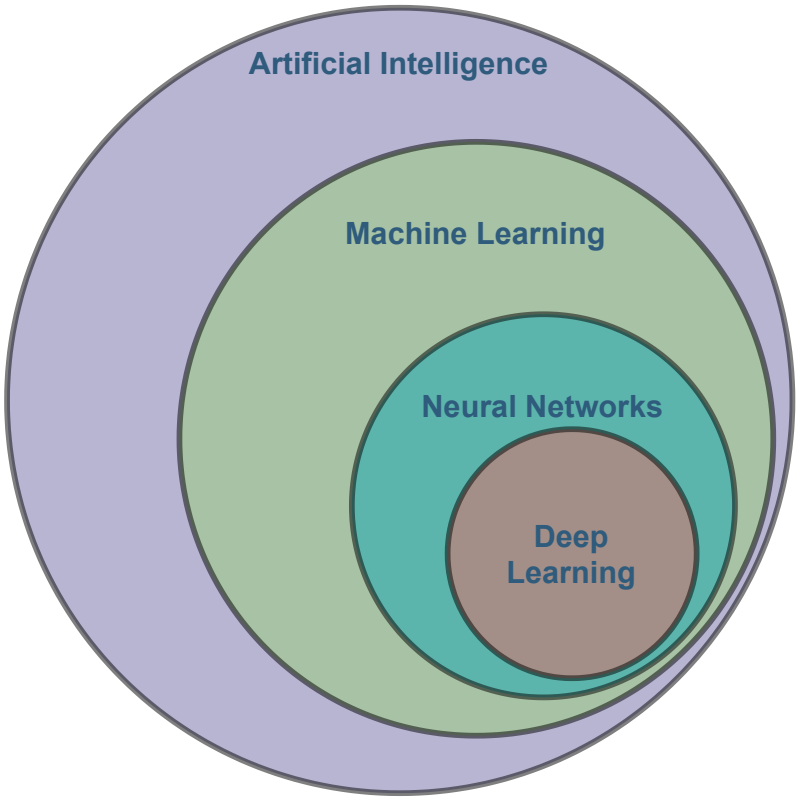

Figure 2.3: Venn diagram representing the relationship among Artificial Intelligence, Machine Learning, Neural Networks and Deep Learning

To be eligible of being called *Deep Neural Network* (*DNN*), a Net must feature more than three layers (counting both the input and the output layers); the adjective "deep", in fact, refers to the application of several layers to the Network.

Given their nature, these Networks are complex, but in return they are also very powerful, provided they get sufficient amount of training data. Again due to their structure, there is theoretically no limit to the classification accuracy they can achieve, but such performance is obtained via progressively heavier computational demands and training time. Nowadays, DNNs are capable of solving complex tasks such as object detection, speech recognition, and language translation.

2.3 Pruning of Neural Networks

One of the hardest challenges to overcome for Network Verification is undoubtedly the large number of parameters that need to be processed. For this reason, Verification techniques available today are able to output a fast enough response only when dealing with Nets of modest size. To tackle this problem, several pruning techniques have been proposed. The first one to become public in literature has been *Network pruning* (or *Network slimming*), introduced by [11]. Network slimming is a technique that allows to reduce the size of Neural Nets in an automated way, by removing from them low-importance neurons, together with their corresponding input and output connections. This approach also allows to initially exploit larger Nets, which guarantee higher probabilities of success (i.e., a too small Network may not feature enough neuron in order to learn the given task), and then to remove the extra neurons once training is complete; this process usually also leads to results that are better than the ones provided by the original Net. A similar approach is represented by *neuron pruning*, introduced by [27]. In this case, the aim is focused on automatically removing superfluous weights from neurons, rather than the whole neuron itself, in order to provide higher flexibility with respect to the previous technique. With this approach, irrelevant weights get set to 0 and remain frozen to that value. Indeed, while Network slimming severs all the connections of a neuron deemed as unnecessary, neuron pruning removes that same neuron only if all its connections end up being considered irrelevant. Both techniques take place after the Network training, and operate on weights/neurons whose values are below an user-specified threshold.

In the context of Network Verification, the two approaches have been detailed and extensively tested in [16]. Additionally, the concepts and results reported in [16] are also planned to be exported to NeVer2 [17]. pyNeVer fully supports weight and neuron pruning, and specifies peculiar parameters for both techniques. In particular, it features ρ, i.e. the *sparsity*

rate, which indicates the percentage of neurons or weights to prune.

2.4 Verification of Neural Networks

Neural Network Verification is among the most promising solutions to the problem of uncertainty of results provided by Neural Nets themselves. Verification algorithms allow to determine whether a given Neural Network can be considered safe to use or not for a given environment. This is achieved by picking some predetermined ranges for the Network's input(s), and by then checking if all the possible output(s) generated by these ranges fulfill some constraints enforced over the output(s) themselves. In other words, the idea is that given a predetermined range for the input(s) fed to the Neural Network, and the safe range(s) of the output(s) that should be yielded, one can formally compute whether all the computed output(s) fall inside the wanted range(s), that is, whether the Net can be declared safe or not. Verification algorithms available in literature are usually divided into four categories, depending on their algorithm's characteristics:

- *Constraint solving*: the Neural Net gets encoded into a set of constraints, and then the Verification process is reduced into a constraint solving problem
- *Search based approach*: algorithms belonging to this set imply that the Verification procedure is based on exhaustive search
- *Global optimisation*: Verification algorithms inside this category are inspired by global optimisation techniques
- *Over-approximation*: these Verification techniques are able to guarantee the soundness of the result, but not their completeness. In effect, when an algorithm belonging to this category states that a Net is safe, it really is, but Nets deemed as unsafe may instead turn out to be safe, in which case the negative response is imputable to the approximation error itself

In recent times, also other policies have been designed in order to divide Verification approaches into subgroups; for example, [24] proposes to split the techniques currently available in literature into four subgroups, based on the kind of formal guarantees they provide. [24] also acknowledges [32], which represents the basis of the NeVer project, as already mentioned in section 1.3. These are the four guarantee-based categories presented in [24]:

- *Exact deterministic* guarantee, which states exactly whether a property holds or not. Algorithms inside this subgroup do not resort to approximations, and usually take more time to execute
- *One-sided* guarantee, that provides either a lower bound or an upper bound to a variable, and thus can serve as a sufficient condition for a property to hold
- Guarantee with *converging* lower and upper bounds to a variable
- *Statistical* guarantee, which quantifies the probability that a property holds

According to the first list, NeVer and NeVer2 [17, 18] fall inside the *constraint solving* and the *over-approximation* categories, like [42, 30]. With the second criterion instead, they are comprised inside the *Exact deterministic* subgroup, featuring robustness and reachability properties. They are deterministic in the sense that the algorithm outputs a Boolean answer to the proposed Verification query: either the property holds (i.e., the Network is safe) or it does not hold (i.e., the Network is not safe with respect to that property). Different exact deterministic approaches leverage various constraint solvers, such as *SATisfiability* (*SAT*) solvers, *Linear Programming* (*LP*) solvers, *Mixed Integer Linear Programming* (*MILP*) solvers and *Satisfiability Modulo Theories* (*SMT*) solvers: our Verification techniques exploits this last option [28, 34], thanks to the abstraction-refinement approach originally proposed in [32]. The idea is to abstract a network into a set of Boolean combinations of linear arithmetic constraints. In practice, linear transformations between layers can be encoded directly (i.e., Affine layer Abstractions never feature approximation, and are fast to compute), while non-linear activation functions such as ReLUs are approximated - with both lower and upper bounds - by piecewise linear functions. It is shown that whenever the abstracted model is declared to be safe, the same holds true for the concrete model. Spurious counterexamples, on the other hand, trigger refinements and can be leveraged to automate the correction of misbehaviors [3, 12].

2.5 Abstraction/Refinement for Verification

The concept of *Abstraction* is key for the real-world application of Network Verification. It was first introduced in literature by [32], and it is a strategy

that allows to generate, from a given Neural Net, a new Net that over-approximates the real one, but requiring much less computations in order to be tested for safety. This turns the Verification test in a sufficient condition, i.e., we are assured that if the over-approximated Net fulfills the property, also the real one does. If the abstract Network breaks the property, we have no guarantee about the safeness of the concrete Network; we must either resort to less over-approximating abstractions, or test the real Net itself. So, in addition to pruning (see section 2.3), we can resort to an additional mean of decreasing computational load. this is pivotal in order to succeed, as Network safety Verification problems belong to *NP* (*non-deterministic polynomial time*) complexity class [25]. This degree of complexity is also why, in literature, Verification techniques always aimed at being sufficient conditions for certifying Net robustness. Our approach makes no exception, thus we know that some Networks we label as unsafe may actually be robust, and the Verification property rejection may be due to approximation errors.

In recent years, many abstraction solutions have been proposed, such as [38, 13, 36]. In particular, [38] has a very similar approach to the one exploited by PYNEVER, which is the engine of our Verification tool (see section 2.6). As already mentioned in section 2.4, Verification properties require input(s) bounds and safe output(s) areas. These input and output ranges outline multi-dimensional regions (in the case of two dimensions they define a polygon, with three a polyhedron, etc.), hence they are in general represented as *polytopes*. The approach of [38] uses *star sets* as a symbolic representation of sets of states, which are known in short as stars, and provide an effective representation of high-dimensional polytopes. These polytopes define a multi-dimensional area which is comprehensive of all the possible outputs of the Network for the given inputs. We represent polytopes and define Abstraction by picking a subclass of *generalized star sets*. In our context, a generalized star set is a tuple $\Theta = (c, V, R)$. Here, $V \in \mathbb{R}^{n \times m}$ is the *basis matrix*, that is obtained from the arrangement by column of a set of m *basis vectors* $\{v_1, \ldots, v_m\}$; then $c \in \mathbb{R}^n$ is a point called *center* of the star, and finally R: $\mathbb{R}^m \to \{\top, \bot\}$ is the *predicate*. Thus, the generalized star set comprises the following set of points:

$$\Theta \equiv \{z \in \mathbb{R}^n \mid z = Vx + c \text{ such that } R(x_1, \ldots, x_m) = \top\} \tag{2.5}$$

In general, it can be proven that any bounded and convex polyhedron can be represented as a generalized star set [38]. We only consider star sets in

which R is a conjunction of p linear constraints; in particular we require that $R(x) := Cx \leq d$, where $C \in \mathbb{R}^{p \times m}$ and $d \in \mathbb{R}^p$ for $p \geq 1$. Additionally, we also demand that the set $Y = \{y \in \mathbb{R}^m \mid Cy \leq d\}$ is bounded. C is called *constraints matrix*. From now on, we may refer to generalized star sets, which comply with our restrictions, simply as *stars*. Stars are polytopes in $\mathbb{R}^n$, and we represent the set as $\langle \mathbb{R}^n \rangle$. Given a star $\Theta = (c, V, R)$ and an Affine mapping $f : R^n \to R^m$ with $f = Ax + b$, the Affine mapping of the star is defined as $f(\Theta) = (\hat{c}, \hat{V}, R)$ where $\hat{c} = Ac + b$ and $\hat{V} = AV$. If $\Theta \in \langle \mathbb{R}^n \rangle$, then also $f(\Theta) \in \langle \mathbb{R}^m \rangle$, i.e., the affine transformation of a polytope is still a polytope. Thus, given a star $\Theta = (c, V, R)$, a generic Affine mapping $\bar{\Theta} = \{y \mid y = Wx + b, x \in \Theta\}$, with W (neuron weights) as the mapping matrix and b (neuron biases) as the offset vector, is simply another star, with these characteristics:

$$\bar{\Theta} = \langle \bar{c}, \bar{V}, \bar{R} \rangle,\ \bar{c} = Wc + b,\ \bar{V} = \{Wv_1, Wv_2, \ldots, Wv_m\},\ \bar{R} = R \tag{2.6}$$

Furthermore, [38] also proves that the intersection between a star and an half space (like the ones generated by ReLU's Abstractions), is still a star.

As explained, Abstraction techniques over-approximate layer output areas in order to limit computational load, especially in presence of non-linear layers, but this of course comes at the cost of precision. The reported literature shows that the star set representation features arguably the best trade-off between precision of results (in terms of over-approximation, where, under the same input/output restrictions, smaller obtained areas are better) and computational load. If the over-approximation of the areas gets too big, some properties may fail to get labelled as "safe" even when the Neural Network would not actually break the imposed restrictions. That said, instead of star sets, [36] uses abstract-domains as symbolic representation of states sets, inside a system called DeepPoly. DeepPoly features an abstract domain which is a combination of floating-point polytopes with intervals, coupled with tailored abstract transformers for common Neural Network layers (Affine transforms, ReLUs, Maxpools, etc.). In addition to generic polytopes, [13] also relies on *zonotopes*, a centre-symmetric restricted form of polytopes, as well as on simple bounding boxes, which are the most over-approximating figures of all the ones mentioned so far. The AI^2 algorithm reported in [13] can exploit the zonotope abstract domain, but performs badly with a general polytope domain in terms of computational time required: indeed, it is vastly outperformed by the star set representation proposed in [38]. In general, AI^2's performance is also

worse with respect to DeepPoly, especially considering it does not exploit the intrinsic structure of ReLU layers. The algorithm presented in [13] is the oldest one, and its analysis has been one of the reasons why [36] and [38] tend to outperform it. NeVer2 development decided to exploit star sets like it has been done in [38], as this Abstraction solution is currently the one featuring the most promising results. The effectiveness of generalized star set representations is also proven by [39].

That said, all three algorithms have successfully proved to be capable of withstanding adversarial attacks, which is another crucial issue with Neural Nets. An *adversarial example*, or *adversarial attack*, consists in the slight modification of an original input data into a perturbed version, mimicking an adversary who is trying to trick the Network. The problem of adversarial examples is discussed in detail in [42, 3, 24, 30]. Generating and feeding Networks with adversarial examples is a pivotal element in order to achieve a sound Verification algorithm, as they could cause irreparable damage if ignored, and as it is hard to teach Neural Networks how to correctly handle them without resorting to tailored approaches.

Lastly, the concept of *spurious counterexamples* refers to the weak spots in the over-approximation generated by the Abstraction. Spurious counterexamples trigger the refinement process, and can be leveraged to automate the correction of misbehavior. Indeed, *CounterExample-Guided Abstraction Refinement* (*CEGAR*) [7], has thus far been successfully applied in many Neural Network Verification scenarios.

2.6 pyNeVer

In order to handle Neural Network Verification, this Thesis leverages [17], i.e., the NeVer2 framework, which will be soon finalized and ready to get distributed. NeVer2 consists in a tool that does not just provide automated Network Verification, but it also features several capabilities: in its stable release, users will be also able to draw[1], edit, train, convert to different formats, test, save/load Neural Networks. In the next future, we are planning to also add support for Network pruning/slimming and automated repair. The ultimate goal of NeVer2 can be summarized as follows: "*Current state-of-the-art tools are restricted to Verification/analysis tasks, in some cases they are limited to specific network architectures and they might prove difficult to use for the non-initiated. NeVer2 is a new system*

[1]In this context, "drawing" means rearranging Network layers over NeVer2's canvas

that aims to bridge the gap between learning and Verification of DNNs and solve some of the above mentioned issues" [17].

The NeVer2 tool is completely back ended by pyNeVer [18] for all the non-graphical tasks, i.e., all the above mentioned NeVer2's capabilities except drawing. Indeed, both the case studies presented in Chapter 4 and 3 have been carried out by directly or indirectly leveraging the Python[2] back end of the NeVer2 tool, whose logic is described in [33, 17]. That said, pyNeVer can be used without the NeVer2's GUI, but doing so requires a deeper knowledge of the subject as well as some degree of expertise with the Python programming language. In return, pyNeVer features some additional capabilities that are not available yet in NeVer2, such as Network pruning/slimming. We set for ourselves the goal to erase this discrepancy in the next future. Also, at the moment one of the most relevant shortcomings of pyNeVer is that it only supports sequential Fully Connected Neural Networks with ReLU and Sigmoid activation functions, so we only dealt with this specific subset of layer types. In addition to that, few of the classes now present in pyNeVer are still wrappers of corresponding PyTorch[3] ones, but this will be changed in the future. pyNeVer features an *internal representation* for the Networks, which is translatable at will into the widespread ONNX[4], PyTorch (and soon TensorFlow[5]) data formats. pyNeVer's Python package also features some built-in examples, which are useful to learn how it works and also to test its capabilities.

pyNeVer is currently available as a package on PyPI[6] (acronym for Python Package Index, i.e., the official third-party software repository for Python). Being an open source project, its source code is publicly available on GitHub[7].

When NeVer2 fully releases, pyNeVer will completely be invisible to the user, as the GUI will be in charge of communicating with it, so that non-experts may easily take advantage of its capabilities. As already mentioned, our goal for the future is to give users the chance of leveraging all the functionalities provided by pyNeVer through a graphical user interface.

[2] https://www.python.org
[3] https://pytorch.org
[4] https://onnx.ai
[5] https://www.tensorflow.org
[6] https://pypi.org/project/pyNeVer
[7] https://github.com/NeVerTools/pyNeVer

2.7 Behavior Trees

Another key concept exploited during the making of this Thesis is the one of *Behavior Tree* (*BT*). Behavior Trees are directed acyclic graphs that are used to organize a plan and determine the switching behavior between different sub-tasks of the plan itself [8]. Due to their particularly high human readability, they are becoming more and more used in fields such as Robotics, control systems and video games. The main advantage of Behavior Trees is their ability to describe the interaction of complex tasks, without worrying about the implementation of each one of them, providing both modularity and reactiveness to the overall scheme. This way, it is easy for developers to focus on the bigger picture of a problem first, and to deal with the actual tasks at a later time. Although most, if not all, of the complex processes can be described via other solutions, like for example *Finite State Machines* (*FSM*) or *Hierarchical State Machines* (*HSM*), BTs are still widely renown as one the better solutions [14].

Behavior trees imply a discrete vision of time flow, in which each time instant is called *tick*. The tree is made of nodes, each one connected with one or more of them, in a parent-child structure. These nodes can be *Control nodes* or *Execution nodes*. Among the control nodes, we are interested in the *Sequence* and *Fallback* blocks. As their name implies, Sequence nodes sequentially tick the execution of all their children (i.e., once one has finished, the next one begins), until one of them fails. Conversely, Fallback nodes tick the execution of its children until one of them returns *success*. In both cases, if a child returns *active*, it gets repeatedly ticked until it reaches a terminal state. The leaves of the Behavior Tree can only be Executions nodes, namely *Actions* and *Conditions*. When ticked, Actions may return success, *running* or *failure*: these nodes are better suited for longer and more complex tasks, that are not usually carried out in a single tick. On the other hand, Condition nodes may only return success or failure when ticked: these are best used for simple checks, e.g., for checking whether a robot has reached its goal position or not.

We used Groot[8] to plan and print the Behavior Tree for our case study.

2.8 ROS and Gazebo

In this section we briefly present two staple pieces of software widely renown in the Robotics community, and to which we also decided to resort

[8] https://github.com/BehaviorTree/Groot

for this Thesis. ROS[9] (acronym for Robot Operating System) is a free and open source middleware (or "low-level framework") that simplifies the interconnection and communication between real or simulated robots and software. Nowadays, it is becoming a standard in the Robotics field. One of the most important features of ROS is its modular architecture, allowing developers to publish their software using common interfaces. Furthermore, ROS is language agnostic, meaning that projects and submodules can be written in different languages and they will still be able to communicate with each other. ROS projects are contained in folders called packages. These packages contains various pieces of code that implement the so called *nodes*, which have the ability to communicate among themselves thanks to various solutions. The communication means that ROS supplies are messages, services, and actions:

- ROS *messages* are used mainly for communications that require constant or very frequent updates (e.g. data streams). Each of them is sent to one or more *topics*, which are named buses used to exchange messages between different nodes/processes in an organized way. More nodes can write to the same topic, and more nodes can read from it. Message publishers and subscribers are anonymous

- ROS *services* are instead needed when the communication between two processes must be synchronous. This also means that this kind of interaction is suited for lower frequencies with respect to ROS messages case. In other words, ROS services implement a typical client/server communication scheme

- ROS *actions* implement asynchronous communication between two processes, in which the client can ask the server for information about the status of the request, or cancel the action execution altogether. Also in this case then, the interaction follows a client/server architecture

Nodes communication is handled by the ROS master node. In addition to facilitating the task of implementing communications between different pieces of code, ROS also provides other powerful capabilities out of the box, such as sound implementations of *logging*, *launch files* and the *Parameter Server*[10].

[9] `https://www.ros.org`
[10] `http://wiki.ros.org/Parameter%20Server`

The code of this Thesis has been developed as a series of Catkin packages[11]; concerning the various ROS editions, we decided to use ROS Noetic, which is the last and currently still maintained version of ROS1. We chose Noetic because it directly targets Python 3 (instead of the previous versions, which targeted Python 2.7) and due to the great amount of software available for it. For that reason, we developed our code inside Ubuntu 20.04.4 LTS[12], as it is officially supported by this ROS version. Another essential element for our Work has been the physics simulator. We decided to employ Gazebo[13] (version 11.9.0). One of the main reasons behind this choice is that, albeit being an independent project, it is well integrated with ROS, to the point which major Gazebo releases match the ones of ROS. Furthermore, unlike other equivalent pieces of software, Gazebo is completely open source. Being among the most renown simulators inside the Robotics field, it features a remarkable amount of plugins. In addition to that, it offers command line commands to allow for a deeper control and introspection about the simulation itself. Furthermore, it supports four different physics engines: ODE[14], Simbody[15], Bullet[16] and DART[17].

2.9 Reinforcement Learning

Reinforcement Learning is one of the three major paradigms of Machine Learning, along with Supervised Learning and Unsupervised Learning. The core idea of Reinforcement Learning is to find a balance between *exploration* and *exploitation*. Exploration is the activity through which an *agent* (i.e., a Neural Network) tries different paths, in order to discover new and hopefully better solutions to a given problem. The concept of exploitation, instead, refers to the agent resorting to its past experience in order to try and replicate the rationale behind the choices that produced the best *rewards*. Reinforcement Learning aims to train an agent that is able to perform actions in a given environment, which usually consists in a *Markov Decision Process* (*MDP*). The actions can be picked from a discrete pool, or they can be chosen from a continuous space. Once the agent has chosen the action to perform (A_t), it gets applied to the environment.

[11] http://wiki.ros.org/catkin
[12] https://ubuntu.com
[13] http://gazebosim.org
[14] http://opende.sourceforge.net
[15] https://simtk.org/projects/simbody
[16] https://pybullet.org
[17] http://dartsim.github.io

Then, an interpreter computes the changes to the *state* ($S_t \rightarrow S_{t+1}$) of the environment and the reward (R_{t+1}) to give to the agent as a feedback. The agent receives the updated state of the environment along with the reward, and picks a new action to perform based on these two elements. Figure 2.4 graphically shows this cyclic process, which is usually divided in *epochs*, i.e., the amount of attempts "from scratch" that the agent is given, and *steps*, which represent the time unit for each epoch. At each step, the agent can and must perform a single action. Each epoch is composed by a fixed maximum amount of steps. Ultimately, the goal for the agent is to learn a *policy* that maximizes the reward obtained. Furthermore, rewards are characterized by different schemes/patterns, namely *dense reward* and *sparse reward* ones. Dense reward schemes feed the agent with a non-zero reward after every action taken (or after most of them): this way, the agent's behavior can be tightly controlled the by providing it a constant feedback. On the other hand, sparse reward schemes produce non-zero rewards only in a limited subset of possible scenarios, i.e., most of the actions will not provide any immediate feedback to the agent. The rationale behind this is to let the agent learn the correct behavior without risking to corrupt its choices by designing sub-optimal reward patterns. In general, sparse reward schemes should eventually provide better results overall, but they tend to require a much greater training/learning time.

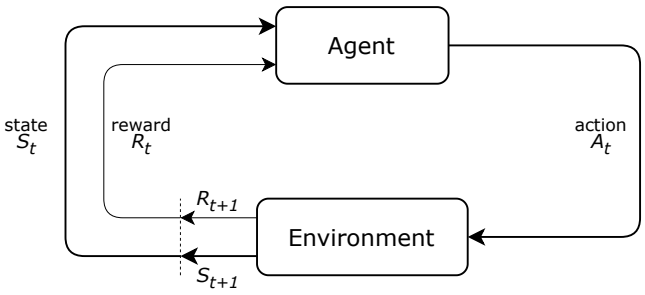

Figure 2.4: Schematic depiction of a generic Reinforcement Learning process

Over the last years, Reinforcement Learning approaches have gained more and more attention in the scientific community, due to their promising results and also thanks to the constant increase of computing power. Out of the many different algorithms proposed, one of particular interest is the *Soft Actor Critic* (*SAC*) algorithm [19]. The Soft Actor Critic algorithm is an off-policy technique consisting in two separate agents, namely an actor and a critic. The actor is the one deciding the action to apply to the environment, while the critic has the duty to evaluate the action proposed by the actor,

based on the changes caused to the environment's state. Being an off-policy Learning Algorithm here translates to the fact that we can update the Q-network and the parameters of the policy with experience data collected from a policy different than the current one; then, for every actor roll-out, we save all the transition data inside a replay buffer. To express it in other words, the learning agent learns the value function according to the action derived from another policy. Indeed, off-policy Reinforcement Learning agents store experiences in an experience buffer: during training, the agent samples mini-batches of experiences from the buffer and uses these mini-batches to update its actor and critic function approximators. One of the key aspects of the SAC is the maximization of the *entropy* (which regulates the degree of predictability of the actions proposed by the agent), aimed at granting both stability and exploration-prone behaviors: the ultimate goal is to achieve an actor that behaves in the best way possible (in terms of obtained rewards) while performing actions that are as random as possible. In general, Reinforcement Learning algorithms aim to maximize the expected sum of rewards $\sum_t \mathbb{E}_{(\mathbf{s_t},\mathbf{a_t})\sim p_{\pi_\theta}} r(\mathbf{s_t},\mathbf{a_t})$; in the Soft Actor Critic case the objective equation $J(\pi)$ is given by:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(\mathbf{s_t},\mathbf{a_t})\sim p_\pi}[r(\mathbf{s_t},\mathbf{a_t}) + \alpha\mathcal{H}(\pi(\cdot|\mathbf{s_t}))] \tag{2.7}$$

where $\mathcal{H}(\cdot)$ is the measure of the entropy and α is the temperature coefficient, which regulates the relevance of the entropy itself with respect to the rewards. In other words, α determines the stochasticity of the policy; it is also called *entropy regularization coefficient/factor*. This coefficient is equivalent to inverse of the reward scale presented in [19].

Another important algorithm featured in recent literature is surely the *Deep Deterministic Policy Gradient* (*DDPG*): a model-free off-policy actor-critic algorithm that combines *Deterministic Policy Gradients* (*DPG*s) and *Deep Q-Networks* (*DQN*s). One of the most important features for our Work is that the DDPG is suited for continuous action spaces, rather than discrete ones, like it is instead for standard DQNs. This is thanks to the actor-critic architecture, as *"it is not possible to straightforwardly apply Q-learning to continuous action spaces, because in continuous spaces finding the greedy policy requires an optimization of a_t at every time step; this optimization is too slow to be practical with large, unconstrained function approximators and nontrivial action spaces"* [29]. The DDPG trains a deterministic policy in an off-policy way. Because the policy is deterministic,

if the agent were to explore on-policy, at the beginning of its execution it would probably not try a wide enough variety of actions to find useful learning signals. Indeed, on-policy approaches imply learning agents that learn the value function according to the current action, derived from the policy that is currently being leveraged. To make DDPG policies explore better, some noise is added to their actions (namely, to their exploration policy) at training time, i.e., $\mu^{'}(s) = \mu_{\theta}(s) + \mathcal{N}$. As already mentioned, while still learning a deterministic policy, the DDPG algorithm extends the DQN approach to the continuous action space, rather than being only applicable to the discrete case. Another advantage of DDPG over DQN is its "conservative policy iteration" [41] for the both actor's and the critic's parameters, with $\tau \ll 1 : \theta^{'} \rightarrow \tau\theta + (1 - \tau) \cdot \theta^{'}$. In this way, the target Network's values are forced to change slowly, while in the case of the DQN the target Network does not get adjusted at all for prolonged time frames. Lastly, another detail showed in [29] is particularly interesting for Robotics, namely the normalization of different units of low dimensional features. For instance, let us assume that a model is designed to learn a policy with a robot's positions ad velocities as input: these two elements are different by nature; furthermore, the same physical unit may also vary a lot from robot to robot. [29] solves these issues by applying batch normalization, so that each dimension of all the samples of a mini-batch gets normalized. In particular, we are interested in a further improvement over [29], i.e., the DDPG algorithm featuring *Prioritized Experience Replay* (*PER*) [23, 22], whose core idea is to frequently replay experiences associated with particularly successful attempts, or with extremely ineffective ones. The replay buffer size, which determines the amount of samples that can be stored in the buffer, can be adjusted in order to tune the algorithm's performance.

Other than algorithm-specific parameters, it is also important to briefly introduce all the most important ones which are common to the majority of the Reinforcement Learning approaches. The *discount factor* γ is a hyperparameter that represents how much future events lose their value according to how far away in time they are: it is used to tune how "greedy" the Network should be. It is limited between $[0; 1]$, in which bounds a value of 0 means only caring about immediate rewards (thus maximizing greediness), while greater values imply farther reward propagation through time. Next, τ is the *smoothing factor*, and is used to update target Network weights to match the ones of their value functions. The *weight decay* regularization parameter indicates the rate at which the Net weights lose

their values, by effectively subtracting, from the current weight values, the weight decay constant times the weight value itself. The weight decay is used to balance underfitting and overfitting phenomenons: a weight decay value close to 0 favours overfitting (a 0 implies no decay at all), while higher values may instead lead to underfitting.

In addition to state-of-the-art Reinforcement Learning Algorithms, we also need to briefly introduce OpenAI Gym[18][2]: a toolkit that simplifies Reinforcement Learning techniques creation and comparison. Gym is an open source project that offers an easy-to-learn syntax, along with many built-in examples to draw inspiration from. The mission of OpenAI with Gym is to offer a standard for Reinforcement Learning tasks definition, and to provide a way to generate better benchmarks by creating a large and open database of Reinforcement Learning environments. Gym's interpretation of Reinforcement Learning is based on few key concepts. A Gym *environment* is a particular setup, consisting of an algorithm, an agent (in our context, a robot), a set of possible actions, an environment (the world in which the agent operates), and a reward scheme. Each Gym environment can be invoked and trained or tested for any number of *episodes*. An episode is an agent's new attempt to solve the given task. It corresponds to the concept of "epoch", presented earlier in this section. Each episode can have any number of *steps*, which is the unit defining the time flow: an agent can and must perform an action at each step. After the action is performed, if the task is not solved, the episode can continue with the next step. If the agent has triggered a task failure condition or has reached its goal, that episode ends and the next one begins.

Thanks to its recent grow in popularity, many developers have started working with Gym and, among the others, also Robotics engineers started integrating their software with it. Thus, the last essential element we introduce in this chapter is the openai_ros package[19], which provides an interface between Gym and ROS/Gazebo (see 2.8). The core principle of openai_ros is modularity: for instance, this package allows users to apply different tasks and different algorithms to the same robot with ease. The structure of openai_ros projects is composed by these elements:

1 - *Training Script*

1a - *Learning Algorithm*

1b - *Learning Launcher*

[18] `https://gym.openai.com`
[19] `http://wiki.ros.org/openai_ros`

2 - *Training Environment*

2a - *Task Environment*

2b - *Robot Environment*

2c - *Gazebo Environment*

The Training Script holds the the Learning Algorithm, i.e., the Reinforcement Learning algorithm chosen for the training, and the Learning Launcher, i.e., the piece of code dedicated to starting the whole training or testing process. The Learning Launcher usually also holds the required OpenAI Gym instructions and parameters, such as the number of episodes and steps to carry out, as well as the instantiation of the Gym environment itself. Relative to the training environment, the Task Environment is the one containing the code describing the job that the robot has to carry out. This element contains, for example, the starting configuration of the robot of each episode, the different conditions that trigger the reward (and its values), the definition of the actions that the robot may perform, and which are the elements defining the state of the environment. The Robot Environment holds the description of the robot used, in terms of the sensors it has at disposal, as well as the implementation of the code that allows it to perform the action chosen by the algorithm. Lastly, the Gazebo Environment is the interface between ROS/Gazebo and Gym's Reinforcement Learning architecture, and allows the user to watch the robot learn through a graphical simulation.

We leveraged Gym and openai_ros in order to link our case study's robot simulation to the Neural Network training process (see Chapter 4), and we created and adjusted Training Scripts, Task Environments and Robot Environments in order to tailor these resources to match the specifications required by our Work.

Chapter 3

Improvements on NeVer2

This Chapter is dedicated to the contributions of the Thesis to NeVer2, described in section 3.2, and to the presentation of a demonstrative experiment (section 3.3) that we used to test and present our tool.

3.1 Introduction

In this Chapter we are going to show the improvements over the NeVer2 tool, that are aimed at making the use of pyNeVer accessible to Robotics engineers, regardless of their knowledge about Deep Learning techniques and Neural Networks. For that reason, most of the focus was given to the modelling of a simple graphical user interface capable of integrating all pyNeVer capabilities, especially Network learning and Verification. We also provide some of the results of the Adaptive Cruise Control experiments, which can be seen as a small showcase of both pyNeVer's and NeVer2's capabilities.

3.2 NeVer2

One of the contributions of this Thesis is the improvement of the NeVer2 tool, which is an open source project currently available on GitHub[1]. NeVer2 represents a novel resource for easily creating, editing, saving, converting, training and verifying Neural Networks. We are planning to add additional capabilities, such as Network automated repair and pruning/slimming, and to support more nodes/Network formats. NeVer2 features an easy-to-use graphical interface, in which the user can pick

[1] https://github.com/NeVerTools/NeVer2

and arrange the desired blocks to create a Net, like it has been done for the example in Figure 3.1.

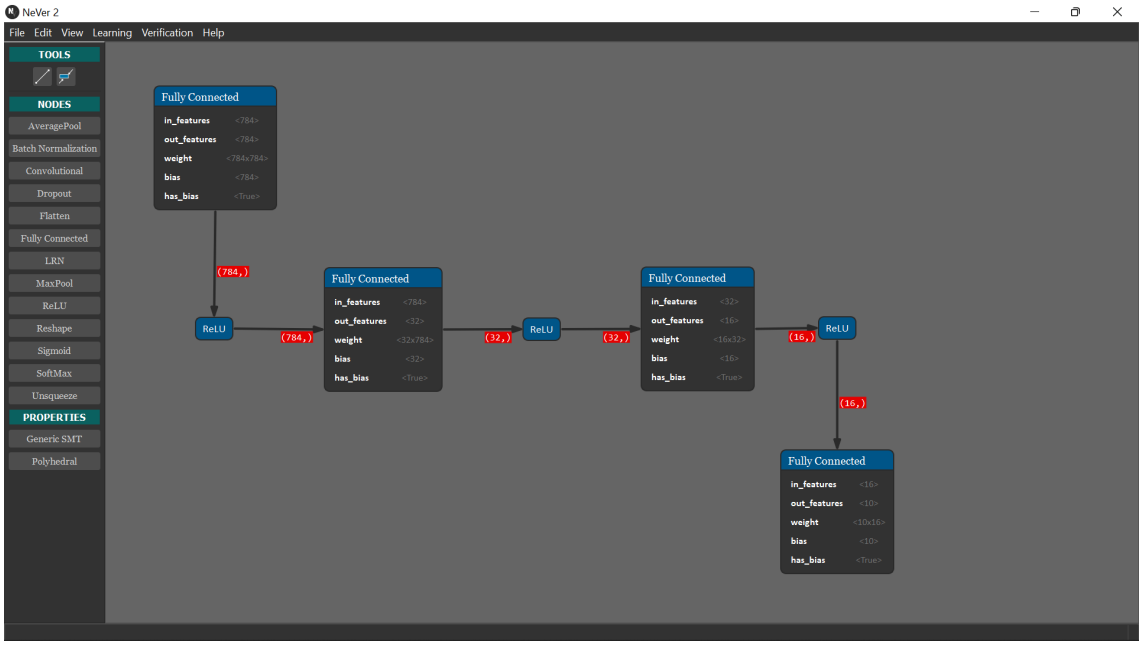

Figure 3.1: The GUI of NeVer2 showing a sample Neural Network layout

Taking advantage of an early prototype available in our University, we decided to implement the graphical interface of pyNeVer, i.e., the computational back end presented in section 2.6. From that idea, we gave birth to CoCoNet[2], a PyQt5[3] Python graphical user interface that implemented some of the capabilities featured by pyNeVer, namely Network construction, Network loading/saving, *Generic SMT* and *Polyhedral* (see section 2.5) property definition, and Network conversion from one format to another (currently, only to/from ONNX, PyTorch, and pyNeVer internal representation). We want to share this tool with the scientific community, with the aim of providing a reliable and easy and graphical way to perform the above mentioned operations. We also think that CoCoNet could be leveraged by other researchers as a basis for their implementations of Network Verification, pruning, repair, and so on. This further step is what we have done, indeed, with the NeVer2 project, by following the concepts presented in [33, 17]. Starting from CoCoNet's interface and pyNeVer capabilities, part of this Thesis focused on increasing the amount of tasks that can be fully carried out via the GUI, namely Network learning and Verification. We also added some accessory features like keyboard shortcuts and tooltips.

[2] https://github.com/NeVerTools/CoCoNet
[3] https://pypi.org/project/PyQt5

Figure 3.1 also allows us to introduce the core elements of the graphical user interface of both CoCoNet and NeVer2. These are the *canvas*, the *menu bar*, the *tool bar*, and the *status bar*. The canvas occupies the greater portion of the GUI, and is the place where the user can arrange Network blocks, connect/disconnect them, drag new ones from the tool bar (which is located on the left side of the canvas itself) and edit their properties by double-clicking on them. The tool bar hosts all the available Network layers and properties, along with the "insert line" and "insert block in edge" buttons. The former allows to simply draw connections between layers, while the latter allows to insert a block between two of the already connected ones. The status bar at the bottom of the interface shows the ID of currently selected block (or, in the case of a line, the IDs of the previous and next blocks) and the current mode, i.e., the default mode, "GraphicLine drawing" mode, or "Block insertion" mode. The two last are triggered by their keyboard shortcuts or via the already mentioned "insert line" and "insert block in edge" buttons present in the tool bar. Lastly, the menu bar holds a list of all the possible actions the user can perform, along with their respective keyboard shortcuts.

3.3 Adaptive Cruise Control

In order to have a clear picture about how to effectively integrate and organize the capabilities of pyNeVer on NeVer2, one of the steps we took was to create a simpler ad-hoc case study, which revolved around the *Adaptive Cruise Control* (also known as *ACC*). Thanks to a collaboration between the University of Genoa and Stellantis[4] (ex PSA), which revolved around Neural Network Verification in the context of level 1 autonomous driving, we could leverage an algorithm that implements the Adaptive Cruise Control. We present this experiment mainly for demonstrative purposes. Before diving in the experiment description itself though, it may be useful to provide a brief introduction to the subject.

3.3.1 Context

The Adaptive Cruise Control is an evolution of the standard Cruise Control feature available in most modern vehicles. The objective of the Cruise Control is to keep the vehicle at the speed chosen by the driver without having them interacting with the throttle: this is useful during long highway

[4] https://www.stellantis.com

trips, as the user is not forced to handle the gas pedal during that period. This kind of input translates in a more or less constant signal, that can be easily managed by a basic controller. The Cruise Control, engineered to overcome such annoyances, can be deactivated at will by simply pressing the brake pedal: albeit useful, the Cruise Control is not able to handle any unexpected event, which results in the driver needing to take back full control of the vehicle. The ACC is a step forward in terms of usefulness, as its functioning is able to automatically handle the distance from the preceding vehicle. Thanks to its connection with front-mounted proximity sensors, the ACC is able to act ahead of time in order to avoid potential collisions with other cars, and then to return to the desired speed. This is carried out by a closed-loop feedback control system, which constantly monitors the current speed of the vehicle and the distance from the cars in front.

With these notions in mind, what we have done was to try and train various Neural Nets so that they could replicate the behavior of the STELLANTIS ACC algorithm, and then to propose and test some Verification properties. Due to the black box nature of the code (at least from our point of view), we relied on that algorithm "as it is" and trained our networks around it without leveraging data coming from the real world. In this context then, we were working in a classical Supervised Learning scenario; this is also supported by the fact that the ACC script can provide us with all the reference output we need. Algorithm 2 shows how ACC has been implemented. K_{v1}, K, K_d, K_{v2}, v_{ref} and $acons\_factor$ are six parameters whose values have been respectively left to their defaults of 0.36, 1.4, 1.5, 1, $\frac{90}{3.6}$, and 1. The other two parameters, TIV and D_0, will be described in subsection 3.3.2. The Verification properties we wanted to prove are:

- assure that the Neural Network does not output accelerations that are outside of the chosen bounds
- assure that the Network does not output positive accelerations when too close to the front vehicle
- guarantee that the Net does not output negative accelerations when there are no obstacles ahead or the front vehicle is too far

The "too close" and "too far" concepts will be thoroughly defined in subsection 3.3.4.

Algorithm 2 STELLANTIS algorithm for the Adaptive Cruise Control

Require: v_p, v_r, D_0

$D_s = TIV \cdot v_p + D_0$

$K_{Ds} = K \cdot D_s$

$K_{1Ds}(K-1) \cdot D_s$

$K_{2Ds}(2K-1) \cdot D_s$

if $v_r \geq 0$ **then**

$acons_{v_r} = K_{v1} \cdot v_r$

else

if $D > K_{Ds}$ **then**

$acons_{v_r} = -K_{v2} \cdot \frac{{v_r}^2}{2(D-D_s)}$

else

$acons_{v_r} = -K_{v2} \cdot \frac{{v_r}^2}{2K_{1Ds}} \cdot \frac{K_{2Ds}-D}{K_{1Ds}}$

end if

end if

$acons_D = K_d \cdot \frac{D-D_s}{D_s}$

$acons_D = min(0.3,\ max(-1.5,\ acons_D))$

$acons_{RDD} = min(1,\ max(-3,\ acons_{v_r} + acons_D))$

$acons_{RVV} = min(1,\ max(-1,\ K_{v1} \cdot (V_{ref} - v_p)))$

$acons = min(acons_{RDD},\ acons_{RVV})$

$acons = acons\_factor \cdot acons$

3.3.2 Data Set

As mentioned in subsection 3.3.1, our experiment did not involve real world data. We instead decided to artificially generate them by drawing samples from uniform distributions for the algorithm's inputs, leveraging their given lower and upper bounds. We then obtained the reference outputs of training by applying the generated samples to the ACC algorithm. The inputs (and the two parameters we thoroughly tweaked) are:

- v_p [$m \cdot s^{-1}$], which is the current speed of the vehicle carrying the Adaptive Cruise Control. It is bounded between [$0\, m \cdot s^{-1}$; $50\, m \cdot s^{-1}$]
- v_r [$m \cdot s^{-1}$], which is the current relative speed with respect to the vehicle ahead. If the vehicle holding the ACC is going faster, this value is negative; it is going slower, v_r is positive instead. It is bounded between [$-50\, m \cdot s^{-1}$; $50\, m \cdot s^{-1}$]
- D [m], that represents the current distance between the two vehicles. It is limited inside [$0\, m$; $150\, m$]
- TIV [s], a parameter that represents the supposed human reaction time. Increasing this translates into imposing a greater safety distance from the vehicle ahead
- D_0 [m], a parameter which represents the base safety distance value

The only output of the algorithm is *acons* [$m \cdot s^{-2}$], that represents the output acceleration for the vehicle carrying the ACC. The allowed output values are bounded inside [$-3\, m \cdot s^{-2}$; $1\, m \cdot s^{-2}$]. Figure 3.2 shows input and output box plots for a million data samples.

Given the nature of the ACC inputs, i.e., that two of them are fixed parameters, we decided to generate 16 different data sets, each composed by a million samples, that featured 16 different combinations of TIV and D_0. TIV was given the values of 1, 1.5, 2, 2.5, while D0 was given the values of 2.5, 5, 7.5 and 10. We split all these data sets in two parts, one for training and one for testing, with the ratio of 4:1. Our training phase lasted 100 epochs for each of the 16 data sets. We chose the *Adam* optimizer [26] and the *ReduceLROnPlateau* scheduler[5]. For both our loss function and our precision metric we leveraged the *Mean Squared Error* (*MSE*) loss. We set batch sizes to 32 for training, validation, and test sets. In our setup, we dedicated 30% of the training set to the validation process. Concerning

[5]See PyTorch documentation at `https://pytorch.org/docs`

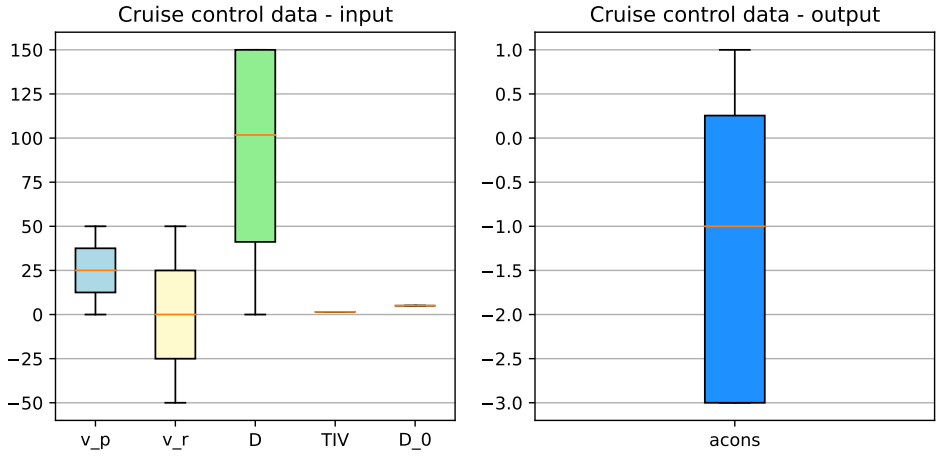

Figure 3.2: Box plot for a million samples of the Adaptive Cruise Control data set ($TIV = 1,5; D_0 = 5$)

NEVER2's optional parameters, we also set the learning rate to 0.01, the weight decay to 0.0001 and the training scheduler patience to 3. The last value represents the number of consecutive epochs without loss decrease that trigger training procedure abortion.

3.3.3 Neural Networks

We also took the chance to test various Neural Network architectures, while still keeping in mind current PYNEVER limitations in terms of supported layers and overall Net size. After some preliminary experiments, we decided to train, test, and Verify three different architectures, all of them composed by Fully Connected and ReLU layers. We named them *NET0*, *NET1* and *NET2*. They feature increasing complexity both in terms of layers number and in the amount of neurons per layer. NET0's architecture is composed by five layers in total:

$$FC_1(3,20) \longrightarrow ReLU_2 \longrightarrow FC_3(20,10) \longrightarrow ReLU_4 \longrightarrow FC_5(10,1) \tag{3.1}$$

where the subscripts indicate the layer number, and where the values inside the parentheses show the layer sizes. For example, FC_3 is the third layer of the Network, its type is Fully Connected, and holds $20 \cdot 10 = 200$ weights plus (10 biases).

The next Network, NET1, has the same amount of layers as NET0, but it features more weights and biases:

$$FC_1(3,50) \longrightarrow ReLU_2 \longrightarrow FC_3(50,40) \longrightarrow ReLU_4 \longrightarrow FC_5(40,1) \tag{3.2}$$

Lastly, NET2 's architecture is characterized by four more hidden layers than the others:

$$\begin{aligned} &FC_1(3,20) \longrightarrow ReLU_2 \longrightarrow FC_3(20,20) \longrightarrow ReLU_4 \longrightarrow \\ &\longrightarrow FC_5(20,20) \longrightarrow ReLU_6 \longrightarrow FC_7(20,10) \longrightarrow ReLU_8 \longrightarrow \\ &\longrightarrow FC_9(10,1) \end{aligned} \tag{3.3}$$

Figures 3.3, 3.4 and 3.5 show their structure over NEVER2 canvas.

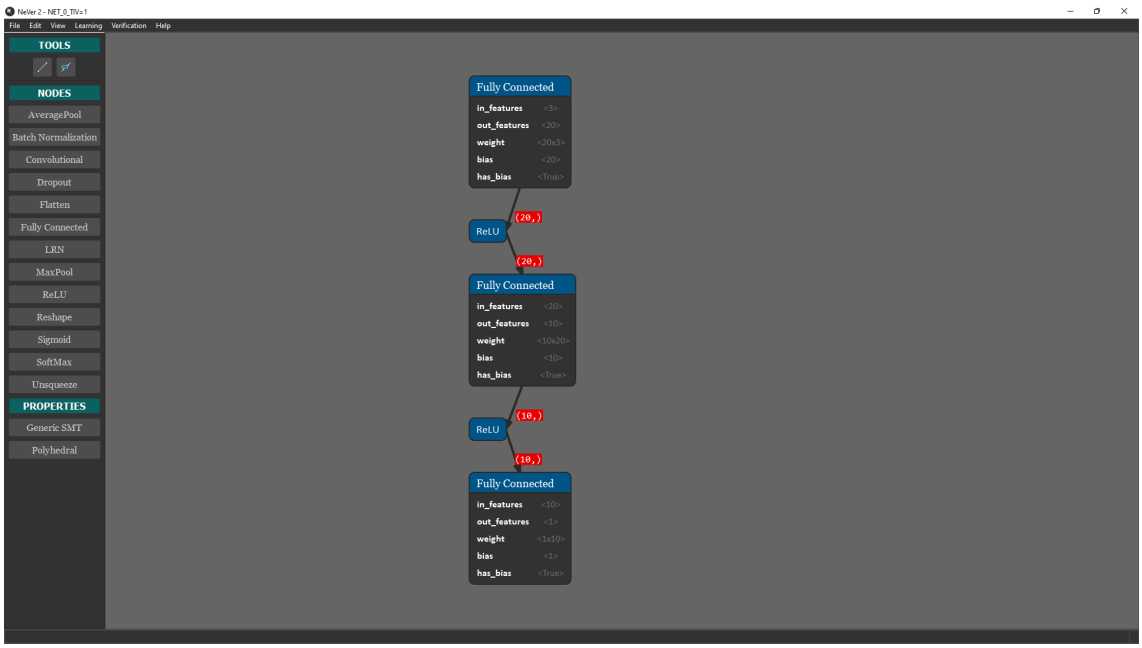

Figure 3.3: NEVER2 representation of NET0's architecture ($TIV = 1; D_0 = 2.5$ data set)

3.3.4 Verification Properties

We decided to design three properties for the ACC case study, and we tested them with four different Verification parameter sets. The first property that we defined, *OutBounds*, simply checks that output accelerations do not exceed the bounds of the real ACC algorithm script, provided that we supplied our Nets with inputs inside the bounds expected by the script itself. In other words, we want to make sure that our Networks do not output values that would be never proposed by the original ACC algorithm. Figure 3.6 shows the NEVER2 implementation of this property for one of the 16 data sets. We wanted to Verify that, given these preconditions:

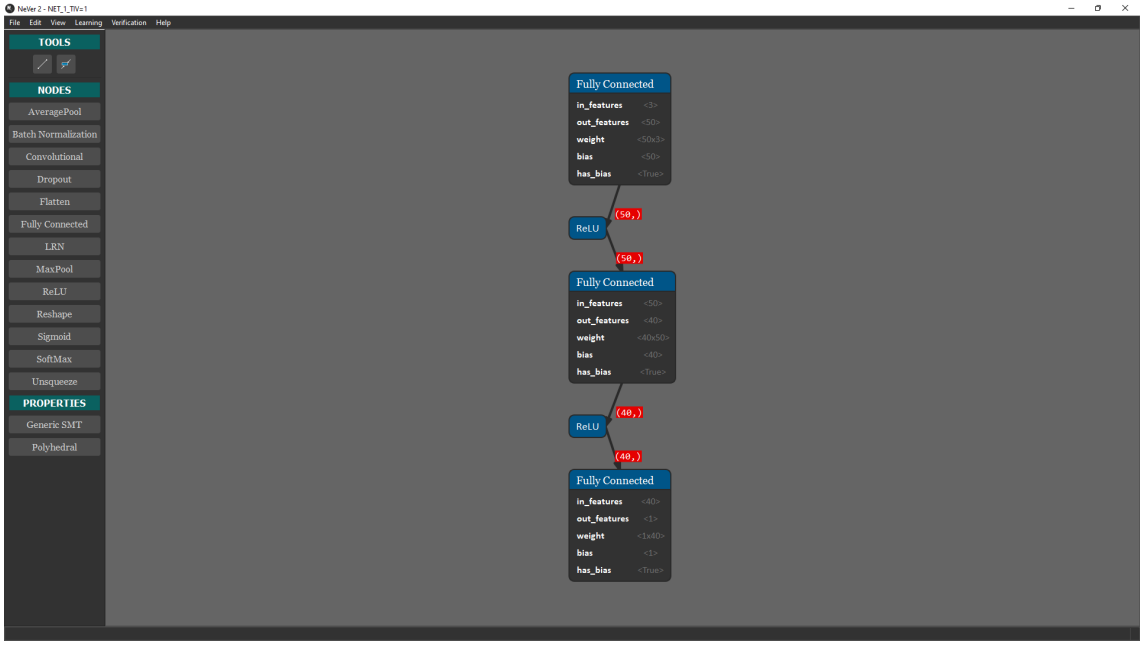

Figure 3.4: NEVER2 representation of NET1's architecture ($TIV = 1; D_0 = 2.5$ data set)

$$\begin{aligned} 0 \leq v_p \leq 50 \\ -50 \leq v_r \leq 50 \\ 0 \leq D \leq 150 \end{aligned} \tag{3.4}$$

the output lied inside these bounds:

$$-3 \leq acons \leq 1 \tag{3.5}$$

The second property we designed is *Near0*, which is aimed at making sure that the ACC system does not output positive accelerations when the vehicle ahead is "too near". We defined this concept via an input inequality:

$$TIV \cdot v_r + D_0 \geq D + \epsilon \tag{3.6}$$

where ϵ is the distance tolerance value. Other than this inequality, we also added the inputs' lower and upper bounds in the same way we did for OutBounds. That said, the structure of the additional input precondition comes from the fact that $TIV \cdot v_r$ is the safety distance necessary to stop the ACC car in time if the one ahead suddenly stops. To this, we add the D_0 buffer value (which, like TIV, is constant for each data set), and subtract the tolerance, which is adjusted in order to tune the Verification itself, i.e., to test the perturbation robustness of the Network for this property. The

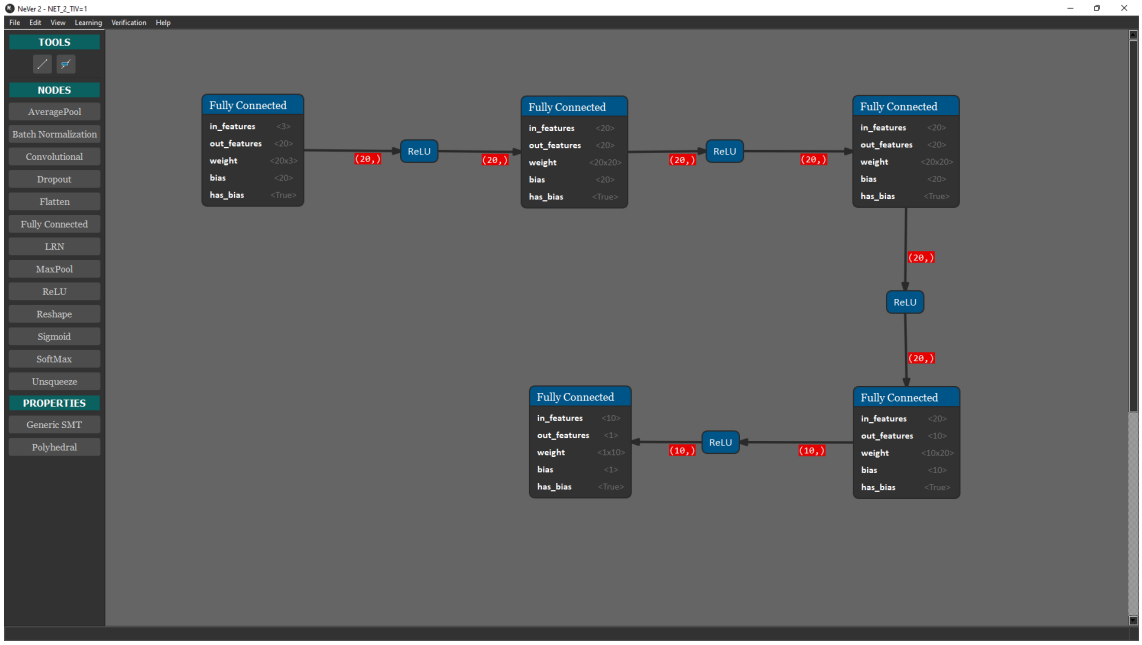

Figure 3.5: NEVER2 representation of NET2's architecture ($TIV = 1; D_0 = 2.5$ data set)

corresponding output safe area is:

$$-3 \leq acons \leq 0 \tag{3.7}$$

We thus do not want the Network to output positive accelerations in this specific scenario. Then, the last property we designed is *Far0*, which is symmetrical to Near0. In addition to the inputs' lower and upper bounds expected by the ACC algorithm, we formulated this precondition:

$$TIV \cdot v_r + D_0 \leq D - \epsilon \tag{3.8}$$

where ϵ is again the distance tolerance value. With this property, we wanted to Verify that when the vehicle carrying the ACC is too far from the vehicle ahead (or when there is no vehicle ahead at all), the Net does not suggest negative accelerations. Thus, the output safe area in this case is:

$$0 \leq acons \leq 1 \tag{3.9}$$

In addition to the properties themselves, we also defined different parameter sets. The sets of parameters determine which Verification technique is leveraged. We distinguish between three heuristics: *Over-approximation*, *Mixed*, and *Complete*. The Over-approximation heuristic provides a sufficient condition only, so that we know that if the property is verified, the

Figure 3.6: Creation process of the OutBounds property with NeVer2. Detached property blocks are treated as input preconditions, while property blocks linked to the Neural Network are the output areas to Verify

Net is safe with respect to that property. If the property is not verified, the Net may still be safe anyway. This lack of precision is compensated by a lower computational effort required, i.e., this set is usually the fastest to be processed. The Complete parameter set instead does not approximate the computation thus, albeit usually being the slowest, its results are precise (i.e., sound and complete). The Mixed heuristic lies between these two antipodes, and is aimed at being simultaneously more precise than the over-approximation set and significantly faster than the Complete one. Like the for Over-approximation, Mixed results are sound, but not complete. Our ultimate goal is to prove the usefulness of this heuristic. In this experiment, we decided to feature two different parameter sets for the Mixed category. These sets, Mixed and Mixed2, differ for the number of neurons to refine during abstraction (1 and 2, respectively). Over-approximation does not feature neuron refinement, while for the Complete parameter set the value is 100. In general, increasing the amount of neurons decreases the approximation, but lengthens the computation.

3.3.5 Results

We ran our tests on a machine featuring Ubuntu 20.04.03 LTS, a pair of Intel Xeon Gold 6234 CPUs, three NVIDIA Quadro RTX 6000/8000 GPUs (with CUDA enabled), and 125.6 GiB of RAM. For the sake of brevity, we

are only going to report here a fraction of the experiment results. Overall, we could appreciate that all our theories are respected:

- OutBounds is always True
- If a property is verified by Over-approximation, then all the other heuristics verify it too
- If the Complete parameter set fails to verify a property, then all the other heuristics fail the Verification too
- On average, Mixed and Mixed2 are significantly faster than Complete, but Over-approximation is slightly faster
- Increasing the tolerance value makes properties easier to Verify

Tables 3.1 and 3.2 show the complete results for the data set with $TIV = 1.5$ and $D_0 = 5$, and with $\epsilon = 0$ and $\epsilon = 20$ respectively. Here, the interesting portion is the Far0 property for NET0, which gets verified only with a tolerance of 20 for Mixed, Mixed2 and, necessarily, Complete, thus contributing to prove the usefulness of our parameter set. Given the modest Network size, computational time gaps between the first three parameter sets are modest and, in general, Mixed and Mixed2 feature negligible additional computational loads with respect to their Over-approximation counterpart.

We also performed some tests with different distance tolerance values, but we then decided to focus our attention on this couple, in order to concentrate on the main case study of this Work, which is presented in Chapter 4.

Table 3.1: Results for the ACC data set with TIV set to 1.5 seconds, D_0 to 5.0 meters and with 0 meters of tolerance (ϵ). Elapsed times [s] have been rounded to the third decimal place

ACC: $TIV = 1.5$ **|** $D_0 = 5.0$ **|** $\epsilon = 0$				
Network	**Property**	**Param. Set**	**Result**	**Time elapsed**
NET0	OutBounds	Over-Approx	True	5,139
		Mixed	True	5.055
		Mixed2	True	5.112
		Complete	True	6.273
	Near0	Over-Approx	False	5.666
		Mixed	False	5.251
		Mixed2	False	5.203
		Complete	False	6.319
	Far0	Over-Approx	False	5.078
		Mixed	False	4.986
		Mixed2	False	5.139
		Complete	False	5.186
NET1	OutBounds	Over-Approx	True	5.931
		Mixed	True	6.662
		Mixed2	True	7.309
		Complete	True	51.683
	Near0	Over-Approx	False	5.906
		Mixed	False	6.676
		Mixed2	False	8.071
		Complete	False	50.469
	Far0	Over-Approx	False	5.709
		Mixed	False	5.888
		Mixed2	False	6.301
		Complete	False	13.041
NET2	OutBounds	Over-Approx	True	9.525
		Mixed	True	10.482
		Mixed2	True	12.525
		Complete	True	26.958
	Near0	Over-Approx	False	9.515
		Mixed	False	10.292
		Mixed2	False	13.636
		Complete	False	24.496
	Far0	Over-Approx	False	9.753
		Mixed	False	9.944
		Mixed2	False	12.148
		Complete	False	13.27

Table 3.2: Results for the ACC data set with TIV set to 1.5 seconds, D_0 to 5.0 meters and with 20 meters of tolerance (ϵ). Elapsed times [s] have been rounded to the third decimal place

ACC: $TIV = 1.5$ **|** $D_0 = 5.0$ **|** $\epsilon = 20$				
Network	**Property**	**Param. Set**	**Result**	**Time elapsed**
NET0	OutBounds	Over-Approx	True	5.037
		Mixed	True	5.063
		Mixed2	True	4.996
		Complete	True	6.203
	Near0	Over-Approx	False	5.034
		Mixed	False	5.101
		Mixed2	False	4.965
		Complete	False	5.345
	Far0	Over-Approx	False	5.008
		Mixed	True	5.016
		Mixed2	True	5.068
		Complete	True	5.62
NET1	OutBounds	Over-Approx	True	5.948
		Mixed	True	6.934
		Mixed2	True	7.232
		Complete	True	52.318
	Near0	Over-Approx	False	5.436
		Mixed	False	5.797
		Mixed2	False	5.955
		Complete	False	7.667
	Far0	Over-Approx	False	5.344
		Mixed	False	5.776
		Mixed2	False	6.226
		Complete	False	8.212
NET2	OutBounds	Over-Approx	True	9.532
		Mixed	True	10.149
		Mixed2	True	12.065
		Complete	True	26.794
	Near0	Over-Approx	False	9.379
		Mixed	False	9.872
		Mixed2	False	11.653
		Complete	True	10.696
	Far0	Over-Approx	False	9.453
		Mixed	False	9.848
		Mixed2	False	11.558
		Complete	False	10.854

Chapter 4

Case Study

This chapter describes the main experiment we carried out during the development of the Thesis. We start with a presentation of the robot and the environment used for the tests in section 4.1. Then, in section 4.2, we proceed with showing the behavior tree (whose theory is introduced in section 2.7) for our robot. Next, sections 4.3 and 4.4 will discern between the robot's scripted skills and its learned ones. Finally, section 4.5 will detail the algorithms and the resources leveraged for learning, and describe their peculiarities.

4.1 Introduction

Our case study revolves around safe robot navigation and manipulation. We developed a simulation of a domestic environment in which a robot is free to roam. The goal of the robot is to clean the rooms in which it wanders, and it does so by reaching different target locations and picking up junk from the ground, which will be then disposed in a dropbox. For this case study, we decided to implement an ad-hoc robot model: Turtlex. The name is a crasis from Turtlebot[1] and WidowX[2], as they are two of the main building blocks of our robot. Indeed, Turtlex features a differential-drive Kobuki basis, on top of which some poles and stacks support the five-degree-of-freedom arm. All the arm's joints are revolute ones, while the gripper is equipped with two prismatic joints, one for each finger. The robot also features an Asus Xtion PRO LIVE[3] RGBD camera for object recognition, and an SCIP 2.0-compliant Hokuyo laser range finder for obstacle avoidance. Our robot is shown in Figure 4.1.

[1] https://www.turtlebot.com
[2] https://www.trossenrobotics.com/widowxrobotarm
[3] http://xtionprolive.com/asus-xtion-pro-live

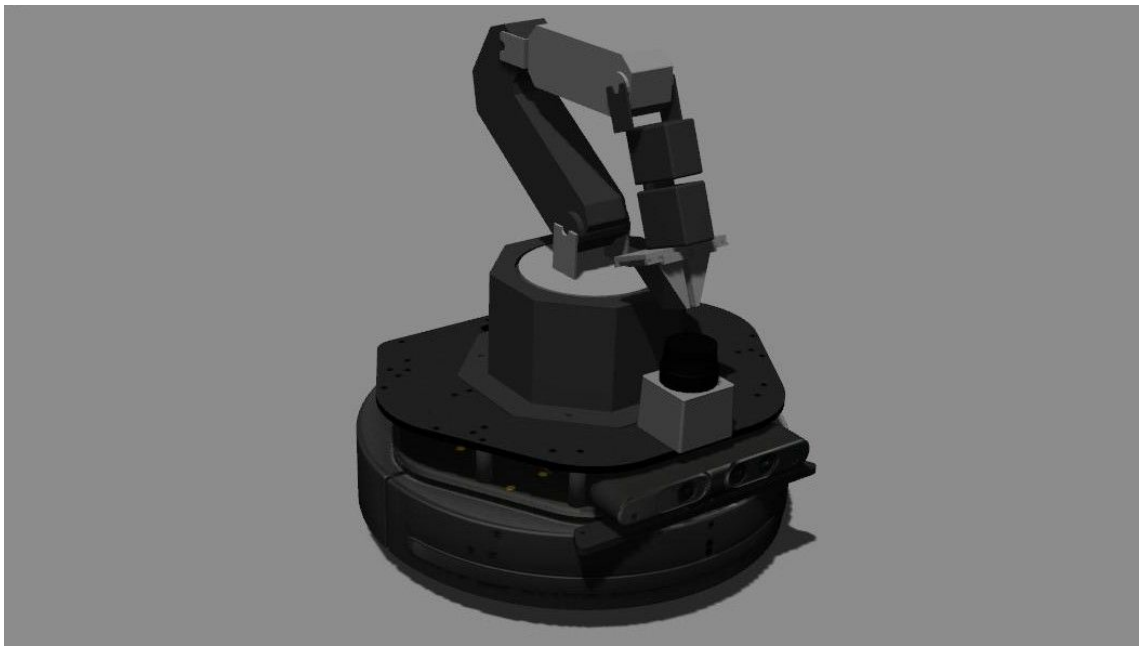

Figure 4.1: The Turtlex robot

We decided to split the robot's capabilities into scripted skills and into learned ones. The first category encloses the abilities that Turtlex is able to perform via a predetermined pattern (namely, a script), thus not resorting to Machine Learning. The other group instead comprises the skills that the robot learns to perform by using Neural Networks. While the latter group a the pivotal element of our Work, the former is also important nonetheless, as it allowed us to clearly define and restrict the tasks that the Networks had to learn.

The domestic environment we created is shown in Figure 4.2. The junk elements/toys to be picked up are represented by the colored spheres. The starting point of the robot is located at the center of the smaller room.

4.2 Turtlex's Behavior Tree

This section is dedicated to the presentation of the Behavior Tree of our case study. As described in section 2.7, Behavior Trees are used to represent the interactions, along with their logic, between the all different actions an agent is able to perform. In our case, the agent is the Turtlex robot, while the possible actions are embedded in the leaves of the Behavior Tree of Figure 4.3.

The robot starts its execution with a Sequence. The Sequence ticks a Fallback node, which in turn has two more Sequence nodes underneath. The first of these handles the behavior of the robot while it is cleaning the

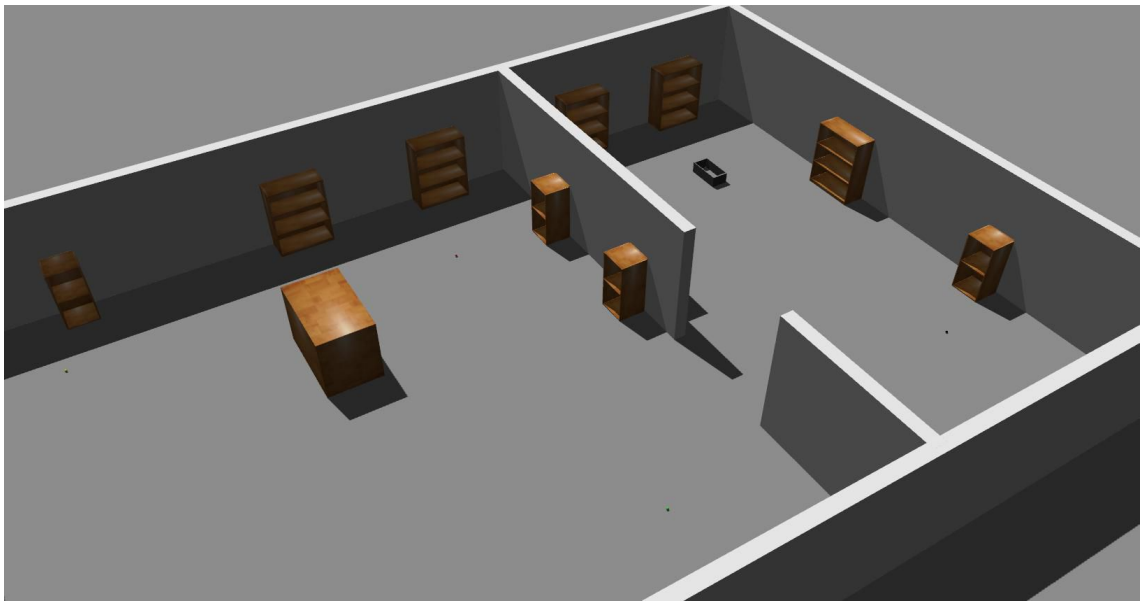

Figure 4.2: The domestic environment designed for our TURTLEX case study. Some obstacles and colored spheres (junk elements) are visible, along with the grey dropbox

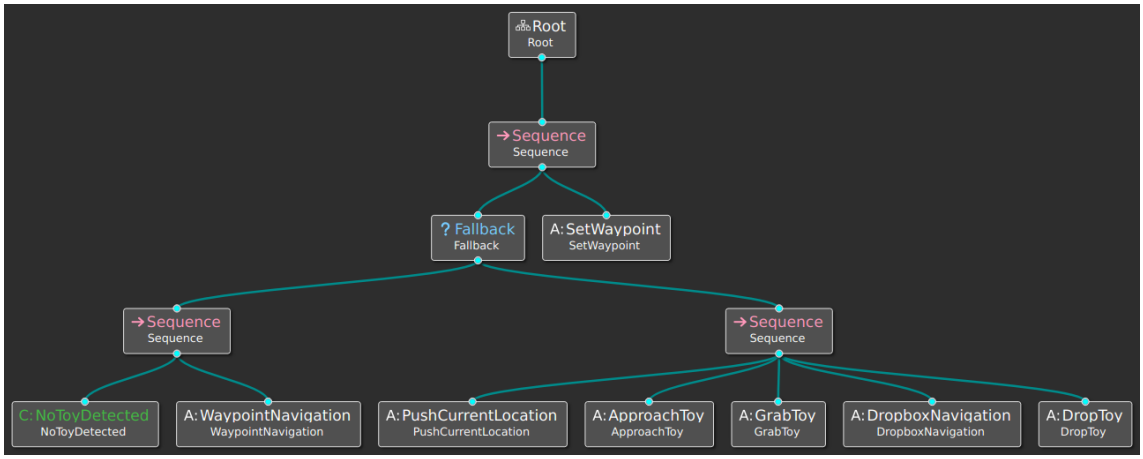

Figure 4.3: Behavior Tree for the TURTLEX case study

house(i.e., wandering around) by picking its goal location from a *Last-In-First-Out* (*LIFO*) waypoint queue. When the execution of the experiment starts, the LIFO queue holds all the target goals that the robot has to reach. These can be randomly generated (from a pool of achievable ones) or scripted, and the process can be finite or endless (i.e., the LIFO queue gets replenished with the same goals once empty). The first leaf of the Sequence is a Condition node, named *NoToyDetected*, that returns success until some junk is found (e.g., a toy). This Condition block is handled by the camera mounted on the robot, which is able to recognize when there is something for the robot to pick up. If no junk has been found in that tick, the robot can continue performing its *WaypointNavigation* Action: the TURLTEX keeps

on moving, trying to reach the current goal. When the Sequence ends with a success, i.e., the current target position has been reached, the Fallback node is over: its father Sequence node ticks the *SetWaypoint* Action. This leaf pops the first element of the LIFO queue and that becomes the new goal target for the robot navigation. Getting back to the Fallback node, if the Turtlex finds some piece of junk before reaching its target, the next active leaf will not be WaypointNavigation. In that case indeed, the Fallback node ticks the next Sequence node, and the next Action to be executed is *PushCurrentLocation*: the robot stores the interrupted goal location on top of the LIFO queue, so that it will be attempted again with the next tick to SetWaypoint. This way, regardless of the displacement of the junk over the floor, all the goals will always be carried out by the robot in the end. Then, the next leaf is *ApproachToy*: here the robot simply gets closer to the spotted element, up to a predetermined distance. After that, the *GrabToy* Action can start: the robotic arm approaches the object and lifts it up. Next, *DropboxNavigation* pushes the location of the dropbox into the LIFO queue, and then orders the robot to reach it, in the same way as it is performed in WaypointNavigation. Finally, once the robot has got to the goal (in this case, the dropbox location), the last Action *DropToy* triggers the release of the object into the dropbox. As this Sequence gets completed, the parent Fallback also ends, and SetWaypoint restores the previously interrupted navigation goal, making it again the active one.

We decided to implement the Actions of the Behavior Tree as ROS action servers (see section 2.8), as they are well integrated into ROS/Gazebo environments, and as they provide status feedback features "out of the box", which are useful in our scenario. The LIFO queue is hosted on the ROS Parameter Server, in order to allow different pieces of code to easily interact with the stored data (i.e., the code handling navigation and the one handling junk recognition).

As it will be shown in the rest of the Chapter (in particular, inside sections 4.3 and 4.4), some of our robot's abilities are scripted, hence certain portions of our Behavior Tree do not feature additional alternative behaviors/ recovery routines (i.e., Fallback nodes).

4.3 Scripted Skills

As mentioned in section 4.1, our robot features a set of scripted skills. Scripted skills are all the hard-coded abilities in which Neural Networks have not been involved. While in future contributions these might be

eventually replaced by learned ones, they have so far been needed in order to isolate and efficiently tackle the core tasks we wanted to learn through Reinforcement Learning.

If we take a look at Image 4.3, we can notice that all the skills are represented by the Execution nodes (i.e., the leaves of the tree); the only outliers in this group are SetWaypoint and PushCurrentLocation, which do not properly fit into the "skill" category. That said, the capabilities we decided to implement as scripted are: the Condition node NoToyDetected, the Action node ApproachToy, and a portion of both GrabToy and DropToy Action nodes. NoToyDetected is handled by a script which gets fed with the images of the front camera: thanks to edge and color detection, it is able to adequately distinguish between junk laying on the ground (represented by colored spheres) and unmovable obstacles like walls or pieces of furniture. This script does nothing until it finds a piece of junk laying on the ground; only then, it takes control of the robot by interrupting its current navigation goal pursuit. Our script relies on OpenCV[4] Python libraries to process incoming data. We tried to plan ahead and organize the structure of our code, so that in the future this skill could be easily replaced with a learned one by training a Neural Network for object recognition, while still leveraging incoming data from the RGB-D camera. The other fully scripted node, ApproachToy, again uses OpenCV and incoming camera images in order to let the robot reach the desired distance from the colored sphere along with centering its view over it. This means that the script is able to have the robot move away or get closer to the sphere, while also adjusting the heading in the meantime: the algorithm makes sure that the process always ends with the same relative configuration between the piece of junk and the robot itself. The last two skills mentioned, GrabToy and DropToy, are only partially scripted. Indeed, these two Action nodes expect a trained Neural Network to move the joints of the robotic arm, but the eventual gripper movement is always handled by a script. This is due to various reasons. First, object picking and manipulation in general can be quite difficult to pull off in simulations, especially without any ad-hoc setup or the implementation of some exceptions/"tricks" with the physics engine. Doing this would have translated in an excessive level of complexity for the scope of the Thesis over this particular aspect. Second, Turtlex's gripper does not feature a complex architecture (i.e., it is equipped with just two prismatic joints as fingers) and an additional Deep Network solution for it would have been an overkill. So, we decided that in the case of GrabToy

[4] https://opencv.org

the scripted portion would allow the gripper to close their fingers and grab the sphere while, conversely, in the case of DropToy it would allow it to release the object, by opening the fingers and getting them back to their initial configuration.

The movement of both the gripper and the rest of the arm in the scripted portions of GrabToy and DropToy is carried out by by leveraging the MoveIt Motion Planning Framework[5]: MoveIt is a powerful open source tool able to solve inverse kinematics problems and to perform collision checking, among other capabilities. Once setup, MoveIt allows to easily issue new configurations to the arm joints, either by specifying the desired end effector position (i.e., MoveIt computes the inverse kinematics for the arm) or by directly listing the desired values of the joints (forward kinematics computation). We thus decided to keep the gripper management fully scripted via MoveIt as the interaction with other objects in the Gazebo simulation adds further degrees of complexity; this may be addressed in the future.

4.4 Learned Skills

Besides the scripted skills described in section 4.3, Turtlex also features some learned skills, i.e., acquired via Reinforcement Learning approaches. The first one is the *navigation skill*, while the second one is the one for object manipulation (*manipulation skill*). Our robot makes use of Neural Networks in order to reach target locations avoiding obstacles in the meantime, and also in order to move the joints of its arm to reach a goal position for the end effector. Our main interest was to get a pair of small and reliable Nets that allowed us to achieve safe robot navigation and manipulation. The first skill is usually carried out through the aid of *Simultaneous Localization And Mapping* (*SLAM*) algorithms, but this approach features some issues. Instead, for instance our approach completely overcomes errors due to odometry approximation, as the Network learns to achieve goals regardless of physical shortcomings/biases. Relative to the arm case, our Neural Network model learns to handle forward kinematics queries, without suggesting the actuators impossible or self-damaging configurations. As mentioned in section 4.3, we leveraged the MoveIt planning framework. In the context of the learned skills, we planned a couple of named and predetermined configurations via MoveIt, for both the gripper

[5] `https://moveit.ros.org`

and the rest of the arm ; their self-explanatory names are “closed_gripper”, “open_gripper”, “rest_arm” and “picking_arm”. The two gripper configurations are used to grab the colored spheres by increasing the relative distance of the end effector fingers, and then reducing it when the object is in between them, while generating enough friction to lift it. Relative to the named configurations for the arm, rest_arm identifies the default and starting position of the arm itself, that is also kept during *WaypointNavigation*, *ApproachToy*, *DropboxNavigation* Actions (see section 4.2), and in general whenever the TURTLEX robot does not need to directly operate with the arm. The other named configuration, picking_arm, represents the goal configuration of the arm when there is an object to pick up. From this configuration, we extracted the corresponding end effector relative x, y, z position, so that we could feed it to the manipulation Network (more on this in section 5.3). During manipulation tasks then, the action flow works as follows: the arm, which starts in rest_arm position and with the open_gripper configuration, is supervised by MOVEIT, which listens to the manipulation Network’s outputs. These try to bring the arm to the picking_arm configuration (or to an equivalent one: here the important thing is that the end effector arrives in the correct position). Then, the gripper is switched to its closed_gripper configuration. Next, the Network guides the arm to its rest_arm configuration (or an equivalent one). When it is time to drop the object, the Network brings again the arm in its picking_arm configuration (or an equivalent one), the gripper releases the object by switching to the open_gripper configuration, and finally the manipulation Network brings the arm back to its rest_arm configuration, or to an equivalent one. If the manipulation Neural Network issues a configuration that is considered harmful for the robot by MOVEIT, the planning framework intervenes and ends the episode.

4.5 Learning Techniques

In this section, we are going to present the techniques and algorithms we chose for the learning side of our case study. We leveraged OPENAI GYM and OPENAI_ROS, which have been introduced in section 2.9, in order to create a tailored bridge between Deep Learning and Robotics simulations. These two resources let us leverage a working code stack so that we could focus, for the most part, on the choice of the learning algorithms and their adjustments to suit our use case. We also adapted the OPENAI_ROS package so that we could work with continuous space tasks, as it was originally

intended for simpler discrete ones. Indeed, it was designed to leverage volatile algorithms such as the off-policy *Q-learning* [40] and the on-policy *SARSA* [5], while, as already mentioned, we adapted it so that it could manage more complex Deep Reinforcement Learning tasks.

Concerning the navigation task, we opted for a Soft Actor Critic (SAC) approach [19]. We knew the potentialities of this algorithm thanks to recent publications, and we also found some promising applications to similar tasks as ours. Among these, [10] is one of the most relevant, and the one from which we took inspiration. The Soft Actor Critic was also chosen thanks to its ability to work with continuous action spaces, so that its outputs were not limited to a pool of scripted ones. One of our first contributions has been to translate the SAC version reported in [10] to the OpenAI Gym lexicon and syntax. This way, the code is now more modular, thanks to the clear distinction between algorithm, task, robot and simulation environments, which leads to a piece of software both easier to understand and to edit. We adapted the Soft Actor Critic algorithm in terms of the Networks it leverages in order to comply with our Verification aspirations (more on this in section 5.2) and pyNeVer's current capabilities. We re-imagined the terminal output, in order to provide useful live feedback to the user, and fixed some bugs and compatibility issues. We provided a more challenging simulation environment and tested several reward schemes. We also proposed new training patterns, and more robust logging features, along with accessory scripts such as a graphical plotter for training and testing results.

Relative to the manipulation task, we instead chosen the Deep Deterministic Policy Gradient (DDPG) with Prioritized Experience Replay (PER) algorithm, which again allowed us to work in a continuous action space. For this off-policy algorithm, we took inspiration from online resources[6], and then again changed some elements to tailor the code to our needs. Like for the navigation skill, we imported the algorithm inside of an OpenAI Gym framework, and effectively built from scratch all the necessary environments. We decided to expand the Robot Environment we created for the navigation task, so that both our learned skills could run on the same base, just like it is for the simulation one (i.e., the Gazebo Environment). This way our Robot Environment encompasses all the capabilities of the Turtlex robot, and handles all the machine's ROS Publishers and Subscribers. Once more, we adjusted some parameters for the DDPG algorithm in order to accommodate Verification and pyNeVer requirements:

[6] https://github.com/CUN-bjy/gym-ddpg-keras

among other adjustments, we considerably shrunk the hidden layers of the actor Network, and changed the type for some of them (but these aspects are better detailed in section 5.3). For consistency reasons, we tried to preserve the features we designed during the implementation of the navigation task's Task Environment and Learning Algorithm as much as possible. This way, we kept all the advantages of our contributions, while also providing a further level of standardization with respect to the Gym lexicon alone. Indeed, we designed a learning patterns that in the future might be applied to other robot simulations.

In Chapter 5, we will fully detail the Turtlex learning processes, along with our contributions and the decisions we took relative to the resources mentioned in this section.

Chapter 5

Learning

This Chapter will introduce the learning portion of our case study and present some preliminary contributions (in section 5.1); it will then proceed with a detailed description of how we implemented Neural Network learning for our unscripted skills, i.e., the navigation and manipulation tasks (sections 5.2 and 5.3, respectively). We will then show, inside section 5.4, the test results of our learned skills while in isolated environments. Finally, in section 5.5, we will describe the integration process between the learned skills and the rest of the code.

5.1 Introduction

When dealing with Neural Network training, one of the most important choices that has to be made is the one relative to the learning algorithm. We surveyed several optionsh before picking what we thought that would best suit our needs. But, as we wanted to train Neural Nets by feeding them data coming from a physics simulator, we first had to leverage some interfaces, such as OpenAI Gym and openai_ros (see section 2.9). Our first step was to clone and update the openai_ros package in order to make it compatible with the latest OpenAI Gym version currently available (version 0.18.3). After that, we created our Catkin packages following the model proposed by openai_ros. Both our navigation and manipulation packages have a similar structure, as they feature:

- A *configuration file*. This file stores all the learning algorithm's parameters and the training/testing ones. In general, all the values that an user may want to change are found in here
- A *launch file*. This piece of code is the one that launches all the necessary nodes and environments for the experiment to run, and

that also contains some adjustable configuration parameters for the physics simulation. These values should however be left as they are, as they are mostly useful for debugging purposes

- A *learning script*. This is the file that contains the algorithm and that starts the Gym environment. This piece of code is the one responsible for the training or testing episode flow and for the majority of the logging

- A *task environment*. This is one of the most important files, as it contains the definition of the task that the robot has to perform. Inside the task environment lie the implementations of the methods required by the Gazebo environment: the action selection, the state/observation acquisition, the reward scheme, the environment reset functions, and the check for episode completion. In addition to that, this piece of code also holds some additional functions, which are directly or indirectly required by the task itself

- A *robot environment*. This file is the same for both the learned skills, and provides an interface with the Gazebo environment (defined inside openai_ros). It also contains the implementation of low-level tasks such as sensor data acquisition, robot movement, odometry polling.

With respect to the original templates featured in openai_ros for these elements, we deeply customized all of them. In particular, the details for each of our learned skills are described in the next two sections. We thus improved some aspects of the openai_ros package, such as the rationale behind logging messages, and the clearness of its output over the terminal emulator. That said, we did not modify the Gazebo environment file for compatibility reasons. We thus require users to install openai_ros inside their Catkin workspace, as our software is still built on top of it.

During our experiments, we set a 0.2 seconds *running step* time for the navigation task. The running step is the time that the robot is given to physically carry out the chosen action inside the Gazebo simulation, i.e., to reach the output linear and angular velocities, and to let the action itself translate into a noticeable effect on the world. Concerning the manipulation task instead, we did not set a running step time at all, as we let MoveIt carry out the arm movement before proceeding with the subsequent episode step.

5.2 Navigation

Our navigation learned skill has been implemented by leveraging the Soft Actor Critic (SAC) algorithm [19]. Our actor Network has 14 inputs and 2 outputs. Their inputs are:

1.-10. 10 laser readings [$\approx 0.1000\,m;\ 30.0\,m$]. These come from the 720 Hokuyo laser scan rays, which then are divided in 10 equal sectors. Each value represents the average of that sector's readings. The laser scan has a scope of $180°$, with the front of the TurtleX being its center

11.-12. the agent's previous action values, bounded inside [$0\,m \cdot s^{-1};\ 0.22\,m \cdot s^{-1}$] and [$-2\,rad \cdot s^{-1};\ 2\,rad \cdot s^{-1}$] (see actor's outputs below)

13.-14. the current heading with respect to the goal [$-\pi;\ \pi$] and the current distance from it [$0\,m;\ \approx 18.0278\,m$]

The outputs of the actor Network are:

1. the linear velocity value [$0\,m \cdot s^{-1};\ 0.22\,m \cdot s^{-1}$]
2. the angular velocity value [$-2\,rad \cdot s^{-1};\ 2\,rad \cdot s^{-1}$]

We dedicated an abundant amount of time to experiment with this learned skill, and we tried to find a balance between Network performance and the possibility to verify it, as it is currently feasible only with Networks of modest size. Given that what ultimately we wanted to verify was the actor Network alone, that is the only one which we needed to keep relatively small; we thus decided not to shrink the size of the critic accordingly, as some of our tests proved that such an action resulted to be detrimental to the overall results. Thus, our Soft Actor Critic features a policy Network with a restrained number of hidden layers' neurons. Our actor sends its 14 inputs to a pair of ReLU-activated Fully Connected layers. Both feature an hidden dimension of 30, like for all the rest of the Network. The output of the second ReLU gets sent to two parallel Fully Connected layers. One of these, which we call *mean layer*, has its output values doubled before being activated by a *sigmoid* function (Σ; $\Sigma = \frac{1}{1+e^{-x}}$), whose exit values are then again doubled, and reduced by 1. This last step is done so that the two final output values are bounded inside $[-1;\ 1]$: indeed, we could not leverage a $\tanh$ due to pyNeVer currently not fully supporting it. Its parallel layer will be referred as *logarithmic standard deviation layer*. In the end, we obtain a function that features the same shape of a $\tanh$, but

which is instead compatible with our tools:

$$x_i = 2 \cdot \frac{1}{1 + e^{-2t_i}} - 1 \tag{5.1}$$

Where t_i is the i-th output of the previous layer, and where x_i is i-th action value normalized between $[-1;\ 1]$.

We now proceed to detail how the algorithm works relative to the actor's action selection. The behavior of the algorithm changes based on if the experiment is in training or testing mode. Notably, when training, each 10 episodes the actor is queried as if the experiment was in evaluation mode. For every step of each episode, the actor is asked to choose a single action, whose decision originates from a sampling procedure. The actor is given the current state of the environment, and this data is forwarded up to the two parallel layers, the mean and the logarithmic standard deviation ones. The latter gets translated into a simple standard deviation by applying its value as an exponent of the e base. From that standard deviation and the mean value, the algorithm creates a normal/Gaussian distribution. From that distribution, a sample x_t is picked. Then, the training action value is obtained by the application of the Σ activation function with a doubled input value, that gets then multiplied by 2 and reduced by 1, so that, as already mentioned, its output lies inside $[-1;\ 1]$ (see equation 5.1). The sample x_t is then also leveraged by the function that updates the actor's parameters. If instead this sampling process is carried out while in evaluation mode, the action picked by the agent is simply the mean layer output value, activated by the modified Σ function mentioned above. This way, the output actions proposed by the actor always lie inside $[-1;\ 1]$: we can then denormalize these output values and fit them to our effective output bounds. The formula we use to denormalize is shown in equation 5.2, in which $x_{i_{ub}}$ and $x_{i_{lb}}$ represent the bounds of the actor's i-th output layer (in our case, always -1 and 1), while $action_{i_{ub}}$ and $action_{i_{lb}}$ stand for the desired output bounds of that action's component. These last values are different for each action component (in our case, the robot's linear velocity and the angular one).

$$\begin{aligned} \text{action}_\text{i} &= \frac{action_{i_{ub}} - action_{i_{lb}}}{x_{i_{ub}} - x_{i_{lb}}} \cdot (x_i - x_{i_{lb}}) + action_{i_{lb}} \implies \\ \implies \text{action}_\text{i} &= \frac{action_{i_{ub}} - action_{i_{lb}}}{2} \cdot (x_i + 1) + action_{i_{lb}} \end{aligned} \tag{5.2}$$

We then have two identical Q-Networks for the critic, that are composed by four Fully Connected layers of hidden dimension equal to 500, all featuring

ReLU activation functions. They have 16 inputs, i.e., the 14 state values plus the 2 action ones; they have a single output, which corresponds to the metric used to evaluate the actor's choice: thus, it based on the decision made by the actor, relative to the current status of the environment. The critic also features a target network that helps with the training process.

Even if most of the literature we reviewed claimed to obtain better results with a sparse reward scheme, we acknowledged the opposite for our case study, and chose to opt for a dense reward scheme. Our sparse reward scheme gave the robot a positive reward when it got to the goal, and the opposite of that value when it instead got too close to an obstacle ($\leq 0.25m$). For all the other scenarios, no reward was provided. This did not work well for us, or, in the best case scenario, it would have required much more training episodes to be effective. We instead decided that, at every step, the robot received a base negative reward in order to nudge it towards being proactive. If, after that step, the episode was not prematurely terminated and the robot managed to decrease its distance from the goal, it would receive -0.5 points. On the contrary, not decreasing it would reward -1.0 point. If a step's action is the direct cause of an episode termination instead, the robot gets again rewarded differently based on the outcome of that action. If the episode is over because of the robot reaching the goal, the robot gets rewarded 300 points. If the episode is over due to the robot being too close to an object, the robot gets -300 points as reward. Algorithm 3 summarizes the whole reward scheme for this learned skill.

Algorithm 3 TURTLEX dense reward scheme for the navigation learned skill

```
if episode done then
    if goal reached then
        reward = 300
    else
        reward = −300                          ▷ Obstacle too close
    end if
else
    if goal distance reduced then
        reward = −0.5                          ▷ Reduced step punishment
    else
        reward = −1                            ▷ Full step punishment
    end if
end if
```

The Soft Actor Critic parameters that we employed for training were:

- $\alpha = 0.2$

- $\gamma = 0.99$
- $\tau = 0.01$
- learning rate $= 0.0003$
- batch size $= 64$
- replay buffer size $= 50000$

Where the learning rate is relative to the Adam optimizer, which is leveraged by the actor, the critic, and also for by optimization process of the α coefficient.

We first planned to train our TURTLEX robot with a sparse reward scheme, and allowing it to take an high number of steps per episodes (up to 2000). We increased the steps number because it looked like our robot did not have enough time per episode to learn properly. As that kind of training did not yield particularly interesting behaviors, we then changed our approach, and eventually switched to a dense reward scheme. During both approaches, we planned to train our robot by giving it a fixed array of 13 goal positions to reach tailored for our specific world, each of increasing difficulty: once the TURTLEX managed to achieve a single goal 10 times in a row, the goal itself was changed to the next one in the list. We designed a 5000 episodes training with 600 steps limit, and we chose Adam [26] as optimizer for both the actor and the critic. Like for the ACC experiments that have been described in section 3.3, we again carried out our training/learning and tests on a machine featuring UBUNTU 20.04.03 LTS, a pair of INTEL XEON GOLD 6234 CPUs, three NVIDIA QUADRO RTX 6000/8000 GPUs (with CUDA enabled), and 125.6 GiB of RAM.

Our initial training results with sparse reward schemes were not promising, like it can be seen in Figure 5.1. As it can be appreciated from the picture, after almost 800 episodes our TURTLEX robot was slowing down its learning process. This was due to the difficulty faced by the actor to receive positive feedback, especially when confronted with harder objectives. In particular, our robot started struggling when asked to reach goals that required wall circumnavigation. As it completely failed its objectives in that scenario, one of the solution we attempted was, as already mentioned, to give it more steps per episode, but this lead the TURTLEX to learn to stand still (and receive a null reward) rather than to try and reach a goal, while risking to obtaining negative feedback in the process. From our experiments, it is evident that the number of steps per episodes to set for

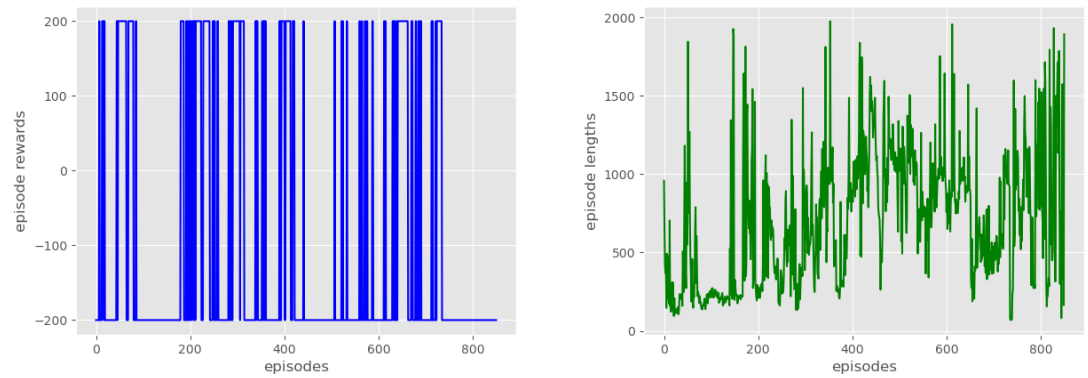

Figure 5.1: Learning episode rewards and lengths for the navigation task, with sparse rewards and incremental goals

the task must be close to the expected number of steps it would take a "perfect" agent to reach said goal. For our scenario, we computed a value of 600. Before only sticking to this rule though, we also tried to implement a dense reward scheme in which standing still for a fixed amount of consecutive steps would trigger higher and higher negative rewards. The result is shown in Figure 5.2. With the dense reward scheme, our robot started

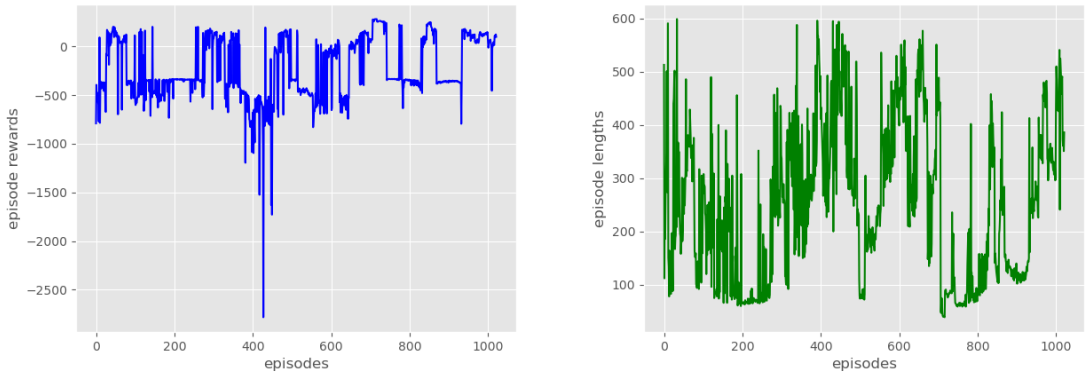

Figure 5.2: Learning episode rewards and lengths for the navigation task, with dense rewards, incremental goals, and a routine to prevent prolonged stillness

triggering our anti-stillness routine around episode 400: rewards with modulo greater than 1000 were not achievable otherwise. We soon proved that, for our navigation task, a well-designed dense reward scheme was more than enough in order to let our robot learn correct behaviors (e.g., to avoid standing still for several steps). Figure 5.3 reports our final training session, which relies uniquely on the dense reward scheme detailed in Algorithm 3. After less than 1500 episodes, our robot managed to reach two times all the 13 handcrafted goals, by accomplishing each objective 10

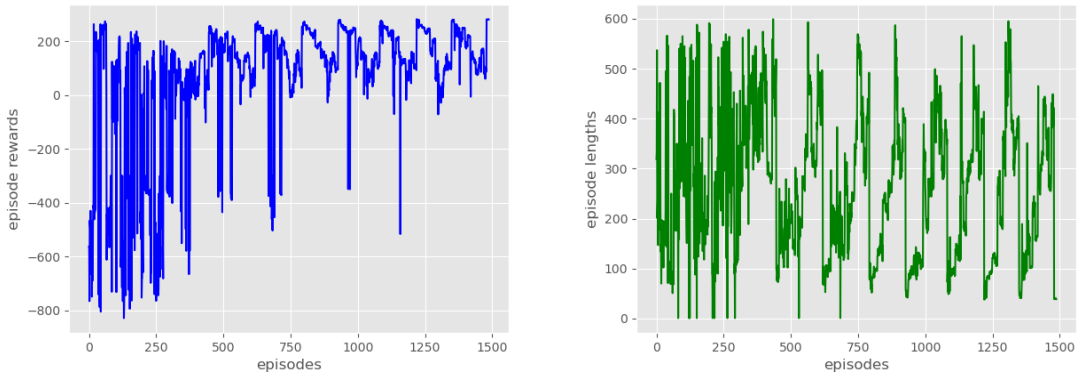

Figure 5.3: Learning episode rewards and lengths for the navigation task, with dense rewards and incremental goals for our final training session

times in a row. By leveraging a dense reward scheme, and by fine-tuning the SAC learning parameters, we managed to obtain an actor network able to reliably reach different goals while also requiring a restrained amount of steps per episode. Indeed, the number of steps taken by the second set of attempts to reach the 13 objectives was significantly reduced: this serves as an important hint about the policy being able to fit this navigation task with more and more confidence over time.

5.3 Manipulation

Relative to the manipulation learned skill, we wanted to have a Reinforcement Learning algorithm capable of providing actions in continuous state and action spaces, like for the navigation skill. This is one of the major reasons that led us to choosing the Deep Deterministic Policy Gradient (DDPG) algorithm [29]. As already mentioned in section 4.1, the robotic arm of our case study features 5 revolute joints. Thus, this time our learned skill involves 5 separate values for the actions (i.e., the actor's outputs) and 9 state/observation values (i.e., 9 actor inputs). The elements characterizing the current state are:

1. Current position of the first joint $[-2.617\ rad;\ 2.617\ rad]$
2. Current position of the second joint $[-1.571\ rad;\ 1.571\ rad]$
3. Current position of the third joint $[-1.571\ rad;\ 1.571\ rad]$
4. Current position of the fourth joint $[-1.745\ rad;\ 1.745\ rad]$
5. Current position of the fifth joint $[-2.617\ rad;\ 2.617\ rad]$

6. End effector's goal position x coordinate [$-0.27\,m;\ 0.36\,m$]
7. End effector's goal position y coordinate [$-0.36\,m;\ 0.36\,m$]
8. End effector's goal position z coordinate [$0.08\,m;\ 0.63\,m$]
9. Current distance of the end effector from the goal position [$0\,m;\ 1\,m$]

where the x, y, z end effector coordinates are relative to the robot's base frame. The actor outputs the proposed configurations of all the 5 revolute joints, whose allowed ranges are of course the same as the ones listed above for the input vector. The initial joints configuration is [$0,\ -1,\ 1,\ 1.2,\ 0$], which corresponds to rest_arm, i.e., one of our four named arm/end effector configurations, the other being picking_arm, open_gripper and closed_gripper. See section 4.4 for a detailed description about them. Our DDPG algorithm requires the action space to be symmetrical and features Prioritized Experience Replay (PER) [23, 22]. Like for the navigation learned task (see section 5.2), one of our main interests is to keep the actor Network's size relatively small. Again, we decided not to shrink the critic's dimension accordingly, for the same reasons expressed in the previous section. Both the actor and the critic Networks feature target Nets to aid with the training process. As already hinted, the actor Network features an input size of 9 elements, and outputs the 5 action values. The input is sent to a Fully Connected layer of size 30, followed by a Batch Normalization layer and a ReLU activation layer. Then the actor features three more layers that are identical to the ones just described. After those, the actor has a last triplet of layers: a Fully Connected of size 5 (that matches the Net's output dimension), followed by a Batch Normalization layer, and a Σ layer (see section 5.2) which is adjusted as in the navigation case (equation 5.1), so that its output values lie between [$-1;\ 1$]. This is done to mimic a $\tanh$ final activation layer, as that function is currently not fully supported by pyNeVer (section 2.6). Lastly, a final layer is applied in order to bring the output values from [$-1;\ 1$] to the wanted output ranges for the manipulation task: this is done by multiplying each output value by its corresponding action space upper bound (e.g., by 2.617 for the first input, or by 1.745 for the fourth output). Doing so works as a alternative to the denormalization procedure which we applied to the navigation learned skill case. Relative to the critic Network, which implements a Q-value approximator function, it requires 14 input values (the 9 of the state/observation values plus the 5 of the action ones), that get sent to a Fully Connected layer of size 300. Then, like for the actor case, a pair of Batch Normalization and ReLU layers follows. Next, this

triplet is again repeated: the critic then features a Fully Connected layer of size 400, followed by a Batch Normalization layer and a ReLU one. There is a last similar triplet which leads to the single output value: this time, the Affine layer has therefore size 1, and the activation layer is not a ReLU, but it is a Linear one. As the name implies, Linear activation functions do not introduce any non-linearity; in particular, in our case the output of the Linear layer corresponds exactly to the neuron weight's value. Indeed, when this layer type is leveraged for the output layers, it is also referred as *Identity* or "No activation" layer.

For this learned skill we again chose to stick with a dense reward scheme, like we opted for the navigation learned skill case. If the current episode is not directly terminated by the step action's outcome, two different rewards can be triggered. If the end effector current location is closer to the goal with respect to the previous one, the Network receives 10 points. Otherwise, the reward scheme outputs a "step punishment" of -1. Conversely, in the case that the action of the agent is the cause of a premature episode termination, the robot receives 100 points if it led the end effector to reach the goal, or -100 if it attempted to reach an impossible/harmful joints configuration. Algorithm 4 illustrates the reward scheme in its entirety.

Algorithm 4 TURTLEX dense reward scheme for the manipulation learned skill

```
if episode done then
    if goal reached then
        reward = 100
    else
        reward = −100                    ▷ Impossible configuration proposed
    end if
else
    if goal distance reduced then
        reward = 10
    else
        reward = −1                                        ▷ Step punishment
    end if
end if
```

The training/learning phase is carried out by repeatedly feeding 6 predetermined goals, each of increasing difficulty, that in general tend to lead the end effector to the x, y, z location of the picking_arm configuration (see section 4.4). The algorithm continues feeding the same goal until the Network has reached at least a 90% success rate over a moving window of 30 episodes. We planned to set the training phase to 400 episodes, with

a 500 step limit, with every new episode resetting the robotic arm to the rest_arm configuration. Once again, we chose the Adam algorithm [26] as learning optimizer for both the actor Network (with a learning rate of 0.0001) and the critic one (with a learning rate of 0.001). Below is a list of the other learning parameters for the DDPG:

- $\gamma = 0.99$
- $\tau = 0.001$
- weight decay $= 0.01$
- batch size $= 64$
- replay buffer size $= 20000$

The action selection process carried out by the actor involves the addition of a noise component; we chose an Ornstein-Uhlenbeck Noise Process model.

As already mentioned in section 4.4, we leveraged MoveIt for the implementation of this learned skill. We designed this robot task so that, at each step, the Network proposes a new joints configuration, that has to get validated by MoveIt. Indeed, thanks to the configuration process we carried out for our robot, MoveIt is able to compute beforehand whether a specific joints configuration would be feasible/safe or not. If so, the arm gets guided by MoveIt to the configuration proposed by the Network; otherwise the episode is ended and the Network receives a negative reward (-100).

5.4 Test

In this section, we are going to describe the results obtained by the navigation task testing procedure. The tests have been carried out in an isolated environment, i.e., all the other skills of the Turtlex robot were disabled, and the world did not feature junk elements, as they are not handled by the navigation Network.

In order to provide a meaningful testing scenario, we relied on randomly generated valid goals, i.e., sampled from the reachable areas of the house. In order to achieve this, we divided each room of the domestic environment in 4 areas (for a total of 8 zones). The smaller room features an area for each side of the dropbox, while the bigger one follows the same principle, but does so around the wooden table. At the beginning of the testing execution,

the Task Environment code fills an array with the amount of desired test goals by picking, for each sample, a random area of the house, and then a random (x, y) coordinate pair inside the drawn zone. With this approach, not all the randomly generated goals have the same chance of being picked (due to the different sizes of each zone), but all the 8 zones have the same chance of being chosen. It is possible to trigger this random goal generation also for learning sessions. Indeed, we also ran some training experiments with this approach, but the results were almost equal to the ones achieved with our thirteen-handcrafted-goals technique (introduced in section 5.2).

With the aid of Figure 5.4, we can now report the results of our tests. During testing, the TURTLEX is fed a new goal each episode, regardless of the outcome of the previous attempt: this way, our one-hundred-episode test involved 100 different objectives to reach. Our robot completed the

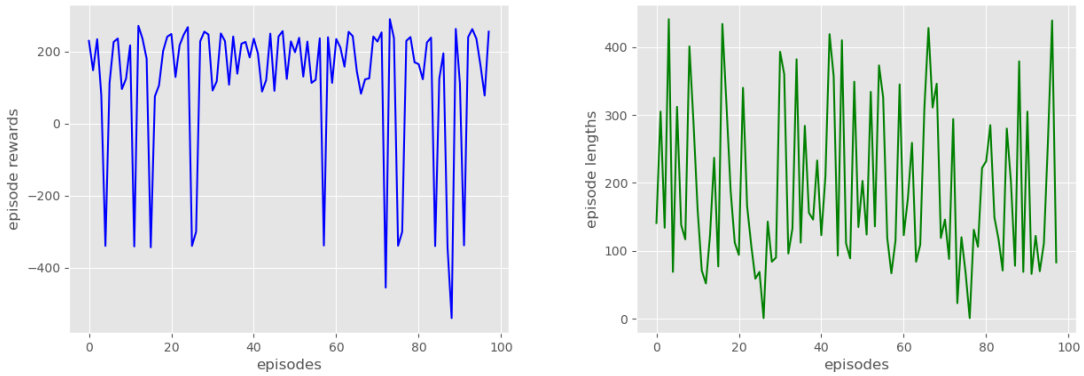

Figure 5.4: Testing episode rewards and lengths for the navigation task, with sparse rewards and 100 random goals

simulation with a success rate of 85.0%, and the test itself took a total of one hour, seventeen minutes and fifty-five seconds. We reviewed the goals that lead to an episode failure, and noticed that our robot only struggles with objectives that imply a closer circumnavigation of the dropbox. In other words, even if our robot learned not to greedily try to decrease its goal distance at all costs, it fails in doing so in this specific case. Indeed, as mentioned in section 4.1, our simulation starts with the robot in the middle of the smaller room, so if the given goal is lying close to the opposing face of the dropbox, our TURTLEX has not yet learned how to properly setup its trajectory in advance. This issue is of course relative to the actor's current behavior, and these are by no means unreachable goals per se. We also ran some specific training with these more problematic objectives and, even if the robot eventually learned how to reach them, it then still struggled to generalize that behavior, and thus failed again during subsequent tests

featuring similar targets. That said, we think that there is enough evidence to believe that further training sessions would eventually lead to an high degree of success also for this particular scenario. Apart from this minor inconvenient, our testing session featured in Figure 5.4 confirms that, after only ≈ 1500 training episodes, the robot learned to generalize its navigation skills in a convincing way: it proved to be able to reach even the farthest areas of the house, while circumnavigating walls and obstacles, often thanks to a robust degree of planning ahead. Furthermore, goals were reached by taking a very limited average amount of steps. Indeed, by our computations, and by the visual feedback of the simulation, we acknowledged that the actor Network often got close to the minimum possible amount of steps that these goals would require, relative to the experiment constraints we imposed (i.e., house topology, robot's allowed maximum velocities, etc.). The visual feedback of the graphical simulation also showed that the TURTLEX learned to head towards the goal before starting its journey to the goal; then, while travelling it repeatedly performs small variations of its heading, and we assume that it does that in order to scan greater portions of its surroundings, while wasting the least amount of time in doing so.

5.5 Integration

We prepared the integration between the learned and scripted skills by leveraging a couple of online resources, with the aim of synchronizing the execution of the pieces of code of our case study, and in order to implement its Behavior Tree in a way that could benefit from ROS capabilities. For the first objective we tweaked TASK_BEHAVIOR_ENGINE[1]. Taking advantage of its ROS implementation[2], we made it compatible with PYTHON 3, implemented an easier installation method, and adjusted other minor elements. We then implemented the Behavior Tree structure of our case study with that tool, and completed its Control nodes integration. We then outlined all the classes for the Execution nodes.

In order to provide a reliable mean of synchronization between ROS or Behavior Tree nodes, we also implemented a custom version of ROSNODE, a resource that had been originally implemented inside the ROS_COMM[3] package. This package is *a command-line tool for displaying debug information*

[1] https://github.com/ToyotaResearchInstitute/task_behavior_engine
[2] https://github.com/ToyotaResearchInstitute/task_behavior_ros
[3] https://github.com/ros/ros_comm

about ROS Nodes, including publications, subscriptions and connections. It also contains an experimental library for retrieving node information. We re-implemented the ROSNODE package alone and slimmed its structure. This piece of code allows us to start the simulation exactly when the robot model is fully loaded, and it also helped us avoiding some bugs that occur when the code relative to the Reinforcement Learning portion begins too quickly with respect to the Gazebo simulation.

Chapter 6

Verification

6.1 Introduction

The Verification Chapter introduces the topic of Network Verification for our case study, and describes the Verification properties for the navigation task, along with the results obtained.

The aim of this Chapter was to provide formal Verification guarantees for both the learned skill relative to the main case study of the Thesis. Unfortunately, due to the lack of time, we were able to only verify the navigation skill, and will leave the manipulation task Verification for the future.

6.2 Navigation Verification

This section will first detail the definition of the Verification properties we designed for the navigation learned skill, then it will show their results along with some comments about them. In order to define sensible properties, it is important to have a clear view of the context in which the Network acts. For the navigation task, we wanted to be sure that the robot does not crash into walls or other obstacles, and that its velocity outputs are not only reasonable, but also that, provided small variations (similar to noise perturbations) over the input state vector, they do not feature sudden substantial changes to their values. In order to outline a wholesome Verification procedure, we thus designed several properties that aim to provide an exhaustive picture of the analysis. We restricted our process to three parameter sets for the Verification, i.e., one for each heuristic featured by NeVer2. We picked the same *Over-approximation*, *Mixed*, and *Complete* parameter sets presented for the ACC Verification procedure,

inside subsection 3.3.4. This time though, we had to drop the Complete parameter set computation, as our Network owns too many unstable ReLU layers, each of which takes roughly 48 hours to be computed (over the same machine we leveraged for learning and testing). So, our Verification results are sound, but incomplete: this means that every verified property here reported would be also verified by the Complete verification parameter set (which yields sound and complete results), but properties that failed to verify with our parameter sets may indeed hold true, when computed without the injection of any approximation. We also dropped *Mixed2* from this analysis, as it did not feature any relevant advantage over the Mixed parameter set, at least in this context. We remind that for this task we have 14 state input values: 10 laser sensor readings, the goal heading value and the goal distance, and the values of the previous output. The outputs are composed by 2 elements: the linear and the angular velocities proposed by the actor Net. We omit input and output measurement units for the sake of brevity, as they have been already reported in section 5.2. Furthermore, we define the actor Network input vector (namely, the state/observation one) as $\underline{x} = [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}]^T$.

The first property we designed is *GlobalReach*. As the name suggests, it is a global property, meaning that it attempts to verify a property for the whole input bounds. Indeed, this property checks whether the Net in analysis, given its full input bounds, only outputs values inside the allowed output bounds. Our propriety is defined by Inequality 6.1.

$$\text{if } \underline{x} \in \begin{bmatrix} lb_1; & ub_1 \\ lb_2; & ub_2 \\ lb_3; & ub_3 \\ lb_4; & ub_4 \\ lb_5; & ub_5 \\ lb_6; & ub_6 \\ lb_7; & ub_7 \\ lb_8; & ub_8 \\ lb_9; & ub_9 \\ lb_{10}; & ub_{10} \\ lb_{11}; & ub_{11} \\ lb_{12}; & ub_{12} \\ lb_{13}; & ub_{13} \\ lb_{14}; & ub_{14} \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \underline{y} \leq \begin{bmatrix} lb_{out1} \\ -ub_{out1} \\ lb_{out2} \\ -ub_{out1} \end{bmatrix} = \emptyset \tag{6.1}$$

where $\underline{y}$ represents the output vector and corresponds to $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$. Furthermore, all the lb and ub stand for the lower and upper bounds of the inputs, while their counterparts for the outputs are represented by the two lb_{out} and ub_{out}. Their values have been again reported in section 5.2. The implication in 6.1 states that if all the components of the input lie inside their lower and upper bounds, the set encompassing the unsafe area is empty. We always declare Verification properties outlining unsafe areas; these are represented as $C\underline{y} \leq d$, where C stands for the constraints matrix/unsafe matrix, and d stands for the unsafe vector.

The next series of properties are local ones, i.e., they are centered around a predetermined input vector that leads to a certain output, and their aim is to guarantee that for small shifts in that vector, also the output values lie close to the original ones. The values for the small variations over the inputs are represented by the components of the vector $\underline{\epsilon}$, while the equivalent one for the outputs is $\underline{\delta}$. Indeed, each property presented in this section features their own $\underline{\epsilon}$ and $\underline{\delta}$, with the sole exception of GlobalReach, which features none of them. The property *Local1*, *Local2* and *Local3* are defined as reported in Inequality 6.2, which serves as a template for all the possible strictly local properties.

$$\text{if } \underline{x} \in \begin{bmatrix} t_1 - \epsilon_1; & t_1 + \epsilon_1 \\ t_2 - \epsilon_1; & t_2 + \epsilon_1 \\ t_3 - \epsilon_1; & t_3 + \epsilon_1 \\ t_4 - \epsilon_1; & t_4 + \epsilon_1 \\ t_5 - \epsilon_1; & t_5 + \epsilon_1 \\ t_6 - \epsilon_1; & t_6 + \epsilon_1 \\ t_7 - \epsilon_1; & t_7 + \epsilon_1 \\ t_8 - \epsilon_1; & t_8 + \epsilon_1 \\ t_9 - \epsilon_1; & t_9 + \epsilon_1 \\ t_{10} - \epsilon_1; & t_{10} + \epsilon_1 \\ t_{11} - \epsilon_2; & t_{11} + \epsilon_2 \\ t_{12} - \epsilon_3; & t_{12} + \epsilon_3 \\ t_{13} - \epsilon_4; & t_{13} + \epsilon_4 \\ t_{14} - \epsilon_5; & t_{14} + \epsilon_5 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \underline{y} \leq \begin{bmatrix} z_1 - \delta_1 \\ -z_1 - \delta_1 \\ z_2 - \delta_2 \\ -z_2 - \delta_2 \end{bmatrix} = \emptyset \qquad (6.2)$$

For the Local1 property, $\underline{\epsilon} = [.1,\ .2,\ .1,\ .05,\ .2]$, $\underline{\delta} = [.05,\ .2]$. The deterministic pair of output values relative to the local property is represented by $\underline{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$. The actor computes it for the specific input state vector $\underline{t} \in \mathbb{R}^{14}$,

which for this property is $\underline{t} = [.25, .25, .25, .25, .25, .25, .25, .25, .25, .25, 0, .1, .21, 0]$. The next local property, Local2, differs from the previous one for the input values it features: $\underline{t} = [3,\ 3,\ 3,\ 3,\ 3,\ 3,\ 3,\ 3,\ 3,\ 3,\ .1,\ 4,\ .12,\ .5]$. Finally, Local3 again implies yet another input state vector. This time, we fixed it to $[4.2, 4.2, 4.2, 4.2, 4.2, .25, .25, .25, 4.2, 4.2, 3, 7.1, .1, -2]$. These last two properties have the same $\underline{\epsilon}$ and $\underline{\delta}$ as Local1.

The next property, *GlobalPartial*, verifies that the network outputs a high angular velocity along with a low linear velocity, when input conditions are met. Those input conditions are: at least one of the front laser readings (in this case, the fifth out of ten) provides a low reading (i.e., the robot is close to an obstacle), while the other inputs are confined inside their lower and upper bounds. It is depicted by Inequality 6.3.

$$\text{if } \underline{x} \in \begin{bmatrix} lb_1; & ub_1 \\ lb_2; & ub_2 \\ lb_3; & ub_3 \\ lb_4; & ub_4 \\ lb_5; & lb_5 + \epsilon_1 \\ lb_6; & ub_6 \\ lb_7; & ub_7 \\ lb_8; & ub_8 \\ lb_9; & ub_9 \\ lb_{10}; & ub_{10} \\ lb_{11}; & ub_{11} \\ lb_{12}; & ub_{12} \\ lb_{13}; & ub_{13} \\ lb_{14}; & ub_{14} \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \underline{y} \leq \begin{bmatrix} ub_{out1} \\ -lb_{out1} - \delta_1 \\ ub_{out2} - \delta_2 \\ -lb_{out2} - \delta_2 \end{bmatrix} = \emptyset \tag{6.3}$$

where $\underline{\epsilon} = \epsilon_1 = [.05]$ and $\underline{\delta} = [.11,\ 1]$

The last property we defined is *SpeedThreshold*, which verifies that the actor Network decreases the output linear velocity while increasing the angular one, when input conditions are met. Those input conditions are: at least one of the front laser readings (again, we picked the fifth one) provides a low reading (i.e., the robot is close to an obstacle), the input previous linear velocity is over a certain threshold, and the input previous angular one is under a predetermined value. Inequality 6.4 reports the

description of this last property prototype.

$$\text{if } \underline{x} \in \begin{bmatrix} lb_1; & ub_1 \\ lb_2; & ub_2 \\ lb_3; & ub_3 \\ lb_4; & ub_4 \\ lb_5; & lb_5 + \epsilon_1 \\ lb_6; & ub_6 \\ lb_7; & ub_7 \\ lb_8; & ub_8 \\ lb_9; & ub_9 \\ lb_{10}; & ub_{10} \\ lb_{11}; & ub_{11} \\ lb_{12}; & ub_{12} \\ ub_{out1} - \epsilon_2; & ub_{13} \\ -\epsilon_2; & \epsilon_3 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \underline{y} \leq \begin{bmatrix} ub_{out1} - \delta_1 \\ lb_{out1} \\ \delta_2 \\ \delta_2 \end{bmatrix} = \emptyset \quad (6.4)$$

where $\underline{\epsilon} = [.05,\ .02,\ .2]$ and $\underline{\delta} = [.02,\ .2]$.

Now that we presented all the properties we designed, we can describe Table 6.1, which reports our Verification results for this specific learned task. As we already mentioned, these results lack the Complete parameter set results, as we discovered that they would require too much time to compute. That said, we already proved that we do not need that data for properties that are declared safe by other sets. Given the results we obtained, this is particularly convenient for us: we managed to verify all of the properties with both the remaining heuristics. We are then satisfied with the safeness outcomes, and we also acknowledge that, at least with Nets of similar sizes, the computing time differences between our Mixed approach and the Over-approximation one are encouragingly marginal. That said, the employment of CUDAs to carry out this task also played a part in flattening their time performance gap.

We can thus conclude that our Verification results are overwhelmingly positive and promising for the future. Anyway, we still feel the need of expanding the analysis also on this task, as our property prototypes are not generic enough to provide an unconditioned safety guarantee for this Network. Furthermore, future work could focus on also stressing the properties we designed, in order to find the maximum values for $\underline{\epsilon}$ and $\underline{\delta}$ that still allow the properties to hold true.

Table 6.1: Verification results for the TURTLEX navigation task. Dashes indicate missing entries. Elapsed times [s] have been rounded to the third decimal place

Turtlex navigation task verification			
Property	**Param. Set**	**Result**	**Time elapsed**
GlobalReach	Over-Approx	True	7.8
	Mixed	True	7.975
	Complete	—	—
Local1	Over-Approx	True	7.223
	Mixed	True	7.379
	Complete	—	—
Local2	Over-Approx	True	7.198
	Mixed	True	7.120
	Complete	—	—
Local3	Over-Approx	True	7.086
	Mixed	True	7.155
	Complete	—	—
GlobalPartial	Over-Approx	True	7.64
	Mixed	True	8.206
	Complete	—	—
SpeedThreshold	Over-Approx	True	7.453
	Mixed	True	8.059
	Complete	—	—

Chapter 7

Conclusions and Future Work

The last chapter draws the conclusions of our work (section 7.1) and proposes the next steps to carry it on in the next future (7.2).

7.1 Conclusions

In this Work we tackled the challenge of linking the world of Robotics simulation with the one of Deep Learning. Thanks to our extensive tests, we managed to provide the scientific community with a sound Network for the navigation task, along with a modular structure that easily allows users to adapt the various modules of our environment to their needs. We found out that the dense reward scheme better fit our case study, and after several trials we managed to obtain a solid actor Network that is able to reach goal locations in a very limited amount of steps/time. Then, we proceeded with the Verification process and achieved convincing results, which contribute to prove both the usefulness of Verification approaches and the safeness of our Network.

Additionally, we worked towards the release of NeVer2, with the aim of closing a gap that, as of today, is still open in the scientific community: the availability of a simple-to-use tool that allows even non-experts to train/verify/repair their networks in an automated way, and through an user-friendly GUI. We developed our tool and tested it via an educational case study revolving around the Adaptive Cruise Control technology. In particular, NeVer2 is fully capable of handling ONNX and PyTorch Networks, and allows the user to save/load/graphically edit them at will. NeVer2's GUI now integrates Neural Network training, testing and Verification, by providing interactive windows that let users choose the desired parameters for each capability.

7.2 Future Work

This Thesis proposes several paths to continue the work on the 2 main projects it dealt with, namely NeVer2 (along with CoCoNet and pyNeVer) and the Turtlex case study. NeVer2 is currently lacking a complete integration with TensorFlow, which would allow our tool to be able to autonomously handle most of the Neural Network formats currently in use in the scientific community. Another important feat would be to finish the GUI integration of all the currently unsupported features of pyNeVer, such as Network pruning/slimming, in order to even out the discrepancies between the back end and front end of the tool. Once that is done, a subsequent step may be the outlining of an automated repairing algorithm able to fix unsafe Networks, so that they become safe to use, while still retaining their performance and core behaviors. A solid starting trace is surely provided by [15]. Other than this, another important aspect to work on would be the increase of Net layer types that pyNeVer (and, by extension, NeVer2) can support, or to try to export its Verification capabilities outside of the FFNN set, while sticking to VNNLIB guidelines. Furthermore, NeVer2 computing speed could immensely benefit from the implementation of its Abstraction techniques over NVIDIA's CUDAs (as well as over their competitor hardware architectures) by wrapping the necessary primitives. Finally, researchers could leverage the core purpose of CoCoNet, and thus develop their own implementations of Neural Network operations, and then interface them with CoCoNet's GUI, like we have done with pyNeVer.

Relative to the Turtlex case study, one of the most important contributions would be to carry out the learning and testing for the manipulation learned skill, along with the designing and computation of some fitting Verification properties. Furthermore, the creation of more generic properties for the navigation task would provide a definitive answer over its Network's safety. Once that gets done, the next step would be to complete the Behavior Tree integration, by leveraging the software resources provided with this Work. Next, the manipulation Network could be expanded in order to also handle the gripper. Relative to the scripted skills, the junk detection pattern could be implemented by leveraging a third Neural Network, instead of resorting to canonical Computer Vision solutions. Ultimately, our simulated model could be implemented on a real robot; our code has been designed with this objective in mind, and all the physical counterparts of the components of our Turtlex are available online and are compatible

with ROS Noetic.

During the making of this Thesis, we also started working on a Docker[1] image that would allow anyone to run our Turtlex case study, in order to check our results and to easily create new experiments in a closed environment, which already features a connection between ROS/ Gazebo and the Deep Learning world. This side project has then been discarded for the moment, but it would match well with the rationale behind NeVer2. With that additional resource, users could run their own tests on the Docker images, export their Networks to NeVer2, and eventually Verify them with our tool, without ever needing to directly interact with the underlying code.

[1] https://www.docker.com

Bibliography

[1] B. K. Bose. Neural network applications in power electronics and motor drives—an introduction and perspective. *IEEE Transactions on Industrial Electronics*, 54(1):14–33, 2007. 6

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016. 25

[3] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. J. Goodfellow, A. Madry, and A. Kurakin. On evaluating adversarial robustness. *CoRR*, abs/1902.06705, 2019. URL `http://arxiv.org/abs/1902.06705`. 14, 17

[4] S. Cavalieri, P. Maccarrone, and R. Pinto. Parametric vs. neural network models for the estimation of production costs: A case study in the automotive industry. *International Journal of Production Economics*, 91(2):165–177, 2004. 6

[5] S.-L. Chen and Y.-M. Wei. Least-squares sarsa (lambda) algorithms for reinforcement learning. In *2008 Fourth International Conference on Natural Computation*, volume 2, pages 632–636. IEEE, 2008. 48

[6] G. Cicala, A. Tacchella, F. Leofante, and D. Guidotti. The VNNLIB standard for benchmarks, 2019. URL `http://www.vnnlib.org/standard`. 9

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000. 17

[8] M. Colledanchise and P. Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018. 19

[9] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989. 10

[10] J. C. de Jesus, V. A. Kich, A. H. Kolling, R. B. Grando, M. A. d. S. L. Cuadros, and D. F. T. Gamarra. Soft actor-critic for navigation of mobile robots. *Journal of Intelligent & Robotic Systems*, 102(2):1–11, 2021. 48

[11] J. S. R. Dow. Neural net pruning-why and how. In *Proc. of International Conference on Neural Networks*, pages 325–333, 1988. 2, 12

[12] Y. Y. Elboher, J. Gottschlich, and G. Katz. An abstraction-based framework for neural network verification. In *International Conference on Computer Aided Verification*, pages 43–65. Springer, 2020. 2, 14

[13] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2018. 2, 15, 16, 17

[14] R. Ghzouli, T. Berger, E. B. Johnsen, S. Dragule, and A. Wąsowski. Behavior trees in action: a study of robotics applications. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, pages 196–209, 2020. 19

[15] D. Guidotti, F. Leofante, C. Castellini, and A. Tacchella. Repairing learned controllers with convex optimization: a case study. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 364–373. Springer, 2019. 72

[16] D. Guidotti, F. Leofante, L. Pulina, and A. Tacchella. Verification of neural networks: Enhancing scalability through pruning. *arXiv preprint arXiv:2003.07636*, 2020. 2, 12

[17] D. Guidotti, L. Pulina, and A. Tacchella. Never 2.0: Learning, verification and repair of deep neural networks. *arXiv preprint arXiv:2011.09933*, 2020. 2, 12, 14, 17, 18, 28

[18] D. Guidotti, L. Pulina, and A. Tacchella. pynever: A framework for learning and verification of neural networks. In Z. Hou and V. Ganesh, editors, *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings*, volume 12971 of *Lecture Notes in Computer Science*, pages 357–363. Springer, 2021. doi:

10.1007/978-3-030-88885-5_23. URL https://doi.org/10.1007/978-3-030-88885-5_23. 2, 14, 18

[19] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018. 22, 23, 48, 53

[20] R. Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992. 10

[21] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29 (6):82–97, 2012. 6

[22] Y. Hou and Y. Zhang. Improving ddpg via prioritized experience replay. *no. May*, 2019. 24, 59

[23] Y. Hou, L. Liu, Q. Wei, X. Xu, and C. Chen. A novel ddpg method with prioritized experience replay. In *2017 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 316–321. IEEE, 2017. 24, 59

[24] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, and X. Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review*, 37:100270, 2020. 2, 13, 17

[25] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017. 15

[26] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 32, 56, 61

[27] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990. 2, 12

[28] F. Leofante, N. Narodytska, L. Pulina, and A. Tacchella. Automated verification of neural networks: Advances, challenges and perspectives. *arXiv preprint arXiv:1805.09938*, 2018. 14

[29] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015. 23, 24, 58

[30] M. Mirman, T. Gehr, and M. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pages 3578–3586. PMLR, 2018. 2, 14, 17

[31] F. N. Ogwueleka, S. Misra, R. Colomo-Palacios, and L. Fernandez. Neural network and classification approach in identifying customer behavior in the banking sector: A case study of an international bank. *Human factors and ergonomics in manufacturing & service industries*, 25(1):28–42, 2015. 6

[32] L. Pulina and A. Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *International Conference on Computer Aided Verification*, pages 243–257. Springer, 2010. 2, 6, 13, 14

[33] L. Pulina and A. Tacchella. Never: a tool for artificial neural networks verification. *Annals of Mathematics and Artificial Intelligence*, 62(3): 403–425, 2011. 2, 6, 18, 28

[34] L. Pulina and A. Tacchella. Challenging smt solvers to verify neural networks. *Ai Communications*, 25(2):117–135, 2012. 14

[35] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. 6

[36] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019. 2, 15, 16, 17

[37] Y.-C. Tang. An approach to budget allocation for an aerospace company—fuzzy analytic hierarchy process and artificial neural network. *Neurocomputing*, 72(16-18):3477–3489, 2009. 6

[38] H.-D. Tran, D. M. Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson. Star-based reachability analysis of deep neural

networks. In *International Symposium on Formal Methods*, pages 670–686. Springer, 2019. 2, 15, 16, 17

[39] H.-D. Tran, X. Yang, D. M. Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson. Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *International Conference on Computer Aided Verification*, pages 3–17. Springer, 2020. 17

[40] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3): 279–292, 1992. 48

[41] L. Weng. Policy gradient algorithms, 4 2018. URL `https://lilianweng.github.io/posts/2018-04-08-policy-gradient`. 24

[42] E. Wong and Z. Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*, pages 5286–5295. PMLR, 2018. 14, 17

[43] A. Zell. *Simulation neuronaler netze*, volume 1, number 5.3. Addison-Wesley Bonn, 1994. 8