University of Trieste

Department of Mathematics and Geoscience

Master Degree in
Data Science and Scientific Computing

Adversarial Learning of Robust and Safe Controllers for Cyber-Physical Systems

Author:
Francesco Franchina

Supervisor:
Luca Bortolussi

Cosupervisors:
Francesca Cairoli
Ginevra Carbone

Academic Year 2018/2019 - April 2020

Alle persone che tengo sempre vicino al cuore
che, anche nelle tempeste, riescono a donarmi tranquillità.
A tutti coloro che hanno caratterizzato questa mia esperienza triestina.
Grazie!

Contents

Abstract **5**

Abstract in italiano **6**

1 Introduction **7**
1.1 Context 7
1.2 Problem statement 8
1.3 Contributions 9
1.4 Structure 9

2 Background **10**
2.1 Cyber-Physical Systems 10
2.1.1 Definition 11
2.1.2 Physical component 12
2.1.3 Cyber component 14
2.1.4 Hybrid model 14
2.1.5 Modelling and Design 15
2.1.6 Safety and monitoring 17
2.2 Generative Adversarial Networks 18
2.2.1 Neural Networks 19
2.2.2 Zero-sum games 23
2.2.3 Description of a GAN 25
2.2.4 Usage of GANs 26
2.3 Signal Temporal Logic 28
2.3.1 Syntax 29

2.3.2 Boolean semantics 30
2.3.3 Quantitative semantics 31

3 Method **34**
3.1 Description 34
3.1.1 CPS architecture 35
3.1.2 Controller-Environment architecture 37
3.2 Training 42
3.3 Testing 45

4 Case study **47**
4.1 Cruise control 47
4.1.1 Model 48
4.2 Car platooning 50
4.2.1 Model 50

5 Experimental results **53**
5.1 Implementation details 53
5.2 Cruise control 54
5.2.1 Experimental settings 55
5.2.2 Metrics and results 56
5.3 Car platooning 59
5.3.1 Experimental settings 59
5.3.2 Metrics and results 61

6 Conclusions and future works **68**
6.1 Summary 68
6.2 Conclusions 68
6.3 Future work 69

References **69**

Abstract

Data-driven models are bringing huge advantages in terms of performance and effectiveness in many sectors. In the field of Cyber-Physical Systems, although, they are not widespread since their opaqueness and lacks formal proofs pose a serious threat to their employment.

We propose and analyze a novel approach in creating robust and safe controllers drawing from the literature of the adversarial learning.

Our method trains in an adversarial way two Neural Networks to reach a twofold goal: obtain one network that is able to generate difficult configurations of the environment and another that is able to overcome them in a safe e robust way. The aim is to create a formally verified controller and, at the same time, to give insights on the most demanding corner cases of a given model.

The approach is promising and worthy of further investigation.

Abstract in italiano

I modelli data-driven stanno permettendo di raggiungere efficacia e performance mai viste prima in molti settori. Tuttavia, nel campo dei Cyber-Physical System non sono ancora molto impiegati dal momento che, per via del loro funzionamento difficilmente formalizzabile, non permettono di fare le adeguate verifiche di sicurezza che li renderebbero delle valide alternative ai sistemi attualmente in uso.

Noi proponiamo e analizziamo un nuovo approccio alla creazione di controllori robusti e sicuri attingendo dalla letteratura dell'apprendimento avversario.

Il nostro sistema addestra due Reti Neurali avversarie per ottenere un duplice obbiettivo: ottenere una rete in grado di generare configurazioni insidiose dell'ambiente e un'altra rete in grado di affrontare tali situazioni in modo robusto e sicuro. Lo scopo è quello di ottenere un controllore verficato formalmente e, allo stesso tempo, ottenere informazioni su quali sono i casi più difficili da gestire per un dato modello.

Il nostro approccio è promettente e sicuramente degno di ulteriori approfondimenti.

Chapter 1

Introduction

This work is highly interdisciplinary and aims at using advanced frameworks like Generative Adversarial Networks and formal methods for model verification to propose a new approach on how to learn safe and robust controllers for Cyber-Physical Systems.

The challenges for creating truly autonomous and safe system are currently being actively investigated and we believe that our method could give some concrete contribution to the already flourishing literature on this topic.

1.1 Context

Every day we unconsciously use of an increasing number of automatic systems that have not been programmed explicitly. They have been virtually trained on human examples in order to learn the proper behaviour in the most common scenarios. The massive usage of Deep Neural Networks to solve many of today's challenges [1] is a striking example of how effective this approach can be. The adoption of such technologies is growing in both industrial and consumer sectors and we are increasingly relying on data-driven models that work very well but that, due to their complexity, we fail to understand in their whole [2].

This practice brought numerous advantages like speed and precision in the most common cases but, once we test the models in some unusual environment or with some weird configurations, they can fail miserably. While in some field of application this can be quantified as an economic loss, in others the situation is much more delicate, especially in those where human health is involved.

Due to such practical reasons, the development of advanced models is confined to controlled environments and, usually, it is not a viable option in those that would benefit from automation, like the big complex artificial systems on which we daily rely on: transportation, power grid, production and so on.

The biggest challenge for the deployment of autonomous systems that can have a larger impact in everyday life, is posed by the complexity of the exploration of **open worlds** like ours. *Open worlds*, in fact, are difficult to model and control due to the significant amount of stochastic variables that are needed in their modelling and the variety of unpredictable scenarios that they present. Therefore, while trying to ensure safety and robustness, we need to be cautious about not trading them with the models' effectiveness.

1.2 Problem statement

When it comes to the control of Cyber-Physical Systems, the industry tends to rely on well-established and well-understood techniques, belonging to classic Control Theory [3]. These algorithms allow a high degree of predictability along with good robustness in presence of noise.

Nevertheless, research is trying to fill the gap with Deep Learning [4] that, on one hand, guarantees more flexibility and resilience but on the other sacrifices the full comprehension of the whole model. The big problem posed by a partial comprehension of the model is the impossibility of checking the

behaviour of the controller in the case of unpredictable scenarios that could lead to unintended or harmful behaviours.

Finding an efficient, safe and comprehensible solution is still an open challenge that offers a large space for improvements and fertile ground for new ideas.

1.3 Contributions

Most of the research that combines the usage of Deep Learning for robust control is done in the field of Reinforcement Learning [5]. It offers different approaches, but the most promising trend involves learning from an *expert* how to face the most common scenarios [6]. This approach performs reasonably well but presents some limits in case of unexpected situations.

What we are going to investigate is the possibility of autonomously learning a safe and robust controller in a *open world scenario*. The proposed approach consists in the training of two Neural Networks and is inspired by Generative Adversarial Networks [7]. The two networks have opposite roles: while the *attacker* tries to generate troubling scenarios for the controller, the *defender* learns how to face them without violating some safety constraints.

The outcome of such training procedure is twofold: on one side we get a robust controller, on the other we get a generator of adverse tests.

1.4 Structure

After covering the background knowledge required to understand the whole architecture, we will show the proposed solution. Later, we will show the case studies on which we tested our model and the respective results. Finally, we will draw some conclusions about this work and its possible future development.

Chapter 2

Background

This project is quite broad and combines knowledge coming from different fields. Here we cover the theoretical background required to understand it.

2.1 Cyber-Physical Systems

The field of CPS, Cyber-Physical Systems, is relatively new: the name was coined and started gaining ground only in the first decade of the 2000s. The knowledge employed in this field, however, comes from well-known disciplines: CPS is, in fact, a highly interdisciplinary area where Physics, Control Theory, Embedded Systems, Formal Verification and Robotics.

Despite the recent foundation of this field, the first medical devices that can be classified as CPS are quite dated: pace-makers and insulin-pumps, for instance, were invented around the '60s. Due to the increasing computational power of the controllers classified as Edge Computing [8] and to the new possibilities given by the advent of the Internet of Things, we are going to hear about CPS new fields of application: they are already effectively used in growing fields like automotive, manufacturing, energy management and in any other field that involves the employment of robots [9].

Each of this fields deals with *Complex Systems* in which **continuous**,

discrete and **stochastic** processes are involved.

2.1.1 Definition

As the name suggests, a Cyber-Physical System is composed of two parts:

- the "**cyber**" one is a computational system that processes logical operations,
- the "**physical**" one is the real phenomenon that we are modelling.

We define it CPS only when the two components are deeply interconnected by some communication devices that can affect each other's state.

The major difference between the two components is their way of dealing with time. The *cyber* part operates in a discrete time-scale where the internal state of the model is governed by the execution of algorithms. The *physical* part, instead, lives in a continuous time-scale where the model's state evolves according to the physical laws of nature.

The communication between the two components of the CPS takes place through **transducers**, allowing the exchange of information between them. This allows the two parts to co-evolve simultaneously following their respective rules: the instructions of the algorithm for the *cyber* part and physical laws the *physical* one.

Sensors are transducers that discretize an analog signal, therefore, they allow the flow of information from the *physical* part to the *cyber* part. For instance, we call *sensor* whatever is able to digitally measure a property of the physical world: thermometers, photoresistors, camera sensors, etc.

Actuators, instead, are transducers that use a digital input to transform the physical world. They work in the exact opposite way with respect to *sensors*, in fact, they allow the information from the *cyber* part to flow toward the *physical* one. Examples of actuators are all those devices that, given

a digital input, can produce an observable change in the world: motors, speakers, electromagnets, etc.

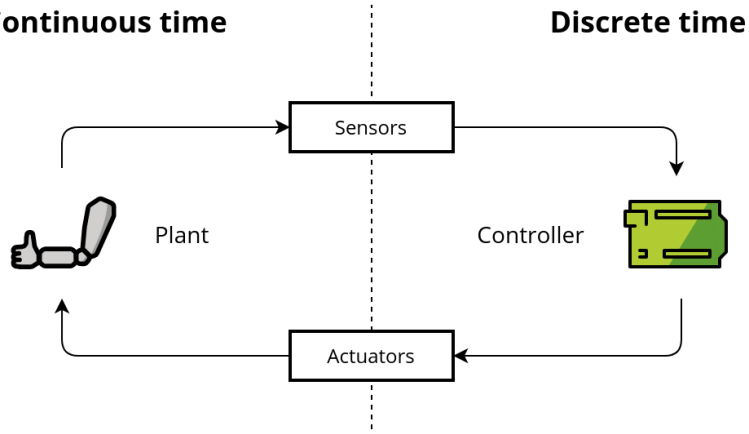

Figure 2.1: The two components of a CPS interact through sensors and actuators, these allow them to exchange information and act as a single system.

The communication between the two parts is fundamental because a CPS acts as a whole due to the constant flow of information that allows the two parts to adapt to each other. We are now going to describe a practical example of CPS: the *cruise-control*, whose aim is to keep constant the speed of a vehicle.

2.1.2 Physical component

In the cruise-control example, the vehicle and its dynamic represent the physical part of the CPS. This part is often referred to as the **plant**. We call *state* a specific configuration of the variables describing the model's properties we are interested in. For instance, in the cruise-control problem, the *state* is composed of the vehicle's speed v and the *steepness* of the road θ. The *plant* evolves its *state* in time according to the physical laws involved.

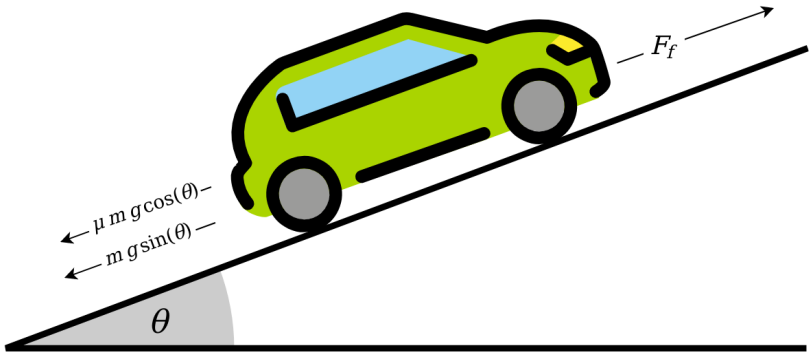

Figure 2.2: The physical forces involved in the cruise-control problem.

In this case we can describe the evolution of the vehicle's *state* through a differential equation that captures the continuous time-scale of the behaviour:

$$m\frac{dv}{dt} = \frac{F_f}{m} - mg\sin(\theta) - \mu mg\cos(\theta)$$

In the equation above, F_f is variable that we can control: it is the force that pushes the vehicle forward, directly proportional to the amount of fuel provided to the engine. Apart from the inclination of the road θ, all the other symbols that appear in the equation are constants: the vehicle's mass m, the gravity g and the friction constant μ. The equation shows the link between the variable that we can control and the rest, the *environment*. By knowing the properties of the *environment*, therefore, it is possible to compute deterministically the value of the input F_f for which the speed of the vehicle v remains constant.

The physical model of a system should be able to describe the relationship between all the variables that are involved in its dynamics. The more complex a system is, the more complex are its equations.

2.1.3 Cyber component

The *cyber* part of the CPS is referred to as the **controller**. Its role is to plan how the system should behave in order to achieve its goal by executing a pre-programmed algorithm. In our example of cruise-control, the aim of the *controller* is to keep the velocity constant.

The *controller* is – usually, but not always [10] – an embedded computer that executes a sequence of programmed operations in a discrete time-scale. The way of modelling the algorithmic part draws directly from the literature of embedded systems [11], where the employed formalisms – like *Finite State Machines* (FSM) or other kind of more complex finite-state automata – have been refined over the years.

This kind of formalism describes how the *cyber* part evolves in time through the discrete *states*. The *state* changes **deterministically** when a certain internal (a timer, for instance) or external (a value form a sensor) condition is met. Moreover, it can change in a **stochastic** way if the transition happens randomly according to a specific probability distribution.

Sensors and *actuators* play a crucial role for the *cyber* part since, through them, it can get information about the *physical* part's state and compute how to control the actuators to reach – or get closer to – the desired state. For instance, in the cruise-control problem, the *controller* receives information about the current state of the physical system through the *speedometer* and, according to its algorithm, manages how much fuel to provide to the engine to keep the speed constant.

2.1.4 Hybrid model

Due to the twofold nature of the CPSs, they can be represented as **Hybrid models**. The name *hybrid* comes from the fact that we include in the same model both continuous and discrete systems. In the example of the cruise-

control, in fact, we have the continuous state's variable of the velocity that determines whether the controller should or should not (Boolean variable), provide more fuel to the engine.

Hybrid models can be formalized using **hybrid automata**. In a **hybrid automaton**, like in the FSM, each *state* can change deterministically or randomly. Such formalism extends the FSMs by allowing conditions on the continuous variables observed from the CPS' physical part. In this way, it is possible to model even a fairly complex interaction scheme in a compact way.

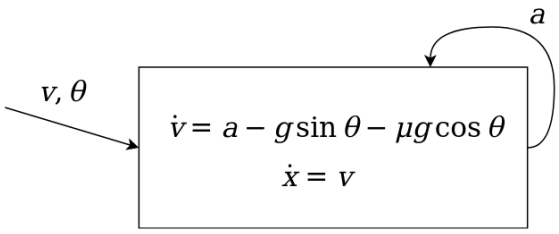

Figure 2.3: This hybrid model handles the problem of the cruise control. It has only one state and takes as input the vehicle's velocity v and the road's steepness θ. It gives as output the acceleration a that updates the state of the system. The other terms that appear are constants: μ is the friction constant and g is the gravity.

2.1.5 Modelling and Design

The modelling of the continuous part consists in obtaining the equations that appropriately describe the behaviour of the *plant*. With the aim of deriving the equations involved in the problem, we need to consider the relevant dynamics that take place into our setting. Once we obtain the differential equations that describe the system, we can make some considerations on its controllability and, in particular, on the way the control should occur.

The modelling of the discrete part, instead, aims at identifying the possible discrete *controller*'s states. This procedure is fundamental to obtain the FSM that can completely represent the *controller*. The implementation

of the FSM as an algorithm is often not-trivial due to the many machine's limitations: memory, compute power, precision and execution's speed are all finite resources.

In particular, the last constraint has become an important bottleneck that is preventing CPS from scaling massively [12]. When designing a controller, in fact, it's very important to take into account the latency that will occur between the moment in which the sensors will get information about the state and the moment in which the actuators can react accordingly, after that the *controller* has determined the strategy to follow. In *time-critical* systems this latency should be kept as low as possible in order to keep the system into a safe state and deliver the response action in time. Traditional methods are still able to stand out with respect to the newer counterparts due to the fact that the most sophisticated control techniques require complex algorithms that, sometimes, cannot run fast enough to make the system a *real-time* one [3].

A fundamental part of the design process consists in paying attention to the kind of feedback we expect to receive from the system once that the planned action has taken place through the actuators. There are two possible choices: designing an *open-loop system* or a *closed-loop system*.

In an **open-loop** system there is lack of feedback: there is no way for the controller to check whether a given action actually took place or not. Of course the limits of this approach are evident but they can work reasonably good in certain scenarios. Examples of **open-loop** systems are many appliances that, once that they have been instructed on what to do, they execute the program and terminate. On the other hand, the advantages are lower complexity and fewer resources needed.

A **closed-loop** system, instead, is characterized by the feedback's presence right **after** the action has occurred. The purpose of the feedback is to check whether the action actually brought the system into the desired state

or if it still needs more operations in order to reach it. These kind of systems are *adaptive* and more reliable respect to the *open-loop* ones since they can detect unexpected behaviours and handle them adequately. By contrast, these kind of systems are more expensive since they require specific sensors and control algorithms in order to provide the feedback.

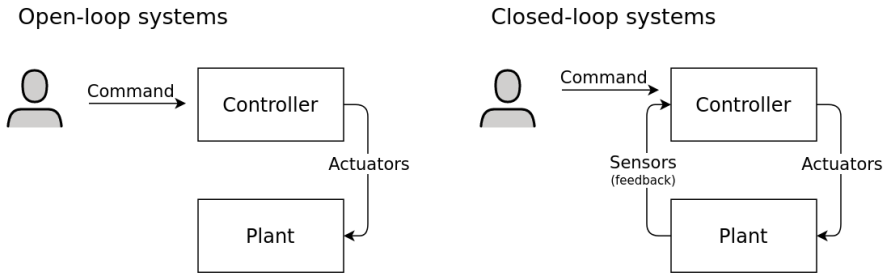

Figure 2.4: Illustration of the difference between open-loop systems and closed-loop systems. The term closed-loop refers precisely to the circular communication that is present in the system.

Once we have modelled the whole CPS, it is possible to build software simulators of both the continuous and discrete dynamics in order to test the performance and the capabilities of the system.

2.1.6 Safety and monitoring

The interest in CPS is steadily growing, and as such the research landscape of CPS is complex and rich [13]. One theme receiving a lot of attention is that of CPS *safety* and *security*. While **security** is about preventing malicious attacks of the CPS [14], **safety**, in particular, concerns the behaviour of CPS and their interaction in an open world, which should be safe and not lead the CPS device or other interacting agents – especially humans – to accidents. Safety can be framed in a formal setting by requiring a CPS to satisfy a given set of formal requirements, describing in a mathematically sound way properties like "the autonomous vehicle never bumps into a human" or "the

vehicles's speed should always remain close to $50km/h$", in the case of the cruise-control.

A popular language to specify safety properties is Signal Temporal Logic [15], a *temporal logic* tailored to deal with continuous quantities that we are going to cover more in detail in the last section of this chapter. Given a STL property, in principle, we would like to formally verify that it holds for the CPS under consideration, or at least for its mathematical model. This means having a formal proof that each trajectory of the model is safe. Such procedure is named **verification** or **model checking**. Verification of STL properties on hybrid systems, however, is often undecidable or just computationally too complex – especially in the case of *stochastic models* – therefore it is an active field of research [16].

In such cases a common approach is that of **monitoring** the system trajectories at *run-time* with STL properties and leveraging this information to assess safety in a weaker way, e.g. only with statistical guarantees.

Another technique to assess the safety of a CPS is **falsification**. This technique applies *monitoring* to simulations specifically conceived in order to drive the system to an unsafe state [17]. STL propositions are used to identify the dangerous corner-cases that can be used as *counterexamples* for gaining insight on how the problem occurred and to tune the model's parameters accordingly [18] or to train the model to avoid such cases [19].

2.2 Generative Adversarial Networks

A Generative Adversarial Network (GAN) is a **generative model**, a class of statistical models whose aim is to learn the probability distribution of the input data.

In particular, this family of models learns a mapping function between a given distribution P_z and the distribution learnt from the data P_{latent}. The

learning procedure is supervised and aims at reducing the difference between the generative distribution P_{latent} and the dataset distribution P_{data}. A peculiar characteristic of these models is that, once the learning procedure has been completed, it is possible to sample from them using the noise distribution P_z.

Summarizing, a generic *generative model* takes as input a noise vector $z \sim P_z$ and maps it to a sample drawn from the distribution P_{latent}, learnt from the data distribution P_{data}. The term "generative" refers to the capability of the model of **generating** new artificial samples that resemble the provided ones.

GANs, in particular, use NNs and a specific learning approach in order to learn the mapping function between the two distributions. In the following sections we are going to cover the concepts used by GANs.

2.2.1 Neural Networks

The concept behind Neural Networks (NNs) is quite old despite only in the last decade the related literature is flourishing [20]. This is happening because of the availability of *big data* and the increased computational power allow a faster and more accurate training, allowing their employment to deal with increasingly complex problems. The ancestors of the modern NNs root their history in the '50s, when scientists begun understanding and formalizing how biological neurons work [21].

Perceptron

The term *perceptron* was coined during the '50s and its way of working was already incredibly similar to how modern NNs work. Since the *perceptron* can be considered an ancestor of the NNs of today and they still have a lot in common, we'll describe how a simple *perceptron* works.

The whole *perceptron* can be represented by a function that takes a vector of inputs $\bar{x}$ and gives a binary output. For each input the perceptron has one internal parameter called **weight** denoted by $\bar{w}$ and a **bias** term b, that encodes the prior knowledge. The *perceptron* does one very simple thing: it *weights* the external input $\bar{x}$ with the internal parameters $\bar{w}$, it sums them up along with b and it applies to the result an *activation function* that determines the final output of the *perceptron*. Nowadays *non-linear activation functions* are the most used although there is a vast literature about them [22].

The activation function of the perceptron is called Heaviside – a simple step function – and is defined as follows:

$$act(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$$

The output of the single perceptron will be given by

$$act(b + (\bar{x}^T \bar{w})) = act\left(b + \sum_{i=1}^{n} x_i w_i\right)$$

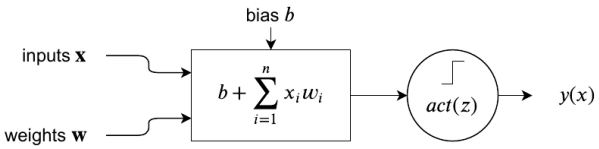

Figure 2.5: Schematic view of the perceptron.

This definition is inspired by biological neurons and their electrophysiology: they receive some inputs, combine them and then, through the equivalent of the activation function, decide whether or not to propagate the signal [21].

Deep Neural Networks

Like in the animal brains, we can achieve interesting flexibility of the model when we link multiple perceptrons together. In particular, during the years the community started following the approach of structuring the NNs in a layered fashion [23] although, in the recent years, researchers are exploring new ways of optimizing and structuring differently the NNs to gain in performance [24] [25].

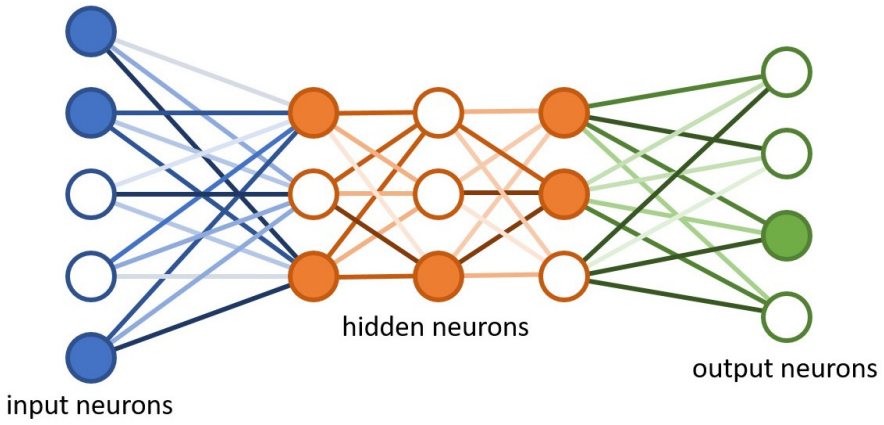

Figure 2.6: Schematic representation of a Deep Neural Network ans its layers.

What makes a neural network "deep" is actually the number of layers between the *input layer* and the *output layer* of the network. These layers are called *hidden*. In a layered NN, the outputs of the previous layer are the inputs to the next layer. With the exception of a few more sophisticated structures, NNs usually form an acyclic graph, known as *feed-forward network*.

It can be proven that a NN is a *universal approximator* of functions [26], this means that it is possible to approximate with arbitrary precision any measurable function depending on the number of neurons present in the NN. It is worth mentioning that choosing a *non-linear activation function* allows the NN to approximate even *non-linear* behaviours. This is usually a

common practice, especially when using DNN.

Training of a Neural Network

What makes the NNs so relevant is the learning procedure that is used to train them. The vast majority of the NNs, in fact, make use of *supervised learning* [27] in order to adjust the internal parameters and, therefore, learn to approximate a given function. The term "supervised" means that the model learns being exposed to many labelled examples. For instance, let D be a training set $\{(\bar{x}_1, y_1), \ldots, (\bar{x}_n, y_n)\}$ of arbitrary length n. Each element of the set is composed of a feature vector $\bar{x}_i$ and the respective label y_i. The training procedure allows a generic NN to learn the underlying structure that links the input to the output, in the case of the training set D, how to map each feature vector $\bar{x}_i$ to the respective label y_i.

As an example, let f be a NN with one layer only, a function of the input vector $\mathbf{x}$ and the matrix of its weights $\mathbf{W}$ that gives us, as output, the predicted values. The vector notation for NNs naturally extends the one of a single perceptron:

$$f(\mathbf{x}, \mathbf{W}) = act(\mathbf{W}\mathbf{x})$$

Let us take a generic training dataset composed by row vectors $\mathbf{x}_i$ of predictors and the respective vector $\mathbf{y}_i$ of the dependent variables. Since we are in the context of *supervised learning*, once the full training is completed, we would expect that our neural network has learnt how to map $\mathbf{x}$ to $\mathbf{y}$. Let us provide our NN f the training example $\mathbf{x}$, we obtain the predicted value $\hat{\mathbf{y}}$:

$$f(\mathbf{x}, \mathbf{W}) = \hat{\mathbf{y}}$$

We initialize the weights $\mathbf{W}$ of the NN randomly and, therefore, the predicted $\hat{\mathbf{y}}$ will not be very close to the desired value $\mathbf{y}$. The training procedure tunes the internal weights $\mathbf{W}$ in such a way to make the predicted

value $\hat{\mathbf{y}}$ as close as possible to the true label $\mathbf{y}$.

We introduce the concept of *loss function*, that is a measure of the difference between $\mathbf{y}$ and $\hat{\mathbf{y}}$ or, in other words, the prediction error of the NN. An common example of *loss function* is the *Mean Squared Error*:

$$\mathscr{L}(\mathbf{y}, f(\mathbf{x}, \mathbf{W})) = \mathscr{L}(\mathbf{y}, \hat{\mathbf{y}}|\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2$$

$\mathscr{L}$ depends on the weights $\mathbf{W}$ of the NN, therefore we can minimize it w.r.t. the weights $\mathbf{W}$ and find the optimal ones for the provided training examples. Minimizing such *loss function* is not easy due to the high dimensionality of the problem and the underlying high non-linearity. The standard procedure that set a turning point in the world of NNs and that helps to solve efficiently this problem is called **backpropagation** [28].

The backpropagation algorithm computes, for each unit of the NN, the derivative of the error with respect to the weights $\frac{\partial \mathscr{L}}{\partial w_{ij}}$ in order to come up with the *gradient* of the error. Once the *gradient* has been computed, we use an *optimization algorithm* that minimizes the error and updates the weights of the NN accordingly. A very common example of *optimizer* is **Stochastic Gradient Descent**, which searches for the optimal parameters in the directions where the *gradient* is lower. The stochasticity is used to add noise to the trajectory and to avoid getting stuck into a local minima [29].

The training procedure is repeated until the NN reaches the desired performances.

2.2.2 Zero-sum games

Before analyzing a GAN, we should first introduce some concepts borrowed from Game Theory useful in games with *non-cooperative* setting. The term "adversarial" comes, in fact, from the peculiar competitive interaction that

takes place inside GANs' architecture. In a non-cooperative settings with two players, for instance, a zero-sum game is a kind of game in which, whenever a player wins an specific amount the other player looses the same amount. Basically players can only steal each others' reward, this is the reason behind the expression *zero-sum*.

This concept is fundamental because it states that there's no other way of maximizing the individual gain without penalizing the other player. Otherwise, in the case of a *non-zero-sum game*, players would not be forced to to overcome the opponent anymore and it would be possible to find other kind of strategies to increase the individual gain. Instead, it is fundamental, in the context of GANs, that we keep the setting competitive.

Coming back to the *zero-sum games*, the best winning strategy for each player is to apply the **minimax** strategy. Let V be the *value function* that computes the expected reward given the choices of the players a and b, we define the maximum loss as the $\inf\{V(a, b)\}$. The objective of both the players becomes the*mini*mization of the *max*imum loss. In particular, the strategy of a player a playing against b is:

$$\min_a \max_b V(a, b)$$

If both players played according to the minimax strategy, they would end up in a stable state of their choice: the choice of each player would no longer depend on the opponent's choice since the optimal strategy would have been found and any change to this choice would not improve the payoff. This situation happens because, being a *zero-sum game*, the best strategy is to choose the action the maximizes the personal reward assuming that the opponent is playing at its best. This peculiar situation in which the dominant strategy is found by the two players, it is called *Nash equilibrium*.

As we will see, this is a core concept for a GAN.

2.2.3 Description of a GAN

GAN architecture is composed by two NNs that play against each other in a zero-sum game [7].

The typical terminology used in GANs refers to the two architecture's NNs naming them **generator** and **discriminator**.

The **generator** is a NN $G(\mathbf{z}, \theta_G)$ where θ_G indicates the network's parameters and $\mathbf{z}$ is a noise vector drawn from a given prior distribution P_z. G is a *generative model* that tries to learn the mapping between P_z and the distribution P_G learnt from the data distribution P_{data}. The output of G, given $z \sim P_z$, is a sample of the distribution P_G. This sample, coming from the estimated distribution P_G, will resemble the samples drawn from the data distribution P_{data}.

The **discriminator** is a NN $D(\mathbf{x}, \theta_D)$ where $\theta_D$indicates the network's parameters and $\mathbf{x}$ is a vector of data. D is a *classifier* that outputs the probability for the vector $\mathbf{x}$ to come from the training distribution P_{data} or from the learnt distribution P_G.

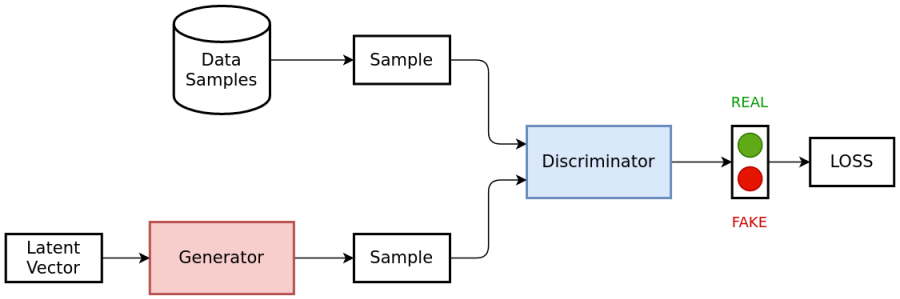

Figure 2.7: Schematic architecture of a GAN.

It should be clear by now that the two networks of the GAN have opposing roles. The aim of the *generator* is to produce data samples that resemble the true data distribution and, consequently, to fool the *discriminator* leading it to a *false-positive* classification. On the other hand, the aim of the *discriminator* that of classifying correctly as much data vectors as possible. The

two NNs are bound to this non-cooperative zero-sum game by the following **minimax** game:

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \sim p(x)}\big[\log(D(x))\big] + \mathbb{E}_{z \sim p_z(z)}\big[\log(1 - D(G(z)))\big]$$

From the formula we can observe how D tries to maximize the number of correct classifications of the original data – the part $\log(D(x))$ – and the generated data – the part $\log(1 - D(G(z)))$ – while G, on the other hand, by tuning its internal parameters θ_G, tries to minimize the number of correctly classified fake images producing more likely results. From the formula above it is possible to derive the individual *loss functions* that can be adapted to the specific problem's domain [30].

Both the NNs are trained in turns and, when one NN is learning, the parameters of the other are kept constant. Both training procedures take place as described above: starting from the *loss function*, the *gradient* is computed through *backpropagation* and the network's weights are updated by the *optimizer*.

Theoretically, after the training, the two NNs should reach the *equilibrium* so that the *generator* should be able to precisely mimic the data distribution P_{data} and the *discriminator* would guess randomly whether the presented sample is fake or not.

2.2.4 Usage of GANs

The GANs have been considered one of the most innovative idea in the field of DNNs. Nowadays, they have many applications, especially where image manipulation is involved. GANs, in fact, are particularly successful in the task of generating or modifying images. For this reason GANs are usually associated with **Convolutional** NNs, a particular architecture of NN that is effective in dealing with images [31].

Figure 2.8: The faces above are artificially generated by StyleGAN.

Some innovative applications include:

- Image super-resolution [32], to increase the resolution and details of low resolution images
- Image inpainting [33], to reconstruct masked/missing parts of one image
- Paint to image [34], from a sketch or a simple paint they can reconstruct a picture
- Data synthesis [35], used to produce synthetic samples of images similar to the ones of the training set
- Style transfer [36], used to apply to a picture the artistic style of a given drawing
- Image harmonization [37], to blend seamlessly two images or parts of them

Figure 2.9: Examples of Neural Style Transfer: a GAN trained to extract the stylistic features of paintings to apply them on pictures.

This was just a quick overview of the most promising applications of the GANs. The research area around them is very vivid and in rapid evolution. The power and the credibility of such technologies became so high that they started to rise some concerns due to malicious and immoderate usage [38].

2.3 Signal Temporal Logic

Temporal Logic (TL) is mostly known for the formal verification of models, both hardware and software. It was conceived to check a discrete signal measured over time against a logical proposition that makes use of special operators.

The term *Temporal Logic* refers to a broad family of logics used for formal verification. Each temporal logic follows precise rules on how to build the propositions using the allowed operators. The aim is to analyze and check some requirements for a given *time series*. TL relies on a basic set of *temporal operators* used to deal with time-related variables [39]. They are used in formal verification to assign a *truth value* to a given *time series* according to the properties expressed by the TL statement.

We are going to cover in detail only one kind of *Temporal Logic*, the **Signal Temporal Logic** (STL). This logic has been created to deal with continuous-time signals by discretizing them [15].

2.3.1 Syntax

The basic component of a STL statement is the **atomic predicate**, generally named μ. Let s be a generic signal over time t, an *atomic predicate* μ is a logical function of the signal, i.e. $\mu = f(s(t)) > 0$ where f is a generic function defined over the signal's state space $\mathcal{S}$, $f : \mathcal{S} \to \mathbb{R}$.

STL syntax allows to chain multiple atomic predicates by means of the **operators** to obtain a STL proposition, generically referred to as φ. The operators defined in STL are *logical* or *temporal*. *Logical operators* are the very same used in every logic: negation $\neg$, and $\wedge$, or $\vee$.

The **temporal operators** are introduced specifically in TLs and are used to check properties of the signal over time. The *temporal operators* defined in the STL syntax are: $\mathcal{G}$*lobally*, $\mathcal{F}$*inally* and $\mathcal{U}$*ntil*. The operator is referred to as *bounded* if it is defined over a finite *interval*, otherwise it is *unbounded*.

An **interval**defines the time window in which a given operator is applied. For instance, the notation $[0, H]$ denotes all the possible intervals $(t_i, \ldots, t_{i+H})$ in the time $t = (t_0, t_1, \ldots, t_n)$ over which the signal s is defined. The operator paired with a specific interval operates on the specified sliding window. Sometimes we refer to $\mathcal{I}$ as the generic interval $\mathcal{I} = [a, b]$.

The notation for specifying a *bounded operator* over the proposition φ is, for instance, $\mathcal{G}_{[a,b]}\varphi$. These operators specify how the given condition φ should be evaluated with respect to the timeframe specified by the interval $[a, b]$.

Giving an intuitive explanation of the three operators:

- $\mathcal{G}_{[a,b]}\varphi$: φ must always be true within $[a, b]$

- $\mathcal{F}_{[a,b]}\varphi$: φ must become true at least once within $[a, b]$
- $\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$: φ_2 must become true at least once within $[a, b]$ and φ_1 must be always true prior to that time

The rules of the grammar STL uses to create any formula φ can be encoded in a very compact way:

$$\varphi := \top \mid \mu \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathcal{U}_{[a,b]} \varphi_2$$

From the operator $\mathcal{U}$, it is possible to derive the other two operators, $\mathcal{F}$ and $\mathcal{G}$:

$$\mathcal{F}_{[a,b]}\varphi = \top \mathcal{U}_{[a,b]} \varphi \qquad \text{and} \qquad \mathcal{G}_{[a,b]} = \neg \mathcal{F}_{[a,b]} \neg\varphi$$

2.3.2 Boolean semantics

Given the STL syntax, we can give a specific meaning to each symbol. As we already said, the temporal logics have been created to *verify* some properties and, in case of STL, signals. The term "verification" implies the computation of a *truth value*, in fact, the most natural semantics that a STL formula can have is the **Boolean semantics**.

Let s be the signal defined over time t and let φ be a STL statement built to express a requirement on s. The *Boolean semantics* allows to *check* whether a given signal $s(t)$ satisfies or not the requirements expressed by the STL proposition φ. The satisfaction of the formula φ is expressed by the *truth value* assigned by the semantics to the specific signal $s(t)$.

It is possible to define recursively the *Boolean semantics* of STL:

$$
\begin{aligned}
(s,t) \models \mu &\iff f(s(t)) = 1 \\
(s,t) \models \neg\varphi &\iff (s,t) \not\models \varphi \\
(s,t) \models \varphi_1 \vee \varphi_2 &\iff (s,t) \models \varphi_1 \text{ or } (s,t) \models \varphi_2 \\
(s,t) \models \varphi_1 \mathcal{U}_{[a,b]} \varphi_2 &\iff \exists t' \in [t+a, t+b] \text{ s.t. } (s,t') \models \varphi_2 \\
&\qquad \text{and } \forall t'' \in [t,t'], (s,t'') \models \varphi_1
\end{aligned}
$$

The trace s satisfies φ if and only if $(s,0) \models \varphi$. Starting from the definition of the $\mathcal{U}$ntil operator, as we already mentioned, it is possible to derive the operators $\mathcal{F}$inally and $\mathcal{G}$lobally.

By adding these basic operations to the usual Boolean logic, we obtain the full power of the STL Boolean semantics. With this semantics it is already possible to monitor a signal to perform verification on it.

2.3.3 Quantitative semantics

Another interesting property of STL is the possibility of attributing different semantics to the same syntax. A **quantitative** or **robust** semantics has been formalized as well [40].

This semantics uses the same notation and operators of the *Boolean semantics*, but provides a different interpretation of the same formulas. Let us introduce a function R that given the triplet (s,t,φ), computes the *robustness* of the signal s with respect to the proposition φ. It is possible to define

recursively how the semantics works:

$$
\begin{aligned}
R(s,t,(f(s) \sim \mu)) &= \begin{cases} \mu - f(s_t) & \sim = \leq \\ f(s_t) - \mu & \sim = \geq \end{cases} \\
R(s,t,\neg\varphi,t) &= -\,R(s,t,\varphi) \\
R(s,t,\varphi_1 \vee \varphi_2) &= \max(R(s,\varphi_1,t), R(s,\varphi_2,t)) \\
R(s,t,\varphi_1 \wedge \varphi_2) &= \min(R(s,\varphi_1,t), R(s,\varphi_2,t)) \\
R(s,t,\varphi_1 \mathcal{U}_I \varphi_2) &= \max_{t' \in t+I} \big(R(s,t,\varphi_2,t'), \min_{t'' \in [t,t']} R(s,\varphi_1,t'')\big) \\
R(s,t,\mathcal{F}_I \varphi) &= \max_{t' \in t+I} R(s,\varphi,t') \\
R(s,t,\mathcal{G}_I \varphi) &= \min_{t' \in t+I} R(s,\varphi,t')
\end{aligned}
$$

The *Boolean* and *robust* semantics are tightly related: $R(s,t,\varphi) > 0 \iff (s,t) \models \varphi$ and $R(s,t,\varphi) < 0 \iff (s,t) \not\models \varphi$.

The intuitive idea behind the quantitative semantics is to measure the relative offset between the signal and the given requirement. This offset represent how much the signal can be shifted before incurring into the change of the truth value for the proposition φ and, therefore, into the violation of the constraint. The *robustness* quantify, in fact, how much variation can be tolerated on the signal. This property of the semantics makes it particularly suitable for this work. We are going to use it to measure the **robustness** of the trajectories proposed by our method.

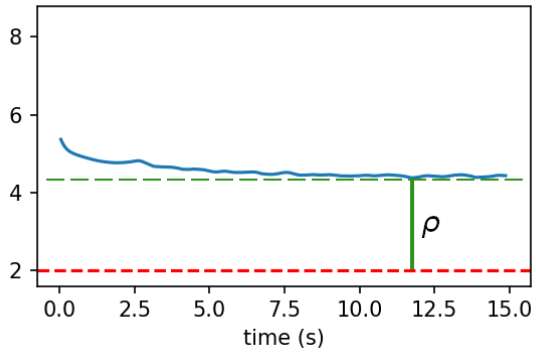

Figure 2.10: The quantitative semantics of STL visualized. Let s be the blue signal in the picture and let $\varphi : \mathcal{G}(s > 2)$ be the STL formula that we want to check on it, whose boundary is depicted in red. The robustness ρ represents the offset between the signal and the constraint φ.

Chapter 3

Method

3.1 Description

The intuition behind the proposed method is to use GAN architectural design to automatically train a controller without providing anything but STL specifications of controller's targets and, at the same time, to obtain a generative model capable of generating difficult tests. Our solution to such problem presents some similarities with classical GANs. The main difference is that our *generator* does not try to fool a classification performed by the *discriminator*. Our architecture is composed of two players that, in a turn-based game, try to overcome each other.

Each player is represented by a NN and the aim is to teach to one of the two NNs, referred to as **defender**, how to safely face the opponent NN, referred to as **attacker**. Basically, the *defender* tries to satisfy an STL requirement Φ, while the *attacker* tries to generate environmental configurations that can lead the opponent toward a violation of such requirement.

The method's achievements are twofold: on one hand we obtain a *defender* able to act safely under adverse conditions, whereas, on the other hand, we obtain an *attacker* able to gain some insights on the scenarios that are

troubling for the *defender*.

The presented method is composed of two different tightly coupled parts:

- the **CPS architecture**, used to run simulations and to estimate the robustness of trajectories generated through them,
- the **Controller-Environment architecture** which is composed of a **defender**, which acts as a *controller*, and of an **attacker**, which acts as the *environment*. Both the *controller* and the *environment* manipulate the inputs and the outputs of the *CPS architecture*.

3.1.1 CPS architecture

The *CPS architecture* is represented by a model $\mathcal{M}$ that captures the dynamics of the addressed system along with its complexity.

Each model $\mathcal{M}$ is composed of a physical component whose continuous behaviour, modelled through differential equations, describes how the whole system evolves in time. The *controller* collects data from the physical world through its *sensors* and decides the best action based on the evolution governed by the differential equations.

We can decompose $\mathcal{M}$ in two subcomponents:

- the **agent**, denoted by α, is the part of the system that can be controlled. In CPS terms we can think about it as the *plant* that, through *sensors* and *actuators*, is governed by a *controller*,
- the **environment**, denoted by β, is essentially what surrounds the *agent* and it cannot be controlled.

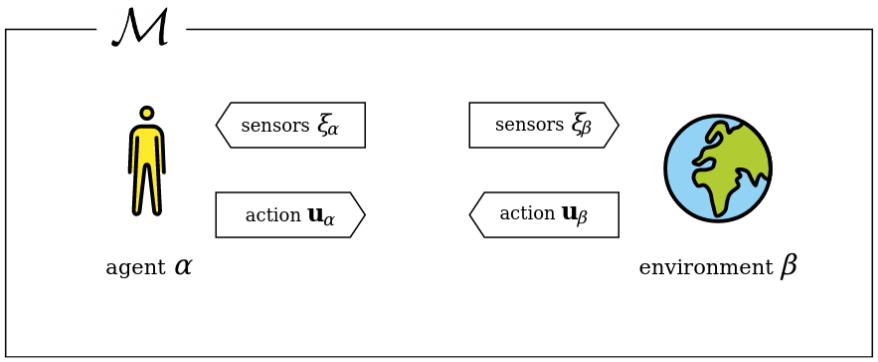

Figure 3.1: The model $\mathcal{M}$ and its components.

Let us define how this structure works.

Let $\mathcal{S}$ be the state space of dimension m, such that $\mathbf{s} \in \mathcal{S}$ is a state of the model $\mathcal{M}$ and let $\mathcal{O}$ be the space of its observable states, having dimension n. Both the *agent* and the *environment* can obtain full or partial information about the observable state $\mathbf{o} \in \mathcal{O}$ through a series of *sensors*. We denote $\mathbf{o}_\alpha \subseteq \mathbf{o}$ the observable state sensed by the *agent* and with $\mathbf{o}_\beta \subseteq \mathbf{o}$ the observable state sensed by the *environment*. Let ξ_α and ξ_β be the two functions defined respectively as $\xi_\alpha : \mathcal{S} \to \mathcal{O}$ and $\xi_\beta : \mathcal{S} \to \mathcal{O}$ that, given a state $\mathbf{s}$, return the observable state $\mathbf{o}_\alpha$ and $\mathbf{o}_\beta$.

$$\xi_\alpha(\mathbf{s}) = \mathbf{o}_\alpha \qquad \text{and} \qquad \xi_\beta(\mathbf{s}) = \mathbf{o}_\beta$$

The distinction can be useful if we want to differentiate the level of the knowledge of the current state $\mathbf{o}$ for α or β. Such flexibility can be very useful to model specific cases of *partial knowledge*.

Let $\mathcal{U}_\alpha$ be the space of dimension l of all the possible actions of the *agent* and let $\mathcal{U}_\beta$ be the space of dimension r of all the possible actions of the *environment*. Let $\mathbf{u}_\alpha \in \mathcal{U}_\alpha$ and $\mathbf{u}_\beta \in \mathcal{U}_\beta$ be the actions taken. Moreover, let t be the time defined over the $\mathbb{R}$ space. The simulator $f_\mathcal{M}$ (or simply f) that computes the evolution of our CPS, is defined as $f : \mathcal{S} \times \mathcal{U}_\alpha \times \mathcal{U}_\beta \times \mathbb{R} \to \mathcal{S}$

and, in particular, it is a differential equation of the form:

$$\dot{s} = f(\mathbf{s}, \mathbf{u}_\alpha, \mathbf{u}_\beta, t)$$

We can discretize it with respect to time $\Delta t = t_{i+1} - t_i$ and we obtain:

$$\mathbf{s}_{i+1} = \mathbf{s}_i + f(\mathbf{s}_i, (\mathbf{u}_\alpha)_i, (\mathbf{u}_\beta)_i, t_i)$$

where i is the i-th discrete instant of time of the simulation. We have now a discrete simulator that is able to compute the evolution of the CPS deterministically given the control actions.

Since the simulator is very dependent on the specific case study, in the next chapter we cover some examples of the equations used.

The simulator plays a crucial role: the NNs are not trained using observed examples. Similarly to what happens in reinforcement learning [41], we simply provide a simulated environment in which different choices can be made without real consequences. This allows for a fast learning procedure that can encourage safe actions and penalize those that lead to undesired results.

In the next chapter, we describe in details how our *case studies* are designed.

3.1.2 Controller-Environment architecture

The architecture is clearly inspired by that of a GAN. In fact, we have two NNs that compete in a *minimax* game for maximizing opposite goals.

The architecture is composed of:

- the **attacker**, denoted by A, similar to a GAN *generator*,
- the **defender**, denoted by D, that tries to keep the CPS safe in the test scenarios.

The main difference compared to GANs is that there is no *discriminator*. The aim of the *attacker* is to generate *environment* configurations in which the *defender* is not able to act safely, rather than trying to fool it.

Let us now describe the interaction between the physical model $\mathcal{M}$ and our adversarial NNs. The *defender* NN D is the controller of the *agent* in the model $\mathcal{M}$, while the *attacker* is in charge of generating the worst possible configuration of the *environment*.

The architecture of the two NNs can be whatever fits the problem at best. Though, the power of the proposed approach is given by how the training is carried out, not by the NNs' design.

The two NNs produce a sequence of actions $\mathbf{u}_\alpha$ and $\mathbf{u}_\beta$. Since in our approach there is no notion of time and only the state of our model $\mathcal{M}$ in a specific instant t_i is known, we need our NNs to produce, for each timestep, a whole sequence of actions. To make this possible we introduce the *policy function* Π_A for the *attacker* NN e Π_D for the *defender* NN.

Let $\mathbf{w}_A$ and $\mathbf{w}_D$ be the coefficients' vectors defined on the spaces $\mathbb{R}^p$ and $\mathbb{R}^q$, respectively. Let the notation $t_i \dots t_{i+h}$ denote a timeframe defined over h instants. From now on, this notation will denote sequences defined on the interval $[i, i+h]$.

The *policy function* is a polynomial functions of the coefficients $\mathbf{w}_i$ that produces as output a function $\mathbf{u}(t)$ that, discretized as $\mathbf{u}(t_i) = \mathbf{u}_i$ for each timestep between t_i and t_{i+h}, gives the sequence of actions $\mathbf{u}_i \dots \mathbf{u}_{i+h}$ for the considered horizon h. In particular we define $\Pi_A : \mathbb{R}^p \times \mathbb{R} \to \mathcal{U}_\beta$ and $\Pi_D : \mathbb{R}^q \times \mathbb{R} \to \mathcal{U}_\alpha$ separately for each NN:

$$\Pi_A(\mathbf{w}_{Ai}, t) = \sum_j^p (\mathrm{w}_j)_{Ai} \cdot t^j = \mathbf{u}_\beta(t) \quad \text{with} \quad \mathbf{w}_{Ai} = A(\theta_A, \mathbf{o}_{\beta i}, \mathbf{z}_i)$$

$$\Pi_D(\mathbf{w}_{Di}, t) = \sum_j^q (\mathrm{w}_j)_{Di} \cdot t^j = \mathbf{u}_\alpha(t) \quad \text{with} \quad \mathbf{w}_{Di} = D(\theta_D, \mathbf{o}_{\alpha i})$$

The choice of the *policy function* can be NN specific: for instance, they could use polynomials with different degree.

Using such *policy function* has the advantage of introducing a natural smoothing and preventing an incoherent behaviour of the actions $\mathbf{u}$ in two subsequent instants.

The problem of finding the best action $\mathbf{u}$ has become the problem of finding the best coefficients $\mathbf{w}_A$ and $\mathbf{w}_D$ for the *policy functions* Π_A and Π_D. And this is precisely the problem addressed by our NNs. Despite the two NNs are very similar, we are going to cover them separately.

Let θ_A be the matrix of the weights and biases (the parameters) of the *attacker* NN, defined on the space Θ_A of dimension dependent on NN's architecture. As said before, the *attacker* controls the *environment*. Therefore, at time t_i it takes as input its own parameters θ_{Ai}, the observable state $\mathbf{o}_{\beta i}$ and a vector of *gaussian* noise $\mathbf{z}_i \in \mathbb{R}^g$, $\mathbf{z}_i \sim \mathcal{N}(0, 1)$. The introduction of such random input vector $\mathbf{z}$ is borrowed from usual GANs literature: as we have seen in the previous chapter, it is possible to use this *latent variable* to sample from the *latent space*. The *attacker* NN is defined as $A : \Theta_A \times \mathcal{O} \times \mathbb{R}^g \to \mathbb{R}^p$ and can be represented as the following function:

$$A(\theta_A, \mathbf{o}_{\beta i}, \mathbf{z}_i) = \mathbf{w}_{Ai}$$

The *defender*, similarly to his opponent, controls the *agent* and take as input its parameters θ_D – the matrix of weights and biases defined over the space Θ_D of dimension dependent on NN's architecture – and its observable state vector $\mathbf{o}_{\alpha i}$. No noise is passed as input to the *defender* since there is no need to learn a distribution. This NN must simply find the best possible set of coefficients $\mathbf{w}_{Di}$. The function is defined as $D : \Theta_D \times \mathcal{O} \to \mathbb{R}^q$ and can be represented as:

$$D(\theta_D, \mathbf{o}_{\alpha i}) = \mathbf{w}_{Di}$$

By doing so, starting from a given state of the system $\mathbf{s}_i$ (from which we can observe $\mathbf{o}_{\beta i}$ and $\mathbf{o}_{\alpha i}$), we obtain the optimal coefficients $\mathbf{w}_{Ai}$ and $\mathbf{w}_{Di}$ and, by means of the *policy functions*, we obtain the optimal sequence of actions $(\mathbf{u}_\beta)_i \dots (\mathbf{u}_\beta)_{i+h}$ and $(\mathbf{u}_\alpha)_i \dots (\mathbf{u}_\alpha)_{i+h}$ for the model $\mathcal{M}$ in the horizon h.

Given the sequences of actions in the time horizon h, we can now simulate via f the complete evolution of the system $\mathcal{M}$. The simulation is carried out step by step from time t_i to t_{i+h}: at time t_i, a pair of actions $(\mathbf{u}_\beta)_i$ and $(\mathbf{u}_\alpha)_i$ is passed to f that makes the system evolve into a state $\mathbf{s}_{i+1}$ and so on. Throughout the simulation we keep track of the sequence of system's states and we collect its evolution $\mathbf{s}_i \dots \mathbf{s}_{i+h}$.

Given the STL formula Φ for the requirement and keeping track of the sequence of the model's states $\mathbf{s}_i \dots \mathbf{s}_{i+h}$ evolved from the actions of both the *attacker* and *defender*, it is possible to compute the **robustness** of the trajectories via STL quantitative semantics (the function R to compute the quantitative semantics has been introduced in the previous chapter):

$$\rho_i = R(\Phi, \mathbf{s}_i \dots \mathbf{s}_{i+h})$$

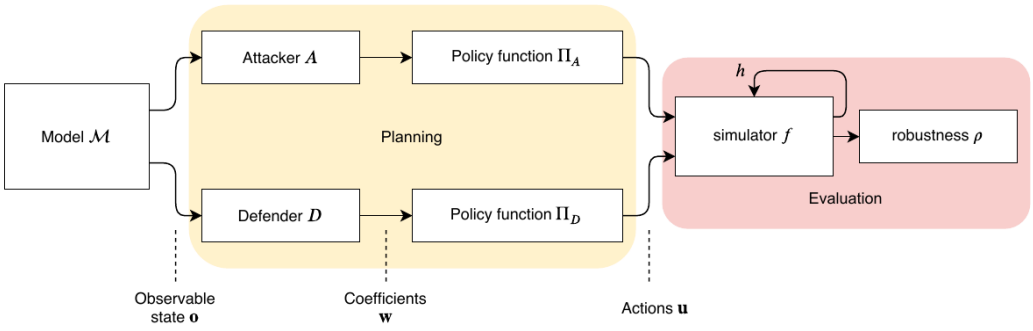

Figure 3.2: Diagram of the architecture. The model $\mathcal{M}$ observable state $\boldsymbol{o}$ is the input of the two NNs that produce the coefficients $\boldsymbol{w}$ for the policy functions Π. These functions gives the actions that are simulated for h steps and the robustness of the system's evolution is computed.

The STL *quantitative semantics*, as said in the previous chapter, measures the maximum shift that can be applied to a trajectory without varying the truth value of the requirements Φ. Therefore it is straightforward to use the *robustness* ρ as *objective function* for tuning of the internal parameters of the NNs.

The *robustness* is the quantity on which the two NNs play the *minimax* game. For convenience, let F be a function $F : \mathcal{S} \times \Theta_D \times \Theta_A \to \mathcal{S}^h$ that, given a initial state $\mathbf{s}_i$ of $\mathcal{M}$ and the parameters of the two NNs θ_D and θ_A, computes the system's evolution in the horizon h:

$$F(\mathbf{s}_i, \theta_D, \theta_A) = \mathbf{s}_i \dots \mathbf{s}_{i+h}$$

Such function applies iteratively the simulator function $f(\mathbf{s}_i, (\mathbf{u}_\alpha)_i, (\mathbf{u}_\beta)_i, t_i) = \mathbf{s}_{i+1}$ for each pair of actions $(\mathbf{u}_\alpha)_i = \Pi_D(D(\theta_D, \mathbf{o}_{\alpha i}), t_i)$ and $(\mathbf{u}_\beta)_i = \Pi_A(A(\theta_A, \mathbf{o}_{\beta i}, \mathbf{z}_i), t_i)$ computed in the interval $t_i \dots t_{i+h}$. Then the *loss function* is defined as follows:

$$\mathscr{L}(\theta_A, \theta_D) = R(\Phi, F(\mathbf{s}_i, \theta_D, \theta_A))$$

Therefore, the *minimax* game between the two is

$$\min_{\theta_A} \max_{\theta_D} \mathscr{L}(\theta_A, \theta_D)$$

In this setting, the *defender* is going to tune its NN's weights in such a way to maximize the *robustness* by generating safe actions for the model, while the *attacker* is going to tune its NN's weights trying to generate more and more challenging scenarios for the opponent with the aim of minimizing its *robustness*.

3.2 Training

The training approach borrows from the *reinforcement learning* literature [41]. The training is expressed as a sequence of **episodes** in which our *agent* receives a feedback – the *robustness* – for its choices and learns from it.

We define an *episode* as a single safety problem to be solved. At the beginning of each *episode*, a new state $\mathbf{s}_0$ of the model $\mathcal{M}$ is generated randomly to allow both the*attacker* and the *defender* to explore and learn from different configurations of the setting. Then, each of them plans a strategy for the time horizon h, the whole evolution is simulated through F, and the *robustness* of the specific sequence of choices is then computed in order to optimize the weights of the NNs.

Since the NNs do not take into account the past history of the opponents' actions and base the predictions solely on the current state $\mathbf{s}_i$, it is possible to imagine each instant of the evolution of the system as a separate problem to solve. This specific property that does not make any assumption on the system's evolution, allows to deal with problems using a simple architecture. However, to explore reasonably well all the state space $\mathcal{S}$, it requires a specific kind of training in order to avoid undefined behaviours due to completely unexplored scenarios.

Due to this architecture's characteristic, in fact, it is compulsory to sample from the entire state space $\mathcal{S}$ in order to expose the NNs even to the less probable cases. The *hyper-grid* – on which all the possible states lie – is explored randomly adding some noise drawn from a normal distribution $\mathcal{N}(\mu, \sigma)$ where μ and σ are respectively the mean and the variance specified according to the case study in order to maximize the diversity of the samples even in case of repeatedly sampled states.

For every sample drawn from the state space $\mathcal{S}$, a training *episode* for a specific NN takes place. The training, in fact, happens in a turn-based

scheme where, both NNs make a choice but only one of them learns from that *episode*.

For instance, suppose we are training D at time t_0 and suppose we sample a state $\mathbf{s}_0$, the parameters θ_A of the other NN remain constant for the specific *episode*. The following operations take place for each *episode*:

$$\begin{aligned} \Pi_A(A(\overline{\theta_A}, \xi_\beta(\mathbf{s}_0), \mathbf{z}_0), t) &= \mathbf{u}_\beta(t) \\ \Pi_D(D(\theta_D, \xi_\alpha(\mathbf{s}_0)), t) &= \mathbf{u}_\alpha(t) \\ F(\mathbf{s}_i, \theta_D, \overline{\theta_A}) &= \mathbf{s}_0 \dots \mathbf{s}_h \\ R(\Phi, \mathbf{s}_0 \dots \mathbf{s}_h) &= \rho \end{aligned}$$

At this stage, the *robustness* value ρ is used only to compute the *loss function* of D and, hence, the backpropagation of the gradient affects only the internal parameters of the *defender*. The training of A is conducted in the very same fashion.

Each training *episode* can be repeated multiple times in order to make the NN try new alternative moves in the same scenario. This strategy encourages the *exploration* of different possible outcomes. In particular, it is possible to make each NN repeat the *episodes* a variable number of times. Such mechanism is borrowed from classical GANs and it is used to balance the game between the two NNs: sometimes the task of one NN is much simpler than the other's, therefore, it needs to be artificially balanced. Setting a different number of repetition for each NN, hence, increases the chances of convergence and prevents a NN from outperforming the other not allowing it to learn at all.

Algorithm 1 Training procedure

procedure TRAINSTEP
  $s_0 \leftarrow$ SampleRandomState()
  for attacker_repetitions **do** ▷ Train A
    $z \leftarrow \mathcal{N}(0, 1)$
    $w_\beta \leftarrow A(\theta_A, \xi_\beta(s_0), z)$
    $w_\alpha \leftarrow D(\theta_D, \xi_\alpha(s_0))$

    states := [ ]
    states[0] $\leftarrow s_0$
    for $i \leftarrow 0 \dots h-1$ **do**
      $u_{\alpha i} \leftarrow \Pi_D(w_\alpha, i)$
      $u_{\beta i} \leftarrow \Pi_A(w_\beta, i)$
      states$[i+1] \leftarrow f(s_i, u_{\alpha i}, u_{\beta i}, t_i)$

    $\rho \leftarrow R(\Phi, \text{states})$
    BackPropagation(A, ρ)
    UpdateWeights(A)

  $s_0 \leftarrow$ SampleRandomState()
  for defender_repetitions **do** ▷ Train D
    $z \leftarrow \mathcal{N}(0, 1)$
    $w_\beta \leftarrow A(\theta_A, \xi_\beta(s_0), z)$
    $w_\alpha \leftarrow D(\theta_D, \xi_\alpha(s_0))$

    states := [ ]
    states[0] $\leftarrow s_0$
    for $i \leftarrow 0 \dots h-1$ **do**
      $u_{\alpha i} \leftarrow \Pi_D(w_\alpha, i)$
      $u_{\beta i} \leftarrow \Pi_A(w_\beta, i)$
      states$[i+1] \leftarrow f(s_i, u_{\alpha i}, u_{\beta i}, t_i)$

    $\rho \leftarrow R(\Phi, \text{states})$
    BackPropagation(D, ρ)
    UpdateWeights(D)

3.3 Testing

Once a sufficiently high number of *episodes* has been completed, the two NNs are ready to be used as *controller* and test generator of our CPS. It is possible to use the two NNs separately: in fact, once trained, they become completely independent one from the other.

The testing phase differs slightly from the training one since, in real scenarios, we do not have *episodes* but a continuous evolution of the system. Therefore, the strategy of computing an optimal plan for the time horizon h and applying it, does not fit such scenario.

We took inspiration from the *Model Predictive Control* algorithm [42] to design the application of our method: in simple words, at each timestep t_i, the optimal strategy over h steps is computed as in the training step but, instead of passing all the actions $\mathbf{u}_i \dots \mathbf{u}_{i+h}$ to the simulator, we execute only the first one and, on the next iteration, the new state $\mathbf{s}_{i+1}$ is used to compute the next move. More precisely, the following operations are repeated iteratively:

$$\Pi_A(A(\overline{\theta_A}, \xi_\beta(\mathbf{s}_i), \mathbf{z}_i), t) = \mathbf{u}_\beta(t)$$
$$\Pi_D(D(\overline{\theta_D}, \xi_\alpha(\mathbf{s}_i)), t) = \mathbf{u}_\alpha(t)$$
$$f(\mathbf{s}_i, (\mathbf{u}_\alpha)_i, (\mathbf{u}_\beta)_i, t_i) = \mathbf{s}_{i+1}$$

Since the NNs are trained and they are not learning anymore, we marked both their parameters with an overline.

Due to the iterative scheme of the policies, the proposed method acts as a **closed-loop** CPS. At timestep t_i, in fact, an action is taken with respect to the time horizon h and the new state $\mathbf{s}_{i+1}$ is reached. Through the *sensors*, our NNs will be able to retrieve the observed state $\mathbf{o}_{i+1}$ that will be used to compute the next action. Such iterative procedure allows to handle automatically cases in which the desired state is not fully reached due to

unexpected causes: the algorithm will plan a new optimal strategy starting from the new observed states, hence taking into account the possible errors that have been made.

Algorithm 2 Test procedure

```
1:  procedure EXECUTE(M, timesteps)
2:      for i ← 0 ... timesteps do
3:          s_0 ← GetState(M)
4:
5:          z ← N(0, 1)
6:          w_β ← A(θ_A, ξ_β(s_0), z)
7:          w_α ← D(θ_D, ξ_α(s_0))
8:
9:          u_β ← Π_A(w_β, 0)
10:         u_α ← Π_D(w_α, 0)
11:
12:         DoAction(M, u_α, u_β)
```

Chapter 4

Case study

4.1 Cruise control

We used as a preliminary test of the architecture the *cruise control* problem. Such problem, even if it does not fully exploit the capabilities of our architecture, was selected since it is one of the classic baseline problem used as CPS. The problem is more than half-century old and has been solved mechanically [43] at first and, finally, electronically [44] around the '70s.

The aim is very simple: to keep a vehicle running at constant velocity regardless of the *steepness* of the road. The model used, due to the nature of the problem, does not present a *reactive environment* and, therefore, it does not fully exploit the potential of the our method. The *environment* of this setting is the road and its steepness that, once defined, cannot vary.

Cruise control systems make use of specific *sensors* to measure the current car velocity and one *actuator*, able to vary the engine fuel intake of the engine according to the *controller* decision.

4.1.1 Model

The model $\mathcal{M}$ in such scenario is composed of the *agent* α, represented by the car c, and the *environment* β, represented by the road.

The road r is static: in fact, once generated by the *attacker* NN, it remains the same for entire *episode*. It is generated as a function that represents its elevation profile. In this specific setting, we generate the road on the physical space x using the **Radial Basis Function** with a *Gaussian* kernel.

Let d be the number of *RBFs* that we want to sum to generate r, let $\boldsymbol{\mu}$ be an arbitrary vector of dimension d that represents the centers of the *RBFs*, let $\boldsymbol{\sigma}$ be the vector of dimension d that represents scale factor of the *RBFs*, let $\boldsymbol{\omega}$ be the vector of dimension d that weights the contribution of each RBF. The function used to describe the road is:

$$r(x) = \sum_{i}^{d} \omega_i \ \exp\left(-\frac{||x - \mu_i||^2}{\sigma_i^2}\right)$$

The vector $\mathbf{u}_\beta$ generated by the *attacker* is built such that $\mathbf{u}_\beta = (\boldsymbol{\omega}, \boldsymbol{\sigma})$. In this way the *attacker* NN can influence the generation of the road.

At each instant, the car c have knowledge of its current velocity v_c and, trough some tilt sensors, it can also measure the steepness of the road in its current position $\theta_c = r'(x_c)$, where x_c is the position of the car. The car can control its acceleration a_c through the actuator that controls the throttle.

Given the setting described above, it is possible to turn the requirement into a STL formula. Since it must hold for the whole trajectory of the car, we used the $\mathcal{G}$lobally operator. Let $\tilde{v}$ be the desired steady velocity and ε be the tolerance on such value. The STL formula for the requirement Φ is:

$$\Phi = \mathcal{G}(v_c \geq \tilde{v} - \varepsilon \ \wedge \ v_c \leq \tilde{v} + \varepsilon)$$

Since the road's steepness is generated only once at the beginning of each

episode, the only part of the CPS that evolves in time through differential equations is the *agent*'s one. In particular, the continuous model of the *agent* operates some simplifications:

- it is seen as punctiform;
- it cannot steer.

Let m be the mass of the car (can be disregarded in this case), g be the gravity of $9.81\frac{m}{s^2}$, ν be the friction constant of 0.01. We call $\left(\frac{dv}{dt}\right)_{in} = a_c$ the controllable acceleration that is provided to the car by the *actuator*. The car dynamics follows the differential equation:

$$m\frac{dv}{dt} = m\left(\frac{dv}{dt}\right)_{in} - \nu mg\cos\theta_c - mg\sin\theta_c$$

The formula above describes how the velocity varies over time. We can see its dependency on the given acceleration a_c, to which we subtract the friction and the vertical component of its weight.

In this case study, the *attacker* is very simple. It does not take any input depending on the state of $\mathcal{M}$, hence the observable state $\mathbf{o}_\beta = \varnothing$. It gives as output the vector $\mathbf{u}_\beta$ to build the function $r(x)$ that represents the profile of the road. We limited the maximum steepness allowed to the interval $[-25°, 25°]$ by setting some constraints on $r'(x)$ to prevent trivial adversarial scenarios (vertical walls).

The *defender*, on the other side, takes as input the observable state $\mathbf{o}_\alpha = (v_c, \theta_c)$ and gives as output the acceleration $\mathbf{u}_\alpha = (a_v)$ of the car c. The simulated model has been limited to allow accelerations in the interval $[-3\frac{m}{s^2}, 3\frac{m}{s^2}]$, while the range of allowed velocities is $[-10\frac{m}{s}, 10\frac{m}{s}]$.

4.2 Car platooning

We selected the problem of *car platooning* [45] to test the full potential of our architecture. This kind of setting is relatively new and it has been developed in the field of *Autonomous Driving*. The platooning, in fact, is the formation of a line of vehicles in which they follow each other in a controlled way: they should never hit the car in front and they should always have enough space in order to brake on time in case of sudden change of leader's speed. This case study is starting to gather interest since it promises to reduce the cost of the transport and increase the overall transportation infrastructures' efficiency [46].

Nowadays this problem has been faced with many composite techniques [46] to coordinate the actions of the car's pool, most of them rely on the so called *vehicular ad-hoc networks* that can be of the type **V2V** (Vehicle to Vehicle) or **V2I** (Vehicle to Infrastructure).

This approach, though, requires specific hardware and a distributed system of coordination that can be an impossible requirement to meet in some cases. Testing our approach to obtain a robust controller on this kind of problem is interesting, at it explores the possibility of an *individual-basis* decision system.

4.2.1 Model

In the car platooning setting we have n cars that follow each other forming a line. Following a strategy of *divide et impera*, it is possible to consider just the problem of two cars before generalizing the solution.

In the model $\mathcal{M}$, that we are going to define, we have one leader l and one follower f. Both leader and follower have an internal state that keeps track of the car's position x, velocity v and acceleration a.

The requirement Φ for the problem can be expressed in STL as

$$\Phi = \mathcal{G}(d \leq d_{max} \ \wedge \ d \geq d_{min})$$

where $d = x_l - x_f$ is the distance between l and f, and d_{max} and d_{min} are respectively the maximum and minimum distance allowed. Such distance can be estimated or measured. In our experiment, we assume that we can measure it by means of a *LIDAR scanner*. This kind of hardware, that makes use of a laser beam to measure distances, is already installed on some autonomous or semi-autonomous cars.

We used the operator $\mathcal{G}$lobally in the STL formula because we want the property to hold for the whole trajectory of the cars. The minimum distance allowed was set at 2 meters and the maximum one at 10.

Since our aim is to have a safe controller that is able to follow the car ahead under the requirement Φ, it comes natural to label the follower f as the *agent* α of our model $\mathcal{M}$, while all the rest is addressed as β, the *environment*. This definition implies that the leader l is considered part of the *environment* too.

Defined the roles inside our abstract model $\mathcal{M}$, we can now analyze how the underlying CPS is configured. To describe the car's physical behaviour, we need to operate some simplifications:

- they are seen as punctiform;
- they can move exclusively on a straight line;
- they move on a flat landscape;
- they can only move forward.

Let ν be the friction coefficient of 0.01, g be the gravity of $9.81\frac{m}{s^2}$ and m the mass (that can be disregarded in this case). The differential equation

that describe the car's dynamic is the following:

$$m\frac{dv}{dt} = m\left(\frac{dv}{dt}\right)_{in} - \nu mg$$

The formula above represents the variation of the final velocity of the car with respect to time, given as input the car's acceleration $\left(\frac{dv}{dt}\right)_{in} = a$ and subtracting the friction due to the movement.

In this specific case study, since both the *agent* and the *environment* are very similar, also their inputs are alike: both are single-element vectors $\mathbf{u}_\alpha = (a_f)$ and $\mathbf{u}_\beta = (a_l)$ that control the cars' acceleration. a_f and a_l are respectively the accelerations that the *agent* and the *environment* are going to provide to the dynamic equation illustrated above. To be more realistic, we limited the acceleration of the car to $[-3\frac{m}{s^2}, 3\frac{m}{s^2}]$ and the velocity to $[0\frac{m}{s}, 20\frac{m}{s}]$.

In this specific case study, moreover, $\mathbf{o}_\alpha \equiv \mathbf{o}_\beta$, hence we can refer to the model's state simply as $\mathbf{o}$. This vector contains information about the state of $\mathcal{M}$, in particular $\mathbf{o} = (v_l, v_f, d)$, where v_l is leader's velocity, v_f is follower's velocity and d is the relative distance.

Chapter 5

Experimental results

5.1 Implementation details

We implemented as a Python framework the whole architecture that has been described so far.

The code of the project can be divided into 3 parts.

- The **model**: it contains the code of the physical model. The latter must be *differentiable* and should comply with some internal API in order to be integrated effortlessly inside the rest of the library;
- the **NNs architecture**: it is the core of the project and it is configurable only via a defined API;
- the **experimental setup**: it is the code where the user can tweak all the parameters and decide how to test the chosen model.

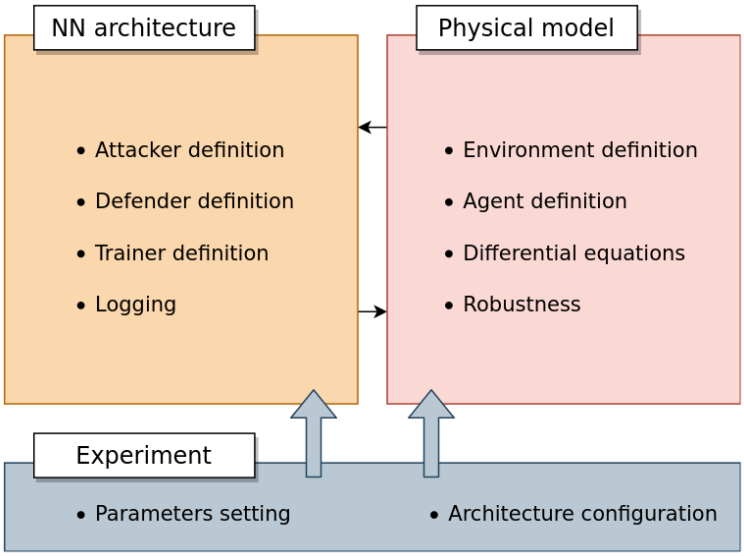

Figure 5.1: Schematic visualization of the interaction between the three main module of the architecture. The structure reflects the organization of the code as well.

In order to compute the robust semantics of the STL formulas in a fast and efficient way, it has been implemented a small separate module that parses and transforms the STL requirements into executable code.

The whole project relies extensively on PyTorch [47], a Python library tailored to build complex numerical models. All the computations leverage the PyTorch's computational graph, which eased the implementation and made the training of the NNs possible. Unfortunately, due to the iterative nature of the training process, porting it on GPU did not provide benefits. In this regards there is room for improvements.

5.2 Cruise control

The problem of the cruise control has been described in the previous section: given a vehicle that moves along a road, the controller should be able to keep its velocity constant regardless of the steepness of the road.

5.2.1 Experimental settings

For the desired constant velocity, we fixed as *setpoint* $\tilde{v} = 5\frac{m}{s}$ and a tolerance $\varepsilon = 0.25\frac{m}{s}$. Therefore, the **constraint** applied to the velocity v_c of the moving car c, is $\Phi = \mathcal{G}(v_c \leq 5.25 \wedge v_c \geq 4.75)$.

Due to the nature of the problem, the attacker controls the generation of the steepness of the road. Such operation takes place **once**, at the beginning of each episode.

The **attacker**'s architecture has 2 layers with 10 neurons each. To each layer is applied the *Leaky ReLU* activation function [48]. The input noise vector $\mathbf{z}$ belongs to the space $\mathbb{R}^5$. This NN does not have other input since the *environment* (the road) is a static object with no knowledge about the setting. The output of the NN is the vector $\mathbf{u}_\beta = (\boldsymbol{\omega}, \boldsymbol{\sigma})$ used to determine the function of the elevation of the road $r(x)$. The dimension d, that defines the number of *RBFs* composing $r(x)$, has been fixed to 3.

Since the environment is static and does not need to describe a dynamic behaviour, the *policy function* of the attacker is defined over the space $\mathbb{R}^1$.

The **defender** NN has 3 layers with 10 neurons each. To each layer is applied the Leaky ReLU activation function.

The *defender* NN uses a *policy function* defined over the space $\mathbb{R}^3$, due to the need of adapting and forecasting different scenarios.

The **initial configuration** of each training episode is sampled from an *hyper-grid* of two dimensions: initial position x_{c0} and initial velocity v_{c0}. The initial position x_{c0} is sampled uniformly from the interval $[0\ m, 50\ m]$. Such choice has been made to introduce variability in the training and force the NNs to explore different strategies. The initial velocity v_{c0} is sampled among 25 equispaced points between $[-12\frac{m}{s}, 12\frac{m}{s}]$.

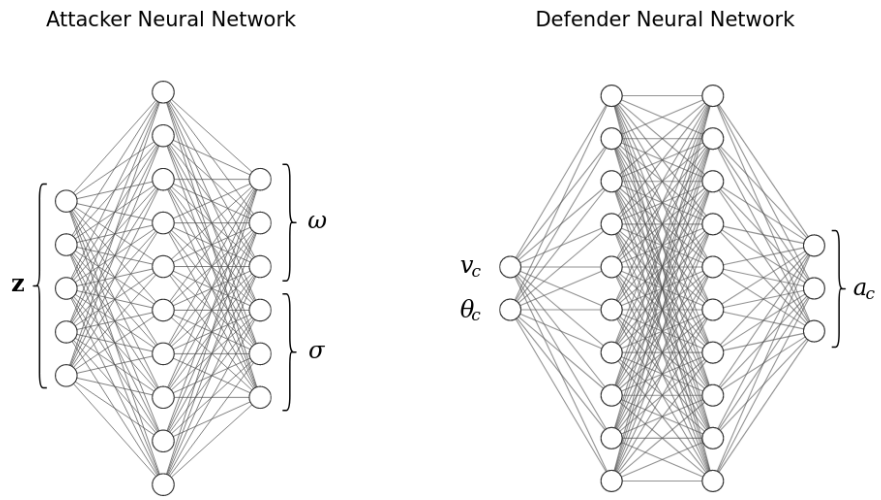

*Figure 5.2: The architecture used for the the two NN in the cruise control case study. The **attacker** takes as input the noise vector **z**, while the output are the two vectors **ω** and **σ** that characterize the . The **defender** takes as input the car's velocity v_c and the inclination of the car (steepness) θ_c to compute the acceleration a_c.*

During the training the **timestep** Δt between two time instant has been set to $0.05s$ while the horizon h for each episode was $500ms$, 10 timesteps. The same *timestep* has been used also in the testing.

The training phase lasted $30k$ *cycles* and each **episode** has been repeated only 1 time for the *attacker* and 10 times for the *defender*.

5.2.2 Metrics and results

The testing configurations is the same of the training except for the the initial position v_{c0} of the car, that is always assumed to be 0 m.

We decided to test the model in three different scenarios. In the first scenario the road is generated by the attacker NN by means of the *RFBs*. They create more complex shapes and produce realistic road profiles, this make them perfectly suitable for training but less suitable for assessing the performance of the controller.

In the second and third scenarios we have a flat landscape and, respec-

tively, only one hill and only one valley. Such simple tests are good to identify the pitfalls of the trained model.

In each figure we show the elevation of the road, the velocity of the car and the acceleration provided by the *defender*.

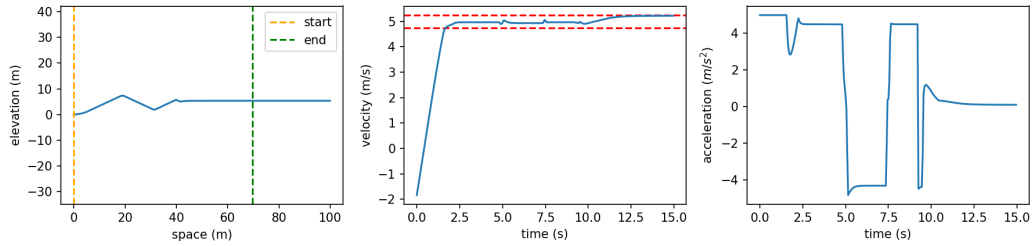

Figure 5.3: Initial configuration: $v_{c0} = -2.04\frac{m}{s}$. *The model manages to reach the area of the desired velocity even starting from afar. Looking at the acceleration, it is possible to notice the adaptation of strategy in presence of a sudden change of steepness.*

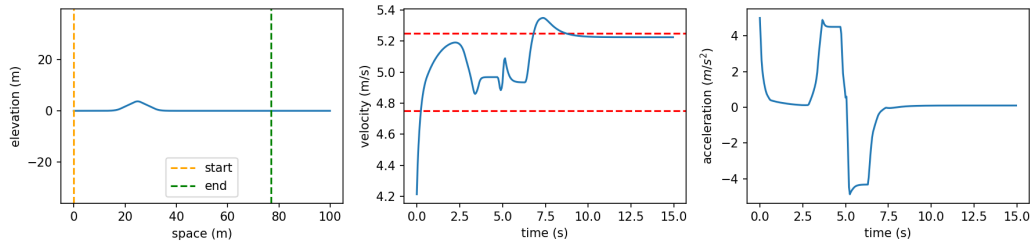

Figure 5.4: Initial configuration: $v_{c0} = 3.96\frac{m}{s}$. *Here the car start with a velocity that is quite close to the setpoint. However, the trajectory presents a bit of overshoot in the correction that takes place right after the descent.*

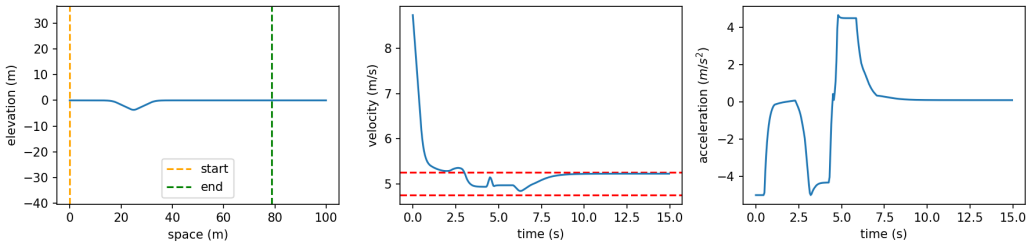

Figure 5.5: Initial configuration: $v_{c0} = 8.98\frac{m}{s}$. *In such scenario we see how the defender is able to tackle two problems at the same time: reaching the setpoint and reacting to the change of slope. Still it succeed in managing the situation in a quite stable way.*

In the figures above we can observe how different are the reaction patterns in the case of *attacker*-generated scenario and artificial one. The *defender* reacts to the landscape generated by the adversarial NN, without noticeable overshoot and handles the situation in a smooth way. In the case of the artificial hill and valley, instead, it varies more frequently the acceleration patterns even if the profile of the road is much simpler. In every case, though, the *defender* is able to rapidly reach the desired velocity and keep it.

In order to perform model comparison and evaluate the performance of the trained model, we ran $10k$ simulations of trajectories. The following plot shows the percentage of trajectories that in a given instant was inside the boundary specified by the requirement.

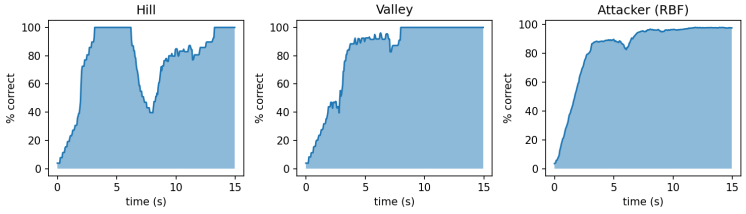

Figure 5.6: This plot shows that every single trajectory simulated reached successfully the target velocity, in the end. It is possible to observe how the training on the attacker was very effective. The experience gained in such scenario, though, is less useful in the others, especially in front of sudden change of slope.

In the figure 5.6 we can see how successful was the adversarial training

for the *defender*: it learnt how to face every possible configuration generated from the attacker. The smooth growth tells that it can gradually improve the situation over time until it reaches the setpoint.

The first two pictures, instead, show that there is room for improvement for the generalization ability of the *defender*'s NN. Since the result are already very promising, a better exploration of the space of the road's configuration could lead to perfect results.

5.3 Car platooning

We decided to split the problem of car platooning into a simpler problem of only two cars. In such simplified setting, we trained both the *attacker* and the *defender* and finally we generalized the behaviour to n cars. We recall that the problem we want to address is the generation of a safe controller that controls the acceleration of its vehicle in order to keep the distance with the other vehicle inside a given range.

5.3.1 Experimental settings

In the simplified setting of the two cars, we have a *leader* l and a *follower* f. The **constraint** applied on the CPS described in the chapter before is on the *distance* d between the two vehicles and is $\Phi = \mathcal{G}(d \leq 10 \wedge d \geq 2)$.

Every **initial configuration** of the experimental runs has been sampled on an *hyper-grid* of three dimensions: leader's velocity v_{l0}, leader's position x_{l0} and follower's velocity v_{f0}. The initial velocities v_{l0} and v_{f0} have been sampled among 40 equispaced values in the interval $[0\frac{m}{s}, 20\frac{m}{s}]$. The initial leader's position x_{l0} have been sampled among 15 equispaced values in the interval $[1\ m, 12\ m]$ while the follower's x_{f0} has been set to $0\ m$.

The **attacker**'s architecture is a dense 3-layered NN with 10 neurons per layer. To each layer has been applied the *Leaky ReLU* activation function

[48]. The dimension of the space to which the noise vector $\mathbf{z}$ – input of the *attacker* NN – belongs, has been set to 2. The **defender** NN has the same structure of the *attacker* except for the input dimension (does not take as input a noise vector). In order to compute the trajectory of the actions, both the NNs use a polynomial **policy function** of order 3. Therefore, the spaces over which $\mathbf{w}_A$ and $\mathbf{w}_D$ are defined, have dimension $q = p = 3$.

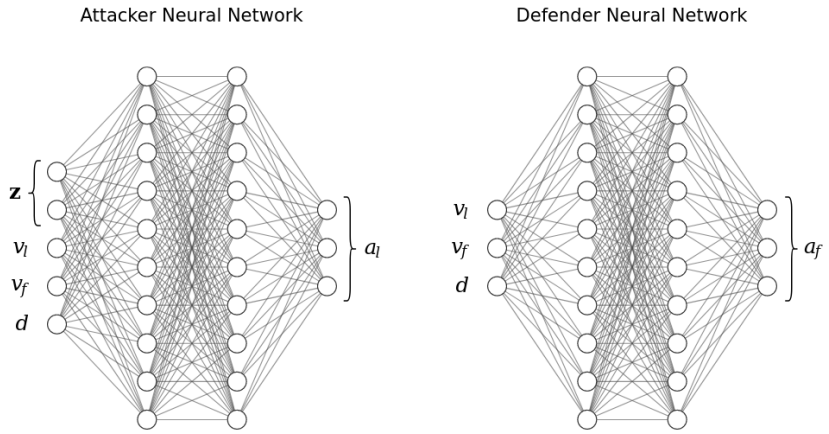

*Figure 5.7: The architecture used for the the two NN in the car platooning case study. The **attacker** takes as input the noise vector $\boldsymbol{z}$, the leader's velocity v_l, the follower's velocity v_f and their distance d. It gives as output the acceleration a_l for the leader's car. The **defender** takes the same input of the attacker except $\boldsymbol{z}$ and gives as output the follower's acceleration a_f.*

During the training the **timestep** Δt between two time instant has been set to $0.05s$ while the horizon h for each episode was $5s$, 100 timesteps. The same *timestep* has been used also in the testing.

The training phase lasted $50k$ *cycles* and each **episode** has been repeated 3 times for the *attacker* and 5 times for the *defender*.

5.3.2 Metrics and results

One leader and one follower

We tried many configuration of the hyper-parameters of the model in order to find the ones that were giving the good results presented.

We tested the **defender-controlled** follower in four different scenarios to check if it was able to adapt effectively to different adversarial strategies. The first scenario is against the *attacker*, in the second the leader accelerates suddenly, in the third the leader brakes suddenly while in the fourth it accelerates rapidly to brake right after. For each scenario we show the position of the cars, the relative distance and the acceleration of both.

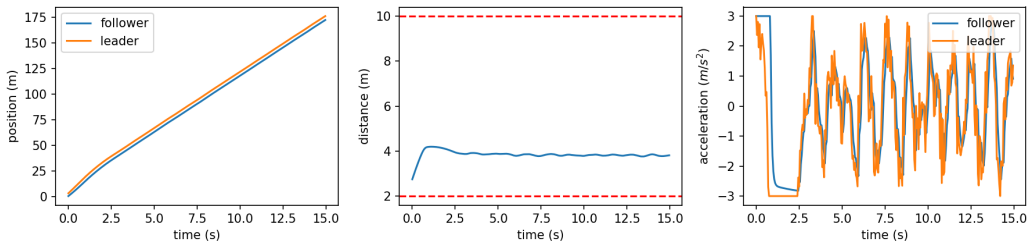

Figure 5.8: Initial configuration: $x_{l0} = 2.63\ m$, $v_{l0} = 15.58\frac{m}{s}$, $v_{f0} = 13.28\frac{m}{s}$. *We can see how the attacker found the strategy of rapidly accelerating and breaking. The defender, though, succeeds in keeping its distance safe and almost constant over time.*

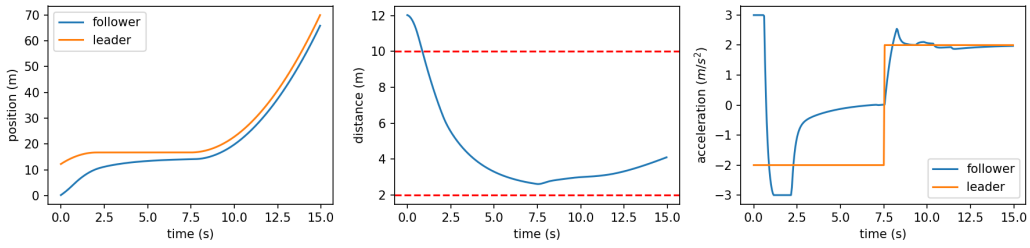

Figure 5.9: Initial configuration: $x_{l0} = 12.03\ m$, $v_{l0} = 4.49\frac{m}{s}$, $v_{f0} = 4.41\frac{m}{s}$. *In this scenario the defender has the initial disadvantage of being too far from the leader. It safely closes the gap and is able to tolerate the sudden acceleration of the leader.*

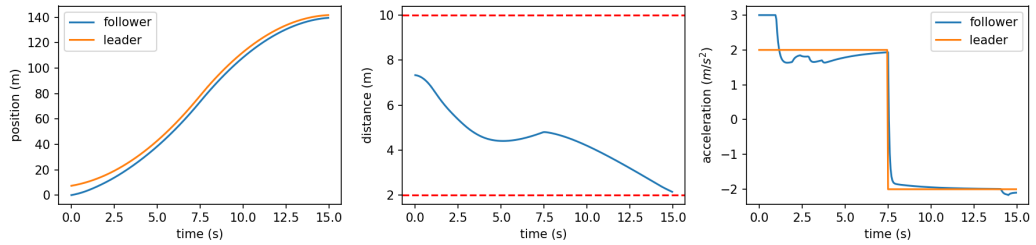

Figure 5.10: Initial configuration: $x_{l0} = 7.33\ m$, $v_{l0} = 2.19\frac{m}{s}$, $v_{f0} = 2.21\frac{m}{s}$. *In this scenario the defender is able to tolerate quite well the sudden brake since it was keeping the evaluated safety distance.*

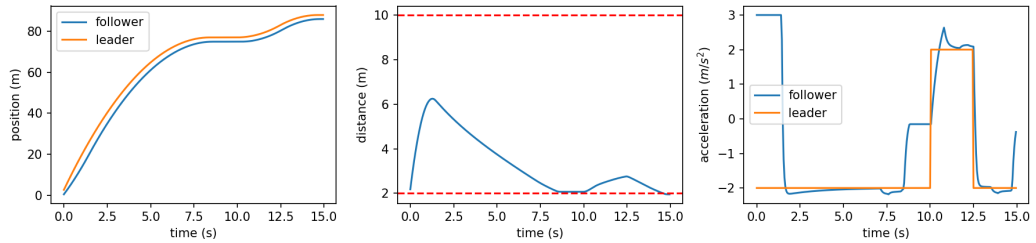

Figure 5.11: Initial configuration: $x_{l0} = 2.63\ m$, $v_{l0} = 15.58\frac{m}{s}$, $v_{f0} = 13.28\frac{m}{s}$. *In this scenario of sudden acceleration and brake, the defender slowly approaches the opponent until it stops waiting for it to move. It adapts to the opponent behaviour arriving at the minimum allowed distance without violating the constraint.*

It is possible to notice how well the follower manages the presented corner cases. In the figure 5.8 we can see how the follower safely keeps an almost steady distance with a leader that behaves in a unpredictable way. The follower mimics perfectly the acceleration pattern of the leader. The same happens in the figure 5.10 where, as soon as the leader suddenly brakes, the follower does the same to keep the safety distance.

In the figure 5.9, it is interesting to notice how the follower is able to exploit the moments in which the leader is almost static in order to close the initial gap. Even if the initial condition was not safe, the follower is able to recover from it and maintain the distance between the safety boundaries.

The follower, during the training, has learnt also to halt completely if the leader does so. Such behaviour is visible in the figure 5.11 in which the

follower halts in correspondence with the minimum distance allowed.

Even though the initial configuration plays a crucial role in the controller's safety, it is possible to observe how the NNs develop consistent strategies to solve at best their task.

In order to compare different models and have an estimate of their performances, we ran $10k$ simulation of different trajectories and, for each instant, we computed the percentage of them that was lying inside the given boundaries. By using this kind of metrics it is possible to have a precise estimate of how well a given model is performing.

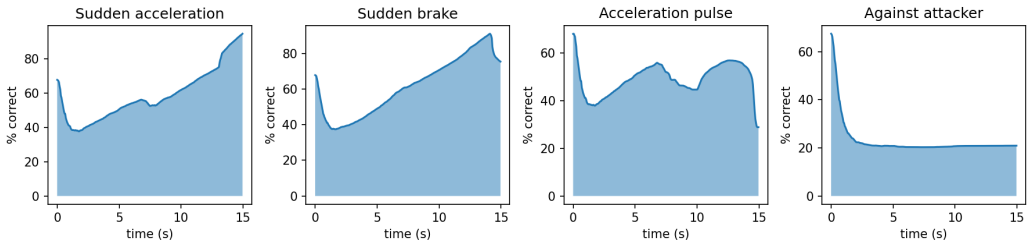

Figure 5.12: Percentage of correct trajectories for each timestep. We can see that the trained model successfully control over the 80% of the scenarios of sudden acceleration or brake. It behaves less well with the acceleration pulse and struggle against the attacker's behaviour. It is evident from the plot that in almost 40% of the cases, the simulation start from forbidden conditions and this should be taken into account to evaluate the model.

It is worth mentioning that the figure 5.12 shows the results of simulations whose initial conditions have been sampled from the training intervals. That is to say that, as we can read from the figure, the 30% of the simulations starts from **unsafe** initial conditions.

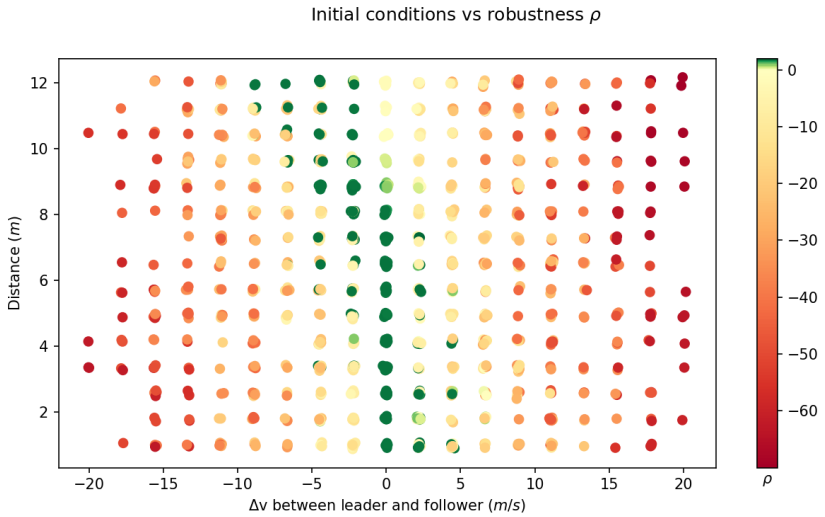

Figure 5.13: Visualization of the simulations' robustness with respect to the sampled initial conditions. The scenario of the simulations is the one in which the leader is controlled by the attacker NN. This figure shows the correlation between the initial conditions and the robustness of the defender NN. It is possible to notice how the defender manages very well the cases in which the two cars have similar initial velocity, given that the initial distance is inside the safety range. The defender can catch up with the attacker even if the latter is further but only if the leader is not going too fast with respect to the follower.

In the figure 5.13 it is possible to notice how the *attacker* is able to exploit favorable scenarios to push the *defender* outside of the safety boundaries. For instance, if the initial distance is already outside the boundaries and the *attacker* is moving at a considerably different velocity with respect to the follower, it is not possible for the *defender* to get to a safe state (as shown in the figure 5.13). The *attacker*'s ability to exploit favorable initial conditions could explain the poorer performances of the *defender* in the simulations in which the leader car is controlled by the *attacker* itself.

One leader and n followers

Given a solution of the problem for two cars, it was straightforward to extend the application to the platooning of n cars. The idea is that each car in the

line should follow the car in front. At this stage it does not matter anymore if the car in front is an *attacker* or a *defender*: the aim of each follower is simply to follow. In such a way it is possible to add an arbitrary number of followers to the line.

The two NNs trained with one leader and only one follower, whose result has been shown above, can be used separately to accomplish such task. In particular, the only leader of the line will be controlled by the *attacker* NN (or any other arbitrary controller), while all the followers will be controller by a copy of the same *defender* NN.

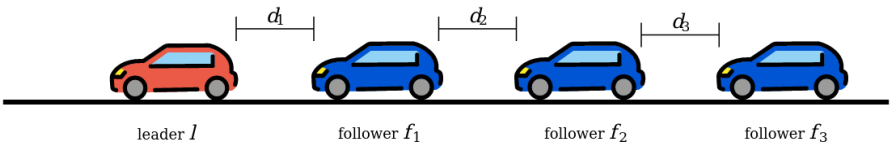

Figure 5.14: The complete platooning setting, each car has knowledge of its status and the distance from the car in front.

In the tested setting there is only a leader l followed by 3 followers f_1, f_2 and f_3. Each follower has to keep the respective distance d_1, d_2 and d_3 with the vehicle in front within the safety range. In such setting, the leader can be controlled by the *attacker* NN or any adversarial policy. The follower, instead, are controlled separately by the *defender* NN.

The vehicles start equispaced with an initial distance $d_{init} = d_1 = d_2 = d_3$ drawn from a $\mathcal{U}(1, 8)$ distribution. Similarly, the initial velocity v_{init} of the cars is drawn from a $\mathcal{U}(1, 5)$ distribution and is the same for all of them.

We performed the simulations for the whole platoon in the same settings used for the basic case. The results are shown in the figures from 5.15 to 5.18.

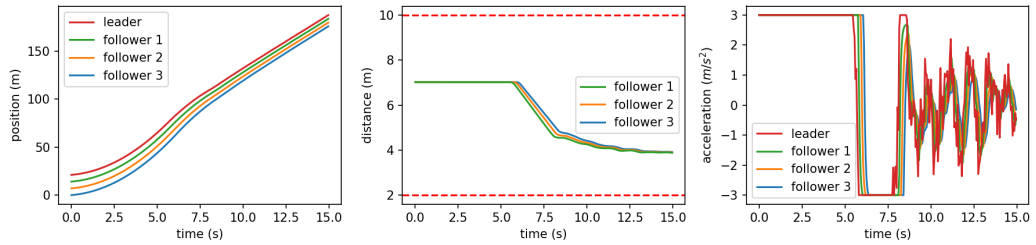

Figure 5.15: Initial configuration: $d_{init} = 7.01\ m$, $v_{init} = 1.30\frac{m}{s}$. *We can see how the attacker tries two strategies (sudden brake and rapid acceleration/brake sequence), but the platoon can still manage the situation safely.*

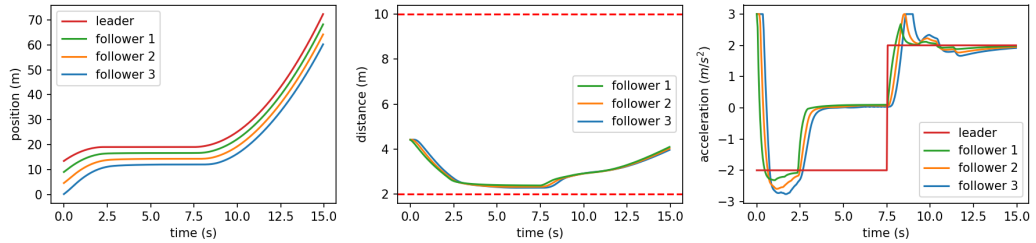

Figure 5.16: Initial configuration: $d_{init} = 4.42\ m$, $v_{init} = 4.99\frac{m}{s}$. *It is worth noticing how the whole platoon stops at some point still keeping a safe distance. It restarts right after and hold the safety distance for the rest of the simulation.*

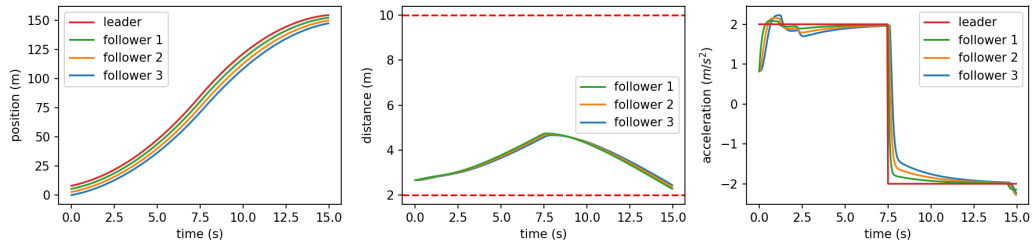

Figure 5.17: Initial configuration: $d_{init} = 2.66\ m$, $v_{init} = 3.01\frac{m}{s}$. *The trajectories are robust despite the sudden brake of the leader.*

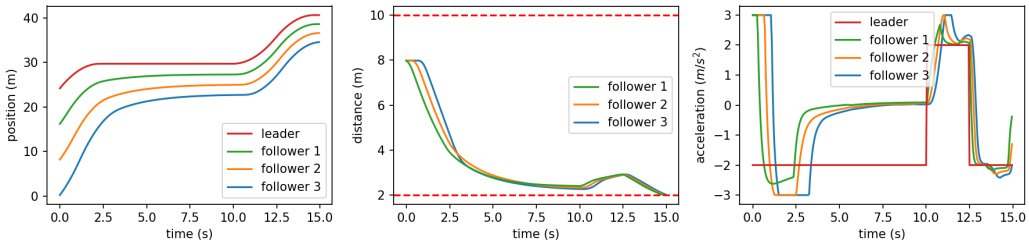

Figure 5.18: Initial configuration: $d_{init} = 7.97\ m$, $v_{init} = 4.96\frac{m}{s}$. *The whole platoon reacts interestingly to the sudden changes. As it would happen in the real world, getting further from the head of the platoon, the accelerations changes appear damped and here it is clearly visible.*

The platoon's followers behave exactly like the follower of the simple case with one leader and one follower. This is not surprising since they all use the same NN as controller.

Chapter 6

Conclusions and future works

6.1 Summary

The potential of the newest models of machine learning in the CPS field is underexploited: while in controlled scenarios many classical control methods are still effective, in open world scenarios they are not always a viable option. Better alternatives like reinforcement learning, though promising, struggle to give an adequate safety guarantee for real-world applications.

We wanted to address such problem drawing inspiration from GANs literature: two opposing NNs that learn from each other. The outcome of the learning procedure is one **defender** NN that is able to face in a safe and robust way the scenarios generated by the **attacker** NN.

We tested our method in two different settings: the traditional **cruise control** problem and the **platooning** one.

6.2 Conclusions

In both the case studies, our architecture behaved well and showed promising results, a potential for further research on the topic.

With the presented architecture we showed a possible way of achieving both safety and interpretability of the model. The *defender*, in fact, can safely overcome most of the presented situations, some of which are indeed unrealistic corner-cases. The *attacker*, on the other side, is able to give insight on which are the most insidious configurations of the environment, allowing to deploy better systems to face them.

The architecture proved itself to be robust even in case of sparse exploration of the space of the initial configuration. Starting for an unknown state, in fact, is almost always able to reach a known and safe state.

The results presented are encouraging.

6.3 Future work

The presented architecture is quite complex and requires a deeper exploration in order to uncover the full potential.

The architecture would greatly benefits of some optimizations to enable the processing of simulations *mini-batches* in parallel.

It would be interesting to try the architecture on different CPS and different problems. A possible future development, in fact, could be to apply this architecture to control non-differentiable models. It could be achieved by training a differentiable NN as a surrogate of the non-differentiable system and use it as model $\mathcal{M}$.

Another matter that requires more attention is the choice of the points on the *hyper-grid* for the training. While at the moment we sample it uniformly, other smarter strategies could be applied since not every configuration is equally probable.

References

[1] Shaveta Dargan, Munish Kumar, Maruthi Rohit Ayyagari, and Gulshan Kumar. A survey of deep learning and its applications: A new paradigm to machine learning. *Archives of Computational Methods in Engineering*, Jun 2019.

[2] Diogo V. Carvalho, Eduardo M. Pereira, and Jaime S. Cardoso. Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8(8):832, Jul 2019.

[3] Steve Howes, Ivan Mohler, and Nenad Bolf. Multivariable identification and pid/apc optimization for real plant application. In *ACHEMA–World Forum and Leading Show for the Process Industries*, 2018.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[5] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[6] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, pages 663–670, 2000.

[7] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv:1406.2661 [cs, stat]*, Jun 2014. arXiv: 1406.2661.

[8] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[9] Teodora Sanislav and Liviu Miclea. Cyber-physical systems - concept, challenges and research areas. *Control engineering and applied informatics*, page 7, Jan 2012.

[10] Rahul Mangharam and Miroslav Pajic. Distributed Control for Cyber-Physical Systems. *Journal of the Indian Institute of Science*, 93(3):353–387, 2013.

[11] M. Adamski, Andrei Karatkevich, and M. Wegrzyn. *Design of embedded control systems*. Springer, 2005.

[12] Davide Calvaresi, Mauro Marinoni, Arnon Sturm, Michael Schumacher, and Giorgio Buttazzo. The challenge of real-time multi-agent systems for enabling iot and cps. In *Proceedings of the International Conference on Web Intelligence*, WI '17, page 356–364, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Christopher Greer, Martin J. Burns, David A. Wollman, and Edward Griffor. Cyber-physical systems and internet of things. 2019.

[14] A. Humayed, J. Lin, F. Li, and B. Luo. Cyber-physical systems security—a survey. *IEEE Internet of Things Journal*, 4(6):1802–1831, Dec 2017.

[15] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, volume 3253 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2004.

[16] Xi Zheng and Christine Julien. Verification and validation in cyber physical systems: Research challenges and a way forward. In *2015 IEEE/ACM 1st International Workshop on Software Engineering for Smart Cyber-Physical Systems*, page 15–18. IEEE, May 2015.

[17] Ram Das Diwakaran, Sriram Sankaranarayanan, and Ashutosh Trivedi. Analyzing neighborhoods of falsifying traces in cyber-physical systems. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ICCPS '17, page 109–119, New York, NY, USA, 2017. Association for Computing Machinery.

[18] T. Yamaguchi, T. Kaga, A. Donzé, and S. A. Seshia. Combining requirement mining, software model checking and simulation-based verification for industrial automotive systems. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 201–204, Oct 2016.

[19] Nauman Sohani, Geunseob Oh, and Xinpeng Wang. A data-driven, falsification-based model of human driver behavior, 2019.

[20] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, 2018.

[21] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

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

[23] B. Widrow, Rodney Winter, and Robert Baxter. Layered neural nets for pattern recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 36:1109 – 1118, 08 1988.

[24] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018.

[25] Adam Gaier and David Ha. Weight agnostic neural networks. *CoRR*, abs/1906.04358, 2019.

[26] Kurt Hornik, Maxwell Stinchcombe, Halbert White, et al. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[27] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[28] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[29] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 116, 2004.

[30] Karol Kurach, Mario Lucic, Xiaohua Zhai, Marcin Michalski, and Sylvain Gelly. The GAN landscape: Losses, architectures, regularization, and normalization, 2019.

[31] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2015.

[32] Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew P. Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi. Photo-realistic single image super-resolution using a generative adversarial network. *CoRR*, abs/1609.04802, 2016.

[33] Deepak Pathak, Philipp Krähenbühl, Jeff Donahue, Trevor Darrell, and Alexei A. Efros. Context encoders: Feature learning by inpainting. *CoRR*, abs/1604.07379, 2016.

[34] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. *CoRR*, abs/1711.11585, 2017.

[35] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[36] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks.

In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.

[37] Huikai Wu, Shuai Zheng, Junge Zhang, and Kaiqi Huang. GP-GAN: towards realistic high-resolution image blending. *CoRR*, abs/1703.07195, 2017.

[38] Ruben Tolosana, Ruben Vera-Rodriguez, Julian Fierrez, Aythami Morales, and Javier Ortega-Garcia. Deepfakes and beyond: A survey of face manipulation and fake detection, 2020.

[39] Valentin Goranko and Antje Rumberg. Temporal logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2020 edition, 2020.

[40] Georgios E. Fainekos and George J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262 – 4291, 2009.

[41] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, Nov 2017.

[42] Alberto Bemporad and Manfred Morari. Robust model predictive control: A survey. In *Robustness in identification and control*, pages 207–226. Springer, 1999.

[43] Ralph R Teetor. Speed control for motor vehicles, August 11 1948. US Patent 2,519,859.

[44] Daniel A Wisner. Speed control for motor vehicles, March 16 1971. US Patent 3,570,622.

[45] Lejla Banjanovic-Mehmedovic, Ivana Butigan, Fahrudin Mehmedovic, and Mehmed Kantardzic. Hybrid automaton based vehicle platoon modelling and cooperation behaviour profile prediction. *Tehnicki vjesnik - Technical Gazette*, 25(3), Jun 2018.

[46] Dongyao Jia, Kejie Lu, Jianping Wang, Xiang Zhang, and Xuemin Shen. A survey on platoon-based vehicular cyber-physical systems. *IEEE Communications Surveys & Tutorials*, 18(1):263–284, 2016.

[47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[48] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network, 2015.