Performance exploration of the REST API using online GAN

Cédric Léonard - 2100775
Master's Thesis in computer science
ÅAU Supervisor: Ivan Porres Paltor
INSA Supervisor: Wassim Hamidouche
Faculty of Science and Engineering
Double Degree INSA Rennes - Åbo Akademi University
2022

Abstract

Performance Testing is a critically important step in the application development process. Manually achieving performance exploration comes with high human-cost and, in general, inefficiency. Software testing automation is a research area with many contributions which observes frequent advancements. In the context of the SBST 2022 CPS tool competition, Åbo Akademi University laboratory has published a new Machine Learning technique that merges a GAN-based model and the online learning process: Online Generative Adversarial Network (OGAN).

This thesis approached the performance exploration problem using the OGAN algorithm on particular applications: REST APIs. This thesis sets the required context and develops the information background concerning: REST APIs principles, the Performance Testing process and the usage of Deep Neural Network, specifically Generative Adversarial Networks (GANs), in an online learning approach. The Machine Learning technique tackled: OGAN is presented and its algorithm detailed and discussed. Finally, an application of this technique is proposed and explored on the PetClinic REST API.

Keywords: Performance Testing, RESTful APIs, Automation, Generative Adversarial Network, Online Learning.

Preface

The desire for this research work has originally come from my long-time curiosity about Machine Learning and its possible applications. Today, Neural Networks are common and popular tools used everywhere, however, they are still seen as an obscure corner of Computer Sciences. I wanted to strengthen my knowledge about this domain and, also, participate in the effort of making it more accessible to novices.

This lengthy work was possible with the teamwork of our institutions, Åbo Akademi University and INSA Rennes. I would like to thank my supervisor, Ivan Porres Paltor, who helped me find my subject and initial research leads. I also express my appreciation to Wassim Hamidouche, my supervisor at INSA Rennes for his support. I am also indebted to Sébastien Lafond, my tutor at Åbo Akademi for my double degree. Lastly, many thanks to my friends for their advice, reviews and support during hard times.

Contents

1 Introduction **1**
1.1 Problem Formulation 2
1.2 Outline 3

2 RESTful APIs **5**
2.1 REST principles and architecture constraints 6
2.2 REST API operating way 7
2.3 Challenges 9

3 Performance Testing **11**
3.1 Performance Testing: A definition 11
3.2 The importance of Performance Testing 12
3.3 Performance, Load and Stress Testing 13
3.4 Operating performance tests 14
3.5 Automation in Performance Testing 15
3.6 Website Performance Testing 17
3.7 Automated performance exploration 19
3.7.1 Genetic Algorithm approaches for Load Testing 19
3.7.2 ML-assisted Performance Testing framework 21

4 A Machine Learning technique: Generative Adversarial Networks **24**
4.1 Deep Neural Networks, a brief presentation 25
4.1.1 The simplest Neural Network: the Perceptron 25
4.1.2 Layers and Depth: from a single neuron to networks 28
4.1.3 Customizing networks: activation functions 31
4.1.4 Training and Learning: database importance 33
4.2 The Generative Adversarial Networks concept 35
4.2.1 Generative modeling 35
4.2.2 Generate and discriminate: a zero-sum game 37
4.2.3 Challenges and Wasserstein variant 39

5 Performance testing using OGANs **41**
5.1 Online Generative Adversarial Networks 41
5.1.1 A GAN approach applied to system testing 41
5.1.2 Online training and OGAN algorithm 42
5.1.3 Benefits and open problems 44
5.2 Online learning and GANs in Performance Testing 45
5.2.1 Software online performance exploration using a Discriminator Network 45
5.2.2 Online GANs for automatic Performance Testing 47
5.2.3 Wasserstein OGAN for Cyber-Physical Systems 48

6 Application of the OGAN technique to performance exploration of REST APIs **52**
6.1 REST API black-box Performance Testing 52
6.2 Adaptive function in between the OGAN and the API 53
6.2.1 PetClinic requests formats 53
6.2.2 Algorithm presentation 55
6.2.3 Discussion and generalization 57
6.3 OGAN setup and parameters 59
6.4 Evaluation of the OGAN REST API performance exploration technique . 61

7 Future Work and Conclusion **64**

List of Figures

2.1 REST API model 6

3.1 Types of Software Testing 12
3.2 RESTest framework workflow 18
3.3 Genetic Algorithm workflow 20

4.1 Neuron growth and connections over time 25
4.2 Schema of a Perceptron 26
4.3 Perception space partitioned with several neurons 28
4.4 Perception space partitioned with one layer 30
4.5 Perception space partitioned with multi-layers 30
4.6 Examples of Supervised Vs Unsupervised Learning workflows 36
4.7 Basic model of a GAN architecture 38

5.1 OGAN algorithm workflow 47

6.1 Entity-Relationship model of the PetClinic REST API 54

List of Tables

2.1 HTTP status codes categories and descriptions 9

Listings

6.1 **GET** request listing every pet 53
6.2 **PUT** request modifying information of a pet 53
6.3 JSON setup object for the OGAN algorithm 59

Acronyms

API Application Programming Interface.

CPS Cyber-Physical System.

DN Discriminator Network.

DNN Deep Neural Network.

GA Genetic Algorithm.

GAN Generative Adversarial Network.

HTTP Hypertext Transfer Protocol.

ML Machine Learning.

NN Neural Network.

OAS OpenAPI Specification.

OGAN Online Generative Adversarial Network.

REST REpresentational State Transfer.

RL Reinforcement Learning.

SBST Search-Based Software Testing.

SUT System Under Test.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

WGAN Wasserstein Generative Adversarial Network.

WOGAN Wasserstein Online Generative Adversarial Network.

Introduction

Today, most companies, associations or entities possess or work with web-based applications. Web-based applications are becoming mission-critical to most private and governmental organizations. The software business is a well developed, valuable and active world.

Most modern web applications expose an Application Programming Interface, or API, that clients can use to interact with the application content. The vast majority of website APIs are based on a RESTful design. REpresentational State Transfer, or REST, is an architectural approach to design web services. This well-known architectural style has the advantage of providing a great deal of flexibility allowing APIs to meet developers' and diverse customers' needs.

Furthermore, the ever-increasing need for high-quality web applications implies methods and tools of equally high quality to ensure that an application works as expected. Therefore, testing has become one of the main activities in software development. Testing is one of the most important steps for evaluating a software application's functionality. However, the success of an application is not limited to its functional aspects, as many cases have shown that even a perfectly functional application can fail [1].
A software system quality is intrinsically linked to end users' satisfaction and, thus, also closely related to non-functional requirements, such as performance. Even though Performance Testing is highly important, it is often still underestimated [1].

Performance is a non-functional property indicating the operational efficiency of a software system. Performance Testing is realized with respect to different execution conditions, such as various types of workload and allocation of available resources.
The continuously increasing usage of web-based applications makes performance an essential quality attribute in determining that the system accomplishes its functionality as it is supposed to. User's end satisfaction critically depends on responsiveness or efficiency, e.g., a slow or inefficient website will receive poor feedback. Satisfying these

non-functional requirements in terms of performance can lead to customer satisfaction improvements and may, consequently, increase business profits.

Ensuring software quality through manual Performance Testing, which requires human resources, is time-consuming and prone to error. Therefore, automated testing methods have been developed to detect performance faults and bottlenecks in applications. Even though classic approaches and algorithms perform well in this task, the glowing recent past of Machine Learning (ML) has created a focus in the development of automated Performance Testing methods through ML. These techniques are used in autonomous Performance Testing tools or frameworks. However, most of the existing solutions may not generate efficient test sets for Performance Testing [2].

In the context of automated test generation, Åbo Akademi University laboratory recently published a new ML technique. This algorithm has been shown to be very promising in the Search-Based Software Testing (SBST) 2022 Cyber-Physical System (CPS) tool competition. This competition deals with finding road scenarios that cause the Artificial Intelligence (AI) of the BeamNG.tech [3] driving simulator to drive a car out of its designated lane. The model submitted by I. Porres, F. Spencer and J. Peltomäki is named Wasserstein Online Generative Adversarial Network (WOGAN) [4].

1.1 Problem Formulation

Although Performance Testing is one of the most important testing methods, it still encounters many challenges. Most of these challenges are found in different automated Performance Testing techniques that need access to the System Under Test (SUT) model or source code. This is known as white-box testing. Using SUT model analysis, source code analysis or user behavior patterns is also known as clear or open-box testing, as the internal structure of the system is available. This is an effective approach that most of the current flourishing testing techniques rely on.

However, the source code is not often available for web-based systems, it is thus necessary to resort to black-box testing techniques. Black-box approaches leverage the SUT specification to automatically derive test cases from it. It can be seen as an external or end-user type perspective testing approach.
Most of Performance Testing techniques rely on white-box approaches and, therefore, on different artifacts such as SUT model analysis, source code analysis, or user behavior

patterns. These approaches might be high in costs and focus on testing the logic rather than the behavior of the SUT. However, if black-box testing approaches are less time-consuming and do not require implementation knowledge, generating optimal test sets is a complex optimization problem. Unlike white-box techniques, it is difficult to efficiently and automatically generate well-suited test cases. Thus, ML-based methods and optimizing search techniques like the WOGAN approach can handle and address the optimization problem.

For this reason, this study aims to investigate and evaluate a performance exploration method based on OGAN. This technique will serve as a solution for efficiently generating tests that reach performance breaking points for a SUT without accessing any of the mentioned artifacts (i.e., black-box testing).

In addition, this study will review related techniques, evaluate and compare them with the OGAN method. This state-of-the-art literature study will be conducted in the domain of software testing, especially in RESTful API Performance and Stress Testing. This review will also include modern automated tests generation ML-based techniques.

Finally, an evaluation of this method application will be realized. This evaluation will include a presentation of an adaptive method in between the OGAN testing tool and the SUT. This presentation will describe a possible algorithm used to adapt the GAN output to an endpoint call in the API.

The objectives of this thesis are described as follows:

- Evaluate how it would be possible to realize automated Performance Testing on REST APIs using OGANs;
- Review and compare related work and literature;
- Present a first method adapting the OGAN testing tool to the SUT;
- Expand and theorize about future work.

1.2 Outline

This thesis is organized into seven chapters. The first three chapters present the background and context of the thesis. Chapter II introduces the knowledge required about REST APIs. Chapter III presents the ML technique used in this thesis. This presentation

introduces from scratch the notion of Neural Networks (NNs) and explains the concept of Generative Adversarial Networks (GANs). In Chapter IV, an introduction to Performance Testing and its variations is given. This chapter highlights the research problem and its questions, and also presents related works. Chapter V describes the online variant of the GAN algorithm that will be used in this thesis and presents the articles affiliated to this thesis. Chapter VI outlines and motivates an application of the OGAN technique to the addressed problem. Finally, Chapter VII concludes the thesis with some discussions and future work.

RESTful APIs

The ever-growing demand of web-based applications has led to an omnipresence of web services. A web service is a software system allowing machine-to-machine interactions over a network. For a web service to be completely functional, it has to be able to answer requests from other services. Such a feature implies that web services have to expose and maintain an interface in between the service sending requests and their back-end data. This interface facilitates the interaction between two distinct software services, the Client and the Server, and is called an Application Programming Interface (API) (cf. Figure 2.1).

An API is a set of rules that defines how applications or devices can connect to and communicate with each other. An API can be seen as composed of two fundamental elements: a technical specification establishing how information can be exchanged between programs, and a software interface publishing that specification. The technical specification is a set of routines, functions and/or commands, whereas the interface is the explanation of how the technical rules should be used and what can be achieved with them. Hence, APIs allow one software service, the Client, to access data from another software service, the Server, without the developer needing to know how the other service works. OpenAPI Specification (OAS) is a description language that can be used to write APIs specification interfaces. It is sufficiently readable to allow both humans and computers to discover and understand the capabilities of a service without accessing the code.

In order to behave consistently and predictably, an API should follow a design. Declaring that an API follows a specific design improves the user experience. Following a specific design implies that any decision, made during the planning and architecture of the API, is consistent with the respectively specific constraints and pursues well-defined objectives. Therefore, following an API design influences how well developers are able to consume and use this interface. Several designs exist, all focusing on different aspects and having different constraints. However, the most popular API design is the REpresentational State Transfer (REST) architectural style.

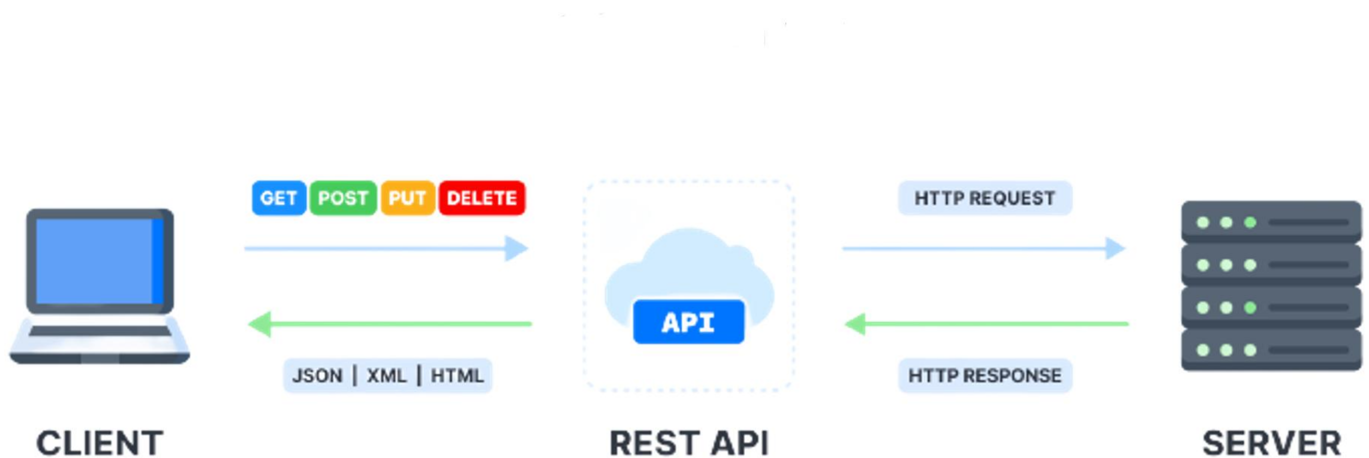


Figure 2.1: REST API model

Source: R. Walker, AppMaster, "How to create an API", *https://appmaster.io/blog/how-create-api-no-code*, Feb. 2022 (accessed March 2022)

2.1 REST principles and architecture constraints

A REST API is an API that conforms to the design principles of the REST architectural style. For this reason, REST APIs are sometimes referred to as RESTful APIs. The REST architectural style was first presented in 2000 by Roy Thomas [5] and involves six constraints or guiding principles.

The first principle is generality, also called the Uniform Interface. It aims to simplify the overall API architecture and to improve the visibility of interactions. The interface should provide a uniform way to identify and access resources. Resources are the basic building blocks in a REST API. For example, an online book store application will include one resource for a book, one for a client, another one for an order, etc. The Uniform Interface constraint states that, in a RESTful API, resources must be uniquely identified and contain every piece of information that the user might need, without being excessively large. Request formats for similar resources should look the same, in order to facilitate consumer's usage.

The second main principle is named Client-Server decoupling. In web applications, the system owning the data and providing the service is the Server, whereas the entity using the service and requesting the data is the Client. The decoupling between these two entities is the second constraint stated by the REST architectural style. The client application should only know and access the server through Uniform Resource Identifiers

(URIs) and vice versa. This separation of concerns allows the two components to evolve independently. URI and Uniform Resource Locator (URL) are ways to identify and locate web resources; they will be further explained below.

Another main constraint for REST APIs is statelessness. This principle remits that each request from the client to the server must contain all the information necessary to understand and complete the request. No context information storage can be made on the server side. This implies that during a session, corresponding to a list of requests and client-server communications, the state of the session must be kept entirely in the client application. Hence, every client request has to be independent from the previous ones and include all information necessary for the server to fulfill the request. This constraint makes REST APIs very convenient to test.

The three next and last REST guiding principles are less relevant for this thesis. The first one is Cacheability: when possible, resources should be cached, stored to be retrieved and used later. This operation should be made in order to save a new request to the server and to improve performance on the client side.
Then comes the layered system architecture. REST APIs architecture may possess different layers, hence, the client and server applications may not be connected directly to each other. There may be a number of different intermediaries in the communication loop. The layered system architecture constraint states that the interface needs to be designed in a way that neither the Client nor the Server can tell whether it communicates with the end application or an intermediary one. Thus, the overall security of the API is improved. It also makes the API maintainable.
And finally, the Code on demand principle: when responses contain executable code, this code should only run on demand. This last constraint is optional, as it may represent a high security risk and not all APIs need the flexibility offered by the Code on demand. On the upside, it can help the Client to implement its own features on the go, with less work being necessary on the API or Server. In essence, it allows the whole system to be much more scalable and agile.

2.2 REST API operating way

Routinely using a massive network of web pages, programs and files (Internet) requires communication. These World Wide Web communications are conducted under a protocol to ensure that any stakeholder on the network can understand them. An analogy is to

make sure that two interlocutors speak the same language.

APIs and REST APIs also use communications. Clients and Servers exchange representations of resources by using a standardized interface and protocol. This protocol could be any protocol as long as the stakeholders agree on its definition and implications. However, the most common internet protocol is Hypertext Transfer Protocol (HTTP) or its encrypted version: HTTPS, where the S stands for secure.

The HTTP protocol works with requests and answers. A request is a message sent by the Client to the Server. This message contains a Request-line, some optional Headers fields and an optional Message-Body. The Request-line contains the main information. It states a request method and a URI which identifies the resource upon which to apply the request. It also includes the version of the HTTP protocol. The Request-Header fields allow the Client to pass additional information about the request, and about the Client itself, to the Server. The Message-Body is for requests that require more information than the method name and the URI alone, such as modification request. For example, when modifying the name of a book, the new name of the book needs to be stated in the Message-Body.
The data transmitted in the Message-Body also has to match a format. A common data format for HTTP requests and answers is JavaScript Object Notation, or JSON. JSON is a standard format used to represent structured data with text. It uses an attribute-value pair system and is easily readable by humans. An example of a JSON object is given Chapter VI, see Example 6.3.

An HTTP request is often associated with its method name. These methods are also sometimes called verbs, because they explicitly refer to an action: POST to create a resource, GET to fetch and read it, PUT to update it and DELETE to erase it, see Figure 2.1. These methods are the most common ones and often referred to as CRUD (Create, Read, Update and Delete).
Additional methods exist, such as CONNECT, OPTIONS or PATCH. PATCH is especially useful, as it allows updating a resource, but without having to specify every field of the resource as is the case with PUT.

When an API requests to access data from the Server, a response is always sent back. The location where the API sends a request and where the response emanates is an endpoint. An endpoint is the name given to the location from which Clients can access the resources they need to carry out their function. Simply put, as APIs allow two systems

to interact with each other, endpoints can be considered as the points of contact of these communications. Endpoints are paths to resources and are often URLs followed by the ID of a resource instance.

When a Client submits an HTTP request to a Server, the latter processes the message and sends back a response. Just as the request, the response contains different information, such as the HTTP protocol version. Simultaneously, the HTTP response provides a status code, indicating whether the request was successful or not, and a status message, a non-authoritative short description of the status code. In addition, some HTTP Headers and a Message-Body containing the fetched resource may optionally also be returned with the response.
The status codes represent the response main information. A status code is a 3-digit number reporting major information, such as the success of the request, if any more steps are required or, in case of an error, what the problem was. Conventionally, the first digit of the status code characterizes the class of response, see Table 2.1. Therefore, with the complete status code, the Client knows all the information he can need about his request, even in case of failure. Status codes participate in fulfilling the Statelessness and Uniform Interface constraints.
For some examples, the well-known *Error 404* refers to a response containing a status code 404 and means that the Server could not find the requested instance. Compared to that, a 201 status code means that the request was accepted, completed and that a new resource was created. As a last example, a 301 status code reports that the resource was moved permanently and is accompanied by a new URL where to find the resource.

Table 2.1: HTTP status codes categories and descriptions

Code	Class of response	Description
1xx	Informational	The request has been received and the process is continuing
2xx	Success	The action was successfully received, understood and accepted
3xx	Redirection	Further action must be taken in order to complete the request
4xx	Client error	The request contains incorrect syntax or cannot be fulfilled
5xx	Server error	The server failed to fulfill an apparently valid request

2.3 Challenges

The REST architectural style is convenient for different factors, for example its independence of any underlying protocol (although the most-common REST APIs use HTTP as

the application protocol). However, REST design also encounters different challenges.

In order to fulfill the Uniform Interface constraint, the paths of the endpoints should be consistent by following common web standards, which may be difficult to manage. These paths must be consistent for the different resources, but also support the versioning of the API. Indeed, resources may be added, deleted, regrouped or simply renamed. In these cases, endpoint management becomes a difficult task, as the methods and routines have been designed around the previous API version. After or during a REST API update, the URLs of the endpoints should not be invalidated when used internally or with other applications.

APIs can be extensive and contain a massive amount of resources. More resources entail more endpoints and, therefore, more methods and functions. For this reason, APIs can be very deep and exhaustive. APIs with a vast architecture also imply that, in order to respect Statelessness and return every piece of information the Client may need, resources become considerably large and contain plenty of fields. This amount of resources and the increasing size of resources add to increased load and response times. Hence, excessive data is convenient, but may also degrade end users' experience.

The excessive data of APIs also brings out a new challenge: API testing. API testing can be a long process to set up and run. Because of the amount of resources and endpoints, the number of possible requests and responses increase exponentially. This increase, added to the necessity of exhaustive testing, has made the challenge of RESTful API testing a popular and ongoing topic that has inspired much work and research.

Performance Testing

The process of application development goes through multiple steps. One of these steps is the Testing stage. During this phase, developers will conduct different experiments on the product to identify functional issues and correct them. However, this step also includes some non-functional verification.

In application development, functional requirements correspond to what the system is designed for: its purpose. Opposed to that, non-functional requirements could be considered as a quality attribute: how the system should perform. In a strictly operating view, in order to work, a system does not have compulsory non-functional requirements. However, this is never true in reality. Non-functional norms are extremely important as they define the quality of the product.

3.1 Performance Testing: A definition

Performance Testing is the name of a non-functional software testing technique which focuses on the system performance. Testing the non-functional requirements of an application is equivalent to testing the global way the system operates. This objective contrasts with the purpose of other more famous testing techniques such as Unit Testing or Integration Testing. These methods are functional testing techniques and aim to verify that every function and functionality of the application complies with the given requirement. For a global view of the software testing sphere, check Figure 3.1.

The previous and simple definition already underlines that the Performance Testing phase has to be done once the system is already functional, many would understand the system to be finished instead of functional. This explains that Performance Testing is still mainly unknown or underestimated, and is too often missing in many projects [1].
Among the non-functional aspects of a system, Performance Testing focuses on the performance and, therefore, tries to determine how the stability, speed, scalability and responsiveness of an application holds up under a given workload.

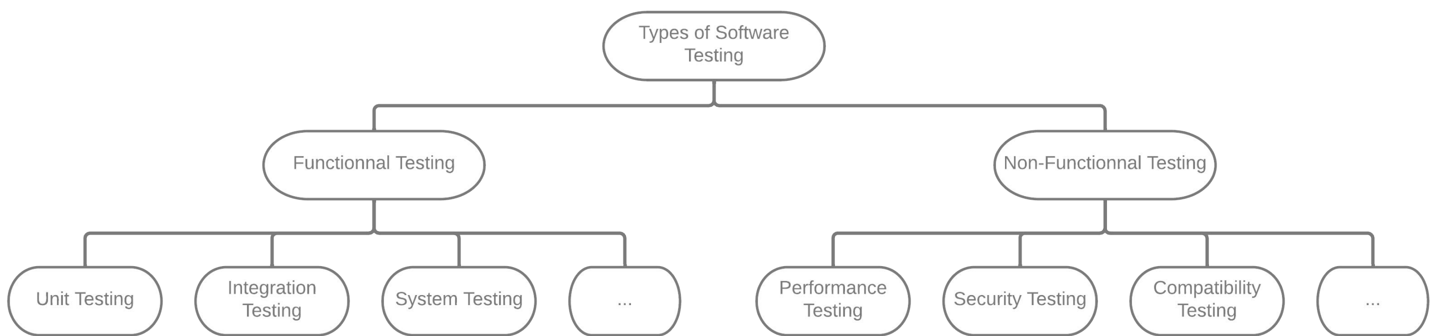


Figure 3.1: Types of Software Testing

3.2 The importance of Performance Testing

Many of the following explanations come from [1], in which Ian Molyneaux states that executives do not appreciate the importance of Performance Testing. He, therefore, writes a commonsense guide about application Performance Testing.

Performance Testing is done to ensure that an application is fast, stable, and scalable for users. More importantly, Performance Testing uncovers what needs to be improved before the product is released. Without Performance Testing, an application is likely to suffer from several issues, such as running slowly while several users use it simultaneously, inconsistencies across different operating systems or poor usability. By simulating traffic and concurrent users, software testers can see how the application responds and then identify bottlenecks in both code and infrastructure.
Realizing performance tests during the development of an application also allows to track the progress of the project. Recording and comparing the results of the performance tests at several stages of the development helps to identify scaling-up bottlenecks or to notify an unplanned compromise, e.g., a new feature slowing down an old one.

An application actually refers to the sum of a whole, since it consists of many different components. It is often referred to as the application software plus the application landscape. The latter includes the servers required to run the software, but also the network infrastructure that allows all the application components to communicate.

3.3 Performance, Load and Stress Testing

Performance Testing, Load Testing and Stress Testing are three terms which are used in many cases interchangeably [6]. In general, Load Testing has been considered as a behavioral assessment of a SUT under load expected in a real-world execution context, from two perspectives of functional problems and violation of non-functional requirements caused under load. In this context, the load of the application can be understood as the quantity of requests made to the application, thus, the number of actions it has to complete. For example, in the case of a website, the load can be represented by the number of simultaneous users. In software testing, this load is often simulated via automated tools.
Stress Testing has been defined as the behavioral assessment of a SUT under extreme conditions, including heavy load and limited available resources. Stress testing measures a software on its robustness and error handling capabilities under extremely heavy load conditions.
Performance Testing is often considered as a more general term which often includes both Load Testing and Stress Testing. There are many commonalities between these types of testing.

There is a slight difference between Performance Testing techniques and the aspect of performance exploration studied in this thesis. Indeed, the performance, load and stress techniques aim to identify performance bottlenecks in the application and its environment. Performance-related testing methods objectives can be summarized as follow:

- Measure the performance metrics under different execution conditions, such as various workloads or resource allocations;

- Detect violations of non-functional requirements under expected and stress conditions;

- Potentially detect functional problems appearing under certain execution conditions, i.e., under a specific workload or resource configuration.

However, what is referred to as performance exploration is more about exploring and building a map of the SUT software and environment. It is not a question of executing and testing the SUT under extreme or expected workloads, but more about exploring every aspect and feature of the SUT. Performance exploration allows to measure different scenarios and to find the ones with the worst, or the best, performance.

3.4 Operating performance tests

Performance Testing is the process of running a specified set of scripts that emulates customer behavior at different load levels and takes measurements of the SUT performance for each of the load levels. From this execution of tests, results are collected. These results can be response times, error rate and codes, throughput (i.e., bandwidth or amount of data per second), hardware utilization (i.e., CPU and memory), etc. From these results, reports are written and can then be used and analyzed by software testers.

Performance Testing requires access to the SUT. However, the quantity of information available can vary. If the performance tests are conducted by the same team developing the application, it is likely that the source code and its structure are available. However, it is rarely the case and it often happens that, from the SUT to be tested, we only know the input and output spaces, i.e., what can be inputted into the system and what it outputs. In that case, when the source code is not available, the system acts like a closed box: the internal workings of the system are not accessible, it is a black-box testing method. Opposed to black-box testing, when the code and its structure are available, it is called white-box testing. This technique is also known as clear-box testing, structural testing or even code-based testing.

Analyzing the performance of a SUT can be done through two techniques. It is possible to build a performance model of the system. The objective of performance modeling is to build a static model of the system in order to identify proper performance indices. Such an approach can give helpful hints about the bottlenecks and allow developers to easily fix them. However, it is a high-cost method as building a model of a system is directly linked to the complexity of the system. Therefore, very complex systems may not be suitable for this approach.
Moreover, modeling a system means skipping some details, such as the deployment environment. It is impossible to predict the exact deployment conditions (e.g., workload, number of users, server's state). In performance modeling, shortcuts and approximations are made, so that the model may not be completely representative of the system and its environment. Performance modeling is an efficient technique to find bottlenecks, but it can be very costly, especially when the model is realistic.

The other approach is more straightforward and will be simply referred to as Performance Testing. This approach consists in directly executing the SUT under different conditions, it can be done dynamically and is suitable with a black-box testing approach.

This technique gives less information about the bottlenecks as the system is tested from an external point of view. Therefore, the internal conditions causing bottlenecks can be more difficult to identify than with a performance model of the SUT. This also means that testing different levels of workloads or the limited resource availability can be more complicated. However, this technique is almost effortless as it does not require any modeling.

3.5 Automation in Performance Testing

Every previously described approach can be completed manually. However, writing test cases manually can be prone to error, high in human-costs and, in general, inefficient, as it is difficult to find good test cases. For all these reasons, various ML techniques have been tested and many researchers use algorithms to generate realistic test cases [6, 7, 8]. Test generation problems can be approached as optimization problems. Search-Based Software Testing (SBST) is the application of optimizing search techniques, e.g., Genetic Algorithms (GAs), to solve problems in software testing. These techniques are used to perform test generation, reduce human oracle cost, verify software models, validate real time properties, etc.

To perceive the test generation problem as an optimization problem, there is need to determine an optimization objective. In the Performance Testing case, the goal is to generate tests that highlight problems in the system. Therefore, the optimal test is the one that implies the worst performance for the SUT.
Considering the SUT as a function that takes inputs and produces a result, then the goal of the optimization problem is to find the input, i.e., test case, that creates the worst performance, i.e., result. The tests exceeding the performance requirement, i.e., the fixed limit that, once overtaken, makes the application being considered as faulty, are called positive test. On the contrary, tests not revealing a performance issue, that should not be generated by SBST techniques, are called negative tests.

On a more general matter, optimization problems can be very complex and, therefore, difficult to solve with exact algorithms. This leads to the use of heuristics to produce near-optimal solutions efficiently. An heuristic is a technique designed to solve a problem quicker than classic methods. It trades completeness, accuracy and precision for speed. Heuristics are often used when finding the exact solution is impossible or unrealistic. In a way, it can be considered as a shortcut. Techniques using heuristics, such as GAs or greedy algorithms are among the best performing techniques.

Manual performance testing requires the following steps: construction of the test environment, defining the performance metrics and requirements, designing and planning the test cases, writing the performance test scripts, executing the scripts and reporting and analyzing the results [9]. Automated Performance Testing could allow to stop the human-costs after the second step. Creating the test environment and defining the performance metrics and requirements would still be required to test the desired aspects. However, designing, implementing and executing test cases can be done automatically. Even the result analysis and the reports can be generated.

Automated Testing is one of the most up-to-date research subjects. When talking about Performance Testing, we can easily define this problem as an optimization problem [10], see Equation (3.1). We model the SUT as a function S that accepts inputs from a nonempty set I. We also assume that there is a performance instrument m that, for a given system output $S(i)$ with $i \in I$, provides a measurement $m(S(i))$ corresponding to the performance of the system for the input. Based on this, the overall goal of Performance Testing is to find what inputs yield a performance greater than a given threshold p_m:

$$I_p = \{i \in I : m(S(i)) \geq p_m\} \tag{3.1}$$

We call I_p the set of positive tests of the system, i.e., the system inputs that reveal a performance issue. Likewise, we can define $I_n = I - I_p$ as the set of negative inputs.
Automated techniques should generate tests belonging to I_p. However, there is a need to execute these tests on the SUT to prove that they match the generation requirements. Executing test cases on the SUT is an expensive operation, consequently the techniques should try to use the SUT the least. In practice, a budget n is given to the method, this budget defines the maximum number of test cases that can be executed on the system. Therefore, the objectives of a ML technique on a performance test cases generation problem are the following:

- Generate test cases with performance greater than p_m, i.e., effectiveness;
- Explore the input space enough to find different positive tests, i.e., diversity;
- Execute and measure tests on the SUT as little as possible, i.e., efficiency.

3.6 Website Performance Testing

The following literature review was conducted by looking for articles on *dblp* and *Google Scholar*. The keywords used were "Performance Testing", "Performance exploration" or "Load testing" as well as "Website", "REST API" or "Web application". The following sections also look for articles with the "Automation", "generation" and "Machine Learning" keywords. All the selected articles were published in the last 10 years.

The first objective of this thesis concerns the performance exploration of websites, or more specifically REST APIs. This research area has many contributions and many different website Performance Testing tools and services already exist.

Hung Kao *et al.* have created a Performance Testing framework presented in [9]. This tool provides software testers an integrated process from test cases design and scripts generation to test execution. The test generation process is based on the test cases designed by software testers, but also on some appropriate artifacts such as the API specification. From the generated test designs, the framework creates a corresponding performance test script. This test script can then be executed by one of the specific Performance Testing tool included in the framework.
To help software testers define test case designs, the framework also provides a test case template. Software testers can use these test templates and provide the framework with further information, such as an action set (the series of action that should be tested, for example login, download a file, etc.) or the workload related characteristics (e.g., anticipated concurrency, load duration).

In [2], Martin-Lopez *et al.* present an open source black-box testing framework for RESTful web APIs named RESTest. The framework uses the API specification and different testing techniques to generate realistic test cases. A test represents a single HTTP request, which allows to access and check one of the API endpoints. The techniques used in RESTest include fuzzing, adaptive random testing and constraint-based testing. Using several techniques, all of them more elaborated than random testing, allows an efficient testing of APIs requiring semantically complex data or answering potential input constraints. RESTest also uses custom test data generators to create realistic data, such as email addresses or strings matching a regular expression.

The steps followed by the workflow of RESTest, Figure 3.2, are the following. The framework first generates a test model from the API specification, also called system

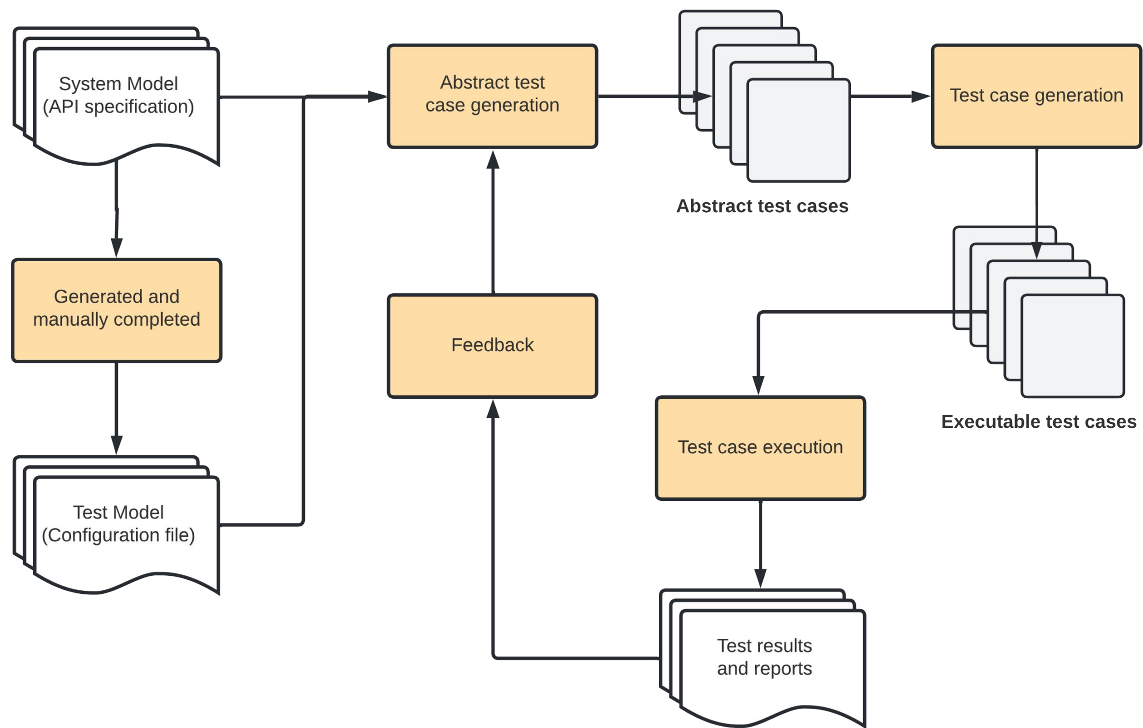


Figure 3.2: RESTest framework workflow

model. This test model is a configuration file that can be manually completed with further information, such as API keys or parameter data types. Abstract test cases are then generated from this model, using the different testing techniques. Finally, these abstract test cases are instantiated into executable test cases and executed. A feedback step is also present and allows the test generators to create more elaborated test cases. However, [2] future work section mentions the will to integrate SBST techniques to optimize the test generation process. Without this automatic improvement method, the user must directly modify the configuration file in order to improve the test coverage or efficiency. That makes RESTest a mainly autonomous testing framework, but not an automated test generator.

The frameworks presented in [9] and [2] aim to support software testers and to make Performance Testing a more efficient and straightforward process. The presented works act as support tools, meaning they cannot accomplish the testing process as a whole, and thereby replace the software testers. These tools still require developers to specify the performance tests awaited or, for the most modern tools, the direction the tests should take.
However, sometimes, issues or performance faults come from unexpected events or overlaying requests. Therefore, this type of tool assisted testing does not cover all possibilities,

as it is still driven and guided by humans, and thus vulnerable to cognitive errors. A possible evolution would be to entirely generate the test cases with an algorithm, to let the model explore the possible input space. This could lead to tests that are unrealistic, but also to scenarios that software testers would not have thought of.

3.7 Automated performance exploration

The previous section has presented tools to assist software testers during REST API Performance Testing. However, when it comes to testing or any other recurrent task, the question about automation emerges. Therefore, this section will review implementations of automated performance test generation through ML.

3.7.1 Genetic Algorithm approaches for Load Testing

The work accomplished by E. Isaku in [8] proposes and evaluates evolutionary search-based methods. These methods are multi-objective optimization approaches and are used for Load Testing in softwares. Evolutionary algorithms are part of the heuristic family of algorithms and are based on natural selection processes. The thesis presents and evaluates four of them:

- Non-dominated Storing Genetic Algorithm II (NSGA-II) [11];
- Pareto Archived Evolution Strategy (PAES) [12];
- The Strength Pareto Evolutionary Algorithm 2 (SPEA2) [13];
- Multi-Objective Cellular Genetic Algorithm (MOCell) [14].

Except PAES, all these evolutionary algorithms are GAs. GAs are the most common evolutionary algorithms and are used to generate high-quality solutions for optimization and search problems. These solutions are obtained through biologically inspired mechanisms.

GAs re-interpret natural evolution processes, such as crossover, mutation and selection, in order to evolve and enhance their performance over time. Figure 3.3 presents the basic workflow followed by GAs. GAs start by randomly creating a population of individuals, with each individual possessing different character traits, i.e., genes, randomly chosen, and then iteratively performing the following sub-steps on the population until the termination criterion is satisfied:

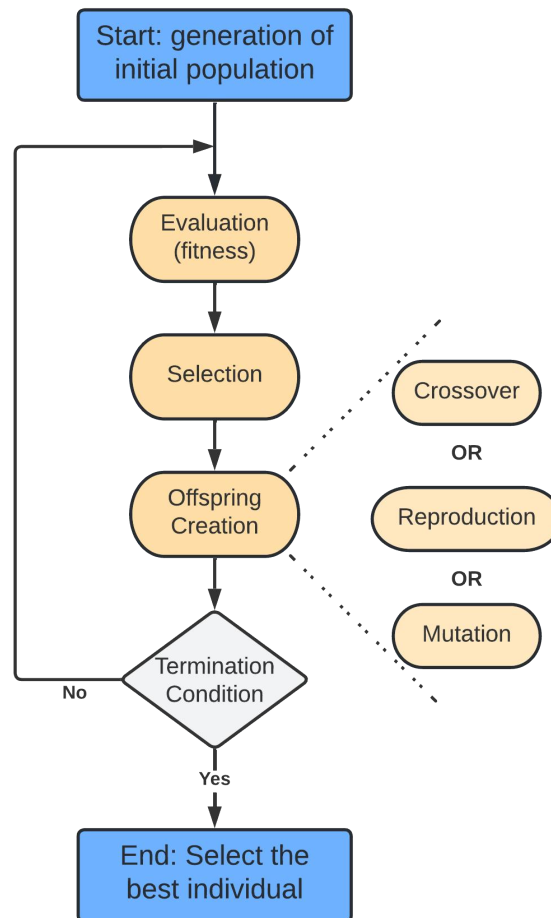


Figure 3.3: Genetic Algorithm workflow

1. Evaluation: Evaluate the fitness of each individual in the SUT environment;

2. Selection (also called Decimation): Decimate the worst individuals in the population. This is equivalent to only selecting the individuals with the best genes.

3. Offspring creation: Create a new population from these best individuals and fill it with variations of these best individuals. In the end, there are 3 different ways to fill the new population:

   - Crossover: create 2 new individuals by genetically recombining randomly chosen characters;
   - Reproduction: copy existing individuals to the new population;
   - Mutation: create a new individual from an existing one by randomly mutating one or several of its genes.

After repeating the previous steps, until the training termination criterion is met, the best individual that appeared in any generation (i.e., the best so far) is designated as the result

of the GA for this training.

The experiments conducted in [8] aim to evaluate the listed evolutionary algorithms and an existing model-free Reinforcement Learning (RL) approach in detecting unhealthy system states. The performance bottlenecks that should be detected concern, beyond other, the following issues: response time, workload, error rate, etc. In addition, the time saved by each testing technique is compared. The capability to find the optimal solution, i.e., the minimum workload size highlighting a performance issue, of each algorithm is also measured.
The results include the following statements. The four evolutionary algorithms, as well as the random search algorithm used for comparison, can be used to generate the optimal effective workload. The Multi-Objective evolutionary algorithms even perform relatively close to the model-free RL method. If the RL approach is faster and saves more time, the evolutionary algorithms perform better in generating the optimal solution.

The experiment conducted in E. Isaku's thesis implemented each GA with a small population size (50 individuals). This specification was sufficient to find the optimal workload because the SUT addressed was rather straightforward (12 possible RUBIs operations). However, GAs with significantly bigger populations can attempt and overcome highly complex problems [15].

3.7.2 ML-assisted Performance Testing framework

In [7, 6, 16], an autonomous Performance Testing framework is built using a model-free RL technique. The goal of the framework is to automatize testing activities to reduce human effort and cost, influence the test coverage positively and, once the optimal policy is learned, reuse the model in analogous situations to improve the efficiency even more.
The framework uses a black-box approach and focuses on Stress Testing. It can learn the optimal policy for test generation. The objective of the framework is to find the performance breaking points of the SUT and to verify the robustness of the system. To address this problem, the framework will generate extreme execution conditions and manipulate the platform, application and workload related factors affecting the performance until a bottleneck is found.

The RL agent used in the model is built during two phases of learning: initial and transfer learning, see Algorithm 1. The initial learning phase corresponds to the classic training phase, the model tackles the problem from scratch and updates until it reaches the end of the training, or finds the optimal policy. During the transfer learning, the frame-

work replays the learned policy in analogous situations, i.e., SUTs with a slightly different performance sensitivity, while keeping the learning running.
The transfer learning phase results in a model more general and robust to similar problems. The final model can perform slightly worse than the original model as it is not specialized anymore. However, it can be used as a basis for different, but similar problems and save training time and resources.

Algorithm 1: Algorithm SaFReL (Self-adaptive Fuzzy Reinforcement Learning-based) presented in [16]. In the following algorithm, Q is the Q-learning function that assigns a utility value to each pair of state and action. s is a state belonging to the state set $\mathbb{S}$, a is an action belonging to the action set $\mathbb{A}$ and ε is the parameter that determines the probability of exploration in a ε-greedy method.

```
1  Initialize: q-values, Q(s,a) = 0 ∀s ∈ S, ∀a ∈ A and ε = υ, 0 < υ < 1;
2  Observe the first SUT instance;
3  repeat
4  |   Fuzzy Q-Learning Episode with initial action selection strategy (e.g.,
   |     ε-greedy, initialized ε)
5  until Initial convergence;
6  Store the obtained experience, the learned policy;
7  Start the transfer learning phase;
8  while true do
9  |   Observe a new SUT instance;
10 |   Measure the similarity;
11 |   Apply strategy adaptation, i.e., adjust the degree of exploration and
   |     exploitation (e.g., tuning parameter ε in ε-greedy);
12 |   Fuzzy Q-Learning Episode with adapted strategy (e.g., new ε);
13 end
```

The ML technique used in [7] is the Q-learning. Q-Learning is a RL algorithm that tries to learn the value of an action in a particular state. The "Q" refers to the expected reward after an action is performed in a given state, it is computed by the algorithm. Ultimately, a Q-Learning algorithm will find an optimal policy that maximizes the total reward after any succession of states. As an example, consider an agent playing a video game. The Q-Learning algorithm will take a decision every step (e.g., go right, left, stop) according to the expected reward it has computed. When the game ends, the overall score is given to the algorithm as a feedback. After enough training, every action in a given state is taken to maximize the final score of the game.
In [7], the state detected by the RL agent corresponds to the quality measurements of the SUT, such as CPU load, response time, memory and disk utilization, etc. The actions available to the agent are operations modifying the factors affecting the performance, i.e.,

workload characteristics or available resource capacity. In order to improve, the agent receives a reward from the SUT. This reward is computed as a combination of the response time deviation and the resource usage.
This framework was tested on different benchmark programs and showed improved efficiency regarding time and resource cost, but also reduced the source code dependency. This tool can be used in software product lines or in CI/CD.

A Machine Learning technique: Generative Adversarial Networks

This section will present nested notions from Machine Learning (ML) to Generative Adversarial Network (GAN).

As seen in the previous chapter, automated testing can be performed through the use of ML [8]. Learning is seen as the core mechanism, or engine, behind what we call intelligent actions. Learning is the search for a behavior that achieves a goal. Machine Learning is a research field subset of AI. It focuses on the study of algorithms that learn and take decisions automatically through experience. These algorithms are generally initialized randomly and gradually upgrade themselves to perform more complex actions. This is a similar approach than the pattern of behavior in life: continuously try random actions and keep doing what works well. This slow learning process is also called "direct experience", it allows to complexify model according to their evolution environment, see Figure 4.1.

ML can be divided into different basic paradigms, such as Reinforcement Learning, Supervised or Unsupervised Learning, Feature Learning, etc. Supervised Learning is a type of ML used to learn models from labeled training data. It is named supervised, because the given training data is already labeled and the model knows all the possible labels. Supervised Learning enables output prediction for future and unseen data.

Neural Networks (NNs) or artificial Neural Networks [17] are used in many different approaches, such as Reinforcement Learning. However, one common application of the NN method is as a Supervised Learning algorithm. NNs are systems using multiple elementary units called neurons connected to each other in layers. A layer is the parallel formation of several neurons, often drawn in columns. The whole system is inspired by the biological brain and roughly tries to model its operations. NNs have remained a more or less obscure corner of Computer Sciences, even though they were conceived in the 1950s [18].

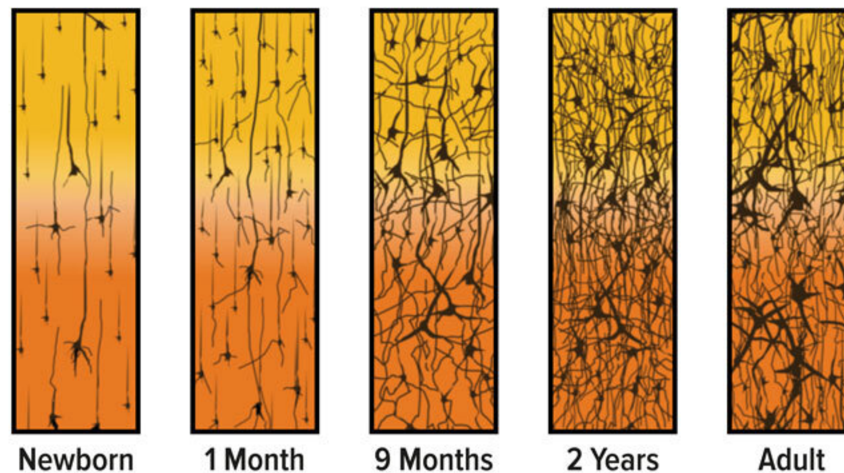


Figure 4.1: Neuron growth and connections over time

Source: Corel, JL. The postnatal development of the human cerebral cortex. Cambridge, MA: Harvard University Press; 1975

A NN learns everything by itself using direct experience. Starting with a random connection pattern, it gradually self-wires into parallel patterns of computation in order to perform complex functions.
NNs are implemented and used in many different situations and fields today, but what made them so famous and powerful is the development of Deep Neural Networks (DNNs). DNNs are artificial NNs with multiple layers between the input and output layers. The adjective "deep" sometimes refers to an unbounded number of layers of a bounded size. These networks have produced results comparable to, and in some cases surpassing, human expert performance.

4.1 Deep Neural Networks, a brief presentation

Before explaining how DNNs work, an overview of classic artificial NNs working principles is necessary.

4.1.1 The simplest Neural Network: the Perceptron

Artificial NNs are usually made of thousands of artificial neurons. However, the simplest NN we can create is made of a unique neuron and is called a Perceptron. This system

takes one or several inputs and, from these inputs, computes and outputs a single value. It is a binary single-neuroned NN.
More specifically, an artificial neuron is a mathematical function, which takes inputs, weights them separately, sums them up and passes this sum through an activation function to produce an output. Figure 4.2 and the Equation (4.1) detail this process.

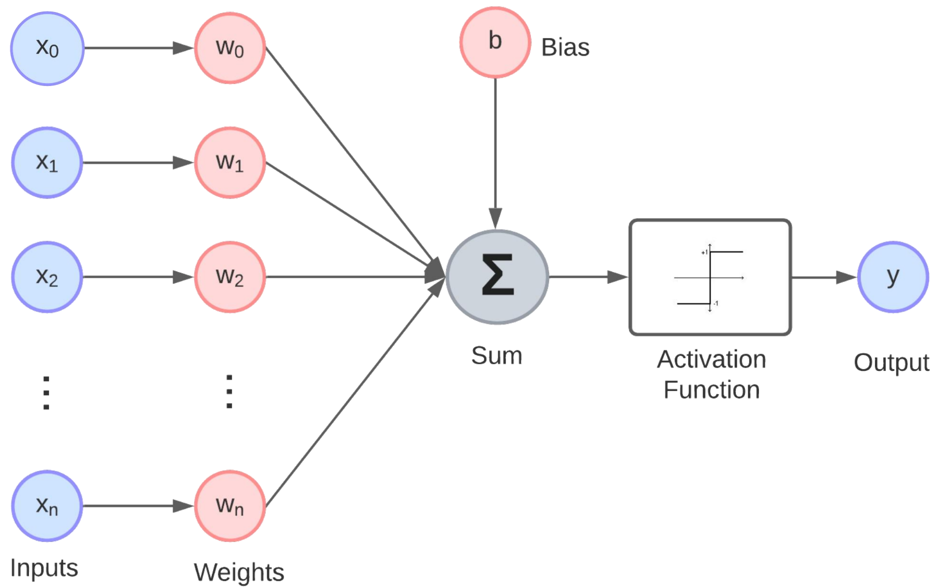


Figure 4.2: Schema of a Perceptron

With $\vec{x} = (x_0, x_1, \ldots, x_n)$ and $\vec{w} = (w_0, w_1, \ldots, w_n)$

$$\text{Output} = sign(\sum_{i=0}^{n} x_i w_i + b) = \begin{cases} 1 & \text{if } \vec{x}.\vec{w} + b > 0 \\ -1 & \text{otherwise} \end{cases} \tag{4.1}$$

In Figure 4.2, on the left we can see $\vec{x}$, the inputs which are multiplied by $\vec{w}$, the weights or the parameters of our neuron. We can also see the bias b, also called threshold, which is added with the weighted inputs. Then, the result of this sum goes through an activation function. If the sum trespasses the threshold, in our example with the $sign()$ function this is equivalent to positive, then the output y is 1: the neuron is called "activated". To summarize Equation (4.1), a neuron is triggered only when weighted inputs reach a certain threshold value.

Equation (4.1) and the previous explanations describe how a NN is used and the system computes an output from inputs. This is also called the Feedforward propagation, as the flow of information occurs in the forward direction (i.e., from left to right). However, when talking about Machine Learning, the system is supposed to learn through experience. This occurs when the second type of information movement comes in, the backward propagation of errors, or in short: the Backpropagation. The Backpropagation allows the network to receive a feedback from its last action and learn from it by modifying its parameters.
For a concrete example, suppose that in our case, with the Perceptron, the system has been used and has computed an output, for example: 1 equivalent to "activated". If we want our model to learn, we will then compare its output with the labeled output, supposedly for these inputs -1 or "deactivated". From this comparison, we will provide a feedback to the model roughly saying "You were wrong the expected answer was -1: you need to update your parameters". This is done by backpropagating the error and updating the weights and the bias of the Perceptron.

The error of the system is often computed as the accuracy of our model, for which we use a so-called cost (or loss) function. Several loss functions of different types exist, the choice of a loss function depends on the type of the learning task, usually either classification or regression. Classification tasks use functions returning a reward for correct predictions and a price for inaccuracy, such as Hinge loss, Tangent loss or Exponential loss functions.
Our example case being a regression task, a possible loss function could be the Mean Absolute Error or the Mean Bias Error. The most-common one is the Mean Squared Error or MSE, see Equation (4.2). In this formula, i represents the index of the sample, $\hat{y}$ is the predicted outcome, y is the actual value, and m is the number of samples.

$$\text{Cost Function} = \text{MSE} = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}-y)^2 \tag{4.2}$$

Ultimately, the goal is to minimize this cost function in order for the model to predict data as accurately as possible. This minimization is often done with gradient descent [19]. During this process, the algorithm adjusts its weights to minimize the cost function. The cost function gradients determine the level of adjustment with respect to parameters such as activation function, weights, bias, etc.

With these mechanisms, the weights of the network connections are, through training, repeatedly adjusted to minimize the difference between the actual output and the desired

output. Hence, optimal weight coefficients are automatically learned, the error function is optimized and the loss minimized: the NN is trained.

4.1.2 Layers and Depth: from a single neuron to networks

In order to further and less rigorously introduce NNs, we also call the input values of the network a "perception" and its output is sometimes called "concept" [20]. With this vocabulary, we can explain the purpose of a neuron in a slightly different way.
For this analogy, let us consider a neuron using the identity function, a linear and basic function, as activation function. Such a neuron allows to separate its input space, also called perception space, with a straight line into 2 regions: "inactive" or "active". Training the network means translating and rotating this line, plane or hyper-plane (depending on the number of inputs and, therefore, the dimension of the perception space) to better represent concepts. By placing new inputs into its perception space and looking whether they are in the "active" or "inactive" region, the network can deduce a concept from perceptions and predict unseen data.

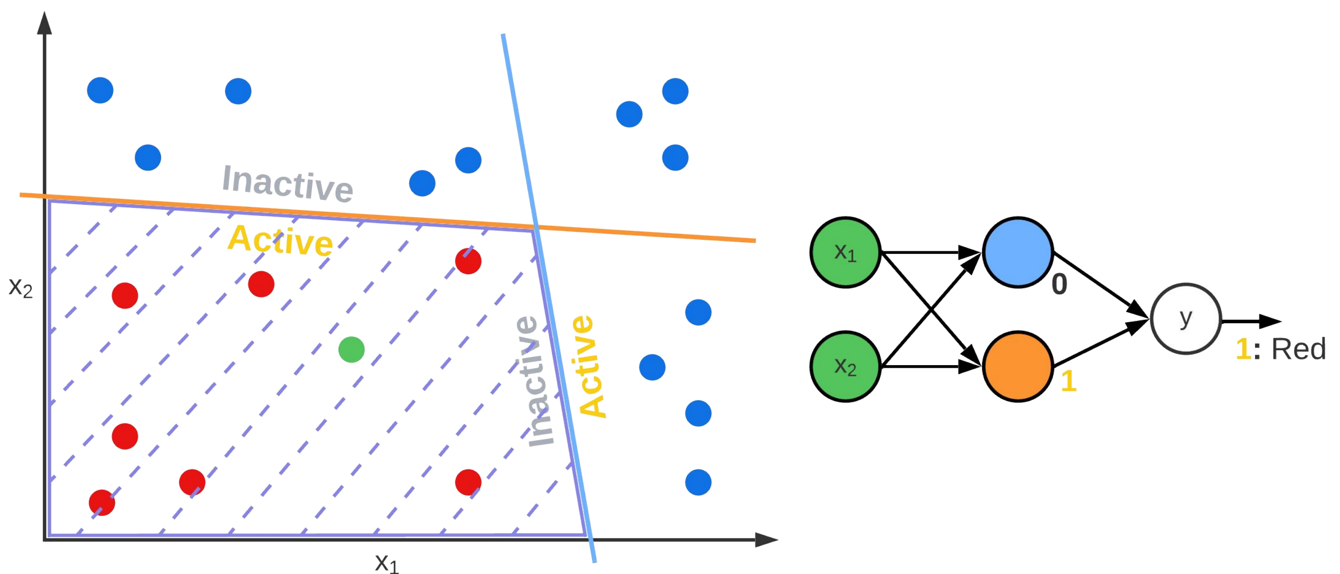


Figure 4.3: Perception space partitioned with several neurons

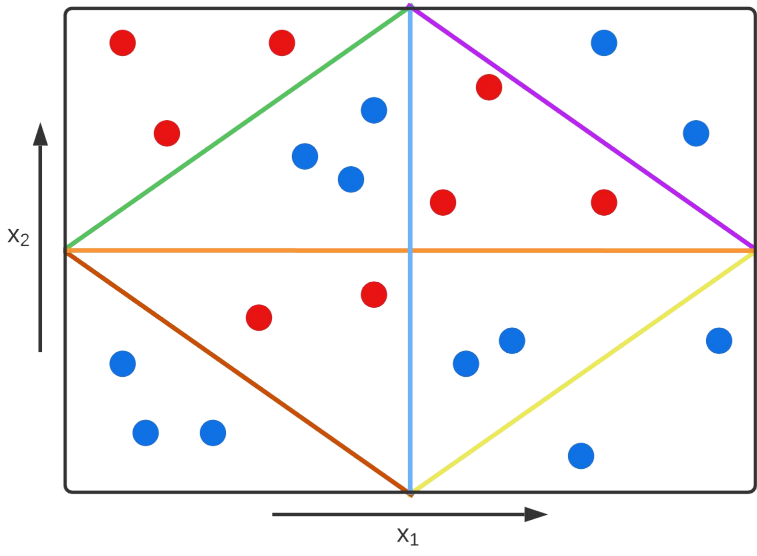
However, problems are often more complex than two concepts which can clearly be separated by a single straight line. A way to overcome this issue is to add more neurons, in order to draw several lines and attempt to solve more complex problems, see Figure 4.3.

In the data distribution of the figure, it is not possible to efficiently separate the samples with a unique straight line. Nevertheless, it is possible to correctly attend this task with two or more straight lines.
In the perception space of Figure 4.3, we can imagine that a trained network with a two-neuron middle layer will only activate its output, meaning the detection of red samples when a particular neuron is active and the other inactive. This case is represented by the network on the right of Figure 4.3. In this example, the input vector $\vec{x} = (x_1, x_2)$ is located among the red dots cluster. This input will not activate the blue neuron but will activate the orange one. On the next layer, the last and output layer, such a pattern is equivalent to the prediction of a red dot. In the end, the network correctly predicts the label of new data.

To summarize, a perception is a list of measurements that are inputted into a network. These vectors can represent coordinates or points in a perception space. The number of dimensions in this space is equal to the number of different input values. A neuron will act as a partition in this space. Several neurons can, therefore, define a region in this space. These regions can carve out inputs which are part of the same concept.

However, real-life problems are in most cases more complicated than that. If we consider more complex problems, such as the MNIST (Mixed National Institute of Standards and Technology) database classification problem [21], the points in the perception space are not exactly clustered into regions but instead scattered all over the space. Carving up such a space into regions to separate concepts is very difficult and would need a layer made of millions of neurons.

A way to simplify this problem is to take inspiration from nature and biological NNs. Organic brains use many layers of neuron activations to process their inputs. The importance of depth or having multi-layer networks is one of the least understood aspects of NNs.
To understand the importance of this concept, we can look at Figure 4.4. This perception space would need six stacked up neurons to be correctly partitioned. We can think of the lines as folds in the perception space as if it was a paper sheet. If we now look at Figure 4.5, this partitioning has been achieved without "unfolding" the perception space after each fold. The folds produced by the activation of one neuron are stacked and produce several "lines" in the perception space. For example, adding a 4th consecutive fold would divide the space into 16 different regions (e.g., dashed gray lines in Figure 4.5), and a 5th consecutive fold in 32 regions.



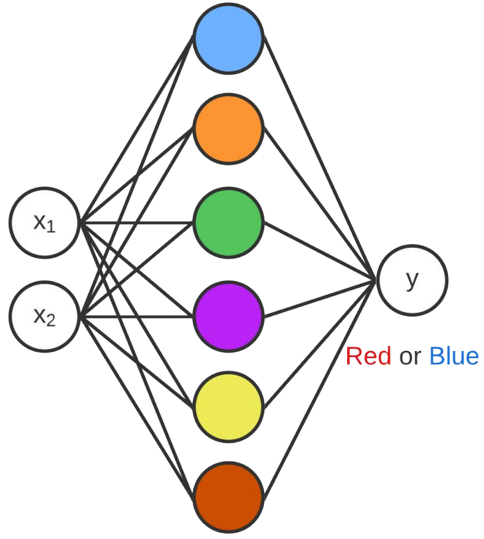
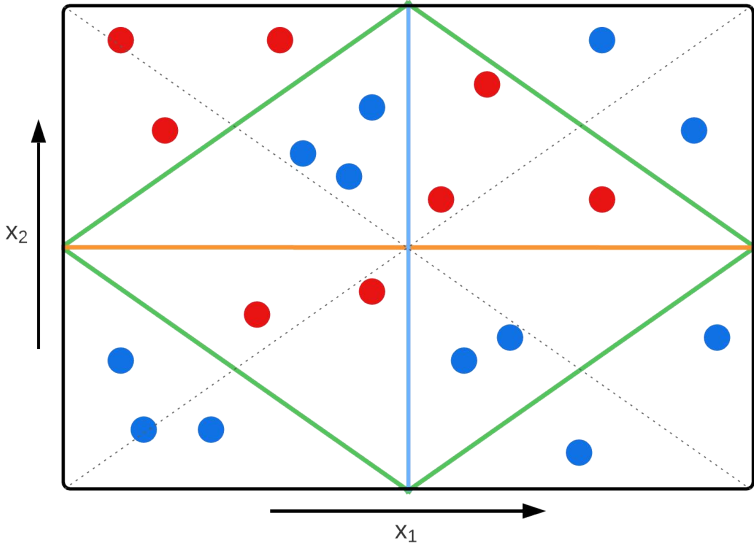


Figure 4.4: Perception space partitioned with one layer



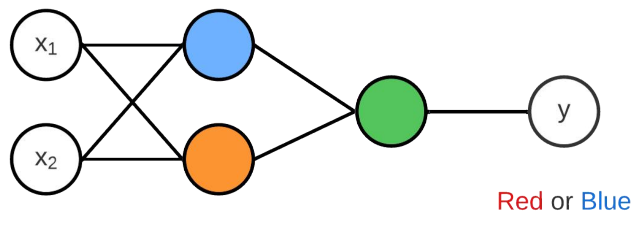


Figure 4.5: Perception space partitioned with multi-layers

This recursive power of folding shows how we can exponentially obtain more partitions using the same number of neurons, if we layer them. More practically, we can also say that neurons deep in a network are not simple linear partitions but are instead activated by a complex pattern of linear partitions.

4.1.3 Customizing networks: activation functions

In the Perceptron example, the network outputs a boolean: 1 or -1 (it can also be 0). This boolean comes from the use of a binary neuron model with a simple and binary activation function: the $sign()$ function or, more generally, a step activation function. In practice, many other activation functions exist like the previously used linear identity function, the Sigmoid (4.3) function or the Hyperbolic tangent (4.4) function. Using these types of activation function implies that the output can be a real number and not only a boolean. If that feature creates more complex computations, it allows a more efficient and precise training.

Having neurons that activate more gradually, i.e., that are not binary activated, allows to determine the direction and the magnitude of the change of the node parameters on the output. To explain it in a simpler way, consider that wiggling the parameters of a binary neuron will very unlikely change the final output of the whole NN. This means that, in order to improve this network, we will have to blindly modify the parameters of the network until we finally observe a change on the output. However, while using real numbers and therefore obtaining gradual outputs, a change in the parameters of a node will always result in a change in the output. Even a very slight change will give information about its direction and magnitude. Also, wiggling last layer nodes parameters will result in bigger changes in the output than modifying the first layer, due to the Backpropagation principle.

Above all, using real numbers allows the network to express a result probability. For example, in a case of a network classifying cat and dog images, the two-neuron output layer may output 0.79 on the neuron responsible for the dog identification. This can be interpreted as "The network is sure at 79% that this picture represents a dog". Thus, using non-binary activation functions that output real numbers allows a probabilistic interpretation and a more overall precise training.

The purpose of using functions such as Sigmoid, see Equation (4.3), or tanh, see Equation (4.4), is also to introduce non-linearity in the NN. Non-linearity in the NN system gives it more abstraction and, therefore, allows it to perform more complex tasks. Introducing non-linearity is equivalent to saying that a neuron no longer separates the perception space with a straight line but with a more complex and customizable line such as a curved line.

$$Sigmoid(x) = \frac{1}{1+e^{-x}} \tag{4.3}$$

$$tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{4.4}$$

$$ReLU(x) = max(0,x) \tag{4.5}$$

To conclude with the activation functions, it is worth mentioning that it is not necessary to use a unique function in the whole network. Many activation functions exist and each of them has advantages and drawbacks. For example, in the case of both *Sigmoid* and *tanH*, these functions are exposed to the "vanishing gradients" problem. During Backpropagation, when the error is computed (see Equation (4.2)) and progressively sent back in the network to update the parameters, the gradients of early layers (layers near to the input layer) are obtained by multiplying the gradients of later layers (layers near to the output layer).
Thus, for example if the gradients of later layers are less than one, their multiplication vanishes very fast. These functions saturate and kill gradients. In contrast, different activation functions can encounter the "exploding gradients" problem.
The *ReLU* function, Equation (4.5), avoids and rectifies the vanishing gradient problem. It generally provides the same benefits as *Sigmoid* but with better performance. In any case, the Sigmoid activation function is often used in the last layer to convert the model output into a probability score (as its output is between 0 and 1), which is easier to work with and interpret.

This finally brings out the following question: When should we use which activation function? This question is one of the most frequently asked when building a NN. Usually, the choice of activation function in hidden layers will control how well the network model learns the training dataset. The choice of activation function in the output layer will define the type of predictions the model can make.
The choice of the activation function is not straightforward and often leads to several tests and trainings. However, to give an example, in the case of a binary classifier, the *Sigmoid* activation function can be used in the output layer. Even if the *Sigmoid* and *tanh* activation functions work badly for the hidden layer, they allow an output which can easily be interpreted as a probability. For hidden layers, *ReLU* or its better version leaky *ReLU* should be used. For a multiclass classifier, *Softmax* is the most used activation function. Even though more activation functions are known, the aforementioned ones are known to be the most used ones.

4.1.4 Training and Learning: database importance

Learning implies two phases, the actual learning phase when the subject trains to achieve a specific task or to accurately remember information, and the phase afterwards when the subject applies what it learned. This second phase is often less known and consists in an application of the new knowledge. In ML, these two phases are called Training and Inference. Inference refers to the process of using a trained ML algorithm to make a prediction. For example, if you use your autonomous car, the AI is detecting obstacles in the Inference phase. It has already been trained by the company during the development and once it achieves the required performance, the algorithm is "frozen" and stops updating itself, the Training phase has ended. Then, and only then, the algorithm can be used in the car and fulfill its purpose.

Switching to the Interference phase means stopping the training. However, how do we know the model is trained and will perform as planned in its environment with new and unknown data? This is a difficult question. The best way to prepare a model to its environment would be to train it with every possible data it could encounter during its Inference phase. This is generally impossible, as we cannot simulate every case the algorithm will face. Besides, that is also why we use ML. If we could completely describe all the possible outcomes of a problem, and if we wanted a 100% prediction accuracy, then the best method would be an exhaustive description of every possible case and its corresponding answer.

As ML is not designed to conduct such an exhaustive description of every scenario, our results will necessarily be imperfect. It is thus important to define which error rate is still acceptable, or which accuracy is satisfying. In other words, we need to define how accurate the heuristics our model relies on must be. Heuristics are used in ML when it is impractical to solve a problem with a step-by-step algorithm. A heuristic approach also emphasizes speed over accuracy.

Thus, how can we evaluate if a model will perform well in an open environment when it is trained with a limited amount of data? A common solution is to evaluate it with a different dataset than the one it has been trained with. If both datasets are representative of the upcoming problem environment, then we can assume that if the algorithm performs well on the new testing dataset, it will do so during the Inference phase.

In other words, training a ML algorithm means first introducing data that we want it to learn from and then, once the model has been acclimated to the problem and its data rep-

resentation, we introduce new data from the same problem to the algorithm to evaluate it. This statement underlines the importance of the dataset. In ML, databases are critical factors for producing high-quality solutions. A dataset has to be sufficiently large and diverse to correctly represent a problem environment and prepare the model to deal with unknown data.

Another major problem about datasets is the evaluation of the model. How can you compare it with another solution? Consider that you own one large dataset used during the training phase to fit your model: the training set. Evaluating your model skill on the training dataset would result in a biased score. Because the data used for the evaluation exactly corresponds to the data with which the model has been trained. If the algorithm has been trained long enough, it should obtain a 100% or close score. A comparison can be made with students. If you evaluate your students with the exact same problem presented during the class, then you know if they paid attention, but not if they understood the notion you have been trying to teach.

A solution to this problem is the use of a test dataset. The test dataset is made of data samples that come from the same data environment, that are very similar to the data used for training, but that the model has never been exposed to during the training. Using a test dataset allows to provide an unbiased evaluation of a final and trained model.

Algorithm 2: Pseudo-code showing a train-test-evaluate approach. In this over-simplified algorithm, $data$ represents the whole set of information available for the attended problem, $parameters$ corresponds to the whole set of modifiable parameters in the model (i.e., the weights of a NN), the $fit()$ function corresponds to a training cycle and an update of the model, and $skill$ is the score or the results of the model.

```
1 Input: data
2 Initialize: parameters
3 train_data, test_data, validation_data = split(data)
4 for params ∈ parameters do
5 |   model = fit(params, train_data)
6 |   skill = evaluate(model, test_data)
7 end
8 model = fit(train_data)
9 skill = evaluate(model, validation_data)
```

To summarize, as the evaluation of a model skill on the training dataset would result in a biased score, the model is evaluated on a different dataset made of the held-out sam-

ples to give an unbiased estimate of model skill. This is typically called a train-test split approach to algorithm evaluation. To push the concept further, Algorithm 2 shows an approach including a third dataset used for comparing performance with other models.

The larger and the more diverse a dataset is, the better the results will be. This is also referred as the robustness of the model. A model exposed to a huge number of diverse samples will be more capable of handling new data slightly different from what it has been used to and trained with. However, training a model with a training set for a long time expose it to the risk of overfitting. In this context, overfitting means that the model will perfectly learn the patterns of data in the training set, so that it can achieve an almost perfect score. An algorithm fitting exactly against its training data means that it cannot perform accurately against unseen data, defeating its purpose.

4.2 The Generative Adversarial Networks concept

4.2.1 Generative modeling

Most of the ML algorithms are used to make predictions. Some data is inputted into a model and the model compiles an output, a prediction. The model is trained by showing it examples of inputs, making it predict outputs, and correcting the model to make the outputs more like the expected outputs (i.e., the ground truth from the training data). This is an example of supervised learning, see Figure 4.6.a.

However, ML can be used to generate new data. These algorithms are called generative models and use the patterns and regularities they automatically find in the data to build new examples that could have been drawn from the original dataset. This technique is categorized as unsupervised learning, see Figure 4.6.b. In general, unsupervised learning corresponds to problems which do not have any output variables. That means the model does not predict anything, because there is no correction and, therefore, no possible feedback. The model is built by extracting or summarizing the patterns in the input data. After training and assimilating input data patterns the model can be used to generate new data.

Unsupervised models are often involved in image generation. Generative models can be used to generate novel data samples, such as images of animals, landscapes or objects [22]. Generative Adversarial Networks (GANs) can even generate photorealistic

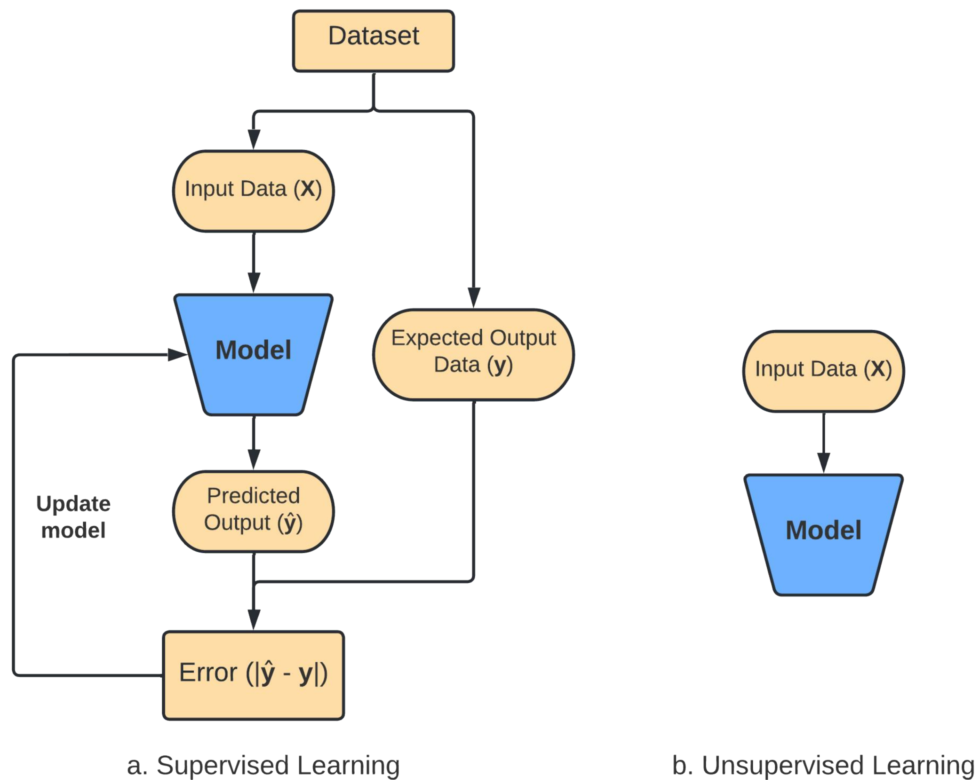


Figure 4.6: Examples of Supervised Vs Unsupervised Learning workflows

pictures such as non-existent human faces that can fool human beings [23]. They can also be used in sophisticated domain-specific data augmentation. Quality improvement applications go from noise reduction to image super-resolution, i.e., upscaling the resolution of an image, but also include restoring missing or deteriorated parts of an image, which is referred to as image inpainting. GANs can also be used in some more elaborated fields such as domain adaptation, i.e., transforming a normal photo in an oil painting while retaining the originally depicted content, or image-to-image translation, i.e., modifying the environment of an image from winter to summer or from day to night.

However, GANs are also used in fields not related to images. The techniques described above can be applied to other types of media, such as audio or text. For example, in 2016 the Google brain team used a structure of adversarial neural networks to implement cryptography without any prior knowledge of cryptographic algorithms [24].

A GAN is an unsupervised model made of two sub-models: the generator and the discriminator. These sub-models are commonly Convolutional Neural Networks or CNNs. CNNs are artificial neural networks adapted to computer vision. Indeed, in computer vision, input data are images that possess two dimensions. Therefore, in order to prevent

a loss of information, images cannot be flattened in arrays as it would be necessary for classic fully-connected NNs, otherwise the information carried by the horizontal or vertical proximity of pixels is lost.

CNNs are networks made of two types of layers: convolution layers and pooling layers. Convolution layers apply the convolution operation. Simply put, the 2D-convolution operation works as a filter and considers a neighborhood of pixels to compute a single value. This operation is used on the whole input of the layer and creates an output of the same size, sometimes called convolved feature. Using such an operation differs from usual fully-connected NNs that consider each neuron independent and, therefore, assign a different weight to each incoming signal.
In addition, pooling layers are responsible for reducing the spatial size of the convolved features. This is done to decrease the computational power required to process the data through dimensionality reduction, but also to extract dominant features and isolate important information.
Alternating convolution layers and pooling layers allows to process high-resolution images into a form which is easier to process, without losing features which are critical for getting a good prediction.

4.2.2 Generate and discriminate: a zero-sum game

The first sub-model of GANs is named the generator and is trained to generate new data samples. During the training phase, the generator faces examples coming from the training dataset. The generator will learn the data distribution of this dataset and it will then be able to generate data plausibly fitting this distribution from an input random vector, also called noise. The second sub-model is the discriminator. The purpose of this sub-model is to predict if a data sample is fake, i.e., has been created by the generator, or real, i.e., comes from the original dataset. To summarize, the generator is a generative model creating plausible fake samples fitting a data distribution, and the discriminator is a classification model predicting if a data sample is generated or real, see Figure 4.7.

Generative Adversarial Networks are called adversarial networks because the two sub-models are competing against each other. The generator and the discriminator are adversaries in the game theory sense: they are playing a zero-sum game. A zero-sum game is a mathematical representation of a situation that implies two sides, and where the result is an advantage for one side and an equivalent loss for the other. Therefore, when the gains or losses are summed, the total is equal to zero. The goal of the generator is to fool the

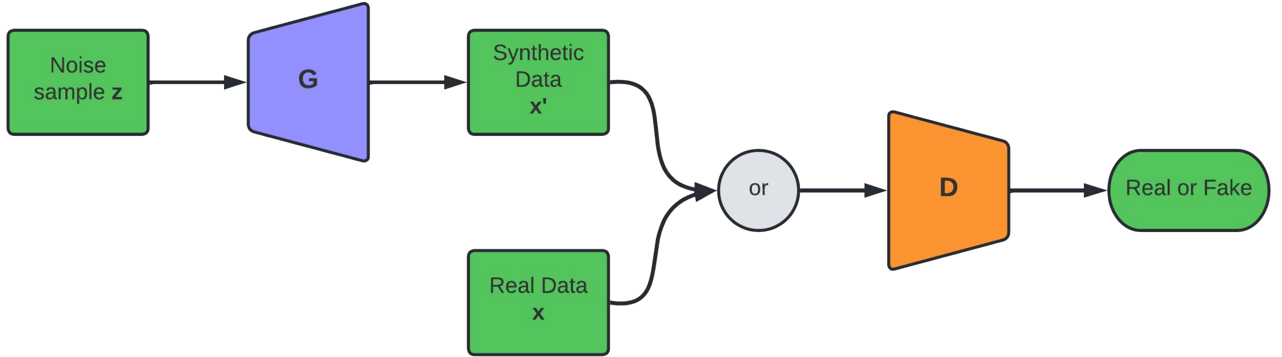


Figure 4.7: Basic model of a GAN architecture

discriminator with very good generated samples, whereas the goal of the discriminator is to perfectly predict the samples and avoid being fooled by the generator. After a prediction, the winner gets to remain completely unchanged, but the loser has to update its model. The training continues until the discriminator is fooled half of the time.

The zero-sum game played by the generator and the discriminator is described by Equation (4.6) [25]. In the following equation, D and G are two models or functions taking respectively data x and z, where z can be interpreted as a noise variable and $p_z(z)$ as a prior on input noise variables. $D(x)$ represents the probability that x came from the data rather than p_g, where p_g is the generator distribution over data x. The input noise z that is fed into the generator G comes from a prior distribution in order to introduce some stochastic behavior to create different outputs.

$$\min_G \max_D \text{V(D, G)} = \mathbb{E}_{x \sim p_{\text{data}}(x)}[log\ D(x)] + \mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z)))] \tag{4.6}$$

In this equation, D is trying to maximize the two terms. The first term corresponds to the probability of the data sample being real, i.e., x belonging to the real data distribution. In addition, D is also trying to maximize the second term of the equation, which corresponds to minimizing the probability of fake data being classified as real. As a two-way classification problem and without any evolution of G, this equation leads to a perfect discriminator. For its part, G is trying to minimize $\max_D \text{V(D, G)}$, i.e., the best possible discriminator.

The game described by Equation (4.6) is named a *minimax two-player game* in game theory. In this kind of zero-sum game, a person can win only if the other player loses. A common strategy is then to try to minimize the worst scenario for you, which is equivalent to the best outcome for your opponent. Trying to minimize what the other player is trying to maximize is a strategy named *minimax*.

4.2.3 Challenges and Wasserstein variant

Training GANs is complex for two reasons. Firstly, a GAN contains two separately trained networks. So, how do we train the GAN as a whole? Secondly, GAN "convergence" is hard to identify because the improvement of one network is done at the expense of the other.
The training process is done by alternation: during a first phase the generator is constant, its parameters are frozen, and the discriminator is trained during one or several epochs [26]. Similarly, during a second phase, the generator trains while the discriminator is frozen, otherwise the generator would be trying to hit a moving target and would never converge. These two phases are then repeated until the GAN converges. However, this convergence is hard to define because as the generator improves, the discriminator performance get worse. At its peak, i.e., for an excellent generator, the discriminator will act as a coin flip and have a 50% accuracy. This means that the discriminator feedback gets less meaningful over time. If the GAN training continues beyond the point where the discriminator feedback is random, then the generator starts to train with junk feedback and its quality may collapse. For these reasons, the convergence of the juggling GAN training is often a volatile, rather than stable, state.

GANs are powerful and versatile algorithms. However, like every ML technique, they are subject to some common failure modes. When the discriminator becomes excellent, the feedback it provides may not contain enough information and lead to the vanishing gradient problem for the generator.
GANs are also subject to a problem named mode collapse [26]. A GAN is supposed to produce a wide range of different outputs, optimally one for every different random input. However, sometimes when the generator produces an especially plausible output, it may learn to produce this output over and over. This leads the discriminator to find that it should always reject this particular output. The next generation of generator, then, discover another plausible output, because the discriminator did not learn the best strategy but got stuck in a local minimum. In this pernicious circle, each iteration of generator over-optimizes for a particular discriminator, and the discriminator never manages to

learn its way out of the trap. This GAN failure, where the generator only rotates through a small number of outputs, is called mode collapse.

A variant of the GAN algorithm, called the Wasserstein Generative Adversarial Network (WGAN), allows to remedy these two problems [27]. The main difference in WGAN is that a discriminator does not actually classify instances. Instead, it outputs a number that is greater for real instances than for fake instances. Because the discriminator is not really discriminating real and fake anymore, it is named a critic. This critic has a different loss function. The generator loss function is now $D(G(z))$ and the critic loss corresponds to $D(x) - D(G(z))$, see Equation (4.6). In other words, the critic tries to maximize the difference between its output on real instances and its output on fake instances. This new loss function solves at the same time the vanishing gradient and the mode collapse problems.

Performance testing using OGANs

A review of state-of-the-art techniques has been conducted in Chapter III about software testing, specifically in Performance and Stress Testing, and Website architecture (REST APIs). This chapter will first present the OGAN algorithm, a GAN-inspired ML technique used to realize performance exploration. In a second step, the work affiliated to this thesis, i.e., the previous articles published by Åbo Akademi University laboratory, will be presented.

5.1 Online Generative Adversarial Networks

The work carried out by I. Porres, J. Peltomäki and F. Spencer in [10, 28, 4, 29] uses the OGAN algorithm or its variant WOGAN. These algorithms, especially the original OGAN, are the ML techniques used and discussed in this thesis. The following section presents the differences with the classic GAN algorithm and introduces the notion of online learning.

5.1.1 A GAN approach applied to system testing

The OGAN algorithm is a GAN-based approach applied to system testing. OGAN is a general-purpose black-box test generator applicable to any SUT having a fitness function for determining failing tests. As previously explained, GANs are renowned in domains such as image generation. Consequently, to our knowledge, [10] was the first work using GANs for automated test generation.

In this GAN-based approach, the generator proposes test cases and the discriminator rejects those candidates if it predicts negative fitness outcome. In case of a positive fitness prediction, the test case is added to a test suite and when the suite contains enough samples, these tests are executed on the SUT. The results obtained from this evaluation are then used as a feedback to train the whole model.

The intuition behind this technique is that the generator explores the input space and produces candidate tests, while the discriminator evaluates the candidate tests and predicts their fitness. Because evaluation with the SUT is assumed to be a timewise expensive operation, predicting the test fitness and rejecting test candidates, which may be negative, avoids time loss.

5.1.2 Online training and OGAN algorithm

This concept of GANs is adapted to work with online supervised learning, i.e., without a prior training set. At the beginning of the training, without prior data, the algorithm acts like a random sampling algorithm, i.e., creates completely random test candidates. Likewise, the untrained discriminator assigns random fitness to tests. It acts as a coin-flip, which means that the first test suites that will be evaluated on the SUT are random and include negative fitness tests.

However, after the first feedback from the SUT evaluation, the empty training set can now be stuffed with the previously generated and evaluated tests. These tests candidates and their respective fitness are used to train the discriminator. Once the discriminator is trained, the whole GAN starts its training. The discriminator is frozen and the generator tries to generate from random noise the maximum possible number of test candidates with a fitness of 1 (according to the actual discriminator).
By repeating this pattern, the OGAN algorithm achieves to train the discriminator and the generator without a prior training set. It only uses the test results obtained online from the SUT, while performing the test generation.
It is also possible to use this algorithm with a prior training set, even a small one. This allows to skip the first random sampling generation step and to accelerate the start of the training.

Algorithm 3 details the OGAN test generation process and uses the problem representation presented in Equation (3.1). Lines 1 to 4 define the inputs and their conditions, and initialize the test suite and the networks. As previously said, it is assumed that a test evaluation with an execution on the SUT is a very time expensive operation. Therefore, an exhaustive search in the input space is not realistic and there is a need to define a budget that limits the maximum number of test executions and corresponding performance measurements. This budget corresponds to a maximum number of tests and adheres to the next condition: $0 < budget \leq |I|$. The budget possesses an upper limit as the algorithm does not generate two times the same test candidate, and because it does not make sense

Algorithm 3: OGAN test generation algorithm presented in [10]. In this algorithm, I is the non-empty set of inputs for the SUT and T is the test suite containing the test candidates. *GAN*, *GN* and *DN* are the networks involved in the training. *target* is a real number acting as a fitness threshold and the target reducer *treducer* is a metaparameter between 0 and 1. The function $generate(NN, S)$ uses the network *NN* to return a subset t of S, whereas $predict(NN, t)$ returns the prediction of the input t by the network *NN*. $measure()$ and $execute()$ are functions executing the test on the SUT and measuring its outcome, i.e., fitness.

```
1  Input: set of inputs I, integer budget
2  Required: budget ≤ |I|
3  T := ∅;
4  Initialize GAN with a untrained generator GN and discriminator DN;
5  while |T| < budget do
6  |   target := 1;
7  |   repeat
8  |   |   target := target * treducer;
9  |   |   t := generate(GN, I − [T]_1);
10 |   until predict(DN, t) ≥ target;
11 |   result := measure(execute(t));
12 |   T := T ∪ {(t, result)};
13 |   train(GAN, T);
14 end
15 Result: test suite T
```

to evaluate twice the same test.
Lines 5 to 14 describe the main loop:

1. Initialize a variable *target* to 1, meaning that only positive test candidates can be added to the test suite. This target acting as a fitness threshold is reduced line 8 in the inner loop by the metaparameter *treducer*. The goal is to make it easier to find a suitable test candidate with every loop iteration.

2. Generate a suitable candidate. The inner loop, lines 7 to 10, uses the generator to produce a test candidate. If the test candidate has a fitness predicted lower than the target, then the loop repeats, slightly reducing the target and generating a new candidate, until a suitable test is found.

3. Evaluate the test candidate. Once a test candidate is found, it is executed on the SUT and its real fitness is measured, line 11.

4. Add the test to the test suite. After the evaluation, the test and its corresponding fitness are added to the test suite. T being constituted of tuples, one for each suitable test, the notation $[T]_1$ corresponds to the set of the first element of the tuples. Said differently, $[T]_1$ represents the test cases already executed on the SUT. For this reason, the generator creates a new test candidate from the set of test inputs that have not yet been executed: $I - [T]_1$ (line 9).

5. Train the GAN. Use the current test suite to train the whole network.

6. Iterate the main loop until the budget is reached: all the desired tests have been generated.

5.1.3 Benefits and open problems

OGANs and WOGANs models represent brand new ML algorithm combining the online learning process and the GAN technique. They benefit at the same time from the abundant research on GANs and from the flexibility and lightweight brought by online learning. The changes implied by the online learning on the GAN algorithm also modify the output of the discriminator model. In OGANs, the discriminator is not just separating samples between two classes, fake or real, but is predicting an actual fitness for each sample. This attribute is particularly fitting for software testing and introduce a new technique in the automated test generation challenge. To summarize, the OGAN algorithm allows creating test suites with a great test fitness without relying on sampling of the solution space.

However, the OGAN generative model is also subject to open research problems. Firstly, the SUT must be stateless, i.e., the whole system can be explored with single tests. For example, online banking is a stateful application. Operations are performed with the context of previous transactions and the current transaction depends on what happened during previous operations.

Secondly, the SUT outcomes must be deterministic. Non-determinism, i.e., when there is no way of predetermining the exact behavior of the system, can be actual (e.g., quantum physics) or apparent (e.g., overwhelming complexity of an environment) but implies that a same input, or test, can result in a different output. Algorithm 3, and online learning in general, is based entirely on the evaluation of the test on the SUT (line 11). A non-deterministic behavior in the output could just fool both the discriminator and the generator and lead to a divergence of the algorithm.
Finally, the OGAN algorithm is recent and its performance have only been measured on a couple of systems. Further evaluations are required to acknowledge the potential of OGAN models. This is discussed in the following related work section.

5.2 Online learning and GANs in Performance Testing

This section will go further in the presentation of the articles that form the basis of this thesis [10, 28, 4, 30, 29]. GANs are a relatively new ML technique as they have been recently theorized in 2014 [25]. GANs can be used in a variety of applications, but are mainly applied in image processing. This includes image synthesis, semantic image editing, style transfer, image super-resolution or image classification.
GANs have great potential in the matter of image processing and they have emerged recently among the ML techniques. This has, in a sense, focused the studies in that field and limited the work in other areas, such as automated test generation.

5.2.1 Software online performance exploration using a Discriminator Network

One of the first work conducted by I. Porres *et al.* in the domain of software Performance Testing is presented in [30]. This work introduces an algorithm automatically generating test cases for a SUT, with the objective of having a maximum number of positive tests in a test suite. The key idea, which later led to use the concept of GANs [10], is to use a

Deep Neural Network acting as a test discriminator and predicting whether a test can be positive or negative. The other concept used in this paper is online learning, which allows the model to construct the test suite iteratively while learning about the SUT.

However, the algorithm presented in [30] is not just a Discriminator Network (DN) and is fairly close to the OGAN algorithm, shown in Algorithm 3. The DN algorithm includes a random (uniform) test generator, providing the test samples to the discriminator, a two-arm bandit, dealing with the exploration–exploitation trade-off dilemma, and a metric, called the positive predictive value (ppv) to measure the effectiveness of a test suite.

The DN algorithm follows the main steps and loops of Algorithm 3 and pursues the following ideas:

- During the first iterations, the inexperienced DN algorithm will roughly act like a random sampling algorithm. However, it is expected to find some positive tests by chance. These tests are then used to train the DN.

- After the first training period, the discriminator will become more and more selective and filter out negative tests, ultimately resulting in a test suite with a high ppv.

- Once the discriminator has reached a sufficient level, i.e., produces test suites with a great fitness, a high ppv, there is a risk of overfitting. The two-arm bandit is an algorithm following an ε-greedy strategy, its objective is to prevent the discriminator to overfit. At the beginning, the value of ε is high enough to promote a high exploration of the input space, but as the ppv increases, ε decreases making the two-arm bandit choose for more exploitation of the discriminator results.

The DN algorithm has been evaluated on two different systems: searching for bottlenecks in a web service and searching for efficient hardware configurations in a single board computer. It has produced test suites with a high proportion of positive tests and performed better than a random tester, further results are presented in the next section. To summarize, the DN algorithm acts as a black-box testing technique: requiring no prior knowledge, except the input and output spaces. The process is completely automated and the use of an online learning method allows it to be used in Continuous Integration.

5.2.2 Online GANs for automatic Performance Testing

To our knowledge, in Software Engineering, [10] is the first work that uses a GAN for automated test generation. This paper presents a novel algorithm for automatic Performance Testing, using an online variant of the GAN algorithm to optimize the test generation process.

The objective of the OGAN algorithm is to produce a test suite containing a high number of tests revealing performance defects. The technique does not require any model of the SUT or any prior training dataset. The algorithm is presented and detailed in section 5.1. In addition, Figure 5.1 shows the OGAN test generation system. Therefore, the following paragraphs will only present its evaluation and results.

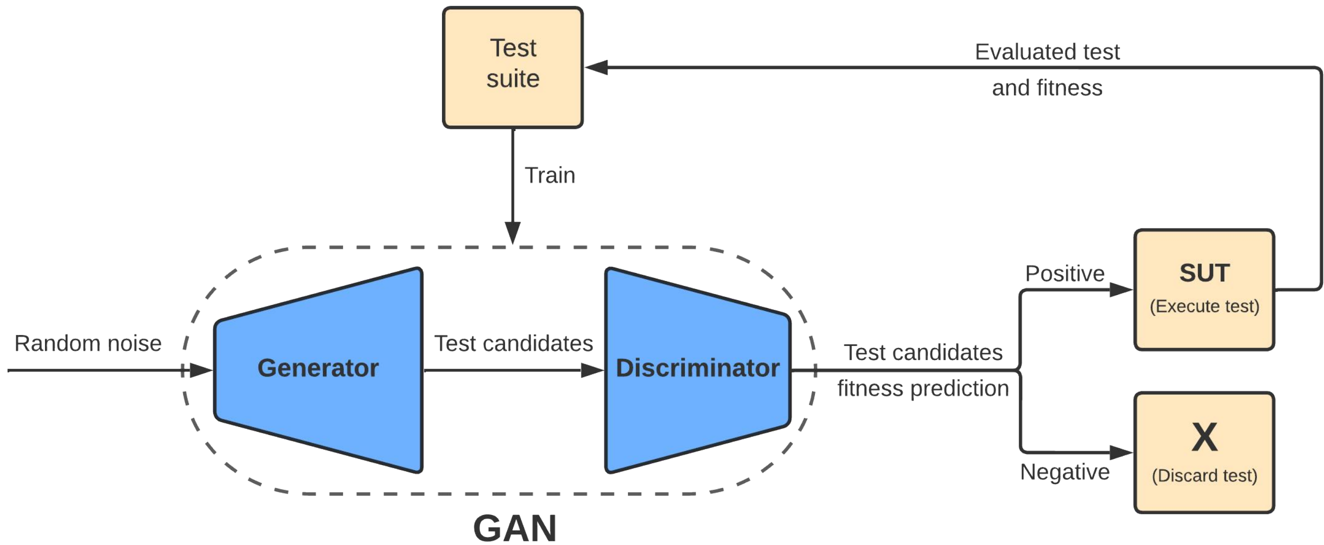


Figure 5.1: OGAN algorithm workflow

In [10], the OGAN algorithm is evaluated with a typical embedded platform performance exploration experiment. The input space represents all the possible board configurations, i.e., the number of CPUs, the clock frequency, the utilization level of a core, etc. In total, the input space has 6 dimensions and can be set in 479 000 different configurations.

The research criteria, i.e., a power consumption $\geq 6W$ equivalent to a positive test, is met in approximately 1% of the configurations. In this experiment, the budget is set at 200 tests and the metaparameter *treducer* at 0.95, see Algorithm 3. The OGAN algorithm is compared to the DN algorithm presented in [30] and to a random search algorithm. In every run, the first 50 tests are randomly generated (to build a basis for the online learning), after that the algorithm is activated and tries to create as many tests yielding a dissipation

power $\geq 6W$, i.e., having a fitness of 1.

As expected, the random search algorithm performs poorly and mainly generate tests with a low fitness, in average ≈ 0.35. The two other algorithms, DN and OGAN, perform better. It is worth mentioning that the DN algorithm is highly dependent on its batch size, i.e., the number of tests generated by the uniform exploration of the input space. The higher the batch size, the quicker the discriminator predicts a positive candidate. For this reason, the OGAN algorithm is compared to a DN algorithm with a large batch size of 32 000.

In such a setup, it takes in average 2.45 trials, i.e., generation and prediction steps in the inner loop (lines 7 to 10 in Algorithm 3), to the OGAN algorithm to generate an accepted candidate. On the other side, the DN algorithm needs 4.46 trials. That also means that the uniform test generator needs to generate, on average, $4.46 \times 32000 = 142720$ tests before one is accepted by the discriminator. Compared to that, the OGAN algorithm only generates in average 2.45 tests, as the generator network has a batch size of 1.
Concerning the average fitness of the test suite, the OGAN and DN performance are comparable. However, the DN network is faster to improve than the OGAN algorithm and quickly reaches its mean fitness, which is independent from its batch size. The conjecture made is that the OGAN is slower to train as it possesses two different networks. After 100 iterations, the mean fitness of the OGAN algorithm becomes better than the one of the DN algorithm. In the end, after 200 tests, the mean fitness of the OGAN test suite is ≈ 0.95.

OGAN models seem to perform relatively well as automated test generators. However, a preliminary evaluation, as presented in [10], is not sufficient to assume the capabilities of the OGAN algorithm, further evaluations on diverse environments are required. Cyber-Physical Systems (CPSs) are complex systems where a software controls mechanisms or physical entities. CPSs represent a good opportunity to evaluate OGANs.

5.2.3 Wasserstein OGAN for Cyber-Physical Systems

CPSs are environments where softwares must be thoroughly verified and exhaustively tested because a fault can lead to severe damage to property or even death. In the context of the SBST CPS tool testing competition organized in 2022, [28] presents a novel test generation algorithm.

In this competition the SUT is the lane assist feature of a car. The main safety requirement is that the percentage of the body of the car which is out of the boundaries of its lane (BOLP) should not exceed an certain limit. Thus, the goal of this Performance Testing experiment is to generate test cases, i.e., road designs, that cause the car to drive out of its expected path. The input space of this problem is made of all the possible road designs on a size-limited map. The output of the system is the maximum BOLP observed during the driving simulation.
In this context, the test generator used is the WOGAN algorithm. The system is seen as a black-box system, only the input and output spaces are known. Moreover, it is an online approach as the algorithm starts with no prior data. The main objective for the model is to generate as many failing tests as possible in a given time budget. Algorithm 4 presents the test generation process of the WOGAN approach in CPS 2022.

Algorithm 4 is very similar to the OGAN algorithm presented in Algorithm 3. The main differences consist in the presence of a third network A, called the analyzer and the use of a dynamic batch of the test suite. The use of a third actor A, acting as a validator, constitutes an alternative approach from the original generator/critic WGAN method, but it is necessary as no prior data samples of failing tests are available in advance. Hence, using the analyzer A in these conditions, speeds up the training at its beginning while also requiring less test executions.
Likewise, to create a faster training and reach, as early as possible, a state where G becomes more able to generate high-fitness tests, a batch of the test suite is used for the training. As a matter of fact, it is counterproductive to always train the WGAN on the whole set of randomly generated tests, because it may contain a significant number of low fitness tests. This batch of the test suite is created with a parameter α. This parameter increases through the training, picking low fitness tests at the beginning, therefore, encouraging exploration. Then, α slowly becomes more exigent, until it only selects high-fitness tests towards the end of the training, encouraging exploitation.

It is also worth mentioning that the GAN model used in this competition is the Wasserstein variant. This choice has been motivated by the will to counteract the GAN mode collapse problem. Indeed, one of the side objective of the competition was to generate tests as diverse as possible with the hypothesis that very diverse tests could better help identifying the cause of the failures.

The results of the WOGAN algorithm in the SBST 2022 CPS tool testing competition are presented in [4]. The simulation time budget, i.e., the time the simulator is running, is

Algorithm 4: WOGAN test generation algorithm presented in [28]. This algorithm requires as inputs: an execution time budget called $time_budget$, a number of initial random test N and a target reducer metaparameter called $treducer$. A, the analyzer, is a NN trained online to approximate the fitness prediction. G and C are respectively the generator and the critic of the WGAN model. α is a parameter allowing to select a batch of tests in the set of random tests, it can be updated to pick more or less high-fitness tests. T is the test suite variable and F the corresponding fitness, therefore, $\{T,F\}$ is a list of tuples representing evaluated tests. *clock* is a parameter symbolizing the time. X is the name given to the set of tests used to train the WGAN. Finally, *target* is the fitness threshold for the test candidates.

```
1  Input: time_budget, N and treducer
2  Initialize: A, G, C and α;

3  {T,F} := Generate_And_Evaluate_Random_Tests(N);
4  while clock < time_budget do
5      Train_Analyzer(A, {T,F});
6      X = Sample_Batch({T,F}, α);
7      Train_WGAN(G, C, X);
8      target := 1;
9      repeat
10         test := Generate(G);
11         if Valid(Test) then
12             Continue;
13         end
14         target := target * treducer;
15     until Predict(A, test) ≥ target;
16     fitness := Execute(test);
17     T := T ∪ {test};
18     F := F ∪ {fitness};
19     α := Update_Batch_Parameter(clock, α );
20 end
21 Result: test suite T
```

two hours and the generation time budget, i.e., the remaining time used by the algorithm, is one hour. The WOGAN algorithm is set with $N = 60$, which is equivalent to $\approx 20\%$ of the tests that can be executed in the given budget.

Among the algorithms competing, the WOGAN algorithm uses, by a magnitude, the least time to propose a new test. It is believed that this is the case because WOGAN is not a traditional SBST algorithm and is relatively small, with almost no training data and compact networks. However, concerning the effectiveness of WOGAN, i.e., how many generated tests were actual positive tests, it did not fare very well: only half of the generated tests were positive. On the contrary, other algorithms could propose only positive tests by the use of a validator allowing to validate internally, before simulating the test, if a candidate was a positive test. However, reaching a perfect effectiveness for WOGAN could also have been possible through the use of this validator.
About the diversity of the failing tests, WOGAN could generate a variety of positive tests, but its performance were significantly lower than the winning algorithms. It is assumed that it could be improved by using SUT domain-specific knowledge, in that case mirroring the road (rotation and translation) to produce similar failing tests.

To conclude, even if the WOGAN algorithm did not perform particularly well in the SBST 2022 CPS tool testing competition, it reached comparable performance. WOGAN achieved to not only find positive tests, but also to generate diverse high-fitness tests.

Application of the OGAN technique to performance exploration of REST APIs

The problem tackled in this thesis concerns the automated generation of performance tests for REST APIs. The technique used to generate test cases is the OGAN algorithm. The objective is to automatically generate positive tests, in a given budget, in order to build an "exploration map" of the API. A perfect "exploration map" can be built with an exhaustive search of the API's input space. In the case of a search technique, the goal is to approximate, as well as possible, this perfect "exploration map".
The API is used in a black-box testing approach and without any modeling. The resulting test suite should contain test cases as effective and diverse as possible, and be generated as fast as possible, with the least SUT executions.

PetClinic is a demo application developed with the Spring framework. The Spring framework is an open-source application framework that provides infrastructure support for developing Java applications. PetClinic presents a state-of-the-art software architecture for an application designed with Spring and has a classic architecture which makes it convenient to test. An implementation of the PetClinic as a REST API will be used as an example test system [31].

6.1 REST API black-box Performance Testing

APIs are complex applications containing many functions and methods to test. Without any model and in a black-box testing approach, there is a need to know the API specification. With this specification, resources are identified, their respective endpoints are known and the request formats are available.
OAS is a standard and language-agnostic interface describing REST APIs. It is the most-common API language specification and has the advantage to be easily readable by humans [2]. The PetClinic REST API application uses OAS for its specification and even owns a Swagger UI that can be used to manually send requests to the API.

Measuring performance implies specifying metrics and even fixing requirements, i.e., limits that we want to overpass and that define positive tests. In the case of performance exploration, requirements are not specified as the overall goal is to build a map of performance. Uncovering performance bottlenecks is still the main goal, but we are also interested in knowing where the application performs well.
Concerning the metrics choice, an overall reward can be estimated from the response time, throughput, resource utilization, but also from the status code received. It is necessary to process the information given by the status code, depending on the testing goals. From Table 2.1, we consider that successful requests should provide a feedback made of ratio type variables, such as the response time, throughput or the resource utilization. Client errors with incorrect syntax or non-supported by the API should also be tested in the same conditions. However, feedback from server errors and redirections need to take into account that the application may not have behaved as planned. Especially for server errors, the feedback should take into account that the API failed to fulfill a request. It may be caused by a particularly serious bottleneck or even a functional fault.

6.2 Adaptive function in between the OGAN and the API

6.2.1 PetClinic requests formats

Our test tool is the OGAN model. This model will train and generate test cases by sending requests to the API. However, the OGAN algorithm outputs numbers between 0 and 1, therefore, there is a need to adapt these numbers into an API call. API requests can be relatively simple, see Example 6.1, but also contains many decisive information such as Example 6.2.

```
curl -X GET "http://localhost:9966/petclinic/api/pets"
     -H "Content-Type: application/json"
```

Listing 6.1: **GET** request listing every pet

```
curl -X PUT "http://localhost:9966/petclinic/api/pets/6"
     -H "Content-Type: application/json"
     -H "accept: application/json"
     -d "{\"birthDate\":\"2022-05-31\",
       \"name\":\"Leo\",
       \"type\":{\"name\":\"cat\"}
      }"
```

Listing 6.2: **PUT** request modifying information of a pet

Examples 6.1 and 6.2 show HTTP requests made from a terminal, using the *curl* command, to the PetClinic API. *curl* is a command allowing to transfer data between servers; with the *-X* option it can be used to send HTTP requests to an API. The Request-Line is located just behind the *-X* option.

If the *-H* options and URLs can be explicitly written in the adaptive function, it is necessary that the OGAN algorithm can generate different IDs and body contents (the Message-Body is specified after the *-d* option).
If the **PUT** example is slightly more complex than the **GET** example, both requests are

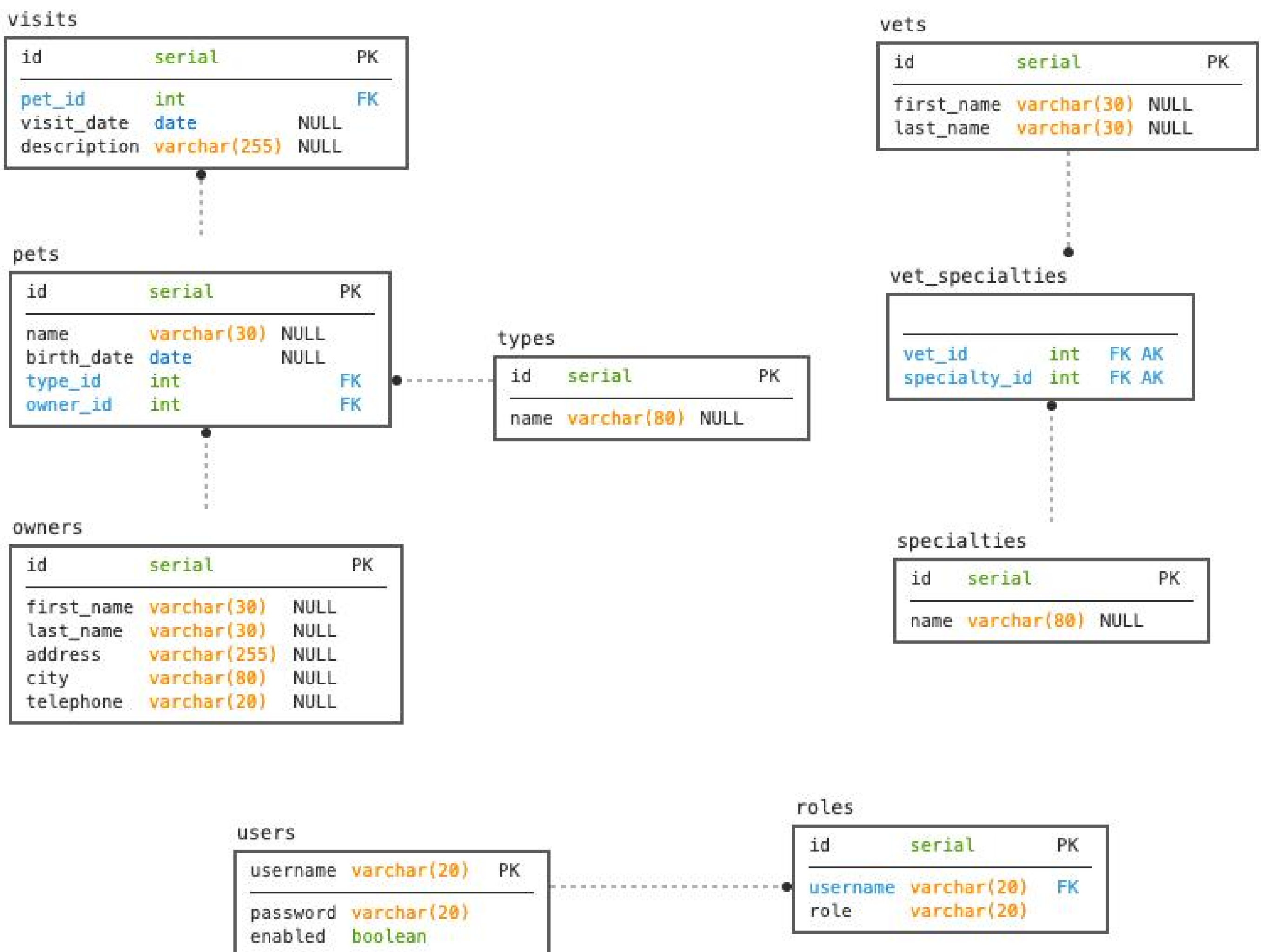


Figure 6.1: Entity-Relationship model of the PetClinic REST API

Source: spring-petclinic, GitHub repository: REST version of Spring PetClinic Sample Application (spring-framework-petclinic extend), 2022

still relatively short. HTTP requests to complex API can contain hundreds of fields and need precise strings or IDs in order to be valid requests.
Figure 6.1 presents the Entity-Relationship model of the PetClinic REST API. Example 6.2 modifies the information of the pet with the ID N°6. After that request that same pet's *birthDate*, *name* and *type* have been changed. It is worth mentioning that the type of the pet is another resource. In this request, a new instance of the *type* resource is created and will be given an ID by the API. The ID of this type, with $name$: "*cat*", will be specified in the API response. Afterwards, if a new request wants to refer to this type, it has to use this ID. However, if the request uses an ID non-existent in the database, then the request is invalid and the response will have the status code 404: *type* not found, see Table 2.1.

6.2.2 Algorithm presentation

The main idea implemented in Algorithm 5 is the following: the OGAN model generates numbers $\in [0;1]$ and we want these numbers to be responsible for a certain endpoint call. Put differently, we create a mapping of the input space and want the OGAN algorithm to be able to access the whole map in order to accomplish an efficient performance exploration.
Depending on the complexity of the test cases we are looking for, it is possible to create variations of Algorithm 5 with a different number of inputs. The simplest version would use only one input. This input would be used to determine which resource, endpoint, and, finally, instance (ID) would be accessed. Moreover, if necessary, this same input would be used to generate the Message-Body, e.g., the *-d* option in the case of Example 6.2.

Algorithm 5 considers that the OGAN generates two inputs: $input[0]$ and $input[1]$. One selects the resource and the endpoint, the other is used as a seed to generate the request body, i.e., the ID and, when necessary, the resource fields. The first step consists in acquiring the API information through its specification. This is done in Line 3 and 4. The algorithm needs a list of available resources, i.e., their URI, and their respective endpoints, i.e., which methods can be called for each resource.

With this information, it is now possible to select the resource and the endpoint to test. This selection is made from $input[0]$. The OGAN algorithm outputs a number $\in [0;1]$, this segment will be divided into smaller segments, as many as resources, see Line 6. Then, an index is selected in Line 7, this formula deduces an integer $resource_index \in [\![0; nb(resource)]\!]$ from $input[0]$ and the number of available resources. Finally, Line 8

Algorithm 5: Adapter function between the OGAN model and the PetClinic REST API. The following pseudo-code presents the main steps mapping the set of numbers from the OGAN to every possible call in the API. This function takes as inputs the *API_Specification*, from which are deduced the available resources and their endpoints, the global URL or *base_address* of the API and the numbers produced by the OGAN algorithm, called *inputs* here.
Get functions lines 3 and 4 initialize the list of available resources and endpoints. *determine* functions select the corresponding resource or endpoint to be called. The $ceil()$ function computes the smallest integer that is greater than or equal to x. It is used to get indexes from *inputs*. *nb_instances* is a metaparameter representing the number of instances of the selected resources; the assumption behind *nb_instances* is explained later.
Line 16, the *generate_arguments* function uses the API specification and the selected endpoint's request format to generate pseudo-random arguments. $inputs[1]$ is used as a seed for this generation.
Finally, the $request()$ method sends the HTTP request to the API and returns the response with the recorded performance metrics. The response's status code, execution time, etc. are then used to deduce a overall *performance* metric used as a feedback for the OGAN algorithm.

1 **Input:** inputs, API_Specification, base_address;

2 **# Get API information from specification**
3 resources := Get_resources(API_Specification);
4 endpoints := Get_endpoints_for_each_resource(API_Specification);

5 **# Select the resource**
6 resource_length := $\frac{1}{nb(\text{resources})}$;
7 resource_index := ceil(nb(resources) * input[0])-1;
8 resource := determine_resource(resources, resource_index);

9 **# Select the endpoint**
10 endpoint_length := $\frac{resource_length}{nb(\text{endpoints})}$;
11 endpoint_index := ceil(nb(endpoints) * (inputs[0] - (resource_length * resource_index)) * nb(resources))-1;
12 endpoint := determine_endpoint(endpoints, endpoint_index);

13 **# Generate ID and body arguments**
14 id := ceil(inputs[1] * nb_instances)-1;
15 address := base_address + resource + id;
16 arguments := generate_arguments(inputs[1], endpoint, resource, API_Specification);

17 **# Send and evaluate the request**
18 response := request(address, endpoint, arguments);
19 performance := evaluate_performance(response);

20 **Result:** performance

extracts the selected resource and its information from the API specification.
Lines 9 to 12 realize the same operation for the endpoint selection. Line 10 further divides the segments depending on the number of available endpoints for this resource. Line 11 computes the selected endpoint index using the same formula as for the resource. The difference is that the $input[0]$ variable is re-scaled in $[0; resource_length]$. Finally, Line 12 fetches the selected endpoint from the specification.

The next step is to generate the last requirements for a valid request. The address that will be requested is computed in Line 15 from: the base address of the API, e.g., *http://petclinic/*, the resource's URI, *resource*, and an ID. This ID may not be necessary if we want to access an entire list of instances. It is assumed that, the API following the REST design, the list of all the instances of a resource has a different name, for example, *pets* for every *pet* in the application. In that case, the ID generation step needs to be skipped. Otherwise, the ID allowing to access a specific instance of a resource is computed from the second OGAN output: $input[1]$.
Finally, the rest of the request arguments, the Message-Body, is generated in Line 12. The $generate_arguments()$ function is discussed in the next subsection, along with the meaning and behavior of the $nb_instances$ metaparameter.

The last step is to call the API, get the response and evaluate the performance. In Line 18, the HTTP request is sent to the API $address$ using the $endpoint$ method, e.g., **GET**, with the $arguments$. Then, Line 19, an overall performance metric is deduced from the response. In practice, it is necessary to observe and record the chosen performance metric before and after the call. As this is a black-box approach, response time, CPU and memory usage, throughput, etc. need to be evaluated from an external point of view. Obtaining this information directly from the API means more precision, but in order to build a performance map and correctly train the OGAN, it is more important to acquire homogeneous records. Finally, we return the API performance as a feedback for the OGAN algorithm.

6.2.3 Discussion and generalization

As previously stated, Algorithm 5 is written in pseudo-code to simplify certain method calls or computations. Furthermore, this algorithm is a template. Depending on the application environment and the goals of the testing process, certain lines or method calls may change.

The Adapter function uses a homogeneous segmentation of the input space to determine the resource and endpoint to be tested. However, in the case of resources with only a couple of endpoints, these endpoints are prioritized compared to a resource with 10 or 12 endpoints. Through time and training, the OGAN algorithm will explore the input space, realize that the space around these endpoints leads to similar performance and adapt. However, upgrading the Adapter function to a different segmentation, e.g., homogeneous regarding endpoints and not resources, can allow endpoints to be equally accessible and, thus, lead to a better sampling of the input space during the initial random test generation.

$nb_instances$ is a metaparameter used to deduce an ID from $input[1]$. The first idea was to fetch it with a first request, in order to know the exact number of instances for this resource. However, it is also possible to fix it, for example with the value 100. This implies that, either the OGAN algorithm will not be able to access every API's resource, or some of the generated IDs will be invalid. In the first case, we can assume than the first 100 resources are representative of the rest, therefore, the overall performance for that request will not change significantly. In the other case, the OGAN algorithm can generate invalid requests. This is useful because we also need to test the performance of the functions managing Client errors. However, in case of a small database, we do not want the OGAN to mostly generate invalid requests. If the $evaluate_performance()$ function takes into account the status code to provide the feedback, then this issue can be fixed over time with training.

Lastly, Line 12 generates the Message-Body. $generate_arguments()$ is a function determining the request format from the API specification and the selected resource and endpoint. It is possible to generate only valid requests, but testing error management is also part of the performance exploration process. We can distinguish two different types of invalid requests. The first type is a request with a wrong format, for example a **POST** request for a new pet with a missing mandatory field, e.g., its *name*. The second type of invalidity is a request with wrong arguments, for example a **GET** request referencing a non-existent pet, i.e., with a wrong ID. These two types of requests produce different responses with diverse status codes, therefore, managed by different parts of the API. If we go further, it is also possible to generate realistic data, such as realistic email addresses. Indeed, in more complex APIs, the data goes through an elaborated data verification (e.g., checking that the string for an email address has a valid format) before processing the request. Generating realistic data can be realized with custom test data generators as in [2], but that also means more computational complexity and training time.

We consider that the presented method can be easily generalized with more inputs. This requires more consideration during the mapping, i.e., the writing process of the Adapter function, but gives to the OGAN algorithm more possibilities as each new input represents a new dimension it can explore. For example, it is possible to use a third input to select the endpoint. Algorithm 5 uses $input[0]$ both for the resource and endpoint selection, but a new input allows the OGAN model to generate more accurate calls. Following the same idea, the body arguments are generated with only one number, $input[1]$. Using more inputs results in more diverse arguments. At its extreme, it is possible to use an *input* for every field we have to generate. In return, more inputs and, therefore, more diversity, means a longer and harder training for the OGAN model. There is need to carefully consider that the model may not converge with a too complex problem.

6.3 OGAN setup and parameters

The OGAN algorithm is a complex algorithm with many parameters. This set of parameters can be broadly customized and is different between each problem. Finding the good combination of parameters is necessary to observe the expected behavior of the algorithm and to optimize its performance.

```
1  job_description =
2  {
3      "sut": "python.PythonFunction",
4      "sut_parameters": {"function": Adapter_function},
5      "objective_func": ["Minimize"],
6      "objective_func_parameters":
7      [{
8          "selected": [0],
9          "invert": True,
10         "scale": True
11     }],
12     "objective_selector": "ObjectiveSelectorMAB",
13     "objective_selector_parameters": {"warm_up": 30},
14     "algorithm": "ogan.OGAN",
15     "algorithm_parameters":
16     {
17         "input_dimension": 2,
18         "use_predefined_random_data": False,
19         "predefined_random_data":
```

```
20          {
21              "test_inputs": None,
22              "test_outputs": None
23          },
24          "fitness_coef": 0.95,
25          "train_delay": 0,
26          "N_candidate_tests": 1,
27          "ogan_model": "model_keras.OGANK_Model",
28          "ogan_model_parameters":
29          {
30              "optimizer": "Adam", "noise_bs": 10000,
31              "d_epochs": 10, "g_epochs": 1,
32              "d_size": 512, "g_size": 512,
33              "d_adam_lr": 0.001, "g_adam_lr": 0.0001,
34              "noise_dimensions": 50, "noise_batch_size": 10000
35          },
36          "train_settings":
37          {
38              "epochs": 1,
39              "discriminator_epochs": 10,
40              "generator_epochs": 1
41          }
42      },
43      "job_parameters": {"max_tests": 80, "N_random_init": 20}
44  }
```

Listing 6.3: JSON setup object for the OGAN algorithm

The OGAN setup is done through a descriptive JSON object, this object is used to start the job and the training, see Example 6.3. This JSON object allows to customize different aspects of the algorithm. The following list is non-exhaustive:

- The objective function and related parameters: what the OGAN should try to do with the feedback, if it needs to invert it and the numbers of inputs required for this function.

- The algorithm used: its main parameters, the number of candidate generated each batch, if the first tests are predefined (i.e., non-randomly generated) and the fitness threshold.

- The OGAN sub-models parameters. The generator and discriminator being artificial NNs, it is possible to customize many of their parameters, including hidden layers size, depth, optimizers, input and output layers, etc.

- Some online training settings, such as the test budget or how many generations (epochs) the discriminator will be trained on, before the generator training.

In order to adapt to our Adapter function between the OGAN model and the PetClinic REST API, we modified the OGAN setup as follows. The objective function has been set as the Adapter function, the model tries to maximize it (equivalent to minimize the inverted feedback of this function), and this function requires two inputs to match Algorithm 5 requirements.

As presented in [10], only one test candidate will be proposed to the discriminator. NNs are of the same size: one hidden layer with 512 neurons. The generator input layer possesses 50 neurons, i.e, 50 noise dimensions, and outputs two numbers between 0 and 1. Therefore, the discriminator input layer is of size 2 and its output layer of size 1, this output corresponds to the predicted fitness of the test.

About the online aspect of the algorithm, the test budget is fixed at 80 executed tests. The algorithm starts with the generation and evaluation of 20 initial random tests, see Algorithm 3. The discriminator is trained 10 times with the current test suite before the generator is trained. Focusing on the discriminator training is required for the generator to correctly be trained. Indeed, the generator feedback highly depends on the capacity of the discriminator to accurately predict positive or negative tests.

6.4 Evaluation of the OGAN REST API performance exploration technique

The OGAN approach tackled in this thesis is a black-box testing approach without any modeling of the system, therefore, it is versatile and can be applied to different REST APIs. The PetClinic application has been chosen as a classic and usual REST API, but the OGAN performance exploration technique should equally perform on same size APIs.

Additionally, the algorithm can properly perform even with a relatively small test budget. The dynamic fitness threshold of the discriminator gives some flexibility in the test generation. Indeed, shortly after the initialization, the generator is still not correctly trained and generate almost "random" tests. The dynamic threshold avoids the generator to be stuck on the first generations, while prioritizing high-fitness tests at the beginning. Another advantage related to the OGAN fitness threshold is that not all the generated tests

are executed on the SUT. This is almost a requirement for classic SBST techniques, but the OGAN algorithm is not a usual search-based algorithm. Moreover, APIs are systems that can be expensive to call.

The aspect making the OGAN algorithm easily applicable is its online learning. Having no prior datasets required to generate tests allows to quickly start the testing process from scratch, because the dataset creation (i.e., manually collecting and evaluating data) is unnecessary. The algorithm starts slowly, due to the randomness of the first tests, but builds its own dataset, which makes it convenient and easy-to-use. Moreover, the generated test suite can be used afterwards as a high-quality dataset, as most of the tests evaluated on the SUT are positive.

In order to properly setup this technique, some work and consideration are still required to select the performance metrics and how the overall performance, used as a feedback for the OGAN, is computed. Even if this step is not the longest, it cannot be completely automated. Also, several trials may be needed, to make sure that the exploration of the API is as complete as possible.

Regarding the evaluation of this technique, the PetClinic REST API application needs to be tested by adding bugs on purpose in order to properly evaluate the OGAN. Prior tests have shown that the OGAN algorithm is able to find performance defects in a reduced version of the PetClinic application (only two resources available: *pet* and *type*). However, implementing this testing process on the whole API to test the OGAN technique may show different results. The PetClinic REST API has 9 resources, each owning in average 5 endpoints and every request to these endpoints can generate around 10 different responses (i.e., status code). A rough approximation shows that the OGAN needs to generate around 450 tests to completely explore the input space. Furthermore, this API stays relatively small and is not comparable to bigger APIs such as the YouTube API [2].

Requirements to generate valid requests can be slow to learn. Certain resources can reference many other resources in the API. For example, the class *vet_specialties* in the PetClinic REST API needs two mandatory IDs (*vet_id* and *specialty_id*), see Figure 6.1. A valid request for the **POST** endpoint of this resource requires the OGAN to generate two existing IDs for the vet and his specialty.
Not updating the API after a request helps the OGAN to map the existent instances in the API and, therefore, to generate valid requests more easily (i.e., the OGAN is not able to **DELETE** an instance it previously learned to use). An alternative and consistent way to

improve the valid requests generation consists in integrating the OGAN arguments generation process with a custom test data generator, in order to create realistic data.

As previously addressed, a limit for the OGAN algorithm is the complexity of APIs. APIs with tremendous number of resources and endpoints are too large to be accurately explored and tested regarding their performance. Trying to achieve this objective would require a serious increase in the test budget.
Finally, testing very complex APIs also means that the GAN-based model can be subject to the mode collapse problem. Selecting the OGAN variant: WOGAN can help solving this problem and be more effective on such complex APIs.

Future Work and Conclusion

Performance Testing is critically important in an application development process. Manually achieving performance exploration comes with high human-cost, test cases prone to error and, in general, inefficiency. In the last years, Machine Learning techniques have been consequently used to automate this process. Framework supporting developers have been created and took part in the complex problem of test case generation. Recently, a new ML technique based on a Generative Adversarial Network (GAN) model has been developed in Åbo Akademi University laboratory. This technique has shown promising results on the SBST 2022 CPS tool competition.

This thesis discussed the REST API performance exploration problem and introduced in details the theory behind Neural Networks and Generative Adversarial Networks. A non-exhaustive literature review concerning the Performance Testing process of websites and its automation has been realized. The online learning process of OGAN has been detailed and discussed, before introducing the related articles published by Ivan Porres *et al.*

Finally, this thesis proposed an application of the OGAN technique on the PetClinic REST API. Primary tests have been driven, to assure that the experiment was possible and interesting, but the implementation of the OGAN as a Performance Testing tool on the PetClinic REST API still needs to be done. This experiment should be planned and realized in the future.

OGANs are solutions adapted to small budget problems due to the online learning process. Moreover, even though the technique includes NNs, these are still relatively compact. The two previous arguments make the OGAN technique interesting for embedded environments and problems. Additionally, the online part of the algorithm makes it very convenient for unattended problems.
On the other side, GANs are known to be subject to some common failure modes, such as the mode collapse problem. If the WOGAN variant helps solving these problems, GAN-based models remain delicate to manipulate.

Bibliography

[1] I. Molyneaux, *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.

[2] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Automated black-box testing of RESTful web APIs," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, pp. 682–685, Association for Computing Machinery. event-place: Virtual, Denmark.

[3] BeamNG GmbH, ""beamng.tech"." [Online], *https://beamng.tech* (accessed Dec. 2021).

[4] J. Peltomäki, F. Spencer, and I. Porres, "Wasserstein generative adversarial networks in the Search-Based Software Testing 2022 Cyber-Physical Systems tool competition," in *The 15th Intl. Workshop on Search-Based Software Testing*, SBST 2022.

[5] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[6] M. Helali Moghadam, M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper, "Machine learning to guide performance testing: An autonomous test framework," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 164–167.

[7] M. H. Moghadam, "Machine learning-assisted performance testing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pp. 1187–1189, Association for Computing Machinery. event-place: Tallinn, Estonia.

[8] E. Isaku, "Machine Learning-Assisted Load Testing," Master's thesis, Mälardalen University, School of Innovation, Design and Engineering, RISE Research Institutes of Sweden AB, 2021.

[9] C. H. Kao, C. C. Lin, and J.-N. Chen, "Performance testing framework for rest-based web applications," in *2013 13th International Conference on Quality Software*, pp. 349–354, 2013.

[10] I. Porres, H. Rexha, and S. Lafond, "Online gans for automatic performance testing," in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 95–100, 2021.

[11] N. Srinivas and K. Deb, "Muiltiobjective optimization using nondominated sorting in genetic algorithms," *Evolutionary Computation*, vol. 2, no. 3, pp. 221–248, 1994.

[12] J. Knowles and D. Corne, "The pareto archived evolution strategy: a new baseline algorithm for pareto multiobjective optimisation," in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, vol. 1, pp. 98–105 Vol. 1, 1999.

[13] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," *TIK-report*, vol. 103, 2001.

[14] A. J. Nebro, J. J. Durillo, F. Luna, and E. Dorronsoro, B. Alba, "MOCell: A cellular genetic algorithm for multiobjective optimization," *Evolutionary Computation*, vol. 24, pp. 726–746, 2009.

[15] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992. Introduction to Genetic Algorithms.

[16] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper, "An autonomous performance testing framework using self-adaptive fuzzy reinforcement learning," *Software Quality Journal*, pp. 1–33, 2021.

[17] E. Grossi and M. Buscema, "Introduction to artificial neural networks," *European Journal of Gastroenterology & Hepatology*, vol. 19, no. 12, 2007.

[18] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, Dec 1943.

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

[20] Brit Cruise, "Brit Cruise Website, Deep Learning serie." [Online]. Available: https://britcruise.com, 2012-2021.

[21] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[22] A. Brock, J. Donahue, and K. Simonyan, "Large scale gan training for high fidelity natural image synthesis," *arXiv preprint arXiv:1809.11096*, 2018.

[23] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[24] M. Abadi and D. G. Andersen, "Learning to protect communications with adversarial neural cryptography," *arXiv preprint arXiv:1610.06918*, 2016.

[25] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[26] Google Developers, "Generative Adversarial Networks | Google Developers." [Online] Available at: <https://developers.google.com/machine-learning/gan/> [Accessed 9 May 2022], 2022.

[27] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 214–223, 2017.

[28] J. Peltomäki, F. Spencer, and I. Porres, "Wasserstein generative adversarial networks for online test generation for cyber physical systems," in *The 15th Intl. Workshop on Search-Based Software Testing*, SBST 2022.

[29] J. Peltomäki and I. Porres, "Falsification of multiple requirements for cyber-physical systems using online generative adversarial networks and multi-armed bandits," in *The 6th. Intl. Workshop on Testing Extra-Functional Properties and Quality Characteristics of Software Systems*, ITEQS 2022.

[30] I. Porres, T. Ahmad, H. Rexha, S. Lafond, and D. Truscan, "Automatic exploratory performance testing using a discriminator neural network," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 105–113, 2020.

[31] spring-petclinic, "REST version of Spring PetClinic Sample Application (spring-framework-petclinic extend)." *https://github.com/spring-petclinic/spring-petclinic-rest*, 2022.