SISSA

Scuola Internazionale Superiore di Studi Avanzati

Mathematics Area - PhD course in
Mathematical Analysis, Modelling, and Applications

A Reduced Order Approach for Artificial Neural Networks applied to Object Recognition

Candidate:
Laura Meneghetti

Advisor:
Prof. Gianluigi Rozza
Co-advisor:
Dr. Nicola Demo
Industrial Supervisor:
Dr. Daniele Turrin

Academic Year 2021-22

What is, then, that makes scientists wander about in universe
of ideas and experimentation? It may be the search for knowledge
or, in more mundane terms, simple curiosity. Nagging questions;
the pressing need to figure something out and the inability to do
anything else until the answer is found; the tingling feeling that a
discovery may be just around the corner; the intuition that a puzzle
is starting to take shape, until eventually one reaches the answer
and feels the thrilling joy of understanding.

—Rodrigo Quian Quiroga—
Borges and Memory: Encounters with the Human Brain

Con lo scudo o sopra di esso.

Ringraziamenti

Arrivati alla fine di questo viaggio mi sento di dover ringraziare tutte le persone che ne hanno fatto parte. Mi scuso intanto per il passaggio all'italiano, necessario ad esprire al meglio i miei ringraziamenti ed emozioni.

Ringrazio innanzitutto il mio relatore Gianluigi Rozza per essere stato in questi anni non solo un'ottima guida nel mondo della ricerca, ma anche una persona con una grande empatia e un gran lato umano. Non è così scontato avere un relatore su cui poter contare e che creda sempre in te, anche nei momenti di difficoltà, e per questo motivo io mi sento estremamente fortunata e grata.

Ringrazio poi il mio correlatore Nicola Demo per il grande aiuto che mi ha dato in questi anni e forse anche per la pazienza che ha avuto nei miei confronti. All'inizio le mie conoscenze nella programmazione erano davvero poche, ma grazie ad un continuo confronto sono riuscita ad imparare un sacco di cose e spero anche di essere migliorata nel mentre.

Ringrazio anche tutto il mio gruppo di ricerca, SISSA MathLab, per avermi accolto fin da subito con grande entusiasmo. La cosa bella di essere in un gruppo non è solo poter conoscere più persone (da disturbare), ma anche avere sempre un punto fisso su cui contare nei momenti felici e anche in quelli più tristi. Non posso elencare ogni persona del gruppo perchè sarebbero troppe e rischierei di dimenticarmi qualcuno, ma ci tengo a ringraziare in particolar modo Andrea Mola per essere stato il mio primo tutor in SISSA e un grande amico.

Un ringrazimento va anche ad Electrolux Professional, ed in particolar modo a Riccardo Furlanetto, per avermi permesso di vivere tutto questo. Un dottorato accademico ed industriale non è sicuramente un percorso facile, ma si è rivelato essere una bella esperienza, in cui ho avuto la possibilità di vedere la produzione di elettrodomestici dal vivo, capire come funzionavano e tutta la bellezza che sta dietro questo mondo. Ringrazio quindi tutte le persone che ho incontrato nel gruppo R&D e nel Research Hub per aver allietato le mie giornate a Pordenone: Riccardo, Daniele, Viktoria, Alessandro, Davide, Mauro, Michele, Emidio, Eleonora, Debora, Bermet, Juri, Yuri, Giulia, Andrea C., Andrea A., Federico.

Ringrazio poi tutte le persone che mi hanno supportata e sopportata costantemente in questi anni. Inizio con la mia famiglia che rappresenta e sempre rappresenterà una parte importante della mia vita. Li ringrazio per esserci sempre e per farmi sentire coccolata ogni volta che torno. Ringrazio in particolar modo mia sorella che è la mia prima sostenitrice da sempre e anche una buona amica.

Un ringraziamento enorme va ai miei compagni di dottorato: Sara, Luigi, Ale, Carlo, Maggico, Cianci, Stefano, Dario. Mi risulta difficile riassumere questi quattro anni senza emozionarmi. Siamo cresciuti così tanto da quando ci siamo conosciuti e non posso che essere felice di aver condiviso questo percorso con voi. Ringrazio poi i miei compagni di 402 passati e presenti, Pier, Moaad, Giulio, Maria, Matteo, Sarina, Audrey, per non avermi (ancora) cacciato dall'ufficio per le troppe lamentele ed aver ascoltato sempre con pazienza le mie storie avvincenti. Un ringraziamento va anche a tutti i miei amici della SISSA e a tutte le persone conosciute in questi anni che hanno allieatato questo bellissimo periodo. Fra pochi giorni inizierà un nuovo percorso e spero sarete ancora tutti qui a tenermi compagnia.

Un ringraziamento va poi a tutte le persone che ho incontrato e che hanno caratterizzato questi anni triestini. Parto con Irene S., una delle prime persone che ho conosciuto a Trieste, amica fidata con cui condividere disagi, pizze e gelati al pistacchio. Poi non posso non ringraziare le mie compagne di avventure in convento: Sofia, Miriana e Giorgia, con cui ho condiviso molte feste, gite, esperienze culinarie e racconti. Ringrazio poi Barty, Pietro e Giorgia (again) per essere parte

di quel Dream Team che tutti sogniamo. Un grazie va anche a Marija e Nicola per tutte le belle serate passate assieme. Ringrazio poi Cassie, Ele ed Enrico per avermi fatto assaporare quegli anni universitari che non ho vissuto appieno, facendomi sentire parte integrante del vostro gruppo. Ringrazio infine Francesco C., perchè nonostante tutto *mai per caso nulla accade*.

Abstract

A major breakthrough in the field of computer vision and image processing has been represented by the introduction of Convolutional Neural Networks (CNNs). They are highly accurate models able to classify and localize multiple objects of different classes in the same image or video. Despite their impressive success in solving many complex tasks, such as image recognition and object detection, these architectures are characterized by a high number of degrees of freedom, resulting in a longer optimization step and, on a practical side, a bigger architecture to manage. In the academic community, the dimension of these networks is not considered a bottleneck of this methodology, contrarily to what happens in many engineering fields, where CNNs may be applied in embedded systems with limited hardware. In this industrial context, real-time performance, robustness of algorithms, and fast training processes are indeed fundamental properties required from the developed models.

This thesis investigates thus the effective deployment of CNNs in engineering fields, and in particular in embedded systems. The restricted resources, such as memory constraints, in these processors, have led to the necessity of designing methods to create light weights versions of the original model. Based on techniques widely used in the Reduced Order Modeling (ROM) community, such as Active Subspaces (AS) and Proper Orthogonal Decomposition (POD), we have developed a dimensionality reduction framework lowering the number of degrees of freedom of the network. The original CNN is split into two cascading parts, the pre-model, and the post-model, based on the choice of a cut-off index, identifying the layer at which we are cutting the net. Since we are retaining only the layer contained in the pre-model, this index represents a key parameter, that takes into account the information we are discarding by replacing the post-model. The core of our approach lies then in the reduction layer, which aims at projecting the high-dimensional feature maps of the pre-model in a low-dimensional space constructed using the aforementioned techniques. Finally, a response surface is introduced to create a mapping between the reduced feature maps and the final predicted output of the model. Two different methods have been tested also in this case: the Polynomial Chaos Expansion (PCE) and a Feedforward Neural Network (FNN).

We have thus employed our proposed methodology to image recognition and object detection, two problems of great interest to the company we have worked with. This project has indeed been conducted within an industrial Ph.D. grant financed by Electrolux Professional. Therefore, we have tested our reduction technique with a custom dataset, collected in the laboratory of the company, as well as a benchmark one for more academic purposes. We have thus obtained a reduced version of the original network, containing just a few of the initial layers and demonstrating a reduction of the network dimension. Furthermore, our experiments show that the reduced architectures can achieve a level of accuracy similar to the original model, gaining a remarkable speedup in the fine-tuning of the network in a transfer learning context.

Contents

List of Figures

List of Tables

Introduction

In the last decades a growing amount of researchers have focused their attention on the study of the human brain, and in particular on algorithms that can mimic its main functions: memorization, learning, recognition, and retrieval of objects [7, 252]. A first attempt at reproducing such tasks is represented by Artificial Neural Networks (ANNs) [165, 42, 197, 115], biologically inspired mathematical models composed of artificial neurons grouped to form different architectures. The power of such algorithms lies in their ability to learn from data, through the so-called *training process*. In particular, the backpropagation phase is responsible for tuning their parameters to gain predicted outputs close to those expected. This great property is also justified by a classical result in this context, the *Universal Approximation theorem for Neural Networks* [58, 137], stating that sufficiently wide (shallow) neural networks can approximate any (continuous) function. This opened the possibility to apply ANNs to a wide range of fields to address approximation problems, such as in the context of Reduced Order Modeling (ROM) [21, 22, 23, 261, 267, 260, 306, 262, 325, 235]. The Proper Orthogonal Decomposition-Artificial Neural Network (POD-ANN) [125, 268, 46] approach couples indeed the POD method [126, 22, 262] with ANNs to reconstruct the functional relationship between input parameters and output solution field in a fast and reliable way. The benefits of this technique has thus led to its application in several fields ranging between automotive [332], casting [279], combustion [316], bifurcating fluid phenomena [239, 122, 124], and hemodynamics [287, 288]. Furthermore, the scientific community has also focused its attention on a better mathematical characterization of the approximation properties of data-driven techniques based on ANNs, as represented by the Neural Network shifted-Proper Orthogonal Decomposition (NNsPOD) [234, 233] algorithm. Another growing impact approach coincides with the Physics–Informed Neural Networks (PINNs) [243, 216, 284, 274], employed successfully for solving both forward and inverse problems for PDEs for various applications [204, 63, 198, 244].

The accuracy of such models is then strictly related to the number of layers, neurons, and inputs [99, 155, 310], therefore, to tackle even more complex problems, architectures are forced to go deep. These huge structures have thus led to the creation of *Deep Learning* [99, 274, 4], a thriving field with many practical applications, as visual object recognition [167, 48], robotics [226, 240], speech recognition [104, 151, 224], natural language processing [328, 70], radiomics and medical imaging [313, 16, 49, 80], autonomous vehicles [139, 145], and digitalization [113]. Alongside this wide range of tasks, it has started to show its impact also on regression problems [20, 291], and on data-driven inverse problems [13, 190], such as image reconstruction and image restoration.

A particular type of ANN, that has achieved impressive results in dealing with computer vision problems, is represented by Convolutional Neural Networks (CNNs) [6, 172, 99]. They are indeed widely used to tackle and solve difficult tasks, such as *image recognition* and *object detection*, the main topics of this thesis. Image recognition faces the problem of recognizing the items represented in pictures, whereas object detection deals also with the localization of the detected objects. The key ingredient for solving such problems is represented by the feature learning process, characterizing the building block of CNNs. Hence, to distinguish between different items in a picture, special structures, called *filters*, are introduced to detect object features. In particular, the first layers are responsible to capture low-level features, such as minor details of the image, while the later layers will learn to recognize high-level features, namely detect objects and larger shapes in the picture. Therefore, by stacking these filters in several layers, several architectures can be constructed to solve the aforementioned tasks. Regarding the object detection problem, there is also the need of introducing an additional structure to localize the items and to construct the *bounding boxes*, that wrap around the object providing the coordinates of its position.

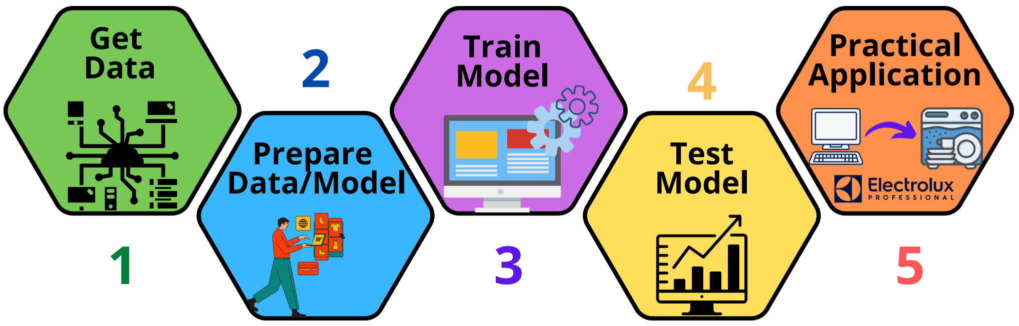


Figure 1: Fundamental steps for developing a neural network model.

This work focuses thus on developing a model that can solve the aforementioned problems of image recognition and object detection for a test case connected with the practical application inside a professional appliance for *Electrolux Professional*. We collaborate with this company during the doctorate, following all the needed steps to prepare and use the CNN framework for these tasks. Such steps are summarized in Figure 1 for sake of clarity. The first key ingredient is represented by the *dataset*. There can be two different choices, based also on the goal of the project: use a benchmark or a custom dataset. The latter is the common case in the industrial field, where there is the need to have data that are not general, but specific to the problem under consideration. In the research framework, instead, the interest is slightly different, more connected to having a comparison of the developed network with the state of the art, done using benchmark datasets. The second fundamental choice is represented by the model. At the beginning, it is typical to use an already implemented CNN model, created to solve a particular task, but this can represent also the starting point for developing a new architecture. The accurate selection of the model and the preparation of the data, to have the correct structure to use, are thus the core of the second step. The third phase is a crucial point in the development process because the accuracy of the final model depends on it. In fact, here is where learning takes place through the training of the CNN. We provide some input-output pairs and we want that the model learns the existing association rule between them. Once a trained model has been created, we should use the acquired knowledge to understand how accurate our CNN is in making predictions on new samples. We are thus testing the generalization capability of our algorithm, namely if it can solve the same problem having different inputs, never seen before. At the end of this phase, we should have a well-performing CNN for the required task, otherwise an additional fine-tuning of the CNN parameters is needed. The last step coincides then with the practical application of our algorithm to solve a real-world problem in the industrial field. In our case, the developed CNN has to run in an embedded system placed inside a professional appliance and characterized by restricted resources, such as strict memory constraints. Our model requires more space than available, leading thus to a dimensionality problem and the necessity to create a reduced version of our CNN. The collaboration with *Electrolux Professional* in this Ph.D. project was thus fundamental to understand the needs and bottlenecks in applying such algorithms inside professional appliances.

Such industrial experience has confirmed last year's trend in developing even more deep (and so heavy) architectures to tackle complex problems. If on one hand, we have an increasing precision, on the other hand, the high number of degrees of freedom results in a longer optimization step and, on a practical side, a bigger architecture to manage. Whereas in the past, the main problem was having enough computational power to make these machine learning algorithms work, nowadays the fundamental goal is providing real-time solutions from the developed models. While the training phase can be performed offline on servers, the final testing phase has to be performed locally on the professional appliance, where it should be applied. Hence, a great effort is made to improve the efficiency of these deep learning algorithms to move the inference phase from servers to embedded devices, usually characterized by limited hardware [259]. The dimension of the network represents thus a bottleneck in the application of CNNs in many engineering fields, and in particular in these limited hardware processors, leading to the necessity of a reduction in the number of degrees of freedom of the network.

Finding the intrinsic dimension of neural networks is a very challenging task that has attracted a lot of interest in the last decades, thanks also to the increasing application of ANNs. To the best of the authors' knowledge, there exist approximation results providing estimates on the necessary size required for certain approximation tasks [17, 213, 327, 280, 241], but there are no rigorous theoretical proofs determining the precise number of ANN parameters. However, for the purpose of obtaining light-weights ANNs, different methods have been proposed, such as network pruning and sharing [110, 196, 192], low-rank matrix and tensor factorization [266, 336, 227], parameter quantization [56, 69], manual architecture design [188, 138, 335, 141], and neural architecture search [259, 342, 40, 302]. Despite the great benefits connected with these approaches, most of these techniques are not changing network architectures by performing an analysis on redundant or non-significant information, but are only deleting model parameters or manually designing layers and network structures. To face this problem, we have instead explored the idea presented in [57], extending it to have a more general approach [210, 211]. Therefore, mimicking the procedure presented in [57], the reduced network is constructed starting from the original architecture, initially split into two cascading parts, the *pre-* and *post-model.* Assuming that the latter brings a negligible contribution to the final outcome, we are retaining only the knowledge contained in the first layers and replacing the remaining ones with an input-output mapping. This response surface is indeed built to fit the data, with the aim of approximating such part of the model without introducing a larger error. Since the output features of the pre-model belong to a high-dimensional space, this implies the necessity of a dimensionality reduction approach to reduce the pre-model outputs, which corresponds also with the input parameters of the response surface. The reduced network is thus constructed by splitting the net into two parts — the pre-model and the input-output mapping — connected by the reduced method, which helps in reducing the typically large dimensions of the intermediate layers by keeping the reduction computationally affordable. In this way, we obtain a reduced version of the network by performing a smart selection of the main parameters of the network, which allows us reducing the required resources and the computing time to infer the model.

Starting from [57], where the Active Subspaces (AS) [51, 53] and Polynomial Chaos Expansion (PCE) [320] are exploited to create a reduced version of the original network, we have investigated other mathematical tools to provide a generic framework for neural network reduction. In [210, 212], we have indeed explored another technique widely used in the ROM community, namely POD [126, 22, 306, 262], that similarly to AS, compress the data by projecting it onto a low-dimensional space. To construct the input-output mapping we have then applied a fully connected Feedforward Neural Network (FNN), recalling the common classification part in a CNN architecture. It is, in particular, characterized by few layers and neurons, reducing further the already minimal space demand of the PCE method. Employing another neural network to approximate part of an original ANN has also the advantage of making the software integration easier, especially when the hosting system is embedded.

Combining all these ingredients, we have designed a reduction method to create a light-weight version of a general ANN. In particular, we have applied the proposed technique to CNNs and object detectors in order to solve the aforementioned problems of image recognition and object detection. Our experiment shows that the reduced nets obtained can achieve a level of accuracy similar to the original model under examination while saving in memory allocation. Furthermore, our method performs better on a simple CNN with respect when applied to the object detection framework, due to the additional complexity of localizing the objects in a picture. Improving the performances in this context represents a possible continuation of this thesis work. We specify also that the proposed method has been tested against a realistic industrial dataset, but also benchmark ones.

In this thesis, we start by outlining in Chapter 1 the most important features of ANNs. We present the building elements of their structure, called neurons, combining which different neural network topologies can be created. We focus then on the description of FNNs and, in particular, on the learning process, the fundamental step to obtain a well-performing model. Finally, we report some commonly used initialization strategies for ANN parameters.

Chapter 2 is devoted to the introduction of CNNs. First of all, we provide a detailed mathematical description of the fundamental layers composing their architectures. Then, we focus on two main problems, introduced before: image recognition and object detection. Hence, we present for each task a review of commonly used datasets and models.

In Chapter 3 we cover in detail our proposed framework to develop a reduced version of an ANN in order to overcome the constraint of memory storage usually connected with embedded systems [210, 211, 212]. We provide, first of all, an algorithmic overview of all the numerical methods involved in the reduction framework, namely AS, POD, PCE, and FNN, and then the application in the context of image recognition and object detection. Furthermore, we present the results obtained by reducing with the proposed methodology a benchmark CNN and object detector testing against different datasets, benchmark and custom ones.

Finally, some conclusions and future perspectives follow.

This thesis has been carried out in cooperation with the Research Hub[1] by *Electrolux Professional* in the framework of an industrial doctoral grant agreement.

[1]The Research Hub is a technology enabling agent, bringing together several universities and research centers within *Electrolux Professional*. More details can be found on the related web-page: *https://theresearchhub.electroluxprofessional.com/*

CHAPTER 1 Artificial Neural Networks

1.1 Introduction

Artificial Neural Networks (ANNs) represent a trending and debated topic, thanks to their power to solve complex tasks and a variety of real-world problems coming from several different fields, such as neuroscience, psychology, mathematics, physical sciences, and engineering. The basic idea of ANNs is the reproduction of the human brain, in particular, the way of thinking and making decisions, using appropriate structures and architectures [7, 252]. Before diving deep into these topics, it is important to understand why they became such popular for many applications.
The first pioneering work that started the success of ANNs was that of McCullogh, a psychiatric and neuroanatomist, and Pitts, a mathematician, published in 1943 [208]. They proposed a first computational model of a neuron, called *MCP neuron*, that mimics the functionality of a biological neuron, by unifying neurophysiological studies and mathematical logic. This type of model is also referred to as *all-or-none* since the inputs are of boolean type and also the output is boolean, in analogy with the biological neuron where the input and output signals can be excitatory and inhibitory. Another essential element introduced in this construction is that of *time* and in particular of *cycle-time*, i.e. the time needed to provide the output of an operation using a network. Hence, a sufficient number of these MCP neurons can then be combined into little temporal sequences to create a network, constituted of these simple units connected through synaptic connections, that was argued to be able to perform any operation of the calculus of propositions.
In 1949, Hebb proposed in [121] an explicit statement of a physiological learning rule for synaptic modification following up an early suggestions by Ramón y Cajál [41]. Hebb's postulate of learning, known as *Hebb's rule* concerns synaptic plasticity, which corresponds to the ability of the brain to change and adapt to new information by strengthening or weakening the synapses over time. This represents a great achievement in the field of neural networks since he was the first to describe brain connectivity as dynamic, i.e. as something continually changing while learning different functional tasks. For this reason, Hebb's work became a source of inspiration for the development and application of computational models of learning and adaptive systems, as happened in 1954 for Farley and Clark, who were the first to use computational machines (*calculators*) to simulate Hebbian networks [84]. Another important attempt to use computer simulation to test a neural theory based on Hebb's postulate of learning was that of Rochester, Holland, Habit, and Duda in 1956 [251]. In the same years, Uttley in [312] suggested the importance of classification in the organization of the nervous system by demonstrating that a neural network with modifiable synapses may learn to classify simple sets of binary patterns into corresponding classes.
In the 1950s the interest in *associative memory* started to grow, i.e. in the ability of the brain to learn and remember the existing relationships between uncorrelated items. Some significant contributions are represented by Taylor in 1956 [304], and by Anderson [8], Kohonen [161], and Nakano [223] in 1972. The novel feature introduced by the latter work is the idea of a *correlation matrix memory*, storing existing associations between patterns, based on an outer product learning rule, as the one of Hebb.
In 1958 Rosenblatt introduced in [257] a novel approach to the pattern recognition problem by creating a supervised learning method called *perceptron*. Starting from the MCP neuron model, some improvements were brought including the Hebbian theory of learning. In particular, by relaxing some rules that characterized the MCP neuron model — e.g. the equal contribution of all inputs, their

integer nature, … — Rosenblatt devised a model, composed of artificial neurons and with a single layer of output neurons[2], able to learn from data and thus figure out the correct weights directly from training data. The *Perceptron learning rule convergence theorem* summarized this achievement, giving the idea that neural networks could solve any problem. For this reason, neural network models started to become very popular with the creation of also multilayer networks [143].
In 1969 Minsky and Papert demonstrated in [215] that perceptrons have fundamental limits on what they can compute and that there was no reason to assume that multilayer neural networks could overcome the problems and limitations of perceptrons. The low processing power of computers at the time was also not able to handle the computation required by large neural networks, slowing the research in that field. The 1970s were thus characterized by a dampening of continued interest in neural networks, except for psychologists and neuroscientists.
The 1980s represent a resurgence of interest in neural networks since new and important contributions to the theory and design of neural networks were made. Firstly, there was the introduction of a class of neural networks with feedback in 1982 by Hopfield [135]. This model, called *Hopfield Network*, attracted great interest because an energy function was introduced to understand the computation performed by recurrent networks with symmetric synaptic connections. In this way, the stored patterns represent the equilibrium points for the dynamics of the net and in particular Lyapunov stable points for the defined energy function. Furthermore, Hopfield Networks established a strong link with physics [7], since Hopfield proved the existence of an isomorphism between such a recurrent network and the Ising model [142], commonly used in statistical physics. This model paved thus the way for an interest in attractor neural networks and in studying the stability of this type of content addressable memory, as done by Cohen and Grossberg [50] in 1983.
In 1985 Ackley, Hinton, and Sejnowski developed the first successful realization of a multilayer neural network, called *Boltzmann machine* [2], using simulated annealing [159], a new procedure for solving combinatorial optimization problems. In the mid-1980s, there was then the rediscovery of the *backpropagation algorithm* by Rumelhart, Hinton, and Williams [263] — initially found by Werbos in 1974 [319] —, that has emerged as the most popular learning algorithm for training multilayer perceptrons. In 1986 Rumelhart and McClelland introduced in [264] the idea of parallel distributed processing to simulate neural networks, known by the name of *connectionism.*
1985 represents then a turning point year since it was held the first *Neural Networks for Computing* meeting by the American Institute of Physics, followed in 1987 by the first *IEEE annual international ANN conference.* ANNs were thus definitely attracting a lot of attention and interest, catching the imagination of the world in solving increasingly complex problems, such as image recognition.
In 1989, Yann LeCun developed a new machine learning method, biologically inspired, to recognize handwritten digits, then called *Convolutional Neural Networks (CNNs)* [171, 170]. 1997 saw then the introduction of a type of *Recurrent Neural Network (RNN)*, called *Long Short-Term Memory* (LSTM), by Schmidhuber and Hochreiter [134], characterized by a truncated backpropagation version to deal with the vanishing gradient problem. In fact, despite the great achievements and results, CNNs, RNNs and in general deep neural networks, i.e. models with more stacked layers, brought also a new issue connected with slow and unstable training processes due to vanishing gradients problems during backpropagation [133].
Despite this, 2010 represented the year of the growth and expansion of *deep learning*. To improve the training step of deep networks, a great effort was put into studying a specific activation function to transmit information, e.g. ReLU [222], and better optimization algorithms, such as ADAM [158]. A great role in this *big bang of deep learning* was also played by Nvidia™, which created the first Graphics Processing Unit (GPU) to support the training of much deep learning neural networks [273]. 2012 was then the year in which CNN brought a lot of attention and interest in them since they

[2]Perceptron is a Feedforward Neural Network with a single layer of output neurons, i.e. composed only of the input and output layer.

were exploited by Krizhevsky in the ImageNet Large Scale Visual Recognition Challenge, a data science competition to classify pictures, achieving great results and error rates using also GPUs for training [167].
Nowadays, ANNs are used in several applications to solve complex tasks such as regression problems [20, 291], data-driven forward and inverse problems [243, 274], visual object recognition [167, 48, 226], natural language processing [328, 70], speech recognition [104, 224], autonomous vehicles [139, 145], and digitalization [113]. The architectures of these models have also undergone a great improvement with respect to the model of McCullogh and Pitts, and the perceptron: they are not composed of only one layer, but they are characterized by particular structures, suitable for overcoming specific difficulties, and by having many layers. Certainly, neural networks have not come to an end, but still, have a great deal of room for growth and development in many research and engineering fields.
In this Chapter, we will thus discuss ANNs by presenting their main characteristics. We will start in Section 1.2 with the base element of a network, that is the neuron. After we have collected the main notions of it, we can construct the ANN itself. Section 1.3 will thus be in charge of presenting what is an ANN and its capacity of learning tasks. In Section 1.4 the typical architecture of an ANN is discussed, providing also an overview of different neural network topologies. Section 1.5 will concentrate on Feedforward Neural Networks (FNNs) and the explanation of the backpropagation algorithm. To conclude, Section 1.6 describes the main strategies to initialize an ANN.

1.2 Neuron: The Base Element

The building block of an Artificial Neural Network is the so-called *abstract neuron* [42, 165], a unit element that tries to mimic how a biological neuron works. It can be formally defined in the following way:

Definition 1.2.1 (Abstract Neuron [42]). *An abstract neuron is a quadruple* $(\mathbf{x}, W, \sigma, \hat{\mathbf{y}})$ *where*

- $\mathbf{x} = (x_0, \dots, x_n)^T \in \mathbb{R}^n$ *represents the input vector*[3]*;*
- $W = (w_0, \dots, w_n)^T \in \mathbb{R}^n$ *is the weight vector;*
- $w_0 = b$ *and* $x_0 = -1$ *are the bias and the corresponding input signal, respectively;*
- σ *is the activation function;*
- $\hat{\mathbf{y}} \in \mathbb{R}$ *is the outcome of the neuron, determined by*

$$\hat{\mathbf{y}} = \sigma(\mathbf{x}^T W) = \sigma\left(\sum_{i=0}^{n} w_i x_i\right). \tag{1.1}$$

Therefore, it can be described as a unit that given some inputs (incoming signals) can provide an output using the corresponding weights (synaptic weights) and an activation function (neuron firing model). We have placed in brackets the corresponding neuroscience terms in order to highlight the structure's analogy with a biological neuron.
Definition 1.2.1 introduces the key ingredients connected with a neuron, depicted in Figure 1.1. As described before, the 0th component of the weight and input vectors represents the bias and the corresponding input signal. The bias is an additional constant parameter added to help the model in

[3]Here we are using the notion of vector for simplicity, but in general the input $\mathbf{x}$ can also be a matrix or a tensor, as in the case of Convolutional Neural Networks treated in Chapter 2.

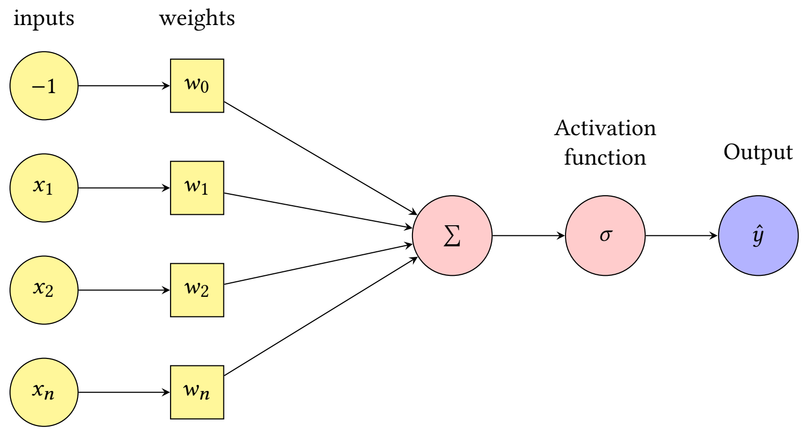


Figure 1.1: Schematic structure of a neuron.

fitting better the given data, therefore the associated input component has to be equal to 1 or -1. In this case $x_0 = -1$ following the same convention used in [42], but equivalently it can be considered the bias $w_0 = -b$ and the corresponding input $x_0 = 1$.
A crucial role is played by the activation function σ since it introduces nonlinearity in the model. We present here a list of common choices [165, 42, 331], depicted in Figure 1.2:

- **Step functions**: biologically inspired activation functions characterized by an upward jump that models a neuron activation [208]. An example is represented by the *signum function*, see Figure 1.2 (a), and is described in the following way:

$$sign(x) = \begin{cases} -1 & \text{if } x < 0, \\ +1 & \text{if } x \geq 0. \end{cases} \tag{1.2}$$

- **Rectified Linear Unit (ReLU)**: commonly used in the context of image recognition since it does not saturate and speeds up the learning process [222, 167]. The ReLU function is linear for $x \geq 0$, see Figure 1.2 (b), and is thus defined by:

$$ReLU(x) = \max\{x, 0\} = \begin{cases} 0 & \text{if } x < 0, \\ x & \text{if } x \geq 0. \end{cases} \tag{1.3}$$

- **Softplus function**: a smoothed version of the ReLU function [75, 98]. As depicted in Figure 1.2 (c), it is an increasing positive function given by:

$$sp(x) = \frac{1}{\beta}\log(1 + \exp(\beta x)), \qquad \text{for } \beta \in \mathbb{R}. \tag{1.4}$$

- **Sigmoid functions**: smoothed version of the step functions [106]. An example is represented by the *logistic function with parameter c>0*, see Figure 1.2 (d):

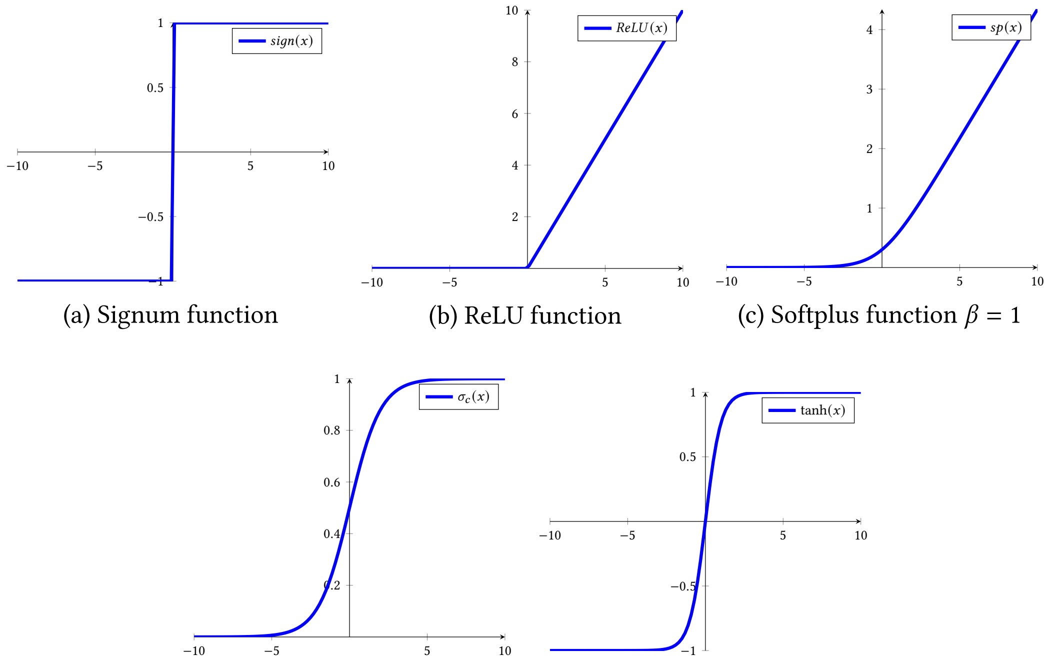
$$\sigma_c(x) = \frac{1}{1 + \exp(-cx)}, \tag{1.5}$$

  where c controls the firing rate of a neuron, since larger values of c lead to a fast change from 0 to 1.

Another example of a sigmoid function is the *hyperbolic tangent* defined by:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{1.6}$$

where, as can be seen in Figure 1.2 (e), it has two horizontal asymptotes at $y = \pm 1$.



(a) Signum function (b) ReLU function (c) Softplus function $\beta = 1$

(d) Logistic function $c = 1$ (e) Hyperbolic tangent

Figure 1.2: Some examples of activation functions.

1.3 Artificial Neural Networks

Starting from the base element described in the previous section, we can define an *Artificial Neural Network (ANN)* [165, 42, 197, 115]:

Definition 1.3.1 (Artificial Neural Network [165]). *An ANN is a sorted triple $(\mathcal{N}, V, w)$, where:*

- *$\mathcal{N}$ is the set of neurons;*
- *$V = \{(i, j) \mid i, j \in \mathbb{N}\}$ is a set whose elements represent the connections between neuron i and j;*
- *$w : V \longrightarrow \mathbb{R}$ is a function that defines the weights of the net, i.e. the value w_{ij} of the connection between neuron i and neuron j. The weights are then stored in a matrix $W = \{w_{ij}\}_{i,j}$, called weight matrix.*

As can be understood, since data are transferred through the connections existing between neurons, the values stored in the weight matrix W play a key role. Hence, based on what is discussed

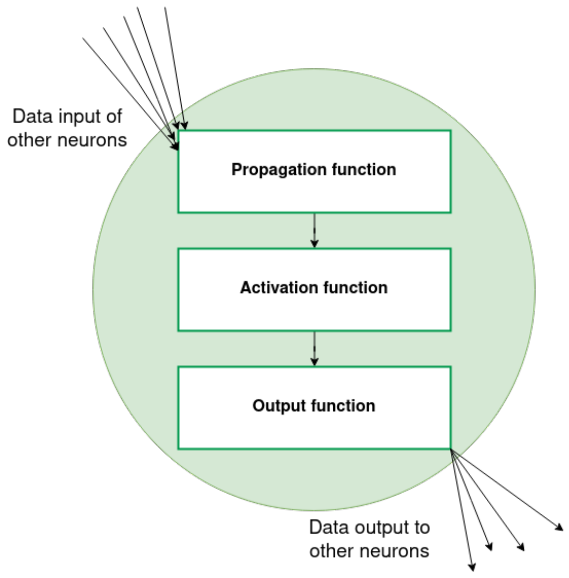


Figure 1.3: Data processing in a neuron [165].

in Section 1.2, the focus is on how every single neuron processes the information arriving in input, as summarized in Figure 1.3 and in Equation (1.1). The data transfer process inside neuron i can thus be described with the following steps:

1. **Data input**: Based on matrix W, it can be determined all the neurons $j_1, \dots, j_n$ that have a connection with neuron i and are transferring their outputs $\{\hat{y}_k\}_{k=j_1}^{j_n}$ to i. The inputs $x_1, \dots, x_n$ to i coincides thus with the outputs coming from the connected neurons.

2. **Propagation function**: A *propagation function* f_{prop} is then applied to the determined inputs, considering the corresponding connecting weights, in order to obtain the propagated signal $\mathbf{h}$. A common choice is represented by the weighted summation, as in Equation (1.1):

$$h_i = f_{\text{prop}}(\hat{y}_{j_1}, \dots, \hat{y}_{j_n}, w_{j_1,i}, \dots, w_{0,i}, w_{j_n,i}) = \sum_{j\in J} w_{ji}\hat{y}_j + w_{0,i}\hat{y}_0 = \sum_{j\in J} w_{ji}\hat{y}_j - b_i, \tag{1.7}$$

   where $J = \{j_1, \dots, j_n\}$ is the set of neurons connected to i and $w_{0.i}$ the bias. It is important to highlight that also the bias represents an input for i, since it can be thought of as a neuron with an output equal to $\hat{y}_0 = -1$ and corresponding weight $w_{0,i} = b_i$. In particular, its role resembles that of the threshold value indicating the activation of a biological neuron.

3. **Activation function**: Once we have the transformed input value h_i, we produce the neuron output by applying an activation function σ, introducing nonlinearity in the network. This function depends on h_i, as defined in the following equation:

$$a_i = \sigma(h_i) = \sigma\left(\sum_{j\in J} w_{ji}\hat{y}_j - b_i\right). \tag{1.8}$$

4. **Output function**: the output function calculates the values which are transferred to the other neurons connected to i. It is usually defined globally for all neurons to be equal to the

identity function. Hence, the output of neuron i coincide with the value computed in step 3, $\hat{y}_i = a_i$.

Therefore, given an ANN, as described in Definition 1.3.1, input data will be processed by the $\mathcal{N}$ neurons following the previous steps and producing at the end the output vector of the ANN. Using these notions, we can then construct different neural networks, that differ from each other in how these interactions between neurons are made, as will be described in Section 1.4.
Before moving to this brief review of the most common Neural Network architectures, we must introduce a crucial notion in this context: *Learning*.

1.3.1 The Learning Paradigm

ANNs have been designed to mimic the behavior of a biological neural network [208, 257], that has the ability to learn, store, and recall information. Hence, one of the main topics connected with ANNs is that of *Learning* [281]: the process of converting experience into expertise or knowledge. The experience is represented by data, whereas the output is the expertise, leading to the ability to perform a particular task T. Given thus T and some data, we want to develop an ANN able to acquire knowledge from these samples about T, to be used to perform T also with new samples.
To implement and then improve the learning process, two steps are needed: the training phase and the testing phase. The *training phase* or *learning phase* coincides with the process in which the network is learning the task T, whereas the *testing phase* is performed after the ANN has been trained. During this process, the network is tested in order to validate its accuracy and performance on new input data. In particular, given a set of data $\mathcal{D}$, we do not use all the data for training and then for testing the model, but usually we have that $\mathcal{D}$ is split into two parts, one for each of the aforementioned phases: a **training dataset** $\mathcal{D}_{\text{train}}$, responsible for the learning of the desired task from the ANN and thus containing most of the data; and a **testing dataset** $\mathcal{D}_{\text{test}}$, corresponding to the remaining part of $\mathcal{D}$, used to test the accuracy of the ANN.
There exists then three main learning paradigms used to teach an ANN how to behave as a biological neural network, which differ from each other in the nature of the interaction between learner and environment [165, 116, 197, 281]: Supervised Learning, Unsupervised Learning, and Reinforcement Learning. **Supervised Learning** [9, 218, 281] is a technique where the input and expected output (target) are provided in the training dataset $\mathcal{D}_{\text{train}}$, and we use an ANN to model the relationship between input and target. Hence, it is describing a scenario in which the experience, namely the training samples, contains important information to be applied in the testing database $\mathcal{D}_{\text{test}}$, where this knowledge is missing. In this case, we can think of the environment as the teacher that is supervising the learner by providing training data, i.e. the pair input-output. In **Unsupervised Learning** [220, 129, 281], this distinction between training and test data does not exist, since labeled data or target variables are not provided to the net. Hence, this is a method that tries to discover patterns and trends within a dataset in order to use them to make predictions about new data. There is also an intermediate learning setting, where the learner is required to predict even more information from the test examples. **Reinforcement Learning** [150, 230, 218] differs indeed from the others because of the presence of rewards and punishments based on the ANN performances. There is also in this case a specific expected outcome towards which we are pushing the network to move, but the ANN will learn the optimal path to achieve it using rewards and punishments signals. Even if there exist several learning paradigms, in our discussion we are going to concentrate on supervised learning, so from now on all the details provided concern this type of learning.
Given now a dataset $\mathcal{D}$, that is composed of input-output pairs $(\mathbf{x}, \mathbf{y})$, we are interesting in approximating the *target function* f, describing the desired relation existing between the target variable $\mathbf{y}$ and the input variable $\mathbf{x}$ — that can be one- or multi-dimensional vectors, random variables or tensors —, namely $\mathbf{y} = f(\mathbf{x})$. In order to do this we can use an ANN, thanks to its property of behaving as

an *universal approximator* [58, 137, 181] for functions with any desired degree of accuracy. The process of obtaining an accurate approximation can thus be interpreted as the learning process of this desired function f.
More technically speaking, an ANN can be described as a function [62, 281], that modifies the input variables using a certain rule to provide an output $\hat{\mathbf{y}}$:

$$\hat{\mathbf{y}} = \mathcal{ANN}(\mathbf{x}) = \mathcal{ANN}_{W,\mathbf{b}}(\mathbf{x}), \tag{1.9}$$

where the subscripts W and $\mathbf{b}$ are indicating the network parameters[4]. As pointed out before, the ANN tries to approximate the relation existing between $\mathbf{x}$ and $\mathbf{y}$, namely the function f, by learning from the data provided in $\mathcal{D}$. In particular, to reach this goal, we need to perform the training phase, where the ANN will tune the parameters (W,$\mathbf{b}$) until the distance between the expected output $\mathbf{y}$ and the predicted output $\hat{\mathbf{y}}$ is small enough. This proximity can be measured in different ways, depending also on the loss function used to model this distance $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = dist(\mathbf{y}, \hat{\mathbf{y}})$. Hence, the net is tuning and finding the optimal parameters by minimizing the loss function. A technical description of this process is carried out in Section 1.5.2, where we will concentrate on learning for FNNs. After this learning step, where we have employed the data in $\mathcal{D}_{\text{train}}$ to train the ANN, we need to evaluate the accuracy of the network using the remaining samples composing $\mathcal{D}_{\text{test}}$. During the testing phase, we are thus computing the distance between the expected output $\mathbf{y}$ and the predicted one $\hat{\mathbf{y}}$ using some techniques and measures, that are described in Section 1.5.9 for the case of a FNN.

1.4 Neural Network Topologies

We provide in this section an overview of the usual *neural network topologies* [165, 180, 42], i.e. the connections between neurons of different layers that define different types of ANN architectures. In the following description we will refer to the structure of neural networks as made up of vertically stacked components, called *layers*, organized as follow:

- **Input layer**: first layer of the network, composed of n_{in} neurons that accept the data and pass it to the rest of the network.
- **Hidden layers**: second type of layer found in an ANN. Hidden layers are either one or more in number for a neural network, depending on the problem at hand. There exists approximation results providing upper bounds on the sufficient size of a network, but also establishing lower bounds on the necessary size required for certain approximation tasks [17, 213, 327, 280, 241]. Hence, they play a central role in an ANN, being responsible for the accurate performance and complexity of the net.
- **Output layer**: last layer of the network, composed of n_{out} neurons that holds the final output of the problem.

A layer of neurons is thus a processing step into a neural network and, depending on the weights and activation function used, different types of layers can be constructed, e.g. *fully-connected layer* if all the neurons in a layer are connected with those in the next layer; *convolution layer* and *pooling layer*, that will be described in details in Section 2.2.
Formally, if we consider an ANN made up of L hidden layers[5], one input layer and one output layer,

[4] From now on we will omit the subscripts, except in cases where it is important to specify them.
[5] We have decided to follow the convention for which L indicates the number of hidden layers without considering also the output layer. Thus, in total, an ANN, with L hidden layers, has $L + 1$ layers.

we can provide a description of an ANN as composition of functions. As discussed in Section 1.3.1, an ANN can be see as a function $\mathcal{ANN}$, that, given an input $\mathbf{x}$, it provides an output following a certain rule [62]. Hence, taking up the notation of Definition 1.3.1, if the set of neurons $\mathcal{N}$ is organized in different layers based on the existing connections between neurons, enclosed in V, we can denote with $\mathbf{x}^{(\ell)}$ the output for layer ℓ, that is made up of n_ℓ neurons[6], where $0 \leq \ell \leq L+1$. Now, based on Section 1.3, layer ℓ, for $1 \leq \ell \leq L+1$, can be described as a function $f_\ell\colon \mathbb{R}^{n_{\ell-1}} \longrightarrow \mathbb{R}^{n_\ell}$ that maps an input that lies in $\mathbb{R}^{n_{\ell-1}}$, i.e. the output of the previous layer, into a tensor[7] in $\mathbb{R}^{n_\ell}$, and in particular as a composition of functions, the propagation function $f^{(\ell)}_{\text{prop}} : \mathbb{R}^{n_{\ell-1}} \longrightarrow \mathbb{R}^{n_\ell}$, the activation function $\sigma^{(\ell)} : \mathbb{R}^{n_\ell} \longrightarrow \mathbb{R}^{n_\ell}$and the output function $f^{(\ell)}_{\text{out}} : \mathbb{R}^{n_\ell} \longrightarrow \mathbb{R}^{n_\ell}$:

$$f_\ell = f^{(\ell)}_{\text{out}} \circ \sigma^{(\ell)} \circ f^{(\ell)}_{\text{prop}}, \tag{1.10}$$

where there could be different choices for the different functions $f^{(\ell)}_{\text{prop}}$, $\sigma^{(\ell)}$, $f^{(\ell)}_{\text{out}}$, yielding different types of layers and thus different neural networks.
We can thus provide a formal definition of an ANN as the composition of functions $f_1, \ldots, f_{L+1}$, which characterize each layer of the net:

$$\mathcal{ANN}(\mathbf{x}^0) = f_{L+1} \circ f_L \circ \cdots \circ f_1(\mathbf{x}^{(0)}), \qquad \mathbf{x}^{(0)} \in \mathbb{R}^{n_0}, \tag{1.11}$$

where the type of layer defined by each function f_ℓ will determine the neural network topology we are constructing, as it will be understood in the following discussion.

1.4.1 Feedforward Neural Network

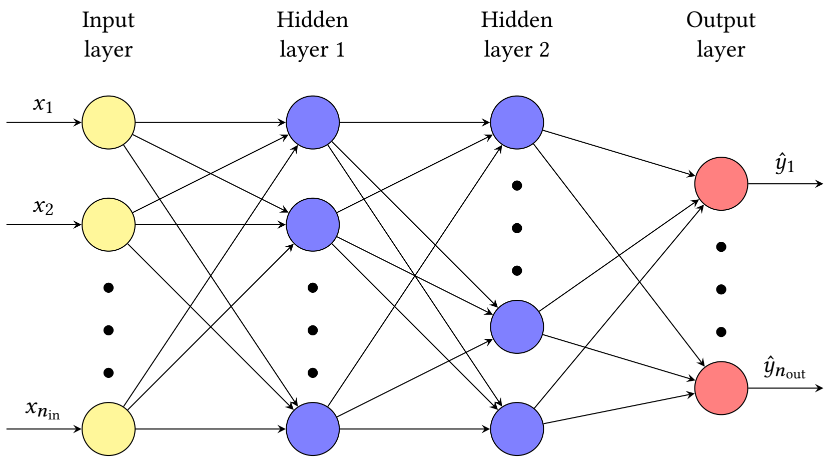


Figure 1.4: Schematic structure of a Feedforward Neural Network.

The simplest form of neural network model is represented by *Feedforward Neural Networks (FNNs)*, also called *multilayer perceptron*. This type of neural network is usually employed for

[6]It is useful to point out that in the following sections and chapter we will refer to the number of input neurons n_0 also with $n$in, and to the number of output neurons n_{L+1} with $n$out or n_{class} in Chapter 3.

[7]As discussed before, the output of a layer can be a vector, a scalar, a general tensor. The notion that includes all these possibilities is that of tensor since scalars and vectors can be seen as particular cases of tensors. For this reason, from now on, we are using the general term tensor to indicate the output of layer ℓ. In particular, if we have a generic tensor we can think of $\mathbb{R}^{n_\ell}$ as the Cartesian product of spaces, e.g. $\mathbb{R}^{n_\ell} = \mathbb{R}^{d_W} \times \mathbb{R}^{d_H} \times \mathbb{R}^{d_C}$.

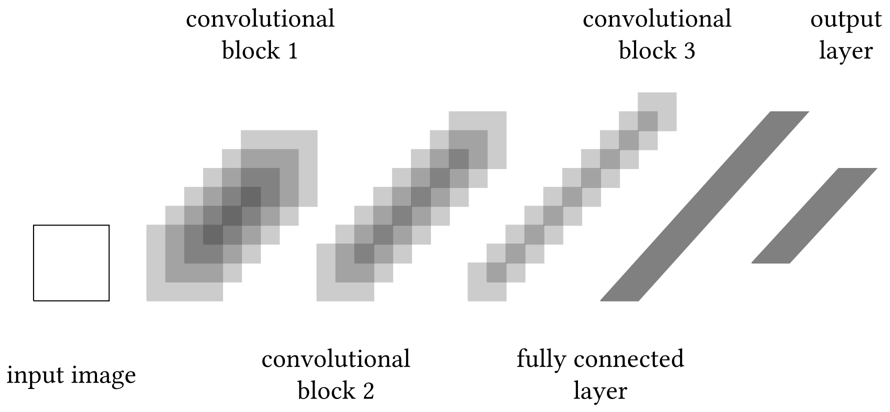


Figure 1.5: Schematic structure of a Convolutional Neural Network.

function regression [85, 215, 257, 291], computer vision [167, 282], data compression [278, 12], pattern recognition [14, 250, 25], financial prediction [290, 207], time series forecasting [303, 162], speech and handwritten characters recognition [104, 48]. Its architecture, depicted in Figure 1.4, is characterized by forward connections, namely input data travel in one direction only from left to right, passing through artificial neural nodes of the several layers and exiting through the output nodes. An exhaustive discussion about FNN will be carried out in Section 1.5 since this represents an architecture we have extensively used in the following chapters.

1.4.2 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are very popular deep[8] neural networks used to solve complex problems such as image recognition, speech recognition, or computer vision [6, 172, 99]. It is a particular type of FNN, since also in this case the net is characterized only by forward connections between its layers. Figure 1.5 presents the common structure of a CNN, which can be seen as a combination of two basic building blocks: the convolutional blocks and the fully connected feedforward layers. A more detailed description of the architecture and the way of using CNNs is provided in Chapter 2 since they represent one of the central topics of this thesis.

1.4.3 Recurrent Neural Network

Recurrent Neural Network (RNN) is a type of ANN commonly used for ordinal or temporal problems, such as language translation, natural language processing (NLP), and speech recognition. As depicted in Figure 1.6, the net is composed of a particular type of hidden layers, called *recurrent neural network layers*, which differ from the usual ones in the dependence of their outputs on the previous elements within the sequence. Therefore, to take into account these existing time relationships, these algorithms are characterized by a *memory*, where information from previous states $\hat{y}^{(t-1)}$ provides an outcome and an input to the next state $\hat{y}^{(t)}$, with t being the time step we are considering [77, 209].

[8]The term ‘deep’ is used to identify the width of the net, i.e. the high number of layers that compose it.

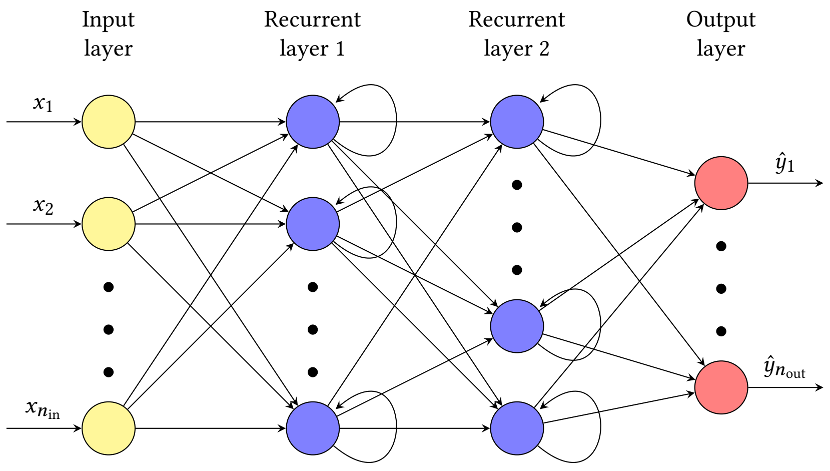


Figure 1.6: Schematic structure of a Recurrent Neural Network.

1.4.4 Autoencoder

Another type of ANN are Autoencoders (AEs) [99, 131, 275, 179], a model widely used for classification, clustering, and feature compression. As depicted in Figure 1.7, they mainly consist of three components: an *encoder* with the corresponding encoding method, a *code* and a decoding method connected with a *decoder*. It is important to highlight that both the encoder and the decoder are fully-connected FNNs and thus they are each other's mirror image. More technically speaking, an AE functions in the following way: first the input $\mathbf{x}$ passes through the encoder to produce the lower dimensional code $\mathbf{z}$, then the decoder maps the code to a reconstruction of the input $\tilde{\mathbf{x}}$. Hence, the AE performs a dimensionality reduction on the input, which is then reconstructed from the reduced version. In order to obtain an output almost identical with the input, AEs are trained the same way as ANNs minimizing, in this case, the reconstruction loss $\mathcal{L}(\tilde{x}, \mathbf{x})$, that quantifies the quality of the reconstruction $\tilde{x}$.

1.5 Feedforward Neural Networks

Feedforward Neural Networks (FNNs) or **multilayer perceptrons** (MLPs) represent a popular type of ANNs widely used in the context of deep learning [99, 85, 42]. As previously introduced in Section 1.4.1 and depicted in Figure 1.4, they mainly consist of an input layer, an output layer, and a certain number of hidden layers in the middle[9]. The base elements that compose each of these layers are the neurons and the associated weights, describing the strength of the existing connections, as discussed in Section 1.2.
As stated in Section 1.3.1, in order to employ a FNN as model to solve a task, we need to let the network learn the problem and the knowledge we have about it, contained in the training dataset $\mathcal{D}_{\text{train}}$, and then test its performances exploiting the testing samples in $\mathcal{D}_{\text{test}}$. We shall now provide more details about how the training and testing phases are carried out for a FNN. There are indeed two main notions connected to these topics: **forward propagation** and **backward**

[9]The number of hidden layers is not determined a priori looking only at the data provided. It depends on the problem under consideration and is usually determined empirically by testing different architectures [310].

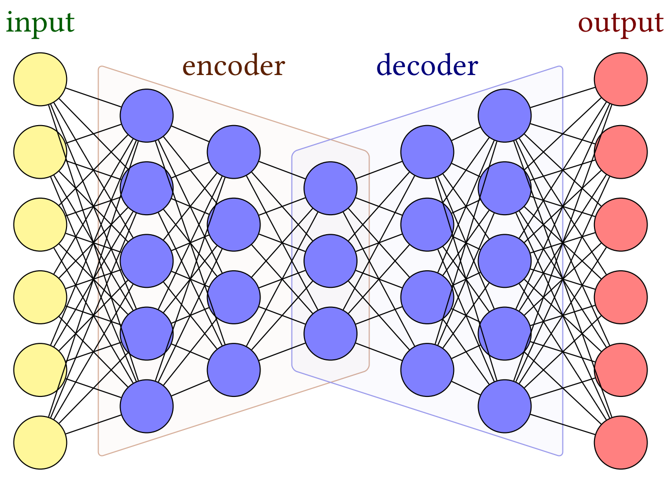


Figure 1.7: Schematic structure of an Autoencoder.

propagation. The forward step indicates the simple evaluation of the output, given some inputs, and is thus strictly connected with the testing phase. On the other hand, backward propagation is the key ingredient of the training process, representing thus the learning method for a FNN.

1.5.1 Forward Propagation

With the term **forward propagation** [42] we are referring to the process of going forward through the network, i.e. finding all the neuron outputs. Therefore, to obtain the final output of the network $\hat{\mathbf{y}} \in \mathbb{R}^{n_{\text{out}}}$, we have to start from Equation (1.1) and the notions presented in Section 1.3. Therefore, let $\mathbf{x}^{(0)} \in \mathbb{R}^{n_{\text{in}}}$ be the input vector and L the total number of hidden layers of the FNN. Since in a FNN neurons are organized in sequential layers without feedback connections in which outputs of the model are fed back into itself, the output of each layer depends only on the output of the previous layer. Coupling this feature with Equation (1.1), we obtain the following expression for the output of the j-th x in layer ℓ, $x_j^{(\ell)}$, [42]:

$$x_j^{(\ell)} = \sigma(h_j^{(\ell)}) = \sigma\left(\sum_{i=0}^{n_{\ell-1}} w_{ji}^{(\ell)} x_i^{(\ell-1)}\right) = \sigma\left(\sum_{i=1}^{n_{\ell-1}} w_{ji}^{(\ell)} x_i^{(\ell-1)} - b_j^{(\ell)}\right), \qquad \text{for } j = 1, \dots, n_\ell, \tag{1.12}$$

where $x_i^{(\ell-1)}$ are the input signals coming from the previous layer $\ell - 1$; n_ℓ, $\ell = 0, 1, \dots, L, L+1$, represents the number of neurons in layer ℓ; $h_j^{(\ell)}$ is the transformed input value or total input of neuron j as introduced in Equation (1.7); $W^{(\ell)} = (w_{ji}^{(\ell)})_{ji},\ j = 1, \dots, n_\ell,\ i = 1, \dots, n_{\ell-1}$ represents the weight matrix of the net, and $x_0^{(\ell)} = -1$ the fake input linked to the bias $b_i^{(\ell)}$ related to layer ℓ. Note that we are using the upper script ℓ to denote the layer number, where $\ell = 0$ denotes the input layer and $\ell = L + 1$ is the output layer. Furthermore, in any weights subscript, we are using the convention that the first number matches the index of the neuron in the next layer and the second number matches the index of the neuron in the previous layer.
Each component of the output vector $x_j^{(\ell)}$ is hence obtained through the application of an *activation function* σ to the weighted sum of all the inputs arriving at it from the previous layer. As discussed in section 1.2 and section 1.3, there are plenty of choices for σ [99, 165], depending on the problem in exam. Usually, the same activation function is used in all hidden layers, but there exists also the

possibility of choosing different functions.
Equation (1.12) can also be written in a more compact form, using matrix notation [42]:

$$\mathbf{x}^{(\ell)} = \sigma\left(W^{(\ell)T}\mathbf{x}^{(\ell-1)} - \mathbf{b}^{(\ell)}\right), \tag{1.13}$$

where we are using the convention that the activation function applied on a vector acts on each of the components of the vector and we have that $\mathbf{x}^{(\ell)}$, $W^{(\ell)}$, and $\mathbf{b}^{(\ell)}$ corresponds to:

$$\mathbf{x}^{(\ell)} = (x_1^{(\ell)}, \dots, x_{n_\ell}^{(\ell)})^T, \qquad W^{(\ell)} = (w_{ji}^{(\ell)})_{ji}, \qquad \mathbf{b}^{(\ell)} = (b_1^{(\ell)}, \dots, b_{n_\ell}^{(\ell)})^T,$$

with $1 \le i \le n_{\ell-1}$ and $1 \le j \le n_\ell$.
Now, since we are considering a FNN made of L hidden layers, the final output can be seen as a weighted sum of the inputs arriving at the neurons in the output layer followed by the activation function, where each of these inputs can be rewritten in the same way generating a recursive formula:

$$\begin{aligned}\hat{y}_j &= \sigma\left(\sum_{i=0}^{n_L} w_{ji}^{(L+1)} x_i^{(L)}\right) = \sigma\left(\sum_{i=0}^{n_L} w_{ji}^{(L+1)}\left(\sigma\left(\sum_{q=0}^{n_{L-1}} w_{iq}^{(L)} x_q^{(L-1)}\right)\right)\right) = \dots = \\ &= \sigma\left(\sum_{i=0}^{n_L} w_{ji}^{(L+1)}\left(\sigma\left(\sum_{q=0}^{n_{L-1}} w_{iq}^{(L)}\left(\sigma\left(\dots\left(\sigma\left(\sum_{k=0}^{n_{in}} w_{sk}^{(1)} x_k\right)\right)\right)\right)\right)\right)\right), \qquad j = 1, \dots, n_{\text{out}},\end{aligned} \tag{1.14}$$

It is important to point out that, when constructing a FNN, usually the weights and the other parameters of the network are randomly initialized following some criteria, as described in detail in Section 1.6. For this reason, at the end of the forward pass, we could not expect to obtain the correct predictions. This is thus preparing the ground for the second step of the method: the *backward propagation*.

1.5.2 Backward Propagation

Let $\mathcal{D}_{\text{train}} = \{\mathbf{x}^i, \mathbf{y}^i\}_{i=1}^{n_{\text{train}}}$ be the training dataset and $\hat{\mathbf{y}} = \mathcal{FNN}(\mathbf{x})$ be the predicted output of a FNN. As introduced in section 1.3.1, in order to measure the distance between the expected output $\mathbf{y}$ and the predicted output $\hat{\mathbf{y}}$, a *cost* or *loss function* $\mathcal{L} = \mathcal{L}_{W,\mathbf{b}}(\mathbf{y}, \hat{\mathbf{y}}) = \mathit{dist}(\mathbf{y}, \hat{\mathbf{y}})$ needs to be introduced[10]. There exist several types of functions that can be used for this purpose, as the following:

- **ℓ^2-Error Function**: it measures the Euclidean distance between the predicted and expected outputs for each sample in the training dataset, i.e. is defined as:

$$\mathcal{L}_{\ell^2}(Y, \hat{Y}) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \mathcal{L}_{\ell^2}^i(\mathbf{y}^i, \hat{\mathbf{y}}^i) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \frac{1}{2}\|\mathbf{y}^i - \hat{\mathbf{y}}^i\|^2 = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \sum_{j=1}^{n_{\text{out}}} \frac{1}{2}(y_j^i - \hat{y}_j^i)^2, \tag{1.15}$$

  where $Y = [\mathbf{y}^1, \dots, \mathbf{y}^{n_{\text{train}}}]$ and $\hat{Y} = [\hat{\mathbf{y}}^1, \dots, \hat{\mathbf{y}}^{n_{\text{train}}}]$.

- **Mean-Squared Error (MSE)**: commonly adopted in regression problems [281, 218], it is defined by:

$$\mathcal{L}_{\text{MSE}}(Y, \hat{Y}) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \mathcal{L}_{\text{MSE}}^i(\mathbf{y}^i, \hat{\mathbf{y}}^i) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} (\mathbf{y}^i - \hat{\mathbf{y}}^i)^2. \tag{1.16}$$

[10]Also here we will omit, from now on, to specify the subscripts identifying the parameters of the FNN for simplicity in the notation.

- **Mean Absolute Error (MAE)**: it measures the mean absolute value of the element wise difference between $\mathbf{y}^i$ and $\hat{\mathbf{y}}^i$, for $i = 1, \dots, n_{\text{train}}$:

$$\mathcal{L}_{\text{MAE}}(Y, \hat{Y}) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \mathcal{L}^i_{\text{MAE}}(\mathbf{y}^i, \hat{\mathbf{y}}^i) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \mid \mathbf{y}^i - \hat{\mathbf{y}}^i \mid . \tag{1.17}$$

- **Cross Entropy (CE) Loss**: commonly used in the context of binary classification problems [99], but can be easily extended for multi-class classification [305], as described in Section 2.2.6. The loss is obtained by computing the following average:

$$\mathcal{L}_{\text{CE}}(Y, \hat{Y}) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \mathcal{L}^i_{\text{CE}}(\mathbf{y}^i, \hat{\mathbf{y}}^i) = -\frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \left(\mathbf{y}^i \log(\hat{\mathbf{y}}^i) + (1 - \mathbf{y}^i) \log(1 - \hat{\mathbf{y}}^i)\right) . \tag{1.18}$$

Also in this case, the correct loss depends on the problem in exam [6, 99] and should be chosen based on some empirical observations. Sometimes to avoid overfitting the training data and to deal with small values of the parameters, a penalization term may be included in the definition of the loss function [42, 331]:

$$\mathcal{L}_{\text{penalty}}(Y, \hat{Y}) = \mathcal{L}(Y, \hat{Y}) + \lambda J(W), \tag{1.19}$$

where λ is the regularization parameter and $J(W)$ is the non-negative regularization term. Commonly choices are represented by the L^1- or L^2-regularization, i.e. the L^1 or L^2 norms of W respectively.
In order to tune all FNN's parameters, we need to perform **Backward propagation** [253, 263, 331]. Going backward through the networks means computing the gradient of the cost function $\mathcal{L}$ to update the parameters set using the **gradient descent method** [43, 108], which will lead to an improved prediction of the expected output. It is important to point out that a fundamental assumption to apply backward propagation is having only forward connections in the network. This is the reason why we are describing these methods for FNNs and not for a generic ANN.
Once we have determined a loss function to model the proximity between expected and predicted outputs, the FNN will tune the parameters (W,$\mathbf{b}$) until this distance is small enough. Hence, we want to find a set of weights and biases which make the loss as small as possible, namely the following minimization problem needs to be solved:

$$\min_{(W,\mathbf{b})} \left\{ \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \mathcal{L}^i_{W,\mathbf{b}}(\mathbf{y}^i, \hat{\mathbf{y}}^i) \right\} . \tag{1.20}$$

The optimal parameters of the net are then in particular obtained using the aforementioned *Gradient Descent algorithm*, where the gradients are computed through the *Backpropagation algorithm*. There exists also other techniques that can be used to solve the minimization problem (1.20). Here we have decided to focus only on the *gradient descent* (see Section 1.5.4) and on three popular extension, the *stochastic gradient descent* (see Section 1.5.5), the *Momentum Method* (see Section 1.5.6) and *Adam* (see Section 1.5.7), since these represent the methods used in our tests. A list of other optimization algorithms can be found in [42, 99].

1.5.3 Backpropagation Algorithm

For the purposes of gradient descent, as it will be discussed in Section 1.5.4, we need to perform the computation of the gradient $\nabla\mathcal{L} = (\nabla_W \mathcal{L}, \nabla_{\mathbf{b}} \mathcal{L})$ [62].
Starting with the partial derivative of $\mathcal{L}$ with respect to the weights W, we can notice that the

weight $w_{ji}^{(\ell)}$ affects the loss only in the total input for neuron j of layer ℓ, $h_j^{(\ell)}$. Therefore, using the chain rule the following relation is provided [42]:

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial h_j^{(\ell)}} \frac{\partial h_j^{(\ell)}}{\partial w_{ji}^{(\ell)}} = \delta_j^{(\ell)} \frac{\partial h_j^{(\ell)}}{\partial w_{ji}^{(\ell)}}, \tag{1.21}$$

where we have introduced the delta notation $\delta_j^{(\ell)}$ to denote the sensitivity of the loss with respect to $h_j^{(\ell)}$. The second term in Equation (1.21) can then be computed explicitly using Equation (1.12):

$$\frac{\partial h_j^{(\ell)}}{\partial w_{ji}^{(\ell)}} = x_i^{(\ell-1)}. \tag{1.22}$$

Substituting this relation in Equation (1.21), this yields:

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(\ell)}} = \delta_j^{(\ell)} x_i^{(\ell-1)}. \tag{1.23}$$

Similarly, we can carry out the same analysis for the biases $\mathbf{b}$. Also in this case the loss $\mathcal{L}$ is affected by the bias $b_j^{(\ell)}$ only through the total input for neuron j of layer ℓ, $h_j^{(\ell)}$. Exploiting the chain rule in the computation of the partial derivative of $\mathcal{L}$ with respect to the bias $b_j^{(\ell)}$, we obtain [42]:

$$\frac{\partial \mathcal{L}}{\partial b_j^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial h_j^{(\ell)}} \frac{\partial h_j^{(\ell)}}{\partial b_j^{(\ell)}} = \delta_j^{(\ell)} \frac{\partial h_j^{(\ell)}}{\partial b_j^{(\ell)}}, \tag{1.24}$$

where we have exploited the delta notation introduced before. Differentiating $h_j^{(\ell)}$, defined in Equation (1.12), with respect to the bias $b_j^{(\ell)}$, we get:

$$\frac{\partial h_j^{(\ell)}}{\partial b_j^{(\ell)}} = -1, \tag{1.25}$$

that can then be substituted in Equation (1.24), providing:

$$\frac{\partial \mathcal{L}}{\partial b_j^{(\ell)}} = -\delta_j^{(\ell)}. \tag{1.26}$$

Both Equation (1.23) and Equation (1.26) contain an unknown term $\delta_j^{(\ell)}$ that we are now going to compute using the backpropagation algorithm. The key step of this method coincides exactly with the backpropagation of deltas, i.e. their computation exploiting deltas coming from the next layer [42]. Therefore, we start with the computation of deltas in the last layer $L+1$, $\delta_j^{(L+1)}$, with $1 \leq j \leq n_{\text{out}}$, that are defined as:

$$\delta_j^{(L+1)} = \frac{\partial \mathcal{L}}{\partial h_j^{(L+1)}}. \tag{1.27}$$

From this relation is clear that $\delta_j^{(L+1)}$ depends on the form of the loss function, as each $\delta_j^{(\ell)}$, for $1 \leq \ell \leq L$. For instance, we can consider the L^2-error function defined in Equation (1.15), focusing on only one generic sample $(\mathbf{x}, \mathbf{y})$ with the associated prediction $\hat{\mathbf{y}}$:

$$\mathcal{L}_{L^2}(\mathrm{y}, \hat{\mathrm{y}}) = \frac{1}{2}\|\mathrm{y} - \hat{\mathrm{y}}\|^2 = \frac{1}{2}\sum_{i=1}^{n_{\text{out}}}(y_i - \hat{y}_i)^2. \tag{1.28}$$

Based on Equation (1.12), we can rewrite each component of the predicted output as:

$$\hat{y}_i = \sigma(h_i^{(L+1)}). \tag{1.29}$$

Hence, if we explicit the computation of $\delta_j^{(L+1)}$ exploiting the L^2-loss introduced in Equation (1.28) and the chain rule, we gain:

$$\delta_j^{(L+1)} = \frac{\partial \mathcal{L}}{\partial h_j^{(L+1)}} = (y_j - \hat{y}_j)\sigma'(h_j^{(L+1)}). \tag{1.30}$$

This relation can be further expanded based on the form of the activation function σ chosen[11]. For example, if we now consider the logistic function with $c = 1$ defined in Equation (1.5):

$$\sigma(x) = \frac{1}{1 + e^{(-x)}} = \frac{e^x}{1 + e^{(-x)}},$$

we can compute its derivative:

$$\frac{d\sigma(x)}{dx} = \frac{e^{(x)}(1 + e^{(x)}) - e^{(x)}e^{(x)}}{(1 + e^{(-x)})^2} = \frac{e^{(x)}}{(1 + e^{(x)})^2} = \sigma(x)(1 - \sigma(x))$$

and substitute in Equation (1.30) to get:

$$\delta_j^{(L+1)} = (y_j - \hat{y}_j)\sigma(h_i^{(L+1)})(1 - \sigma(h_i^{(L+1)})).$$

Once we have understood how to compute deltas for the output layer, we need to find an expression for the deltas of layer $\ell - 1$ in terms of the deltas of layer ℓ. Hence, exploiting the chain rule in the relation defining $\delta_i^{(\ell-1)}$, we have:

$$\delta_i^{(\ell-1)} = \frac{\partial \mathcal{L}}{\partial h_i^{(\ell-1)}} = \sum_{k=1}^{n_\ell} \frac{\partial \mathcal{L}}{\partial h_k^{(\ell)}} \frac{\partial h_k^{(\ell)}}{\partial h_i^{(\ell-1)}} = \sum_{k=1}^{n_\ell} \delta_k^{(\ell)} \frac{\partial h_k^{(\ell)}}{\partial h_i^{(\ell-1)}}, \tag{1.31}$$

where we have used the fact that the loss function is affected by $h_k^{(\ell-1)}$ trough all the $h_k^{(\ell)}$ from layer ℓ: it can be noticed that $h_k^{(\ell-1)}$ in layer $\ell - 1$ represents an input for all neurons in layer ℓ, i.e. for all $h_k^{(\ell)}$, explaining also the need of a summation over the neurons in layer ℓ.

The last thing we shall compute is the term $\frac{\partial h_k^{(\ell)}}{\partial h_i^{(\ell-1)}}$, that measures the sensitivity of the total input $h_k^{(\ell)}$ to neuron k in layer ℓ with respect to the total input $h_i^{(\ell-1)}$ to neuron i in layer $\ell - 1$. This can be done by differentiating Equation (1.12):

$$\begin{aligned} \frac{\partial h_k^{(\ell)}}{\partial h_i^{(\ell-1)}} &= \frac{\partial}{\partial h_i^{(\ell-1)}} \left(\sum_{m=1}^{n_{\ell-1}} w_{km}^{(\ell)} x_m^{(\ell-1)} - b_k^{(\ell)} \right) \\ &= \frac{\partial}{\partial h_k^{(\ell-1)}} \left(\sum_{m=1}^{n_{\ell-1}} w_{km}^{(\ell)} \sigma(h_m^{(\ell-1)}) - b_k^{(\ell)} \right) \\ &= w_{ki}^{(\ell)} \sigma'(h_i^{(\ell-1)}). \end{aligned} \tag{1.32}$$

[11] See Section 1.2 for a list of possible choices for σ.

Using this relation in Equation (1.31), we gain the *backpropagation formula* for the deltas [42]:

$$\delta_i^{(\ell-1)} = \sigma'(h_i^{(\ell-1)}) \sum_{k=1}^{n_\ell} \delta_k^{(\ell)} w_{ki}^{(\ell)}. \tag{1.33}$$

In this way, we have obtained a formula for the deltas in layer $\ell - 1$ in terms of the deltas in layer ℓ using a weighted sum.
Also in this case, if we consider for instance the logistic function with $c = 1$ defined in Equation (1.5), the previous Equation (1.33) becomes:

$$\delta_i^{(\ell-1)} = \sigma(h_i^{(\ell-1)})(1 - \sigma(h_i^{(\ell-1)})) \sum_{k=1}^{n_\ell} \delta_k^{(\ell)} w_{ki}^{(\ell)}.$$

1.5.4 Gradient Descent Algorithm

The **gradient descent method** [43, 108, 42, 99, 165, 62] is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The basic idea is to take, from any starting point, repeated steps in the opposite direction of the gradient of the function, with the size of the steps being proportional to the norm of the gradient. Hence, we are searching the minimum of a function in the direction in which the value of the function is decreasing mostly, called the *steepest descent*, that is identified by the negative gradient.
More technically speaking, we aim at finding the unitary direction $\mathbf{u} \in \mathbb{R}^n$ in which a function f, represented in our case by the loss function $\mathcal{L}$, decreases more within a given small step size μ. To measure the change of the value of a function between an initial point $\mathbf{x}_0 \in \mathbb{R}^n$ and the value after a step μ in the direction $\mathbf{v}$, we need to consider the following linear approximation:

$$f(\mathbf{x}_0 + \mu\mathbf{u}) - f(\mathbf{x}_0) = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(\mathbf{x}_0)\mu u_i + o(\mu^2) = \mu\langle \nabla f(\mathbf{x}_0), \mathbf{u}\rangle + o(\mu^2). \tag{1.34}$$

Since we attempt to find the direction $\mathbf{u}$ in which the function has the change with the largest negative value, we can neglect the effect of the quadratic term $o(\mu^2)$, considering that μ is small enough, and then use the Cauchy inequality for the scalar product:

$$-\|\nabla f(\mathbf{x}_0)\|\|\mathbf{u}\| \le \langle \nabla f(\mathbf{x}_0), \mathbf{u}\rangle \le \|\nabla f(\mathbf{x}_0)\|\|\mathbf{u}\|. \tag{1.35}$$

In the previous relation we can note that the equality is satisfied when $\mathbf{u} = \lambda \nabla f(\mathbf{x}_0)$, $\lambda \in \mathbb{R}$. Hence, since for hypothesis $\|\mathbf{v}\| = 1$, we have that the minimum occurs for[12]

$$\mathbf{u} = -\frac{\nabla f(\mathbf{x}_0)}{\|\nabla f(\mathbf{x}_0)\|}. \tag{1.36}$$

Therefore, based on these considerations, Equation (1.34) can be approximated as:

$$f(\mathbf{x}_0 + \mu\mathbf{u}) - f(\mathbf{x}_0) = \mu\langle \nabla f(\mathbf{x}_0), \mathbf{u}\rangle = -\mu\|\nabla f(\mathbf{x}_0)\|, \tag{1.37}$$

representing the largest change in the function. The step size μ is a positive scalar, called *learning rate*. There exist several ways to determine μ: it can be set to a small fixed constant or determined using a line search approach that seeks the value of μ resulting in the smallest objective function value.
The gradient descent algorithm can thus be summarized with the following steps [42], which attempt to construct the sequence $(\mathbf{x}_t)_t$, with $\mathbf{x}_t$ being the result of the application of t step of this procedure to $\mathbf{x}_0$, i.e. $\mathbf{x}_t = \mathbf{x}_0(t) \in \mathbb{R}^n$:

[12] It is more clear if we think that $\langle \nabla f(\mathbf{x}_0), \mathbf{u}\rangle = \|\nabla f(\mathbf{x}_0)\|\|\mathbf{u}\|\cos(\theta)$. The minimum is reached when $\theta = \pi$, i.e. when $\mathbf{u}$ points in the opposite direction of the gradient.

(i) Choose an initial point $\mathbf{x}_0 = \mathbf{x}_0(0)$ in the basin of attraction of the global minimum $\mathbf{x}^*$.

(ii) Construct the sequence $(\mathbf{x}_t)_t$ as:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \mu \frac{\nabla f(\mathbf{x}_t)}{\|\nabla f(\mathbf{x}_t)\|}, \tag{1.38}$$

that is guaranteeing a negative change in our function f.

As pointed out in [42], this procedure has a drawback: the sequence $(\mathbf{x}_t)_t$ is not converging since $\|\mathbf{x}_{t+1} - \mathbf{x}_t\| = \mu > 0$. It shall be assumed that the learning rate μ is not fixed, but adjustable, in the sense that it becomes smaller as the gradient is smaller, i.e. when the function changes slower. We are thus assuming that exists a constant $\eta > 0$ such that the learning rate at iteration t is proportional to the gradient:

$$\mu_t = \eta \|\nabla f(\mathbf{x}_t)\|. \tag{1.39}$$

Using this new expression for the learning rate, the iteration (1.38) becomes:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \nabla f(\mathbf{x}_t). \tag{1.40}$$

We can now provide a necessary and sufficient condition that guarantees the convergence of this sequence:

Proposition 1.5.1 ([42]). *The sequence $(\mathbf{x}_t)_t$ defined by (1.40) is convergent if and only if the sequence of gradients $(\nabla f(\mathbf{x}_t))_t$ converges to zero.*

Proof. **⇒:** The sequence $(\mathbf{x}_t)_t$ is convergent, hence, using Equation (1.40), we have:

$$0 = \lim_{t\to\infty} \|\mathbf{x}_{t+1} - \mathbf{x}_t\| = \eta \lim_{t\to\infty} \|\nabla f(\mathbf{x}_t)\|,$$

i.e. the sequence of gradients converges to zero.

⇐: In order to prove that the sequence $(\mathbf{x}_t)_t$ is convergent, we need to show that it is a Cauchy sequence. So, let $k \geq 1$, applying the triangle inequality we obtain:

$$\|\mathbf{x}_{t+k} - \mathbf{x}_t\| \leq \|\mathbf{x}_{t+k} - \mathbf{x}_{t+k-1}\| + \cdots + \|\mathbf{x}_{t+1} - \mathbf{x}_t\| = \eta \sum_{i=0}^{k} \|\nabla f(\mathbf{x}_{t+i})\|.$$

Now, if k is fixed, taking the limit to infinity yields:

$$\lim_{t\to\infty} \|\mathbf{x}_{t+k} - \mathbf{x}_t\| \leq \eta \sum_{i=0}^{k} \lim_{t\to\infty} \|\nabla f(\mathbf{x}_{t+i})\| = 0.$$

□

After we have understood the basic notions of gradient descent algorithm, we can apply it to our case [42], i.e. the minimization of the loss function, as described in Equation (1.20). We are thus interested in finding the optimal set of weights W^* and biases $\mathbf{b}^*$ that solves our problem.
Let $W^{(\ell)}(0)$ and $\mathbf{b}^{(\ell)}(0)$ represent the initial system of weights and biases, initialized randomly followed the procedure described in Section 1.6. After the training phase of the FNN with the gradient descent algorithm, the optimum values of weights and biases are determined by using Equation (1.40)

coupled with Equations (1.23) and (1.26). The approximation sequences are thus defined recursively by the following equations:

$$\begin{aligned} w_{ji}^{(\ell)}(t+1) &= w_{ji}^{(\ell)}(t) - \eta\delta_j^{(\ell)}(t)x_i^{(\ell-1)}(t), \\ b_j^{(\ell)}(t+1) &= b_j^{(\ell)}(t) + \eta\delta_j^{(\ell)}(t), \end{aligned} \tag{1.41}$$

where the outputs $x_i^{(\ell-1)}(t)$ and deltas $\delta_j^{(\ell)}(t)$ depend on the weights $w_{ji}^{(\ell)}(t)$ and biases $b_j^{(\ell)}(t)$ at time step t.

1.5.5 Stochastic Gradient Descent Algorithm

One thing that characterizes machine learning is the use of large datasets for training neural networks, leading to computationally expensive training processes [30, 99, 42]. In particular, as we have seen in Section 1.5.2, the loss functions can often be decomposed as a sum over training samples of some per-sample losses:

$$\mathcal{L}_{W,\mathbf{b}}(Y,\hat{Y}) = \frac{1}{n_{\text{train}}}\sum_{i=1}^{n_{\text{train}}} \mathcal{L}^i_{W,\mathbf{b}}(\mathbf{y}^i,\hat{\mathbf{y}}^i). \tag{1.42}$$

In order to solve the minimization problem (1.20), we need to compute the gradient of $\mathcal{L}$, i.e. of $\mathcal{L}^i$, for $i = 1,\dots,n_{\text{train}}$, with respect to the parameters of the net $\boldsymbol{\theta} = (W,\mathbf{b})$:

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta}}(Y,\hat{Y}) = \frac{1}{n_{\text{train}}}\sum_{i=1}^{n_{\text{train}}} \nabla_{\boldsymbol{\theta}}\mathcal{L}^i_{\boldsymbol{\theta}}(\mathbf{y}^i,\hat{\mathbf{y}}^i). \tag{1.43}$$

This operation has a computational cost of $O(n_{\text{train}})$, which becomes prohibitively long as the training set size grows. Hence, a solution to this problem is represented by *batch training*, i.e. sampling a random mini-batch and computing an estimation of the gradient from it. This process of updating the network parameters estimating the loss gradient with an average of gradients measured on randomly selected training samples composing a mini-batch is called **Stochastic Gradient Descent (SGD)** [30, 31, 99, 42].
Specifically, as described in Algorithm 1, on each step of the algorithm, a mini-batch of independent and identically distributed (i.i.d.) samples $\mathcal{D}_{batch} = \{\mathbf{x}^1,\dots,\mathbf{x}^m\}$, with m be the size of the mini-batch, is extracted uniformly from the whole training dataset $\mathcal{D}_{\text{train}}$. Usually, the size of the batch is fixed to a value relatively small with respect to n_{train}, e.g. ranging from 8 to 100. Equation (1.43) is thus replace in SGD with:

$$\tilde{\nabla}_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta}}(Y_{\text{batch}},\hat{Y}_{\text{batch}}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}}\mathcal{L}^i_{\boldsymbol{\theta}}(\mathbf{y}^i,\hat{\mathbf{y}}^i), \tag{1.44}$$

where Y_{batch} and $\hat{Y}_{\text{batch}}$ represents respectively the expected and predicted outputs related to the chosen samples that compose the mini-batch $\mathcal{D}_{\text{batch}}$.
The update of the parameter will follow the same relation described in Equation (1.40), using in this case the estimated gradient in the mini-batch:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta\tilde{\nabla}\mathcal{L}_{\boldsymbol{\theta}}(Y_{\text{batch}},\hat{Y}_{\text{batch}}), \tag{1.45}$$

where $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}(t+1) = (\boldsymbol{W}(t+1),\mathbf{b}(t+1))$. Also in this case, the learning rate η is not fixed but will decrease over time to balance the noise introduced in the random sampling of the mini-batch

Algorithm 1 Stochastic Gradient Descent Update [99]

Inputs:

- Loss function $\mathcal{L}_{\boldsymbol{\theta}}$;
- training dataset $\mathcal{D}_{\text{train}} = \{\mathbf{x}^i, \mathbf{y}^i\}_{i=1}^{n_{\text{train}}}$;
- batch size m;
- predicted outputs of the FNN $\hat{Y} = \{\hat{\mathbf{y}}^j\}_{j=1}^{n_{\text{train}}}$;
- initial value for the parameters $\boldsymbol{\theta}_0 = \boldsymbol{\theta}(0) = (W(0), \mathbf{b}(0))$;
- number of mini-batches $n_{\text{batch}} = \dfrac{n_{\text{train}}}{m}$;
- learning rates at iteration 0 and τ: η_0 and $\eta_{n_{\text{batch}}}$.

```
1: for k=1,...,$n_{\text{batch}}$ do
2:     Sample a mini-batch of $m$ i.i.d. samples $\mathcal{D}_{\text{batch}} = \{\mathbf{x}^1, \dots, \mathbf{x}^m\}$;
3:     Determine the corresponding expected outputs $\mathbf{y}^i$ for each sample in $\mathcal{D}_{\text{batch}}$;
4:     Compute estimation of the gradient: $\tilde{\nabla}_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta}}(Y_{\text{batch}}, \hat{Y}_{\text{batch}})$;
5:     Compute learning rate $\eta_t$ using (1.47);
6:     Update of the parameters: $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \tilde{\nabla}_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta}}(Y_{\text{batch}}, \hat{Y}_{\text{batch}})$.
7: end for
```

and let the SGD gradient estimator be close to zero when approaching the minimum [99]. In practice, the learning rate decays linearly, until a chosen iteration τ, as:

$$\eta_t = (1-\alpha)\eta_0 + \alpha\eta_{n_\tau}, \qquad \text{with } \alpha = \frac{t}{\tau}, \tag{1.46}$$

where τ usually coincides with the number of iterations required to make a few hundred passes through the training set. The initial learning rate η_0 is then chosen based on the learning curve, i.e. how the loss varies on time [99]. Its value will be a trade-off between a slow learning process, common for low values of η_0, and violent oscillations of the loss due to a large learning rate. The value of η_{tau} is then set roughly to 1 percent of the value of η_0 [99].
After iteration τ the learning rate is commonly left fixed to a constant η. Therefore, we have that the learning rate η_t is determined by:

$$\eta_t = \begin{cases} (1-\alpha)\eta_0 + \alpha\eta_{n_\tau}, & \text{for } t < \tau, \\ \eta, & \text{for } t \geq \tau. \end{cases} \tag{1.47}$$

Starting from SGD, several optimization methods with adaptive learning rate have been developed. Two popular modified versions of SGD are represented by *AdaGrad* (Adaptive Gradient) [74, 42] and *Root Mean Square Propagation* (RMSProp) [309, 42].

1.5.6 Momentum Method

In order to avoid getting stuck in a local minimum or slow learning processes, gradient descent is modified by introducing a velocity variable. We have thus that the gradient is modifying the velocity and not the position, which is affected by the changes of the velocity itself. This kind of technique has the goal of accelerating learning while performing the minimization of the loss function and it is called **momentum method** [242, 263, 99, 42].
Formally, a variable $\mathbf{v}$ is introduced with the role of velocity, representing thus the speed at which the parameters move in the parameter space. From a physical point of view, the momentum coincides

Algorithm 2 Stochastic Gradient Descent With Momentum [99]

Inputs:

- Loss function $\mathcal{L}_{\boldsymbol{\theta}}$;
- training dataset $\mathcal{D}_{\text{train}} = \{\mathbf{x}^i, \mathbf{y}^i\}_{i=1}^{n_{\text{train}}}$;
- batch size m;
- predicted outputs of the FNN $\hat{Y} = \{\hat{\mathbf{y}}^j\}_{j=1}^{n_{\text{train}}}$;
- initial value for the parameters $\boldsymbol{\theta}_0 = (W(0), \mathbf{b}(0))$ and for the velocity $\mathbf{v}_0$;
- number of mini-batches $n_{\text{batch}} = \dfrac{n_{\text{train}}}{m}$;
- learning rate η and momentum coefficient α.

```
1: for k=1,...,$n_{\text{batch}}$ do
2:     Sample a mini-batch of $m$ i.i.d. samples $\mathcal{D}_{\text{batch}} = \{\mathbf{x}^1, \dots, \mathbf{x}^m\}$;
3:     Determine the corresponding expected outputs $\mathbf{y}^i$ for each sample in $\mathcal{D}_{\text{batch}}$;
4:     Compute estimation of the gradient: $\tilde{\nabla}_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}}(Y_{\text{batch}}, \hat{Y}_{\text{batch}}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \mathcal{L}^i_{\boldsymbol{\theta}}(\mathbf{y}^i, \hat{\mathbf{y}}^i)$;
5:     Compute velocity update: $\mathbf{v}_t = \alpha \mathbf{v}_{t-1} - \eta \tilde{\nabla}_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}}(Y_{\text{batch}}, \hat{Y}_{\text{batch}})$;
6:     Update of the parameters: $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \mathbf{v}^k$.
7: end for
```

with mass times velocity and the negative gradient represents a force moving a particle through the parameter space, according to Newtons laws of motion. In our case, we are assuming unit mass, so velocity $\mathbf{v}$ coincides with the momentum of the particle.
As presented in [242], the simultaneous updates for the position and velocity are given by the following rule:

$$\begin{aligned} \mathbf{v}_{t+1} &= \mathbf{v}(t+1) = \alpha \mathbf{v}(t) - \eta \nabla f(\mathbf{x}(t)), \\ \mathbf{x}_{t+1} &= \mathbf{x}(t+1) = \mathbf{x}(t) + \mathbf{v}(t+1), \end{aligned} \tag{1.48}$$

with η be the learning rate and $\alpha \in [0, 1)$ a hyperparameter, called *momentum coefficient*, controlling how quickly the contributions of previous gradients exponentially decay [99]. Furthermore, the larger α than η, the more the previous gradients are affecting the current direction. It is also worth noting that, when $\alpha \longrightarrow 0$, the system of equations (1.48) becomes the gradient descent method presented in Section 1.5.4. Usually, α is set to be equal to 0.5, 0.9 or 0.99, but, as for the learning rate, it also may be adapted over time, starting from a small value, then raised during the process. Also in this case, we can provide a convergence result[13] as for the gradient descent method, that guarantees the convergence of both sequences $(\mathbf{x}_t)_t$ and $(\mathbf{v}_t)_t$ when the sequence of gradients tends to zero [42].
By applying equations (1.48) to our case, we gain the expression for the update of the parameters at step t, i.e. $\mathbf{v}_{t+1} = \mathbf{v}(t+1)$ and $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}(t+1)$:

$$\begin{aligned} \mathbf{v}_{t+1} &= \alpha \mathbf{v}_t - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \mathcal{L}^i_{\boldsymbol{\theta}}(\mathbf{y}^i, \hat{\mathbf{y}}^i), \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \mathbf{v}_{t+1}, \end{aligned} \tag{1.49}$$

[13]We are not proving this proposition since this would require the introduction of other notions and theorems and this is not the goal of our discussion. The interested reader can refer to Section 4.4 of [42] for the complete proof.

where $\boldsymbol{\theta} = (W, \mathbf{b})$ are the parameters of the FNN and we are using, as in Section 1.5.5, batch training. Algorithm 2 is summarizing the process of parameters update in the case of SGD coupled with momentum.
A variant of the momentum method is represented by *Nesterov Momentum* or *Nesterov Accelerated Gradient* (NAG) [297, 99], which is a first-order optimization method inspired by Nesterov's accelerated gradient method [225] and characterized by a better convergence rate than the gradient descent [42].

1.5.7 Adam Optimization Algorithm

Adam is an adaptive learning rate optimization algorithm, inspired by the previous AdaGrad e RMSProp methods [158, 42, 99]. The name Adam derives from *adaptive moments*, hence it uses estimations of the first and second moments of the gradient to adapt the learning rate for each weight of the neural network and then includes some bias corrections to these estimates, as summarized in Algorithm 3.
Formally, the loss function $\mathcal{L}(\boldsymbol{\theta})$ can be considered a random variable, since it is usually evaluated on some small random batch of data, differentiable with respect to $\boldsymbol{\theta}$. Hence in this case we are interested in solving this minimization problem:

$$\boldsymbol{\theta}^* = \text{argmin}_{\boldsymbol{\theta}} \mathbb{E}[\mathcal{L}(\boldsymbol{\theta})]. \tag{1.50}$$

The N-th momentum of a random variable X is the expected value of that variable to the power of N:

$$M_N = \mathbb{E}[X^N].$$

In particular, we are interested in the first and second moments, which represent the mean, and the uncentered[14] variance respectively. Denoting with $g_t = g(t) = \tilde{\nabla}_{\boldsymbol{\theta}(t)} \mathcal{L}(Y_{\text{batch}}, \hat{Y}_{\text{batch}})$ the gradient of the loss function at time step t on the current mini-batch $\mathcal{D}_{\text{batch}}$ and fixing an initial value for the parameter $\boldsymbol{\theta}(0) = \boldsymbol{\theta}_0$, in order to estimate the moments, Adam uses exponentially moving averages computed on the gradient evaluated on the current mini-batch:

$$\begin{aligned} \mathbf{m}_t &= \mathbf{m}(t) = \beta_1 \mathbf{m}(t-1) + (1-\beta_1) g_t, \\ \mathbf{v}_t &= \mathbf{v}(t) = \beta_2 \mathbf{v}(t-1) + (1-\beta_2)(g_t)^2, \end{aligned} \tag{1.51}$$

where $(g_t)^2$ denotes the elementwise square of g_t; $\beta_1 \in [0, 1)$ and $\beta_2 \in [0, 1)$ are two hyperparameters representing exponential decay rates for the momentum estimates, with default values of 0.9 and 0.999 respectively [158]. Then, we have that the vectors of moving averages are initialized with zeros, i.e. $\mathbf{m}(0) = 0$ and $\mathbf{v}(0) = 0$.
The moving averages $\mathbf{m}(t)$ and $\mathbf{v}(t)$ can be interpreted as biased estimates of the first and second moment of the gradient defined by the exponential moving average. We now want to correlate them with the first and second momentum of g_t, to gain, in the end, the following property:

$$\begin{aligned} \mathbb{E}[\mathbf{m}(t)] &= \mathbb{E}[g_t], \\ \mathbb{E}[\mathbf{v}(t)] &= \mathbb{E}[(g_t)^2]. \end{aligned} \tag{1.52}$$

[14]Uncentered refers to the fact that we are not subtracting the mean during variance calculation.

First, we need to rewrite Equations (1.51) in a more compact form. Hence, considering the relation defining $\mathbf{m}(t)$, we have:

$$\begin{aligned}
\mathbf{m}(0) &= 0,\\
\mathbf{m}(1) &= \beta_1\mathbf{m}(0) + (1-\beta_1)g_1 = (1-\beta_1)g_1,\\
\mathbf{m}(2) &= \beta_1\mathbf{m}(1) + (1-\beta_1)g_2 = \beta_1(1-\beta_1)g_1 + (1-\beta_1)g_2,\\
\mathbf{m}(3) &= \beta_1\mathbf{m}(2) + (1-\beta_1)g_3 = \beta_1^2(1-\beta_1)g_1 + \beta_1(1-\beta_1)g_2 + (1-\beta_1)g_3,\\
&\dots
\end{aligned}$$

Algorithm 3 Adam Optimization Algorithm [99]

Inputs:

- Loss function $\mathcal{L}_{\boldsymbol{\theta}}$;
- training dataset $\mathcal{D}_{\text{train}} = \{\mathbf{x}^i, \mathbf{y}^i\}_{i=1}^{n_{\text{train}}}$ and predicted outputs of the FNN, $\hat{Y} = \{\hat{\mathbf{y}}^j\}_{j=1}^{n_{\text{train}}}$;
- batch size m and number of mini-batches $n_{\text{batch}} = \dfrac{n_{\text{train}}}{m}$;
- learning rate η and constant ϵ, default values $\eta = 0.001$ and $\epsilon = 10^{-8}$;
- exponential decay rates for moment estimates β_1, β_2 with default values set at $\beta_1 = 0.9$ and $\beta_2 = 0.999$;
- initial value for the parameters $\boldsymbol{\theta}_0 = (W(0), \mathbf{b}(0))$;
- initial values for first and second moment $\mathbf{m}(0) = 0$ and $\mathbf{v}(0) = 0$.

```
1:  for t=1,...,$n_{\text{batch}}$ do
2:      Sample a mini-batch of $m$ i.i.d. samples $\mathcal{D}_{\text{batch}} = \{\mathbf{x}^1, \dots, \mathbf{x}^m\}$;
3:      Determine the corresponding expected outputs $\mathbf{y}^i$ for each sample in $\mathcal{D}_{\text{batch}}$;
4:      Compute estimation of the gradient: $g_t = \tilde{\nabla}_{\boldsymbol{\theta}(t)}\mathcal{L}_{\boldsymbol{\theta}}(Y_{\text{batch}}, \hat{Y}_{\text{batch}}) = \frac{1}{m}\sum_{i=1}^{m}\nabla_{\boldsymbol{\theta}(t)}\mathcal{L}^i_{\boldsymbol{\theta}}(\mathbf{y}^i, \hat{\mathbf{y}}^i)$;
5:      Compute biased first moment estimate: $\mathbf{m}(t) = \beta_1\mathbf{m}(t-1) + (1-\beta_1)g_t$;
6:      Compute biased second moment estimate: $\mathbf{v}(t) = \beta_2\mathbf{v}(t-1) + (1-\beta_2)(g_t)^2$;
7:      Bias correction first moment: $\hat{\mathbf{m}}(t) = \frac{\mathbf{m}(t)}{(1-\beta_1^t)}$;
8:      Bias correction second moment: $\hat{\mathbf{v}}(t) = \frac{\mathbf{v}(t)}{(1-\beta_2^t)}$;
9:      Update of the parameters: $\boldsymbol{\theta}(t) = \boldsymbol{\theta}(t-1) - \eta\frac{\hat{\mathbf{m}}(t)}{\sqrt{|\hat{\mathbf{v}}(t)|}+\epsilon}$.
10: end for
```

Proceeding by induction we can obtain the following formulas for $\mathbf{m}(t)$ and $\mathbf{v}(t)$:

$$\begin{aligned}
\mathbf{m}(t) &= (1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}g_i,\\
\mathbf{v}(t) &= (1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}(g_i)^2.
\end{aligned} \tag{1.53}$$

Now if we apply the expectation operator and we assume that the first and second moments for g_i are stationary, we have:

$$\begin{aligned}\mathbb{E}[\mathbf{m}(t)] &= (1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}\mathbb{E}[g_i] = (1-\beta_1^t)\mathbb{E}[g_t],\\ \mathbb{E}[\mathbf{v}(t)] &= (1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\mathbb{E}[(g_i)^2] = (1-\beta_2^t)\mathbb{E}[(g_t)^2],\end{aligned} \tag{1.54}$$

where in the last two equalities we have exploited the formula for the sum of a finite geometric series. Therefore, from these relations we can deduce that we have a biased estimator, that, in order to obtain (1.52), needs to be corrected by employing, for example, the *bias-correction*. In this case, the obtained bias-corrected moments are thus:

$$\begin{aligned}\hat{\mathbf{m}}(t) &= \frac{\mathbf{m}(t)}{(1-\beta_1^t)},\\ \hat{\mathbf{v}}(t) &= \frac{\mathbf{v}(t)}{(1-\beta_2^t)}.\end{aligned} \tag{1.55}$$

The final recursive formula for parameters update is then constructed using these moving averages to scale the learning rate individually for each parameter, yielding:

$$\boldsymbol{\theta}(t) = \boldsymbol{\theta}(t-1) - \eta\frac{\hat{\mathbf{m}}(t)}{\sqrt{|\ \hat{\mathbf{v}}(t)\ |} + \epsilon}, \tag{1.56}$$

with $\epsilon > 0$ a scalar introduced for numerical stabilization to prevent division by zero. Following [158], some default values for η and ϵ are represented in this case by 0.001 and 10^{-8} respectively. A summary of the procedure can be found in Algorithm 3.
In [158], a variant of Adam, called *AdaMax* or *Adaptive Maximum Method*, is also presented, where $\mathbf{v}$ becomes the L-infinity norm of the value of $\mathbf{v}$ at the previous step and the past gradients.
Another optimization strategy using Adam is represented by *SWATS* [154], which employs Adam together with SGD during the training process, switching from one to another when certain criteria hits. It has been shown in [322] that adaptive methods such as Adam do not generalize as well as SGD with momentum, especially when tested on a diverse set of deep learning tasks. For this reason, SWATS has been proved to achieve results comparable to SGD with momentum [322], since Adam outperforms SGD in earlier stages of the learning process, but then when it saturates, there is the transition to SGD.

1.5.8 Master Equations for a FNN

We can conclude by summarizing the relations obtained in the previous sections in a set of master equations [42]:

$$
\begin{aligned}
x_j^{(\ell)} &= \sigma\left(\sum_{i=1}^{n_{\ell-1}} w_{ji}^{(\ell)} x_i^{(\ell-1)} - b_j^{(\ell)}\right), && 1 \le j \le n_\ell \\
\delta_j^{(L+1)} &= (y_j - \hat{y}_j)\sigma'(h_j^{(L+1)}), && 1 \le j \le n_{\text{out}}, \\
\delta_i^{(\ell-1)} &= \sigma'(h_i^{(\ell-1)}) \sum_{k=1}^{n_\ell} \delta_k^{(\ell)} w_{ki}^{(\ell)}, && 1 \le i \le n_{\ell-1}, \\
\frac{\partial \mathcal{L}}{\partial w_{ji}^{(\ell)}} &= \delta_j^{(\ell)} x_i^{(\ell-1)}, && 1 \le j \le n_\ell, \quad 1 \le i \le n_{\ell-1}, \\
\frac{\partial \mathcal{L}}{\partial b_j^{(\ell)}} &= -\delta_j^{(\ell)}, && 1 \le j \le n_\ell, \\
w_{ji}^{(\ell)}(t+1) &= w_{ji}^{(\ell)}(t) - \eta \delta_j^{(\ell)}(t) x_i^{(\ell-1)}(t), && 1 \le j \le n_\ell, \quad 1 \le i \le n_{\ell-1}, \\
b_j^{(\ell)}(t+1) &= b_j^{(\ell)}(t) + \eta \delta_j^{(\ell)}(t), && 1 \le j \le n_\ell,
\end{aligned}
\tag{1.57}
$$

where we have that:

- the first equation provides a forward recursive formula of the outputs in terms of weights and biases;
- the second one gives an expression for the delta in the output layer (in the case the cost function is the L^2-error function);
- the third equation represents the backpropagation formula for the deltas;
- the fourth and the fifth equations are the partial derivative of the loss function $\mathcal{L}$ with respect to weights and biases;
- the last two equations represent the approximation sequences for the optimal values of weights and biases.

The fourth and the fifth equations can also be used to obtain the differential of the loss function $\mathcal{L}$, assessing thus the sensitivity of $\mathcal{L}$ with respect to small changes in weights and biases:

$$
\begin{aligned}
d\mathcal{L} &= \sum_{\ell=1}^{L+1} \sum_{i=1}^{n_{\ell-1}} \sum_{j=1}^{n_\ell} \frac{\partial \mathcal{L}}{\partial w_{ji}^{(\ell)}} dw_{ji}^{(\ell)} + \sum_{\ell=1}^{L+1} \sum_{j=1}^{n_\ell} \frac{\partial \mathcal{L}}{\partial b_j^{(\ell)}} db_j^{(\ell)} \\
&= \sum_{\ell=1}^{L+1} \sum_{i=1}^{n_{\ell-1}} \sum_{j=1}^{n_\ell} \delta_j^{(\ell)} x_j^{(\ell-1)} dw_{ji}^{(\ell)} - \sum_{\ell=1}^{L+1} \sum_{j=1}^{n_\ell} \delta_j^{(\ell)} db_j^{(\ell)},
\end{aligned}
\tag{1.58}
$$

The previous set of equations (1.57) can then be rewritten in a more compact and simple form using the matrix notation. To do this, we need to introduce the *Hadamard product* of two vectors [42, 136], i.e. the elementwise product between two vectors. Let $\mathbf{u}$ and $\tilde{\mathbf{u}}$ be two elements of $\mathbb{R}^n$, the Hadamard product, denoted with $\odot$, is a vector in $\mathbb{R}^n$ defined as:

$$
\mathbf{u} \odot \tilde{\mathbf{u}} = (u_1\tilde{u}_1, \ldots, u_n\tilde{u}_n)^T. \tag{1.59}
$$

Hence, using the following matrix notations:

$$\mathbf{x}^{(\ell)} = (x_1^{(\ell)}, \dots, x_{n_\ell}^{(\ell)})^T, \quad W^{(\ell)} = (w_{ji}^{(\ell)})_{ji}, \quad \mathbf{b}^{(\ell)} = (b_1^{(\ell)}, \dots, b_{n_\ell}^{(\ell)})^T, \quad \boldsymbol{\delta}^{(\ell)} = (\delta_1^{(\ell)}, \dots, \delta_{n_\ell}^{(\ell)})^T,$$
$$\mathbf{h}^{(\ell)} = (h_1^{(\ell)}, \dots, h_{n_\ell}^{(\ell)})^T, \quad \mathbf{y} = (y_1, \dots, y_{n_\text{out}})^T, \quad \hat{\mathbf{y}} = (\hat{y}_1, \dots, \hat{y}_{n_\text{out}})^T, \tag{1.60}$$

and the convention that the activation function σ acts on each component, equation (1.57) can be rewritten as:

$$\begin{aligned}
\mathbf{x}^{(\ell)} &= \sigma\left(W^{(\ell)T}\mathbf{x}^{(\ell-1)} - \mathbf{b}^{(\ell)}\right),\\
\boldsymbol{\delta}^{(L+1)} &= (\mathbf{y} - \hat{\mathbf{y}}) \odot \sigma'(\mathbf{h}^{(L+1)}),\\
\boldsymbol{\delta}^{(\ell-1)} &= \left(W^{(\ell)}\boldsymbol{\delta}^{(\ell)}\right) \odot \sigma'(\mathbf{h}^{(\ell-1)}),\\
\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} &= \mathbf{x}^{(\ell-1)}\boldsymbol{\delta}^{(\ell)T},\\
\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} &= -\boldsymbol{\delta}^{(\ell)},\\
W^{(\ell)}(t+1) &= W^{(\ell)}(t) - \eta\mathbf{x}^{(\ell-1)}(t)\boldsymbol{\delta}^{(\ell)T}(t),\\
\mathbf{b}^{(\ell)}(t+1) &= \mathbf{b}^{(\ell)}(t) + \eta\boldsymbol{\delta}^{(\ell)}(t).
\end{aligned} \tag{1.61}$$

It should be pointed out that the right side of the fourth equation (also present in the sixth equation) represents a $n_{\ell-1} \times n_\ell$ matrix, where each component is given by $x_i^{(\ell-1)}\delta_j^{(\ell)}$ and not a number.

1.5.9 Testing Phase

Once a FNN has been trained, it needs to be tested in order to understand how accurate is in making predictions[15] [334]. The purpose of the testing phase is thus to compare the FNN outputs, $\hat{\mathbf{y}}$ against targets, i.e. the expected values $\mathbf{y}$, using a different set with respect to the one used for training, namely the testing set $\mathcal{D}_\text{test}$. This represents a fundamental step in a deployment phase because only when we have a performing FNN, we can move to the practical application of the model. There exist different methods to compute the level of accuracy of a FNN, based on the task to solve. Usually, to evaluate the entity of testing errors, we introduce a loss function, that measures the difference between the predicted output of a FNN and the expected output for the sample in $\mathcal{D}_\text{test}$. Some typical examples were provided in Section 1.5.2, but there can be also other options. For instance, we can employ the *Minkowski error*:

$$\mathcal{L}_\text{Minkowski}(Y, \hat{Y}) = \frac{1}{n_\text{train}}\sum_{i=1}^{n_\text{train}} \mathcal{L}^i_\text{Minkowski}(\mathbf{y}^i, \hat{\mathbf{y}}^i) = \frac{1}{n_\text{train}}\sum_{i=1}^{n_\text{train}} (\mathbf{y}^i - \hat{\mathbf{y}}^i)^\gamma, \tag{1.62}$$

where the exponent γ, called the Minkowski parameter, can vary between 1 and 2. It can thus be deduced that this is a general version of the MSE, where $\gamma = 2$, that, thanks to the introduction of γ is more insensitive to outliers[16].

In classification problems, as the one presented in Chapter 2, the most common performance metric relies on *confusion matrix* [112, 314]. Based on the expected and predicted outputs for each sample in $\mathcal{D}_\text{test}$, we can construct that matrix, where the rows represent the target classes in the data set

[15] We should highlight that the topic of this section, namely the testing phase, remains valid when applied to a general ANN. In this case, the hypothesis of having only forward connections is not required, contrary to the backpropagation case.

[16] With the term *outlier* we refer to data points which lie far away from most of the data points.

and the columns the corresponding predicted output classes. Hence, for each output of the FNN under consideration, we are evaluating if the prediction is correct or not. In particular, in order to quantify the level of accuracy of our model, the notions of true positive, true negative, false positive, and false negative are introduced, together with that of recall and precision. However, the explanation of these concepts is postponed to the next chapter and in particular in Section 2.2.7, where we will introduce the problems of image recognition and object detection.

1.6 Parameter Initialization Strategies

When building a neural network, we do not only need to decide the topology of the ANN, but we also need to initialize its parameters. This represents an important step because, if done optimally, then the learning phase will proceed quickly reaching the optimal solution, otherwise converging to a minimum using gradient descent will be impossible or very slow. The magnitude of parameters plays also a fundamental role in avoiding, as much as possible, problems connected with vanishing and exploding gradients [42, 99].
The most natural choice may seem to be to initialize all weights and biases to zero or sample them from a uniform or a Gaussian distribution of zero mean. In the case of a small ANN, made of few layers, this can be a good option since it usually provides satisfactory enough converge results [42]. This is not true when coming to deep neural networks. Having zero initialization or parameters close to zero leads to the loss of meaning in the output signal. In order to understand this statement, we can show that the variance of the input decreases as it passes through each layer of the network [42]. To prove this, we need to recall Equation (1.12), defining the output of layer ℓ:

$$x_j^{(\ell)} = \sigma\left(\sum_{i=1}^{n_{\ell-1}} w_{ji}^{(\ell)} x_i^{(\ell-1)} - b_j^{(\ell)}\right), \qquad \text{for } j = 1, \dots, n_\ell,$$

and to derive a variance approximation formula[17] for $f(\mathbf{x})$, with f a differentiable function [42]. Let $\bar{m} = \mathbb{E}[\mathbf{x}]$ and consider the linear approximation of f at $\mathbb{E}[\mathbf{x}]$:

$$f(\mathbf{x}) = f(\bar{m}) + f'(\bar{m})(\mathbf{x} - \bar{m}) + o(\mathbf{x} - \bar{m})^2.$$

By applying the variance on both sides and by employing variance properties, we gain the following approximation formula for the variance of $f(\mathbf{x})$:

$$Var(f(\mathbf{x})) \approx f'(\bar{m})^2 Var(\mathbf{x}) = f'(\mathbb{E}[\mathbf{x}])^2 Var(\mathbf{x}), \tag{1.63}$$

where we are hypothesizing that the mean and the variance of $\mathbf{x}$ are finite.
Starting again from Equation (1.12), if we compute the variance of both sides of the equation and we use formula (1.63) and the properties of variance and expected value, we obtain the following approximation:

$$Var(x_j^{(\ell)}) \approx \sigma'\left(\sum_{i=1}^{n_{\ell-1}} w_{ji}^{(\ell)} \mathbb{E}[x_i^{(\ell-1)}] - b_j^{(\ell)}\right)^2 \left(\sum_{i=1}^{n_{\ell-1}} (w_{ji}^{(\ell)})^2 Var(x_i^{(\ell-1)})\right), \tag{1.64}$$

where we are assuming that $x_j^{(\ell)}$ are independent and identically distributed (i.i.d.) random variables for each $j = 1, \dots, n_\ell$, $\ell = 0, \dots, L+1$, and we are treating the weights as constants and not as random variables. We can observe that the variance of the output of j-th neuron in layer ℓ is written

[17]For more details the reader can refer to Appendix D.4 of [42].

in terms of the variances of neurons output of the previous layer. Exploiting now Cauchy-Schwarz inequality in the second multiplication factor yields:

$$\sum_{i=1}^{n_{\ell-1}} w_{ji}^{(\ell)} Var(x_i^{(\ell-1)}) \leq \left(\sum_{i=1}^{n_{\ell-1}} \left(w_{ji}^{(\ell)}\right)^2\right)^{1/2} \left(\sum_{i=1}^{n_{\ell-1}} \left(Var(x_i^{(\ell-1)})\right)^2\right)^{1/2}. \tag{1.65}$$

Supposing that the first derivative of the activation function is bounded, i.e. $(\sigma')^2 < c$, then if we take the sum over j in Equation (1.64) and we use the inequality (1.65), this leads to:

$$\sum_{j=1}^{n_\ell} Var(x_j^{(\ell)}) < c \left(\sum_{i=1}^{n_{\ell-1}} \left(w_{ji}^{(\ell)}\right)^4\right)^{1/2} \left(\sum_{i=1}^{n_{\ell-1}} \left(Var(x_i^{(\ell-1)})\right)^2\right)^{1/2}.$$

Taking then the square, we obtain:

$$\sum_{j=1}^{n_\ell} \left(Var(x_j^{(\ell)})\right)^2 \leq \left(\sum_{j=1}^{n_\ell} Var(x_j^{(\ell)})\right)^2 < c^2 \left(\sum_{i=1}^{n_{\ell-1}} \left(w_{ji}^{(\ell)}\right)^4\right) \left(\sum_{i=1}^{n_{\ell-1}} \left(Var(x_i^{(\ell-1)})\right)^2\right). \tag{1.66}$$

Since we have made the hypothesis that the weights are close to zero, this is indicating that the sum of the square of variances in layer ℓ is lower than the sum of the square of variances in the $\ell - 1$ layer. In other words, the signal's variance is decreasing to zero after passing through a few layers, leading to an output too low to be significant.
The opposite case coincides with having too large weights, which induce to have amplified variance as it passes through network layers, exploding values during forward or backward propagation, or a vanishing gradient problem [42]. To understand this we can consider the identity function as activation function $\sigma(\mathbf{x}) = \mathbf{x}$, thus taking the variance in Equation (1.12) yields:

$$Var(x_j^{(\ell)}) = \sum_{i=1}^{n_{\ell-1}} (w_{ji}^{(\ell)})^2 Var(x_i^{(\ell-1)}), \tag{1.67}$$

which implies that the variance of the output of j-th neuron in layer ℓ is also large. On the other hand, if σ is of sigmoid type, large weights bring $\sum_i w_{ji}^{(\ell)} x_i^{(\ell-1)}$ to have larger values and hence the activation function σ tends to saturate, leading to an approaching zero gradient problem and a slow learning process.
Correct initialization of parameters is hence very important since it makes a significant difference in the way the optimality algorithm is converging, but also in the performances of the network. It is thus needed to find a reasonable range of values for the parameters $(W, \mathbf{b})$. In the next section, we are going to describe two approaches, commonly used approaches for parameter initialization: Xavier and Kaiming He.

1.6.1 Xavier Initialization

Following [42, 99], a reasonable initialization of parameters can be derived by assuming that they are uniform- or Gaussian- distributed random variables with zero mean. In particular, since the propagation of the error in a deep neural network can be quantized using the variance of the output of each layer [42], we are interested in keeping the variance under control, i.e. away from exploding or vanishing. It is thus necessary to find the weight values that leave the variance roughly unchanged through each layer of the network.
Starting again from Equation (1.12), we are assuming that $x_i^{(\ell)}$, for $\ell = 0, \ldots, L+1$, are i.i.d. random

variables with zero means, and $w_{ji}^{(\ell)}$, for $j = 1, \dots, n_\ell$, $i = 1, \dots, n_{\ell-1}$ and $\ell = 1, \dots .L+1$, are i.i.d. random variables with zero mean, i.e. such that $\mathbb{E}[w_{ji}^{(\ell)}] = 0$. We are also making the hypothesis that each $x_i^{(\ell)}$ is independent of each $w_{ji}^{(\ell)}$. Using formula (1.63) in Equation (1.12), we obtain

$$\begin{aligned}
Var(x_j^{(\ell)}) &= \sigma' \left(\sum_{i=1}^{n_{\ell-1}} \mathbb{E}[w_{ji}^{(\ell)} x_i^{(\ell-1)}] - b_j^{(\ell)} \right)^2 Var\left(\sum_{i=1}^{n_{\ell-1}} w_{ji}^{(\ell)} x_i^{(\ell-1)} \right) \\
&= \sigma'(b_j^{(\ell)})^2 \sum_{i=1}^{n_{\ell-1}} Var\left(w_{ji}^{(\ell)} x_i^{(\ell-1)}\right) \\
&= \sigma'(b_j^{(\ell)})^2 \sum_{i=1}^{n_{\ell-1}} Var(w_{ji}^{(\ell)}) Var(x_i^{(\ell-1)}) \\
&= n_{\ell-1} \sigma'(0)^2 Var(w_{ji}^{(\ell)}) Var(x_i^{(\ell-1)}),
\end{aligned} \tag{1.68}$$

where we have employed in the second identity the property $\mathbb{E}[w_{ji}^{(\ell)} x_i^{(\ell-1)}] = \mathbb{E}[w_{ji}^{(\ell)}]\mathbb{E}[x_i^{(\ell-1)}] = 0$ and the additivity of variance for independent variables. In the third identity, we have exploited Goodman's formula[18] [101], defining the multiplicative property of variance, for independent variables with zero means. In the last identity we have assumed that the bias is initialized to 0 and we have used that weights $w_{ji}^{(\ell)}$ and $x_j^{(\ell)}$ are identically distributed.
Once we gain this equation, we can impose the condition that the variance is invariant under the ℓ-th layer:

$$n_{\ell-1} \sigma'(0)^2 Var(w_{ji}^{(\ell)}) Var(x_i^{(\ell-1)}) = Var(x_i^{(\ell-1)}), \tag{1.69}$$

from which it can be obtained the following relation for the variance of weights:

$$Var(w_{ji}^{(\ell)}) = \frac{1}{n_{\ell-1}\sigma'(0)^2}. \tag{1.70}$$

Now there can be several cases, based on the distribution chosen for the variables. For example, if we consider having uniform o normal distributed random variables [42], we obtain:

1. *Normal distribution*: $w_{ji}^{(\ell)} \sim \mathcal{N}\left(0, \frac{1}{n_{\ell-1}\sigma'(0)^2}\right)$;

2. *Uniform distribution*: $w_{ji}^{(\ell)} \sim Unif[-a, a]$, i.e. we are considering the weights as uniform distributed random variables on the interval $[-a, a]$ with zero mean. Now using the fact that the variance for this type of variables is given by $a^2/3$, we can equate the relations we have obtained for the variance of $w_{ji}^{(\ell)}$ and get the value of a, that is $\frac{\sqrt{3}}{\sigma'(0)\sqrt{n_{\ell-1}}}$. Hence at the end we have: $w_{ji}^{(\ell)} \sim Unif\left[-\frac{\sqrt{3}}{\sigma'(0)\sqrt{n_{\ell-1}}}, \frac{\sqrt{3}}{\sigma'(0)\sqrt{n_{\ell-1}}}\right]$.

In this approach, we have only taken into account the number of input for the ℓ-th layer, $n_{\ell-1}$, without involving also the outgoing number of neurons. Starting from this and assuming that the activation functions are linear, it can be derived a new formula, called **Xavier initialization** [97],

[18] Goodman's formula [101] is providing an expression for the variance of the product of two independent variables X and Y:

$$Var(XY) = \mathbb{E}[X]^2 Var(Y) + \mathbb{E}[Y]^2 Var(X) + Var(X)Var(Y),$$

that in the case of variables with zero means simply becomes $Var(XY) = Var(X)Var(Y)$.

that aims at preserving the backpropagation signal as well. In this case, we are thus assuming also that the variances of the cost function gradient remain unchanged as the output is backpropagated through the ℓ-th layer:

$$Var\left(\frac{\partial \mathcal{L}}{\partial w_{ji}^{(\ell-1)}}\right) = Var\left(\frac{\partial \mathcal{L}}{\partial w_{ji}^{(\ell)}}\right). \tag{1.71}$$

Using now Equation (1.23), derived for the partial derivative of the loss function with respect to the weights, we obtain:

$$Var(\delta_j^{(\ell-1)} x_i^{(\ell-2)}) = Var(\delta_j^{(\ell)} x_i^{(\ell-1)}). \tag{1.72}$$

Equation (1.33) provides an explicit expression for deltas, where if we use the hypothesis of linearity of σ, e.g. $\sigma(\mathbf{x}) = \mathbf{x}$, we gain:

$$\delta_i^{(\ell-1)} = \sigma'(h_i^{(\ell-1)}) \sum_{k=1}^{n_\ell} \delta_k^{(\ell)} w_{ki}^{(\ell)} = \sum_{k=1}^{n_\ell} \delta_k^{(\ell)} w_{ki}^{(\ell)}. \tag{1.73}$$

It can be noticed[19] that each $\delta_k^{(\ell)}$ is independent of $w_{ki}^{(\ell)}$ and $x_i^{(\ell-1)}$. Hence, taking the expected value in the last relation leads to:

$$\mathbb{E}[\delta_i^{(\ell-1)}] = \mathbb{E}[\sum_{k=1}^{n_\ell} \delta_k^{(\ell)} w_{ki}^{(\ell)}] = \sum_{k=1}^{n_\ell} \mathbb{E}[\delta_k^{(\ell)}]\mathbb{E}[w_{ki}^{(\ell)}] = 0,$$

where we are using the assumption that the weights are random variables with zero mean. Hence, this implies that $\mathbb{E}[\delta_i^{(\ell}] = 0$ for each $i = 1, \ldots, n_\ell$ and for each $\ell = 1, \ldots, L+1$.
Going back to Equation (1.72), if we exploit the fact that $\mathbf{x}_i^{(\ell)}$ are random variables with zero mean and the Goodman's formula in the case of independent variables with zero mean, we have:

$$Var(\delta_j^{(\ell-1)})Var(x_i^{(\ell-2)}) = Var(\delta_j^{(\ell)})Var(x_i^{(\ell-1)}), \tag{1.74}$$

that can be further simplified using the assumption that the variance is invariant under layer $\ell - 1$, i.e. $Var(x_i^{(\ell-2)}) = Var(x_i^{(\ell-1)})$:

$$Var(\delta_j^{(\ell-1)}) = Var(\delta_j^{(\ell)}). \tag{1.75}$$

This last expression is hence proving that also the deltas variance remains unchanged.
Once, we have obtained that deltas are characterized by zero mean and this invariant property of the variance, we can apply the variance in Equation (1.73) obtaining:

$$Var(\delta_i^{(\ell-1)}) = \sum_{k=1}^{n_\ell} Var(\delta_k^{(\ell)})Var(w_{ki}^{(\ell)}) = n_\ell Var(\delta_k^{(\ell)})Var(w_{ki}^{(\ell)}), \tag{1.76}$$

where we have used the fact that the deltas and weights are i.i.d., the properties of variance, and Goodman's formula. Exploiting now Equation (1.75), this yields:

$$Var(w_{ki}^{(\ell)}) = \frac{1}{n_\ell}, \tag{1.77}$$

namely the variance of the weights in layer ℓ is inversely proportional to the number of neurons in that layer. This expression has to coexist with the one derived before (1.70), that in this case of linear activation function becomes:

$$Var(w_{ki}^{(\ell)}) = \frac{1}{n_{\ell-1}}. \tag{1.78}$$

[19] It can be proved by induction using the backpropagation formula. The interested reader can check Chapter 6 of [42] for the proof.

Equation (1.77) and Equation (1.78) are satisfied simultaneously if and only if $n_\ell = n_{\ell-1}$, i.e. the number of neurons in any two consecutive layers is the same. Since this represents a strong condition to impose, a compromise can be to take the arithmetic average of the two different expressions:

$$Var(w_{ki}^{(\ell)}) = \frac{2}{n_{\ell-1} + n_\ell}. \tag{1.79}$$

We can emphasize again what happens for the weights in the two different cases highlighted before:

1. *Normal distribution*: $w_{ji}^{(\ell)} \sim \mathcal{N}\left(0, \frac{2}{n_{\ell-1} + n_\ell}\right)$;
2. *Uniform distribution*: $w_{ji}^{(\ell)} \sim Unif\left[-\frac{\sqrt{6}}{\sqrt{n_{\ell-1} + n_\ell}}, \frac{\sqrt{6}}{\sqrt{n_{\ell-1} + n_\ell}}\right]$, that is also called *normalized initialization.*

1.6.2 Kaiming He Initialization

Xavier initialization is very effective in improving the functionality of the backpropagation process but has been derived for a neural network without nonlinearity. This assumption is obviously too restrictive, even if it has been shown that this initialization strategy performs well also in the case of nonlinear activation functions symmetric around zero and with values in $[-1, 1]$, such as the hyperbolic tangent [99]. Hence, if we are considering other types of activation functions, e.g. ReLU, the **Kaiming He initialization** [119] should be used.
In order to derive the Kaiming He initialization, we should make the following assumptions:

1. $x_i^{(\ell)}$, for $\ell = 0, \dots, L+1$, are i.i.d. random variables;
2. $w_{ji}^{(\ell)}$, for $j = 1, \dots, n_\ell$, $i = 1, \dots, n_{\ell-1}$ and $\ell = 1, \dots .L+1$, are i.i.d. random variables with zero mean;
3. $x_i^{(\ell)}$ and $w_{ji}^{(\ell)}$, with $j = 1, \dots, n_\ell$, $i = 1, \dots, n_{\ell-1}$ and $\ell = 1, \dots .L+1$, are independent of each other.

It is important to highlight that in this case $x_i^{(\ell)}$ does not have zero mean as for Xavier, because it is the result of a ReLU which does not have zero mean.
Let's start by considering the argument of the activation function in Equation (1.12) defining the total input arriving at neuron j in layer ℓ:

$$h_j^{(\ell)} = \sum_{i=1}^{n_{\ell-1}} w_{ji}^{(\ell)} x_i^{(\ell-1)} - b_j^{(\ell)}, \qquad j = 1, \dots, n_\ell. \tag{1.80}$$

For simplicity and in accordance with [119], the bias is set to be equal to zero. Note that the expected value for $h_j^{(\ell)}$ is zero in this case since the weights have zero mean and from hypothesis 3 we have the independence of the variables:

$$\mathbb{E}[h_j^{(\ell)}] = \mathbb{E}\left[\sum_{i=1}^{n_{\ell-1}} w_{ji}^{(\ell)} x_i^{(\ell-1)}\right] = \sum_{i=1}^{n_{\ell-1}} \mathbb{E}[w_{ji}^{(\ell)}]\mathbb{E}[x_i^{(\ell-1)}] = 0.$$

If we take the variance on both sides in Equation (1.80), we obtain:

$$Var(h_j^{(\ell)}) = Var\left(\sum_{i=1}^{n_{\ell-1}} w_{ji}^{(\ell)} x_i^{(\ell-1)}\right) = \sum_{i=1}^{n_{\ell-1}} Var(w_{ji}^{(\ell)} x_i^{(\ell-1)}) = n_{\ell-1} Var(w_{ji}^{(\ell)} x_i^{(\ell-1)}), \tag{1.81}$$

where hypothesis 3 justifies the fact that the variance of the sum is the sum of variances, while hypotheses 1 and 2 the last equality, since $w_{ji}^{(\ell)}$ and $x_i^{(\ell)}$ are identically distributed. Now, applying Goodman's formula and using the fact that weights have zero mean yields:

$$\begin{aligned} Var(h_j^{(\ell)}) &= n_{\ell-1}\left(\mathbb{E}[w_{ji}^{(\ell)}]^2 Var(x_i^{(\ell-1)}) + Var(w_{ji}^{(\ell)})\mathbb{E}[x_i^{(\ell-1)}]^2 + Var(w_{ji}^{(\ell)})Var(x_i^{(\ell-1)})\right) \\ &= n_{\ell-1}\left(Var(w_{ji}^{(\ell)})\mathbb{E}[x_i^{(\ell-1)}]^2 + Var(w_{ji}^{(\ell)})Var(x_i^{(\ell-1)})\right) \\ &= n_{\ell-1}Var(w_{ji}^{(\ell)})\left(\mathbb{E}[x_i^{(\ell-1)}]^2 + Var(x_i^{(\ell-1)})\right) \\ &= n_{\ell-1}Var(w_{ji}^{(\ell)})\mathbb{E}[(x_i^{(\ell-1)})^2], \end{aligned} \tag{1.82}$$

where in the last step we have used the definition of variance. It is now needed to compute the expected value of $(x_i^{(\ell-1)})^2$:

$$\begin{aligned} \mathbb{E}[(x_i^{(\ell-1)})^2] &= \int_{-\infty}^{\infty} (x_i^{(\ell-1)})^2 P(x_i^{(\ell-1)}) dx_i^{(\ell-1)} \\ &= \int_{-\infty}^{\infty} \max(0, h_j^{(\ell-1)})^2 P(h_j^{(\ell-1)}) dh_j^{(\ell-1)} \\ &= \int_{0}^{\infty} (h_j^{(\ell-1)})^2 P(h_j^{(\ell-1)}) dh_j^{(\ell-1)} \\ &= \frac{1}{2}\int_{-\infty}^{\infty} (h_j^{(\ell-1)})^2 P(h_j^{(\ell-1)}) dh_j^{(\ell-1)} \\ &= \frac{1}{2}Var(h_j^{(\ell-1)}), \end{aligned}$$

where in the second equality we have used the fact that $x_i^{(\ell-1)}$ is obtained from $h_j^{(\ell-1)}$ by applying the ReLU activation function.
Substituting this relation in Equation (1.82) gains:

$$Var(h_j^{(\ell)}) = \frac{1}{2}n_{\ell-1}Var(w_{ji}^{(\ell)})Var(h_j^{(\ell-1)}). \tag{1.83}$$

Now rewriting this equation for layer $L+1$, i.e. for the output of the net, we obtain:

$$Var(h_j^{(L+1)}) = Var(\hat{y}_j) = Var(y_1)\left(\prod_{i=2}^{L+1} \frac{n_{\ell-1}}{2} Var(w_{ji}^{(\ell)})\right). \tag{1.84}$$

We need now to impose the condition that the variance at the output and at the beginning is the same, preventing thus exploding or vanishing gradients. In accordance with [119], a sufficient condition is represented by:

$$\frac{n_{\ell-1}}{2}Var(w_{ji}^{(\ell)}) = 1, \qquad \forall\, \ell, \tag{1.85}$$

that implies:

$$Var(w_{ji}^{(\ell)}) = \frac{2}{n_{\ell-1}}. \tag{1.86}$$

Following [119], a good way to initialize the weights is using a zero-mean normal distribution:

$$w_{ji}^{(\ell)} \sim \mathcal{N}\left(0, \frac{2}{n_{\ell-1}}\right). \tag{1.87}$$

As before, after obtaining a relation using only the forward pass, we need to derive one also taking into account backpropagation [119]. Hence, starting from Equation (1.73) without using the hypothesis of having linear activation functions as in the Xavier framework, and taking the variance of both sides we obtain:

$$\begin{aligned} Var(\delta_i^{(\ell-1)}) &= Var(\sigma'(h_i^{(\ell-1)})\sum_{k=1}^{n_\ell}\delta_k^{(\ell)}w_{ki}^{(\ell)}) \\ &= Var(\sigma'(h_i^{(\ell-1)}))\sum_{k=1}^{n_\ell}Var(\delta_k^{(\ell)})Var(w_{ki}^{(\ell)}) \\ &= \frac{1}{2}n_\ell Var(\delta_k^{(\ell)})Var(w_{ki}^{(\ell)}), \end{aligned} \tag{1.88}$$

where in the second equality we have used the three hypotheses done and we have assumed that $\sigma'(h_i^{(\ell-1)})$ is independent of deltas and weights. In the last equality we have then employed[20] that $Var(\sigma'(h_i^{(\ell-1)})) = 1/2$. Imposing also in this case the property that the variance of deltas is invariant, we have that:

$$Var(w_{ki}^{(\ell)}) = \frac{2}{n_\ell}. \tag{1.89}$$

Now using a zero-mean normal distribution, we conclude:

$$w_{ji}^{(\ell)} \sim \mathcal{N}\left(0, \frac{2}{n_\ell}\right). \tag{1.90}$$

We have thus obtained two equations, (1.86) and (1.89), for the variance of the weights that have to be satisfied simultaneously. Substituting Equation (1.89) in Equation (1.84) and using the property that the variance is invariant, we obtain:

$$1 = \prod_{i=2}^{L+1}\frac{n_{\ell-1}}{2}Var(w_{ji}^{(\ell)}) = \prod_{i=2}^{L+1}\frac{n_{\ell-1}}{2}\frac{2}{n_\ell} = \prod_{i=2}^{L+1}\frac{n_{\ell-1}}{n_\ell}, \tag{1.91}$$

that is imposing a condition on the design of the network, not as strict as for Xavier initialization.

[20]In order to obtain this relation, it is only needed to derive the expression for the derivative of the ReLU function and hence compute first the expected value and then the variance, taking into account that the two possible assumed values have the same probability [119].

CHAPTER 2 Convolutional Neural Networks

2.1 Introduction

Computer vision is a broad interdisciplinary scientific field, that deals with extracting information and a high-level understanding from digital images or videos. As for ANNs, the key inspiration comes from the human brain and the will to imitate the biological processes related to vision [72]. For example, a key ability of the human brain coincides with *invariant object recognition*, which refers to an instantaneous and accurate recognition of objects and in particular their similarity with something known in the presence of variations such as size, rotation, illumination, and position. Hence, even in presence of complex, distorted scenes, it allows us to identify objects in a fraction of a second. This increasing interest in understanding how this works and how it can be reproduced artificially leads to a growth of studies and experiments in this field.
In the 1950s and 1960s, David Hubel and Torsten Wiesel addressed the topic of visual perception through neurophysiological research conducted on cats [140]. By studying how neurons reacted to various stimuli, it emerged that each neuron fired when exposed to certain features, but not to all of them, leading to the discovery of different types of cells in the visual cortex — simple, complex, and hypercomplex — responsible for learning simple things such as detecting edges and corner, or, as they become more complex, more and more specific features. In the 1970s, David Marr, a neuroscientist at MIT, published one of the fundamental books in this context [206], paving the way for various biologically-inspired research on computer vision. He discovered indeed that the visual system is *hierarchical*, hence first of all neurons detect simple features, called *low-level features*, like edges, corners, then feed into *high-level features*, namely more complex features, e.g. shapes, or more complex visual representations, taking into account possible relations between features. One of Marr's central contributions was also connected with his *representational framework for vision*, which focused on the vision task of deriving shape information from images. Hence, he proposed a model based on the generation of a sequence of increasingly symbolic representations of a scene. The input image is translated into a 'primal sketch' of the retinal image, where low-level features, such as corners, blobs, edges, and lines, are detected. Starting from this, progressively a more complex image is constructed, called the '2.5D sketch', analyzing the visible surface of the objects in terms of its spatial properties such as orientation, discontinuities, and depth. The final step coincides with the '3D model representation', dealing with spatial and volume properties, appearance, and other relations between items. This theory was later improved by Steven Palmer at the end of 1990s [231], by proposing his *model of visual perception*, characterized by four stages with increasing degrees of abstraction: image-based processing, surface-based processing, object-based processing, and category-based processing. These steps correspond exactly to those proposed by Marr, except for the introduction of the object-based stage, where occluded or unclear parts in a three-dimensional representation are filled in.
The ideas described in these studies have thus paved the way for the development of deep learning algorithms able to reproduce this task [14, 276]. A breakthrough in the computer vision community was indeed represented by the introduction of Convolutional Neural Networks (CNNs) [171] by Yann LeCun in 1989. In [171], he proposed *LeNet-5*, a simple network composed of two convolutional layers, a max pooling layer, and a fully connected layer, for optical character recognition in

documents, i.e. identifying handwritten zip codes on letters using a label data benchmark, called the Modified National Institute of Standards and Technology (MNIST) database. Even if it may not be suitable for complicated tasks, LeNet can be considered the backbone of the most recent CNN architectures. In [175], it was also investigated the possibility of extending the knowledge on ANN optimization by using SGD during the backpropagation phase to train a CNN.
The increased interest in employing such models to solve more complex tasks has forced CNN architectures to go deep, leading to computationally costly neural networks [149]. Furthermore, the lack of available fast computing resources resulted in a practical limitation for training CNNs, until, in the early 2000s, GPUs were introduced to enhance the capabilities to speed up these processes. In this period, the notion of *deep learning*, introduced in [4], started thus to become more and more important and attract a lot of attention. This new term represented not only a novel field of investigation but above all a breakthrough in the machine learning community, thanks to the development of new algorithms and processing units [128]. Deep learning began thus to heat up in the academic community, having its center in several universities, such as Stanford University, New York University, and the University of Montreal, but also started to attract the attention of companies like Microsoft and Google. 2012 represented then a revolutionary year, thanks to the introduction of CNNs in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [68], an annual image classification competition where research teams evaluate their algorithms on the given data set, in order to achieve higher accuracy on several visual recognition tasks. A team led by Alex Krizhevsky, from the University of Toronto, developed indeed for this purpose a CNN, called *AlexNet* [167], that achieved an unprecedented error rate of 15.3%, lower than the previously reached threshold of 26%. In particular, this represented also the first time that a team employed neural networks to tackle the image classification task of the challenge, changing thus the role of these algorithms in the computer vision scenario. The use of deep learning technology, such as GPUs, was then a breakthrough in the image processing field, leading to a performance improvement of CNNs, like that of AlexNet, and their application in numerous domains. In the following years of the competition, all the participants designed similar CNNs to solve the problem. In 2014, *VGGNet* [289] entered the competition, confirming that increasing the depth of the model has a crucial impact on improving performance. Even if VGG was not the winner of ILSVRC2014, it paved the way for further explorations of the key idea behind it. *ResNet* [118], winner of ILSVRC2015, encapsulated this idea in its architecture, by introducing also a new module, called Residual Block, to deal with the problem of vanishing and exploding gradients. Other important improvements are connected with *Inception networks* [301, 298], that proven the necessity of using wider modules, called inception blocks, to push performance in terms of speed and accuracy.
CNNs has thus revolutionized artificial intelligence by tackling complex problems, such as visual object recognition [167, 48], robotics [226, 240], speech recognition [104, 151, 224], natural language processing [328, 70], and autonomous vehicles [139, 145]. In this Chapter and thesis, we will focus only on two problems of interest to *Electrolux Professional*: image recognition and object detection. *Image recognition* has the aim of classifying the items in pictures, whereas *object detection* addresses also the intention to detect the position of these objects as well. The aforementioned CNNs represent a model solving the image recognition task, but also the building block of *object detectors*, deep learning architectures specialized for classification and localization of objects in images.
This chapter will deal with CNNs by explaining how they can be used to solve the problems of image recognition and object detection. In Section 2.2, a detailed description of the several layers composing the CNN architectures is carried out. Section 2.2.8 will then introduce common techniques employed to initialize the parameters for this kind of models.CNNs represents a possible solution for the image recognition task, topic of Section 2.3. We will indeed present this problem by introducing datasets and architectures widely employed in this context. In Section 2.4, we then extend the previous description to object detection by presenting the key notions of this framework.

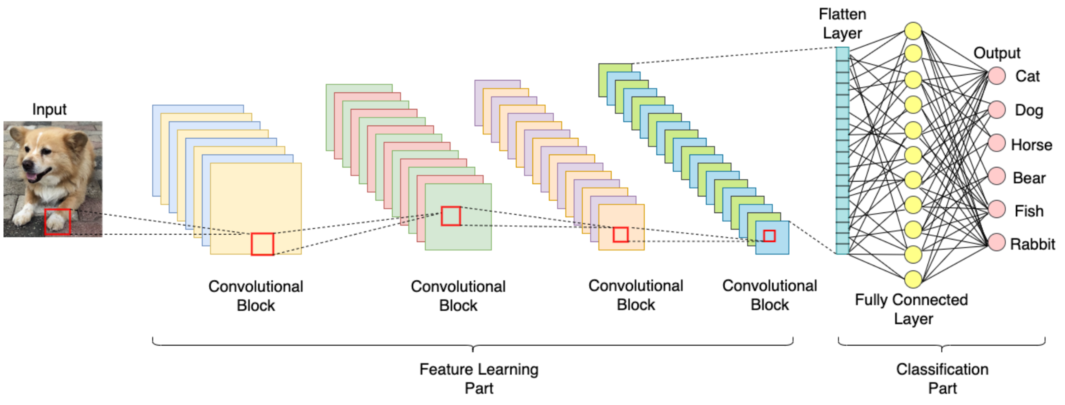


Figure 2.1: Schematic representation of a Convolutional Neural Network.

Also in this case, we briefly review commonly used benchmark datasets and architectures.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [171] are deep learning algorithms specialized in processing data with a known grid-like topology[21], such as time series (1-D grid) or images (2-D grid of pixels). For example, given in input an image, a CNN can extract information from it assigning importance to various features of the objects and hence differentiate between them through the parameters of the net, represented by the weights.

As depicted in Figure 2.1, the structure of a CNN can be mainly subdivided into two parts: the *feature learning part*, responsible for the detection of objects' features, and the *classification part*, that provides the category to which the objects in the picture belong to. The first part is characterized by several convolutional blocks[22], composed of a convolutional layer, a nonlinearity layer, and a pooling layer (see Figure 2.2), whereas a fully connected Feedforward Neural Network, i.e. a FNN made of fully-connected layers, is used for classification. Intuitively, the idea is that in the first part a CNN detects several features, such as corners, lines, or blocks of color in the initial layers (*low-level features*), or combinations gradually more complex in the upper layers (*high-level features*), and then fed them into a FNN to obtain the final classification. In the following subsections, we are going to provide more details on each of these structures [99, 42, 334], in order to understand how a CNN works.

Before diving deeply into the description of the different layers of a CNN, we need to introduce the notation that will be used in the whole chapter, following the one set in Chapter 1. Since a CNN is a particular type of ANN, it can be represented as a function $\mathcal{CNN}$, that maps an input tensor $\mathbf{x}^0$ into the output predicted by the net $\hat{\mathbf{y}}$:

$$\hat{\mathbf{y}} = \mathcal{CNN}(\mathbf{x}^{(0)}), \tag{2.1}$$

where $\mathbf{x}^{(0)} \in \mathbb{R}^{n_{\text{in}}}$ and $\hat{\mathbf{y}} \in \mathbb{R}^{n_{\text{out}}}$. In the context of computer vision, and in particular of image recognition and detection, that is the one we are interested in investigating, the input tensors in

[21]In the further discussion, we will concentrate on images as input for a CNN because this is the case we will then deal in detail and we are interested in studying. However, all the details we are going to provide are still true in other cases, e.g. with tensors with different shapes.

[22]There is not a convention in the number of blocks to use to construct a CNN. As discussed in Chapter 1 for the case of a general ANN, this depends on the problem in the exam. However, there exists some upper and lower bounds on the depth of an ANN that can help in designing its architecture, see Section 1.4.

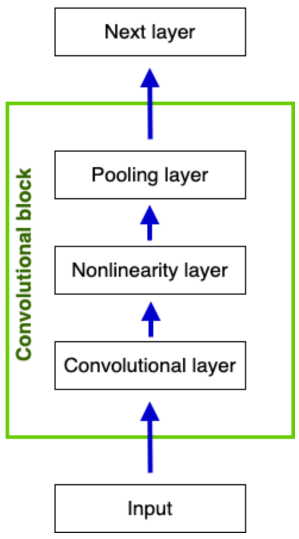


Figure 2.2: The components of a typical convolutional block: a convolutional layer followed by a nonlinear layer and a pooling layer.

exams are images and thus have a rectangular shape with a third dimension representing the color channels. For example, for black and white images there will be only 1 channel, while RGB images have 3 channels. Hence, we have that our input tensor has shape $(d_W^{\text{in}}, d_H^{\text{in}}, d_C^{\text{in}})$, where we are using the notation such that d_W and d_H represent the width and height of the tensor in exam, whilst d_C its number of channels. The output tensor is then the result of the application of the $\mathcal{CNN}$ function to the input image. In particular, since the last part of a CNN is represented by a fully connected FNN, the output is a vector of length n_{out}, that, if we refer to the case of image recognition, corresponds to the number of categories in which the dataset $\mathcal{D}$ is subdivided. Hence, $\hat{\mathbf{y}}$ is a vector whose components are the probability of belonging to the corresponding class.
Also in this case, $\mathcal{CNN}$ can be described as the composition of functions f_ℓ, for $1 \leq \ell \leq L+1$, where each of these represents a layer of the CNN, i.e. convolutional layer, nonlinear layer, pooling layer, fully connected layer:

$$\hat{\mathbf{y}} = \mathcal{CNN}(\mathbf{x}^{(0)}) = f_{L+1} \circ f_L \circ \cdots f_1(\mathbf{x}^{(0)}), \tag{2.2}$$

where L is the total number of hidden layers of the CNN and, as done in the previous chapter, we define $\mathbf{x}^{(\ell)}$ as the output of layer ℓ. If $\mathbf{x}^{(\ell)}$ is the output of a layer belonging to the feature learning part, it is a tensor with shape $(d_W^{(\ell)}, d_H^{(\ell)}, d_C^{(\ell)})$, where $d_C^{(\ell)}$ represents the channel depth, i.e. how many channels there are. Intuitively, in the case of image recognition, each channel can be thought to respond to some different set of features. To be precise, this does not correspond to reality since there is not a single channel for learning a particular feature but rather a direction in channel space for detecting it [334]. This is also justified by the fact that “feature detectors”, i.e. the filters of a convolutional layer, are learned during the optimization process and not chosen during the initialization of the CNN, as will be described in Section 2.2.1.
As discussed in [44, 24, 272, 177, 317], some deep neural networks, and in particular some CNN architectures such as ResNet [118, 156, 315], can also be represented as discretisations of dynamic systems of the form:

$$\dot{\mathbf{x}} = \sigma(W\mathbf{x} + \mathbf{b}), \tag{2.3}$$

where the parameters of the network correspond to the control variables. As presented in [24] and in [272], the training of the deep neural network can thus be formulated as an Optimal Control problem, where Pontryagin's Maximum Principle can be employed to learn the optimal parameters [184]. This is part of a novel approach in the deep learning context, characterized by a growing interest in understanding mathematically deep learning methods.

2.2.1 Convolutional Layer

The primary goal of the convolution step is to extract features from the input by passing many filters over it, each of which picks up a different signal or feature. CNNs derive their name from the *convolutional layer* and in particular from the *convolutional operation* [99, 173, 42]. The operation used in CNNs does not correspond precisely with the one defined in mathematics but can be derived from it [99], as we are going to prove.

Definition 2.2.1 (Convolution)**.** *Let f and g be two real-valued functions. The convolution of f and g, denoted with $f * g$, is defined by the following integral:*

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau, \tag{2.4}$$

where t is a variable, not necessarily representing time.

In the context of CNNs, we are working with discrete data, thus we can derive a discrete version of Equation (2.4):

$$(f * g)(t) = \sum_{\tau=-\infty}^{\infty} f(\tau)g(t-\tau). \tag{2.5}$$

Now, for a CNN the two arguments of convolution are represented by an input I, namely a multidimensional array of data, such as an image, and a *kernel* or *filter* K, a multidimensional array of parameters tuned during the learning process. In general, these multidimensional arrays are tensors, e.g. a two-dimensional image I and a two-dimensional kernel K. Therefore, usually convolution is applied over more than one axis at a time. In the example of a two-dimensional image I with a two-dimensional kernel K, the convolution of I and K is given by:

$$(I * K)(i,j) = \sum_{\alpha}\sum_{\beta} I(\alpha,\beta)K(i-\alpha, j-\beta), \tag{2.6}$$

where we have a summation over each axis of our objects. It is important to note that also in this case the commutative property, proper to classical mathematical convolution, holds and thus we can write:

$$(K * I)(i,j) = \sum_{\alpha}\sum_{\beta} I(i-\alpha, j-\beta)K(\alpha,\beta). \tag{2.7}$$

This is not the relation commonly used in convolutional neural networks and in their implementation [99, 42]. In fact, the convolution function described in Equation (2.7) is substituted with the *cross-correlation* function:

$$(I * K)(i,j) = \sum_{\alpha}\sum_{\beta} I(i+\alpha, j+\beta)K(\alpha,\beta), \tag{2.8}$$

that can be seen to be very close to Equation (2.7). Figure 2.3 provides a practical example of the convolutional operation for a 2D image I with a 2D kernel K.
We now need to derive an operational formula to compute convolutional operation for 3D tensors $\mathbf{x}$, with shape (d_W, d_H, d_C), as usually happens when working with images. In this case, the filter

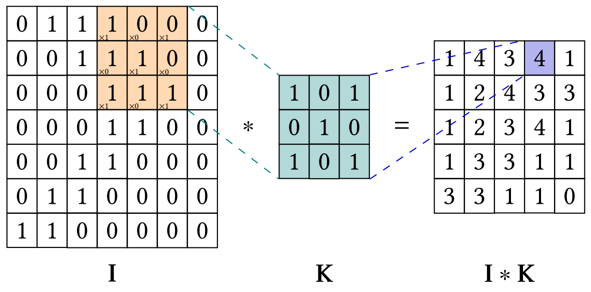


Figure 2.3: Schematic representation of the convolutional operation.

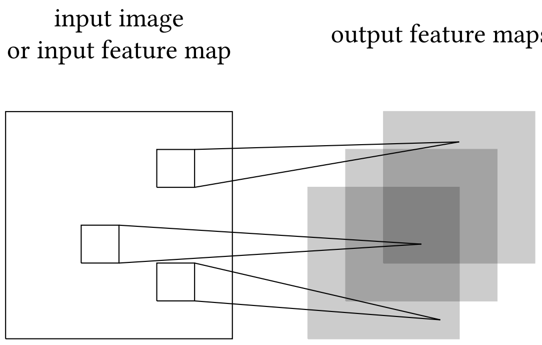


Figure 2.4: Illustration of a single convolutional layer.

K is a 4D tensor, with size $(d_W^K, d_H^K, d_C^{\text{in}}, d_C^{\text{out}})$, where d_W^K and d_H^K are the width and height of the kernel respectively; whilst d_C^{in} are the input channels, corresponding to those of $\mathbf{x}$, and d_C^{out} the number of filters n_K we are using, representing thus the number of channels in the output. Hence, let $x^{(\ell-1)} \in \mathbb{R}^{d_W^{(\ell-1)} \times d_H^{(\ell-1)} \times d_C^{(\ell-1)}}$ be the input for the ℓ-th layer of a CNN, that corresponds to a convolutional layer, and let $K^{(\ell)} \in \mathbb{R}^{d_W^{K,(\ell)} \times d_H^{K,(\ell)} \times d_C^{(\ell-1)} \times d_C^{(\ell)}}$ be the kernel for this convolutional layer. Starting from Equation (2.8), if we convolve $K^{(\ell)}$ across $x^{(\ell-1)}$ (see Figure 2.4), we can derive the following expression:

$$x_{i,j,k}^{(\ell)} = \sum_{\alpha=1}^{d_W^{K,(\ell)}} \sum_{\beta=1}^{d_H^{K,(\ell)}} \sum_{m=1}^{d_C^{(\ell-1)}} x_{i+\alpha-1,j+\beta-1,m}^{(\ell-1)} K_{\alpha,\beta,m,k}^{(\ell)}, \qquad \text{for } k = 1, \dots, d_C^{(\ell)}, \tag{2.9}$$

noticing how the convolution operation is applied separately for each filter $K_{:,:,:,k}^{(\ell)}$ characterizing the ℓ-th layer. The presence of -1 in the sub-indexes is due to the fact we are using 1 as the first index for our arrays.
Since $d_W^{K,(\ell)} < d_W^{(\ell-1)}$ and $d_H^{K,(\ell)} < d_H^{(\ell-1)}$, we have that the convolution kernel is overlapped on the tensor $x^{(\ell-1)}$. Therefore, if we apply convolution using Equation (2.9) this will be only applied locally on $x^{(\ell-1)}$ on the portion of it in which the filter window is placed. It is thus necessary to introduce a parameter that controls the spatial movements (horizontally and vertically) in all possible positions on $x^{(\ell-1)}$. Usually, the kernel overlap starts at the upper-left corner of the input tensor and slides by a certain number s of pixels at a time. This parameter s is called *stride* and is defining the number of pixels by which we move to the right and then down. The previous formula (2.9) can thus be

generalized by taking into account the stride [99]:

$$x^{(\ell)}_{i,j,k} = \sum_{\alpha=1}^{d^{K,(\ell)}_{\mathcal{W}}} \sum_{\beta=1}^{d^{K,(\ell)}_{H}} \sum_{m=1}^{d^{(\ell-1)}_{C}} x^{(\ell-1)}_{(i-1)\times s+\alpha,(j-1)\times s+\beta,m} K^{(\ell)}_{\alpha,\beta,m,k} + b^{(\ell)}, \qquad \text{for } k = 1,\dots,d^{(\ell)}_{C}, \tag{2.10}$$

where $\mathbf{b}^{(\ell)}$ represents the bias vector. In this case, as it usually happens, we have used the same stride s in both directions of motion, but in general, there could be a different value for the height and width components, $(s_{\mathcal{W}}, s_H)$.
As pointed out before, we can then note how the convolution operation is applied separately for each filter $K^{(\ell)}_{:,:,:,k}$ characterizing the ℓ-th layer. In this way, each filter is creating a different output, that corresponds to a different channel of $\mathbf{x}^{(\ell)}$. The output of a convolutional layer is thus generating a tensor with shape $(d^{(\ell)}_{\mathcal{W}}, d^{(\ell)}_{H}, d^{(\ell)}_{C})$, defined as [331, 334]:

$$\begin{aligned} d^{(\ell)}_{\mathcal{W}} &= \frac{d^{(\ell-1)}_{\mathcal{W}} - d^{K,(\ell)}_{\mathcal{W}}}{s_{\mathcal{W}}} + 1, \\ d^{(\ell)}_{H} &= \frac{d^{(\ell-1)}_{H} - d^{K,(\ell)}_{H}}{s_{H}} + 1, \\ d^{(\ell)}_{C} &= n_K, \end{aligned} \tag{2.11}$$

where each channel is also called *feature map* since it is supposed to contain some image features characteristic to the kernel [42]. In these relations we have used, as mentioned above, different values of the stride for height and width, i.e $s_{\mathcal{W}}$ and s_H. Then, by analyzing the role played by $s_{\mathcal{W}}$ and s_H in these expressions, we can deduce that they are parameters determining how much we are reducing the dimensionality, i.e. a larger stride leads to smaller feature maps.
From the previous discussion, we derive that the described convolution operation computed using a filter has the benefit of reducing the size of the input tensor, but is losing information about pixels on the perimeter of the image[23] [334]. Hence, if we want to increase the size of the output and save the information presented in the corners, we can use *padding* [331, 334]. This is a technique that consists in adding extra rows and columns on the outer dimension of the images, basically extending the area of an image in which a CNN processes. There is no rule defining the values to append, but a common choice is represented by *same* or *zero padding* [99, 47], i.e. zeros are added around the border of the input tensor (see Figure 2.5). However, adding padding to an image or in general to a feature map allows for a more accurate analysis of these tensors.
Figure 2.5 provides a practical example of the use of padding inside the convolutional step, where the dimensions of the output are bigger than the one of the input. We need now to derive some expressions to determine the final shape of a tensor $\mathbf{x}^{(\ell-1)}$, once we apply the convolutional step with padding. Let $\tilde{p}_H$ and $\tilde{p}_{\mathcal{W}}$ be the number of columns and rows we are adding, in this case, the output of convolution will have the following shape [331, 334]:

$$\begin{aligned} d^{(\ell)}_{\mathcal{W}} &= \frac{d^{(\ell-1)}_{\mathcal{W}} - d^{K,(\ell)}_{\mathcal{W}} + 2\tilde{p}_{\mathcal{W}}}{s_{\mathcal{W}}} + 1, \\ d^{(\ell)}_{H} &= \frac{d^{(\ell-1)}_{H} - d^{K,(\ell)}_{H} + 2\tilde{p}_{H}}{s_{H}} + 1, \\ d^{(\ell)}_{C} &= n_K. \end{aligned} \tag{2.12}$$

[23]If we think about a pixel on the border of the image, it is immediate to understand how it is considered only when the filter has a side on the perimeter, but this does not happen so often.

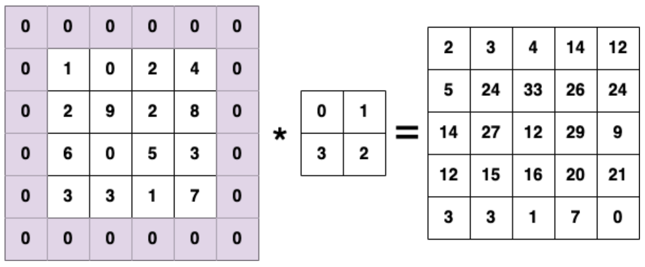


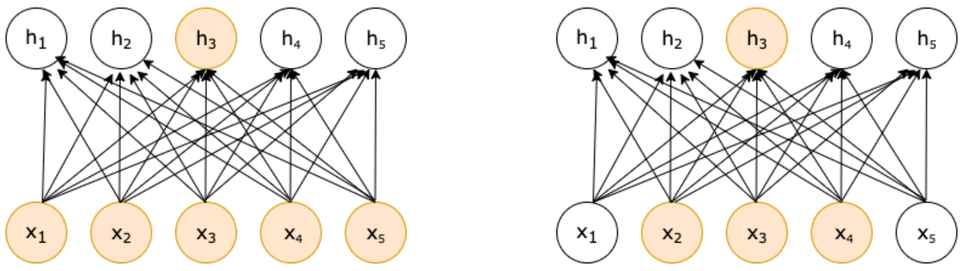
Figure 2.5: Example of application of the convolutional step using zero-padding with $\tilde{p}$=1 and stride s=1.

Also in this case we have introduced two different values for the padding for the height and width, $\tilde{p}_H$ and $\tilde{p}_W$, but usually we have the same value of padding $\tilde{p}$ in both directions of motion, i.e. $\tilde{p} = \tilde{p}_H = \tilde{p}_W$.
Convolution is characterized by the property of equivariance to translations [42, 99]. The following proposition states that if the input is affected by a translation, then also the output of the convolution is affected by the same translation.

Proposition 2.2.2 (Equivariance to translation [42]). *Let f_{conv} be a convolution operation, $\mathbf{x}$ be the input and $T_{\alpha,\beta}$ the translation operation in the direction of the vector (α, β), defined by $(T_{\alpha,\beta} \circ \mathbf{x})_{ij} = x_{i-\alpha,j-\beta}$. Convolution operation preserves translations, i.e.*

$$f_{conv}(T_{\alpha,\beta} \circ \mathbf{x}) = T_{\alpha,\beta} \circ f_{conv}(\mathbf{x}).$$

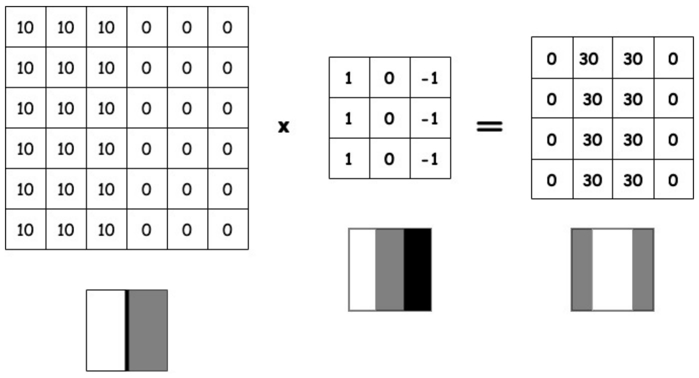
Hence, for example, considering time-series data, this property translates into producing a timeline showing when different features are present in the input. Similarly, with images, this means that convolution produces a 2D map regarding the appearence of features in the picture.
From the previous discussion, we have seen that in a convolutional layer the product between the input and the filter is not describing the interaction between each input unit and each output unit, but between only local portions. To be precise, in a traditional neural network the output is computed using the formula (1.7), where every output interacts with every input through the weight matrix, as can be seen in Figure 2.6 (a). A CNN is instead characterized by having *sparse interactions* (also referred to as *sparse connectivity* or *sparse weights*) [99], i.e. when computing the output only a portion of the input, the one identified by the filter, is involved in its calculation (see Figure 2.6 (b)). In this way, we need to store fewer parameters, and hence the number of operations to compute the output is also decreased. If we denote as n_ℓ and $n_{\ell+1}$ the number of neurons in layers ℓ and $\ell+1$, and we consider a kernel with $d^K_W = d^K_H$ with $d^K_W < n_\ell$, we are moving from $O(n_\ell \times n_{\ell+1})$ operations to $O(d^K_W \times n_{\ell+1})$.
Another relevant property of CNNs is *parameter sharing* [99]. In a generic ANN, we have that each element of the weight matrix W is used exactly once when computing the output of the corresponding layer. In a CNN, weights are *tied*, since the weight value applied to one input is linked to the value of a weight applied elsewhere. In fact, the same kernel is applied throughout the image, in every position. In this way, convolution weights are shared between different locations, leading to the necessity to learn a set of parameters for multiple locations instead of learning a separate set for every location. Also in this case, as consequence, we will have a reduction in the number of parameters to store, $O(d^K_W \times n_{\ell+1})$.
From the previous discussion, it is easy to understand which is the role played by filters in the convolutional step and in general in the feature learning part: extract relevant features from an input image, and in particular from the items depicted in it. To be precise, CNNs use the first



(a) Generic ANN (b) Convolutional Neural Network

Figure 2.6: Comparison between interactions in a generic ANN and in a CNN, that is characterized by sparse interactions.

convolutional layers to learn low-level features, i.e. minor details of the image such as lines or dots, while the later layers will learn to recognize high-level features, i.e. detect common objects and larger shapes in the picture. Figure 2.7 provides an example of a filter for vertical edge detection. Under each matrix (input, kernel, output) we have a representation of it in grayscale: values greater than 0 give rise to a white zone, less than 0 to a black stripe, and when there are 0 columns we have a gray zone. Hence, we want to detect the vertical line that separates the white and gray zone in the input. Using that particular kernel matrix, we obtain in output a matrix with a white zone in the middle, indicating the presence of the vertical line. In this case, it can be seen that the stripe that detects the vertical edge is wide, but this is connected to the fact we have small matrices. If we move to larger dimensions, this results in a more precise detection.



$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}$$

Figure 2.7: Example of a filter to detect vertical edges.

It is important to point out that, in the implementation of a CNN, we are not placing in each layer kernel matrices with some particular structure to detect a defined feature, but they are initialized randomly or using a particular distribution and then fine-tuned during backpropagation and the training phase. A brief insight on the topic can be found in Section 2.2.8.

2.2.2 Nonlinearity Layer

As described in Section 1.3 for a general ANN, after the propagation function, here represented by f_{conv}, a nonlinear function σ is applied. In literature is also not uncommon to see the convolutional layer combined with the nonlinearity layer, without having two separate layers [147].

Given the output of the convolutional layer $\mathbf{x}^{(\ell)}$ and let σ be the activation related to the nonlinear layer $\ell + 1$, we will obtain:

$$x^{(\ell+1)} = \sigma(x^{(\ell)}). \tag{2.13}$$

In Section 1.2, a list of common choices can be found. However, it has been demonstrated that ReLU [222], and in particular one of its variants *Parametric Rectified Linear Unit* (PReLU) [119] (see Figure 2.8), with $PReLU = \max\{\alpha x, x\}$, $\alpha \leq 1$, improve the performances of CNNs in the context of image recognition. Obviously, also in this case, the choice of the activation function is strictly correlated with the problem under consideration, so PReLu can be a good starting point, but other choices should also be taken into account when developing a CNN.

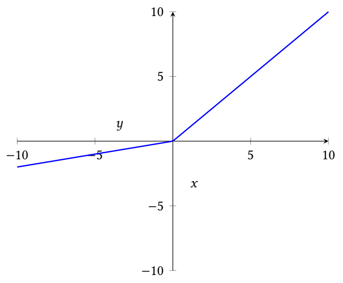


Figure 2.8: Plot of PReLu with $\alpha = 0.2$.

2.2.3 Pooling Layer

Pooling function is a machine learning technique used to reduce the dimensionality of each feature map by retaining the most important information and thus producing a summary of the input [99, 42, 331, 178]. To understand the idea of pooling, we can start by considering a partition of the domain of a function. Pooling determines the most representative value of the function on that set, which is then substituted in place of function values on that partition. There exists several ways to determine these most important values, such as maximum, average, and minimum, that we are now going to briefly describe. Following [42], we start by presenting the one-dimensional case for simplicity, and then we extend everything to a multidimensional case.

Max-pooling

Let $g : [\alpha, \beta] \longrightarrow \mathbb{R}$ be a continuous function defined on an interval $[\alpha, \beta]$, characterized by the following uniform partition of N points:

$$\alpha = q_0 < q_1 < \cdots < q_{N-1} < q_N = \beta.$$

The partition size defined as $\tilde{s} = \frac{\beta - \alpha}{N}$. If we denote with $M_i = \max_{[q_{i-1}, q_i]} g(q)$, the maximum in each sub-interval, we can define the simple function

$$\hat{S}_N(q) = \sum_{i=1}^{N} M_i \mathbb{1}_{[q_{i-1}, q_i)}(q). \tag{2.14}$$

The goal of max-pooling is thus approximating our function $g(q)$ with the simple function $\hat{S}_N(q)$ [341, 42].

Min-pooling

Similar to max-pooling, we can define a similar method by using the minimum value instead of the maximum. Hence, we can consider $m_i = \min_{[q_{i-1},q_i]} g(q)$, and the simple function

$$\hat{s}_N(q) = \sum_{i=1}^{N} m_i \mathbb{1}_{[q_{i-1},q_i)}(q). \tag{2.15}$$

Hence, also in this case min-pooling represents the process of approximating our function $g(q)$ by the simple function $\hat{s}_N(q)$ [42].

Average-pooling

For average-pooling, we are considering the notion of average for a continuous function g, with $g : [\alpha, \beta] \longrightarrow \mathbb{R}$, defined on an interval with the same uniform partition introduced above. The average of g on each sub-interval $[q_{i-1}, q_i]$ is given by $\bar{g}_i = \frac{1}{q_i - q_{i-1}} \int_{q_{i-1}}^{q_i} g(u)du$, thus we can introduce a function $\hat{A}_N$, describing the average-pooling, as:

$$\hat{A}_N(q) = \sum_{i=1}^{N} \bar{g}_i \mathbb{1}_{[q_{i-1},q_i)}(q), \tag{2.16}$$

with which we want to approximate $g(q)$ [42].

It is now needed a result that ensures that these pooling functions are good approximators for g, and is represented by the following theorem[24] [42].

Theorem 2.2.3 ([42])**.** *Let $g : [\alpha, \beta] \longrightarrow \mathbb{R}$ be a continuous function defined on the interval $[\alpha, \beta]$. Then, the three sequences $(\hat{S}_N)_N$, $(\hat{s}_N)_N$, and $(\hat{A}_N)_N$, whose terms are defined by Equation (2.14), Equation (2.15), and Equation (2.16) respectively, converge uniformly to g on the interval $[\alpha, \beta]$, as $N \longrightarrow \infty$. More precise, we have that $\forall \epsilon > 0$, $\exists\, \tilde{N} \geq 1$ such that*

$$|\hat{S}_N(q) - g(q)| < \epsilon, \quad |\hat{s}_N(q) - g(q)| < \epsilon, \quad |\hat{A}_N(q) - g(q)| < \epsilon, \qquad \forall q \in [\alpha, \beta],\ \forall N \geq \tilde{N}.$$

Pooling is then characterized also by the property of *local translation invariance* [42, 99], i.e. approximately invariant to small translations of the input. This means that, if the input is translated by a small amount, the values of the pooled outputs do not change, and thus we do care more about whether some features are present instead of where they are. There exists a result[25] which guarantees this property for pooling [42].

Proposition 2.2.4 (Translation invariance [42])**.** *Let $g : \mathbb{R} \longrightarrow \mathbb{R}$ be a continuous function, f_{pool} be a pooling function and T_γ the translation operator, defined as $(T_\gamma \circ g)(q) = g(q - \gamma)$. There exists a partition of $\mathbb{R}$ such that:*

$$f_{pool}(T_\gamma \circ g) = f_{pool}(g),$$

for any small enough value of γ.

[24] The interested reader can find the proof of the theorem in Chapter 15 of [42].

[25] Also in this case, we are not proving this result since is out of scope for this discussion. The complete proof of it can be found in Chapter 15 of [42].

Since convolution has the property of equivariance to translations (see Proposition 2.2.2), we derive that pooling and convolution are two operations that are compatible and can be applied together [42]. Hence, let $g : \mathbb{R}^2 \longrightarrow \mathbb{R}$, we have:

$$\begin{aligned} f_{\text{pool}} \circ f_{\text{conv}}(T_{\gamma,\delta} \circ g) &= f_{\text{pool}} \circ f_{\text{conv}}(g), \\ f_{\text{conv}} \circ f_{\text{pool}}(T_{\gamma,\delta} \circ g) &= f_{\text{conv}} \circ f_{\text{pool}}(g). \end{aligned} \tag{2.17}$$

We can now show how pooling can be extended to the multidimensional case [42]. Let now $g : A \longrightarrow \mathbb{R}$ be a continuous function defined on a compact $A \subset \mathbb{R}^N$, and consider a covering of A:

$$A = \bigcup_{i \in J} \bar{A}_i,$$

where A_i are open sets, $\bar{A}_i$ denotes the closure of A_i and J is an index set. Hence, the previous pooling functions will now be rewritten as:

$$\begin{aligned} \hat{S}_N(q) &= \sum_{i \in J} M_i \mathbb{1}_{A_i}(q) \qquad \text{with} \quad M_i = \max_{\bar{A}_i} g(q), \\ \hat{s}_N(q) &= \sum_{i \in J} m_i \mathbb{1}_{A_i}(q) \qquad \text{with} \quad m_i = \min_{\bar{A}_i} g(q), \\ \hat{A}_N(q) &= \sum_{i \in J} \bar{g}_i \mathbb{1}_{A_i}(q) \qquad \text{with} \quad \bar{g}_i = \frac{1}{\mu(A_i)} \int_{\bar{A}_i} g(q), \end{aligned} \tag{2.18}$$

where $\mu(A_i)$ represents the measure of the set A_i. Also in this case, there exists a result similar to Theorem 2.2.3 ensuring that we are taking good approximators for our function g [42].
In our test case, we are considering tensors of size $(d_{\mathcal{W}}, d_H, d_C)$, hence, fixed a channel, we have to consider a covering for each $d_{\mathcal{W}} \times d_H$ tensor. More technically speaking, let a pooling layer, with $f^{(\ell)}_{\text{pool}}$ the related pooling function, be the ℓ-th layer in a CNN and let $\mathbf{x}^{(\ell-1)} \in \mathbb{R}^{d^{(\ell-1)}_{\mathcal{W}} \times d^{(\ell-1)}_H \times d^{(\ell-1)}_C}$ be the input for that layer, that represents the feature maps of layer $\ell - 1$ (see Figure 2.9). Fixing a channel k, for $k = 1, \ldots, d^{(\ell-1)}_C$, let $\{A_i : i \in J\}$ be a cover for $\mathbf{x}^{(\ell-1)}_{::k} \in \mathbb{R}^{d^{(\ell-1)}_{\mathcal{W}} \times d^{(\ell-1)}_H}$, i.e. the k-th feature map of layer $\ell - 1$. Note that we are considering the same cover for each channel of $\mathbf{x}^{(\ell-1)}$, since each $\mathbf{x}^{(\ell-1)}_{::k}$, $k = 1 \ldots, d^{(\ell-1)}_C$, has the same dimensions $(d^{(\ell-1)}_{\mathcal{W}}, d^{(\ell-1)}_H)$ and at the end each feature map of layer ℓ should have the same size $(d^{(\ell)}_{\mathcal{W}}, d^{(\ell)}_H)$.
We now assume that each A_i is characterized by having the same rectangular shape of size $a_{\mathcal{W}} \times a_H$, and thus the same measure, $\mu(A_i) = a_H a_{\mathcal{W}}$, $\forall i \in J$. It is important to point out that we are not assuming that the elements of the covering are disjoint sets, therefore we need to define a "measure" of how distant two elements of the covering are in the two spatial directions, e.g. width and height. We want thus to quantize by how many pixels to move to find the next rectangle. This remembers the role of stride in convolution described in Section 2.2.1. For this reason, also in pooling, this parameter is called *stride*[26]Intuitively the idea of pooling is presented by defining a fixed window, similar to the filter of the convolutional layer, that is slid through the tensor with a step $s = (s_{\mathcal{W}}, s_H)$ [47].. Since we have that $d^{(\ell-1)}_H \neq d^{(\ell-1)}_{\mathcal{W}}$, we need to introduce a stride in each dimension $s_{\mathcal{W}}$ and s_H. There is not a default choice for these values, but it is easy to understand that they satisfy this property:

$$\begin{aligned} s_{\mathcal{W}} &\leq d^{(\ell-1)}_{\mathcal{W}} - a_{\mathcal{W}}, \\ s_H &\leq d^{(\ell-1)}_H - a_H. \end{aligned} \tag{2.19}$$

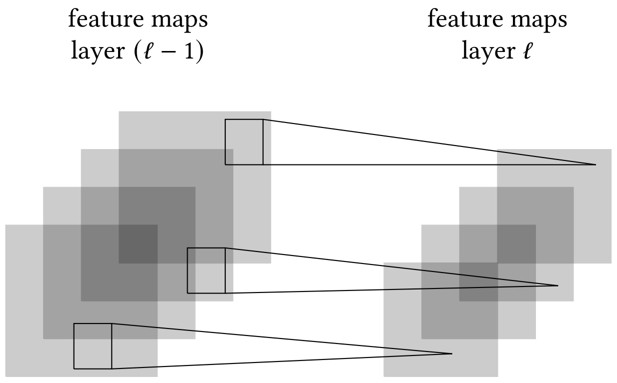


Figure 2.9: Illustration of a pooling layer.

Note that we are assuming that we have always the same stride in each dimension, i.e. the distance between a subset A_i and the next one A_j is always s_H and s_W in the directions of height and width respectively[27].
Setting a value for s_W and s_H, if we apply $f^{(\ell)}_{\text{pool}}$ to $\mathbf{x}^{(\ell-1)}_{::k}$, we gain for each subset A_i:

$$\mathbf{x}^{(\ell)}_{::k|_{A_i}} = f^{(\ell)}_{\text{pool}|_{A_i}}\left(\mathbf{x}^{(\ell-1)}_{::k|_{A_i}}\right), \qquad \forall k = 1, \dots, d_C^{(\ell-1)}. \tag{2.20}$$

where $f_{\text{pool}|_{A_i}}$ is the pooling function applied to the subset A_i, $\mathbf{x}^{(\ell-1)}_{::k|_{A_i}}$ is the k-th feature map of layer $\ell-1$ restricted to the subset A_i, and $\mathbf{x}^{(\ell)}_{::k|_{A_i}}$ represents the element of the k-th feature map in layer ℓ related to A_i. The final output of the pooling layer $\mathbf{x}^{(\ell)}$ can thus be computed as:

$$\mathbf{x}^{(\ell)} = f^{(\ell)}_{\text{pool}}\left(\mathbf{x}^{(\ell-1)}\right), \tag{2.21}$$

where $\mathbf{x}^{(\ell)}$ has size $(d_W^{(\ell)}, d_H^{(\ell)}, d_C^{(\ell)})$ determined by:

$$\begin{aligned} d_W^{(\ell)} &= \frac{d_W^{(\ell-1)} - a_W}{s_W} + 1, \\ d_H^{(\ell)} &= \frac{d_H^{(\ell-1)} - a_H}{s_H} + 1, \\ d_C^{(\ell)} &= d_C^{(\ell-1)}, \end{aligned} \tag{2.22}$$

Note that the function $f^{(\ell)}_{\text{pool}}$ is not applied globally to $\mathbf{x}^{(\ell-1)}$ but locally on each subset of the collection covering each feature map, as described by Equation (2.20). Furthermore, it is also a function acting independently on each channel of our tensor. Hence it is important to point out that pooling acts independently on each feature map and, since is not characterized by the presence of weights, it is not involved in backpropagation.
We should also consider a particular example of the one described, that is the case of an input tensor

[26]

[27] We can set an order in the collection of subsets covering $\mathbf{x}^{(\ell-1)}_{::k}$. First, we can define the distance between two subsets in a dimension —height or width—as the number of pixels that separate them. Then, given a subset A_i, the next subset in the direction of the height is the one with the smaller distance from it, and the same for the width.

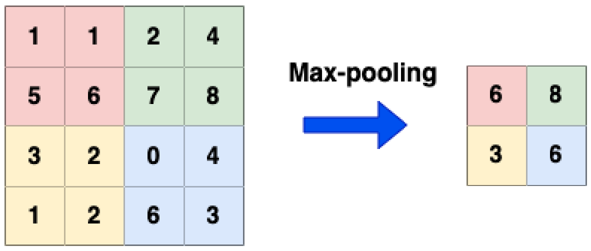


Figure 2.10: Example of application of max-pooling on a squared domain.

that has $d_H = d_{\mathcal{W}}$, as usually happens in the context of image recognition. Therefore, let $\mathbf{x}^{(\ell-1)}$ be a tensor of size $(d_{\mathcal{W}}^{(\ell-1)}, d_H^{(\ell-1)}, d_C^{(\ell-1)})$. In this case, we cover each feature map $\mathbf{x}_{::k}^{(\ell-1)}$ using a partition of it, i.e. with disjoint square open sets of shape (a_Q, a_Q):

$$\mathbf{x}_{::k}^{(\ell-1)} = \bigcup_{i\in J} \bar{A}_i \qquad \text{with } A_{i_1} \cap A_{i_2} = \emptyset, \ \ \forall i_1, i_2 \in J, \ \ i_1 \neq i_2. \tag{2.23}$$

Since we are assuming that each A_i has the same measure, is a square, and is disjoint from the others, we can conclude that the stride in each dimension is the same, i.e. $s_H = s_{\mathcal{W}}$. We can then compute the output of the pooling layer $\mathbf{x}^{(\ell)}$ using Equation (2.20) with the partition described before, as depicted in Figure 2.10, that is providing a practical example of max-pooling.

2.2.4 Fully Connected Layers

The last part of a CNN is responsible for the classification of the image features that have been captured during the feature learning part. Hence, the detected features are fed into a fully connected FNN that drives the final decision, by grouping the information obtained, analyzing them, and at the end providing a number identifying the class to which the input tensor belongs.
In practice, to be a suitable input for the fully connected FNN, the output of the feature learning part, that is a tensor, needs to be flattened into a column vector, as depicted in Figure 2.1. Let $\mathbf{x}^{(\ell-1)} \in \mathbb{R}^{d_{\mathcal{W}}^{(\ell-1)} \times d_H^{(\ell-1)} \times d_C^{(\ell-1)}}$ be the input for the classification part, we need to define a function f_{flatten}, that converts $\mathbf{x}^{(\ell-1)}$ into a 1-dimensional array. A way to do it is to define f_{flatten} as a linear mapping between two spaces, $\mathbb{R}^{d_{\mathcal{W}}^{(\ell-1)} \times d_H^{(\ell-1)} \times d_C^{(\ell-1)}}$ and $\mathbb{R}^{d_{\mathcal{W}}^{(\ell-1)} * d_H^{(\ell-1)} * d_C^{(\ell-1)}}$, as:

$$f_{\text{flatten}} : \mathbb{R}^{d_{\mathcal{W}}^{(\ell-1)} \times d_H^{(\ell-1)} \times d_C^{(\ell-1)}} \longrightarrow \mathbb{R}^{d_{\mathcal{W}}^{(\ell-1)} * d_H^{(\ell-1)} * d_C^{(\ell-1)}}, \tag{2.24}$$

that maps the (i, j, k) component of $\mathbf{x}^{(\ell-1)}$ in the m-th element of the output vector $\mathbf{x}_{\text{flat}}^{(\ell-1)}$, with $m = (j-1) * d_{\mathcal{W}}^{(\ell-1)} + i + (k-1) * (d_{\mathcal{W}}^{(\ell-1)} * d_H^{(\ell-1)})$.
Once we have flatten the input vector obtaining $\mathbf{x}_{\text{flat}}^{(\ell-1)}$, this becomes the input for a fully connected FNN, made of $\tilde{L}$ layers. The output of each layer is determined using Equation (1.12), where in this case ReLU is usually used in the intermediate layers, whereas, to obtain the final output $\hat{\mathbf{y}}$ of the CNN, a common choice for the activation function in the output layer is represented by the *softmax* or *normalized exponential function* [99, 26]. Formally, given a vector $\mathbf{u} \in \mathbb{R}^n$ the softmax function is defined as:

$$\hat{u}_i = \sigma(u_i) = \frac{exp(u_i)}{\sum_{j=1}^{n} exp(u_j)}, \qquad \text{for } i = 1, \ldots, n. \tag{2.25}$$

The application of this activation function is important because it normalizes all the vector components, that at the end will be in the interval $[0, 1]$. In this way, each element of $\hat{\mathbf{u}}$ is between 1 and 0, but also the entire vector sums to 1, providing thus a valid probability distribution.
Hence, following the notation introduced in Section 1.3, we have:

$$\hat{\mathbf{y}} = \sigma(\mathbf{h}^{(L+1)}) \tag{2.26}$$

where $\hat{\mathbf{y}}$ and $\mathbf{h}^{(L+1)}$ belong to $\mathbb{R}^{n_{\text{out}}}$. In this case, since we are solving a classification task, n_{out} corresponds to the number of classes that composes our dataset $\mathcal{D}$.

2.2.5 Equivalence Convolutional Layers and Fully Connected Layers

Intuitively, we can understand from the previous discussion that fully connected layers are a special case of convolutional layers. For example, given a convolutional layer and an input tensor of size $d_{\mathcal{W}}^{\text{in}} \times d_H^{\text{in}} \times d_C^{\text{in}}$, the output is obtained by applying a filter K, with shape $d_{\mathcal{W}}^K \times d_H^K \times d_C^{\text{in}} \times d_C^{\text{out}}$, to it, as described in Equation (2.10), and it will be a tensor of size $d_{\mathcal{W}}^{\text{out}} \times d_H^{\text{out}} \times d_C^{\text{out}}$. Hence, if we consider the case in which $d_C^{\text{in}} = d_C^{\text{out}} = 1$ and $d_{\mathcal{W}}^{\text{in}} = d_{\mathcal{W}}^{\text{out}} = 1$, we are exactly in the case of a fully connected layer. Therefore, to be more precise, we can establish an equivalence between these two types of layers, namely a way of converting one into the other [202, 238]. Hence, to convert the ℓ-th convolutional layer of a net into a fully connected layer, we need to transform its input $x^{(\ell-1)}$ into a 1-dimensional array and its weight matrix $W^{(\ell)}$, that corresponds to the kernel $K^{(\ell)}$ mentioned before, into a 2-dimensional array. For the input tensor $x^{(\ell-1)}$, this is simply done employing the flatten function f_{flatten}, defined in Equation (2.24):

$$\mathbf{x}_{\text{flat}}^{(\ell-1)} = f_{\text{flatten}}(\mathbf{x}^{(\ell-1)}), \qquad \text{with} \quad \mathbf{x}_{\text{flat}}^{(\ell-1)} \in \mathbb{R}^{d_{\mathcal{W}}^{(\ell-1)} * d_H^{(\ell-1)} * d_C^{(\ell-1)}}, \quad \mathbf{x}^{(\ell-1)} \in \mathbb{R}^{d_{\mathcal{W}}^{(\ell-1)} \times d_H^{(\ell-1)} \times d_C^{(\ell-1)}}, \tag{2.27}$$

that maps the (i, j, k) element of $\mathbf{x}^{(\ell-1)}$ into the m-th element of $\mathbf{x}_{\text{flat}}^{(\ell-1)}$, where $m = (j-1) * d_{\mathcal{W}}^{(\ell-1)} + i + (k-1) * (d_{\mathcal{W}}^{(\ell-1)} * d_H^{(\ell-1)})$. For the weight matrix, we can define a similar function f_{flatten}^W:

$$\begin{aligned} f_{\text{flatten}}^W : \mathbb{R}^{d_{\mathcal{W}}^{K,(\ell)} \times d_H^{K,(\ell)} \times d_C^{(\ell-1)} \times d_C^{(\ell)}} &\longrightarrow \mathbb{R}^{d_{\mathcal{W}}^{K,(\ell)} * d_H^{(\ell)} * d_C^{(\ell-1)} \times d_C^{(\ell)}} \\ W_{\text{flat}}^{(\ell)} &= f_{\text{flatten}}^W(W^{(\ell)}), \end{aligned} \tag{2.28}$$

that maps the (i, j, k, c) element of $W^{(\ell)}$ into the (m, c) element of its flatten version $W_{\text{flat}}^{(\ell)}$, with m defined as before.
For the other conversion, i.e. from a fully connected layer to a convolutional layer, we only need to use the inverse of the function introduced. For example, $f_{\text{tensor}} = f_{\text{flatten}}^{-1}$ is the function that maps a 1-dimensional array into a 3-dimensional tensor. In particular, it maps the m-th component of the 1-dimensional array into the $(m - (j-1) * d_{\mathcal{W}}^{(\ell-1)} - (k-1) * (d_{\mathcal{W}}^{(\ell-1)} * d_H^{(\ell-1)}), j, k)$ component of the input tensor for the ℓ-th convolutional layer.

2.2.6 Backpropagation

As described in Section 1.5 for FNNs, after we have initialized the network we need to tune all its parameters using Backward propagation to gain good performances. This can be applied also for CNN, since this method remains valid till we have a network without recurrent connections or loops, hence only with forward connections. As discussed in Section 1.5.2, an important role is played by the loss function. In the context of multi-class or binary classification problems, a common choice is represented by the Cross-Entropy Loss [99, 305], defined in Equation (1.18) for two-classes problems. Hence, a general expression is given by [329]:

$$\mathcal{L}(Y, \hat{Y}) = -\frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \sum_{j=1}^{n_{\text{out}}} y_j^i \log(\hat{y}_j^i), \tag{2.29}$$

where n_{out} is the number of categories in $\mathcal{D}_{\text{train}}$; $\mathbf{y}_j^i$ has value 0 or 1, indicating whether the class label j is the correct classification label or not for $\mathbf{y}^i$ and same for $\hat{\mathbf{y}}_j^i$, where 1 represents that the prediction for the i-th sample corresponds to the j-th category. Note that in this case we are summing also on the classes of $\mathcal{D}_{\text{train}}$ for each training example. It is also important to highlight that the vectors $\{\hat{\mathbf{y}}^i\}_{i=1}^{n_{\text{train}}}$ used in the computation of the loss should have been normalized between 0 and 1, i.e. should have been passed to the softmax function. Hence, if in the last layer the normalized exponential function is not present, to compute the loss function $\mathcal{L}$, we need to introduce this activation function [329], obtaining:

$$\mathcal{L}(Y, \hat{Y}) = -\frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \sum_{j=1}^{n_{\text{out}}} y_j^i \log\left(\frac{\exp(\hat{y}_j^i)}{\sum_{m=1}^{n_{\text{out}}} \exp(\hat{y}_m^i)}\right). \tag{2.30}$$

2.2.7 Testing Phase

As introduced in Section 1.5.9, the testing phase is a fundamental step in the model development process. In the classification context, the error is not measured through the typical loss function. Since we have to deal with n_{class} possible outputs, we have to introduce a performance metric able to take into account the correctness of the prediction for the CNN, namely the *confusion matrix* [112, 314]. It is constructed based on the expected and predicted outputs from our model, where on the rows there are the target classes in the data set and on the columns the corresponding predicted output classes. Based on the values contained in that matrix, we can define Average Precision (AP) [83, 265], a popular metric in measuring the accuracy of CNNs and object detectors. In order to understand its definition we need to introduce the notions of *precision* and *recall* [228, 102, 37, 191]. Precision is the fraction of relevant instances among the retrieved instances, hence it measures the ability of a system to present only relevant items. Recall, on the other hand, is the fraction of relevant instances that have been retrieved over the total amount of relevant instances, i.e. it measures the ability of a system to present all relevant items. In machine learning and statistics, we usually introduce the notions of true positive, true negative, false positive, and false negative [314, 112] to explain these concepts. *True positive* (TP) is a model outcome that correctly indicates the presence of a condition or characteristic, whereas *false positive* (FP) is a test result that wrongly indicates that a particular condition or attribute is present, hence providing an incorrect prediction. Similarly, we have a *true negative* (TN), when the output of the model correctly indicates the absence of a condition or characteristic. On the other hand, a *false negative* (FN) is a model outcome that incorrectly indicates the absence of a condition or an attribute. In this way, precision can be defined as the percentage of predictions that are correct and recall as the accuracy of the model in finding all the positives:

$$\begin{aligned} P = Precision &= \frac{TP}{TP + FP}, \\ R = Recall &= \frac{TP}{TP + FN}. \end{aligned} \tag{2.31}$$

Starting from this, we can express precision as a function of recall and plot his graph. A popular measure that represents a summary of this precision-recall curve, is the AP. In practice, it is defined as the average value of $P(R)$ over the entire interval from the recall value 0 to 1, i.e. the area under the precision-recall curve [292]:

$$AP = \int_0^1 P(R)dR, \qquad \text{with } 0 \leq AP \leq 1, \tag{2.32}$$

that in practice becomes the following finite sum:

$$AP = \sum_{i=1}^{n} P_i \Delta R_i = \sum_{i=1}^{n} P_i(R_i - R_{i-1}), \tag{2.33}$$

where ΔR_i represents the change in recall from $i-1$ and i and n is the number of predictions made. By taking the mean of the APs computed over all the classes in the dataset we obtain the metric commonly used to evaluate the performance of a CNN and an object detector, the Mean Average Precision (mAP) [83, 282]:

$$mAP = \frac{1}{n_{\text{class}}} \sum_{j=1}^{n_{\text{class}}} AP_j, \qquad 0 \leq mAP \leq 1, \tag{2.34}$$

where n_{class} is the total number of categories. Usually, the mAP value is then multiplied by 100 to obtain the percentage of accuracy for all the classes of the dataset.
Starting from recall, another performance metric can be defined: *top-k accuracy* [169, 193]. It computes the number of times the correct label is predicted among the top k labels, ranked by probability prediction scores. Therefore, the outputs of the softmax function, i.e. the last layer of a CNN, are ordered based on their confidence score, and only the top k values are taken into consideration. Top-k accuracy is thus evaluating how often the predicted class falls in the top k values. In particular, when $k = 1$, namely when we consider the top-1 accuracy, we are extracting only the maximum value out from the final outputs of the CNN and hence measuring the proportion of examples for which the predicted label matches the target one. Therefore, top-1 accuracy can provide a measure of how much the model can generalize what has been learned from the training dataset, whereas top-k accuracy gives an insight into the need for an additional fine-tuning of the CNN parameters.

2.2.8 CNN Initialization

In Section 1.6, we have discussed several parameter initialization strategies, that can be applied also to CNNs. Hence, after we have constructed a CNN we can initialize from scratch all the parameters using those methodologies. In this case, a common and highly effective approach is to use *pretrained weights* [47], obtaining thus a pretrained network. This is meaning that we are not using weights initialized following some rules and distributions, e.g. Kaiming He, random, but we are using the weights obtained by training previously that network on a large dataset, such as ImageNet[28] [167], typically solving a large-scale classification task. This technique that exploits those pretrained weights to initialize weights in a new classification task is called *Feature Learning* [232, 283, 99]. The idea behind this method is thus of employing what has been learned in one setting. i.e. to solve a task, to improve generalization in another setting, namely for a different task that is sharing some relevant features with the previous one. For example, in a classification environment, we may want to classify cats and dogs and then use this knowledge acquired to distinguish between ants and wasps. This is very effective since many visual categories share

[28]Refer also to Section 2.3.1 for a brief description of Imagenet.

low-level notions of edges and visual shapes, changes in lightning, the effect of geometric changes, and so on. For this reason, typically, only convolutional layers in the feature learning part use pretrained weights. The classifier part, as well as the entire network, is then trained starting from this initialization. The reason for this choice is that the representations learned by the convolutional layers are more generic and thus more reusable, whereas the ones learned by the classifier are necessarily specific and connected to the set of classes composing the dataset used for training. These layers indeed contain only information about the presence probability of a category in the picture, and not about the localization of it [47].
Pretraining [78] is an approach very similar to the one presented for transfer learning. The main difference lies in the fact that in the latter the network architectures must be transferred as well as the weights. Pretraining, on the other hand, enables the initialization of weights using big datasets, while still enabling flexibility in network architecture design. In this case, pretrained weights can thus be generated using a different architecture with respect to the one in which we are then loading them.

2.3 Image Recognition

In the previous section, we have introduced and explained the notion of CNNs, giving importance to the application in the context of *image recognition*, since is the one we are interested in. Image recognition is thus the task of identifying objects of interest within an image and recognizing which category they belong to. CNNs represent then the algorithms that solve this problem. In this section, we are going to review some of the datasets that are used in literature to test and train CNNs, but also some of the architectures that have been developed in this field.

2.3.1 Datasets

When choosing a dataset to train and test a CNN there are basically two choices that can be followed: create a custom dataset or use one freely available online, commonly used for machine learning research. In the first case, the experiments are strictly connected with a particular problem we may have, in the second one, the choice of using a dataset of this type could be connected with the will to compare the results of our network with the one already present in the literature.
For the benchmark databases, there are plenty of datasets that can be used for this purpose [329]. A first example is represented by **MNIST database of handwritten digits** [170, 175] from the National Institute of Standards and Technology (NIST). It contains $60,000$ training images and $10,000$ testing images, with shape $(28, 28, 1)$, created by re-mixing samples from NIST's original datasets [107]. In Figure 2.11 we can see some examples of samples from the MNIST. Other commonly used databases are represented by **CIFAR-10** and **CIFAR-100 datasets** [166], collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton, from the Canadian Institute For Advanced Research. They contain $60,000$ 32×32 color images divided in 10 and 100 different classes respectively, with $50,000$ training pictures and $10,000$ testing pictures. Figure 2.12 (a) provides an example of samples extracted from all the categories in CIFAR-10, whereas Figure 2.12 (b) represents another dataset commonly employed in the field of computer vision: **ImageNet** [68, 265]. It deals with the *object recognition task*, which, as pointed out in [265], encompasses not only the problem of image classification but also of object detection, which will be the topic of Section 2.4. The ImageNet dataset represents the backbone of the *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* and is organized accordingly to the WordNet hierarchy [214], a lexical database that establishes semantic relations between words in more than 200 languages. It is a very large visual database since it contains more than 14 million hand-annotated images divided into $20,000$ categories and for at least one million of them, object detection annotations are also provided.

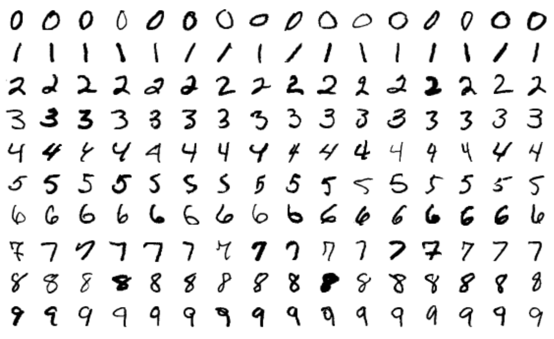
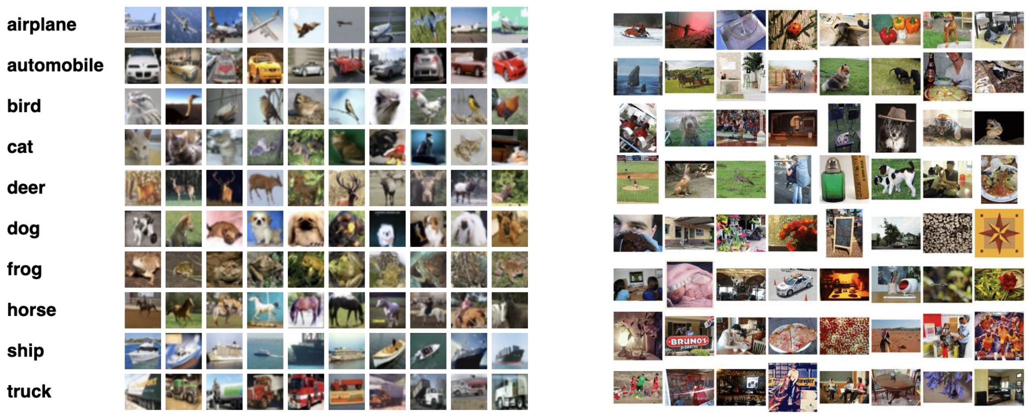


Figure 2.11: Samples from the MNIST Dataset. Image from Yann LeCun, Corinna Cortes, Christopher J.C. Burges, http://yann.lecun.com/exdb/mnist/.



(a) CIFAR-10 Dataset (b) ImageNet Dataset

Figure 2.12: CIFAR-10 and Imagenet Datasets. On the left, (a) presents a complete list of all the classes that compose CIFAR-10 and an example of 10 random pictures for each category. (Source: https://www.cs.toronto.edu/ kriz/cifar.html.) On the right, (b) depicts some random samples from the validation set of the ImageNet Dataset. (Source: original paper [265].)

In both cases, whether we are using custom data or a benchmark dataset, data should be formatted in a proper way [47, 331] to be fed into a CNN. Therefore, after data collection, we need to preprocess images by converting them into a grid of pixels (usually with the RGB format) and then into floating-point tensors. It is also common to normalize data following a certain mean $\hat{\mu}$ and standard deviation $\hat{\sigma}$, using for example the *standard score normalization* or *z-normalization* [331]. Hence, given an input image $\mathbf{x}^{(0)}$, each value is replaced with the z-score:

$$\tilde{x}_i^{(0)} = \frac{x_i^{(0)} - \hat{\mu}}{\hat{\sigma}},$$

leading to a new tensor $\tilde{\mathbf{x}}^{(0)}$ with zero mean and standard deviation 1.
It is widely accepted that bigger datasets result in better Deep Learning models [295, 109, 285]. When using a custom dataset, we may however have few samples to learn from, rendering it ineffective to train a model that can generalize to new data [47]. In many scenarios, such as medicine, it is tough

and expensive to obtain data and then label it. Therefore, it becomes increasingly important to augment training data artificially via several random transformations that yield believable-looking images. *Data augmentation* [203, 47, 285, 99] represents thus this approach of generating more training data from existing training samples. There exist several techniques that can be employed for this purpose [285], e.g. geometric transformations, color space transformations, kernel filters, random erasing, GAN-based augmentation.
The class of augmentations based on geometric transformations is characterized by their ease of implementation. They include techniques such as flipping, rotation, cropping, and translations, that can avoid positional bias in the data. Color space and kernel filter transformations, such as color augmentations, isolation of one color channel, change of the brightness of the image, use of grayscale images, blur and noise addition, can then help CNNs to learn more robust features close to real-life examples. All these methods start thus from the color histogram related to a picture and manipulate it by applying filters changing the color space characteristics of the image.
Random erasing [340] is an interesting Data Augmentation technique developed to face the problem of occlusion, i.e. when some parts of the object are unclear. Hence, a picture is manipulated to guarantee that a network pays attention to the entire image, rather than just a subset of it. Technically speaking, it is performed by randomly selecting a patch of an image and masking it with either value equal to 0 or 255, mean pixel values, or random values.
New data can also be created using another neural network. This is the approach carried out by Generative Modeling and in particular by Generative Adversarial Networks (GANs) [100, 34, 99]. The GAN model architecture involves two sub-models, that can be multilayer perceptron networks or also CNNs: a generator model for generating new plausible examples for the problem domain and a discriminator model for classifying whether generated examples are real, from the domain, or fake, generated by the generator model. The discriminator model is particularly useful during the training process to understand how close are generated picture to the real one. Hence, after the training process, it is discarded, since only the generator is then used to create new samples. In this way, GANs are thus able to unlock additional information from a dataset by constructing artificial instances that retain similar characteristics to the original set.

2.3.2 CNN Architectures for Image Recognition

Over the last 30 years, several CNN architectures have been presented [5, 6, 156, 315] to tackle the problem of image recognition. It is widely recognized that the first CNN that has been proposed was *LeNet*, invented by LeCun, Boser, Denker, Henderson, Howard, Hubbard, and Jackel, in 1989 [170] and further made popular with the work [175]. The success of LeNet opened the way to the birth of great interest in CNNs and their potentiality. In particular, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [265] attracted a large number of researchers aiming to solve the classification task connected with the benchmark dataset Imagenet, leading to a development of a great number of CNNs. In this section, we will thus briefly discuss some of these models [156, 315], starting from the pioneer one: LeNet.

LeNet LeNet-5 [170, 171, 175, 174] is known for its ability to classify and recognize digits, handling a variety of different problems connected with digits, such as variances in position and scale, rotation and squeezing. The introduction of that network is strictly connected with that of the MNIST database [171], which was the standard benchmark in the digit recognition field. Although the development of LeNet-5 means the emergence of CNNs, this model was not so popular in the 1990s because of the lack of hardware equipment, especially GPUs [334].
As depicted in Figure 2.13, it consists of two pairs of Convolutional and Pooling Layers, followed by two fully connected layers and a Gaussian connection layer for classification. A Gaussian connection

is simply a fully connected layer, characterized by using a Euclidean Radial basis function (RBF) as an activation function to provide the final output of the network. The structure of LeNet was able to overcome the main limitations of traditional multilayered fully connected neural networks, that considered each image pixel as a separate input, in which applying some transformation [87]. LeNet, instead, was the first net able to learn features from raw pixels automatically by considering them correlated to the other neighboring pixels. The introduction of convolution with learnable parameters represents also an effective way to extract object features at multiple locations with few parameters.

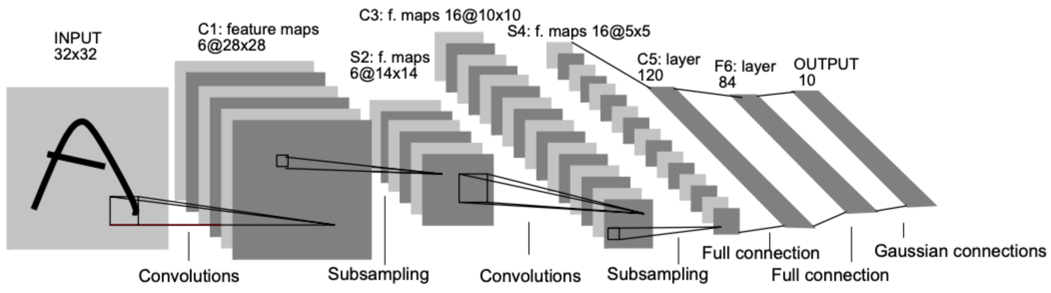


Figure 2.13: Architecture of LeNet-5, original picture taken from [171].

AlexNet AlexNet [167, 156] was designed by Alex Krizhevsky in collaboration with Ilya Sutskever and Geoffrey Hinton in 2012. While LeNet starts the era of CNNs for digit recognition, AlexNet showed groundbreaking results for image classification and recognition tasks connected with a huge dataset as ImageNet [68, 265]. These impressive performances on this database have created a growing interest in using ImageNet for testing new architectures, leading then also to a competition, ILSVRC, for developing the best CNNs for the task of image recognition trained and tested on ImageNet
Figure 2.14 illustrates the basic design of AlexNet. It is characterized by eight layers, where the first five are convolutional layers, with some of them followed by max-pooling layers, and the last three are fully connected layers. As an activation function, ReLU is employed, due to its non-saturating property that improves the convergence rate by alleviating the problem of vanishing gradients [222, 132]. The increase in depth and the use of several parameter optimization strategies then improve the generalization for different resolutions of images and its learning ability.

VGGNet Simplicity is a world that characterizes the CNN introduced by Simonyan and Zisserman in 2014, called *VGG* [289], from the name of their research group Visual Geometry Group. Even if it was not the winner of the ImageNet competition of that year, that was GoogleNet, it has introduced important features to the architecture of a CNN, that opened the path for the creation of ResNet, winner of the ILSVRC 2015. VGG explores deeper structure (11, 13, 16, or 19 weight layers[29]) with simpler layers than the CNNs developed up to that point. Zeiler and Fergus in 2013 suggested that small size filters can improve the performance of CNNs, proposing a CNN called ZfNet [333]. Hence, based on these findings, VGG replaced the 11×11 and 5×5 filters with a 3×3 convolutional layer followed by a 2×2 pooling layer [315, 156]. The use of smaller size filters reduces the number of

[29]The total number of layers having tunable parameters is 11, 13, 16, or 19 of which the first are convolutional layers and the last 3 are fully connected layers. Max-pooling layers or in general pooling layers do not have weights to tune and thus are not used in the backpropagation phase, as discussed in Section 2.2.3.

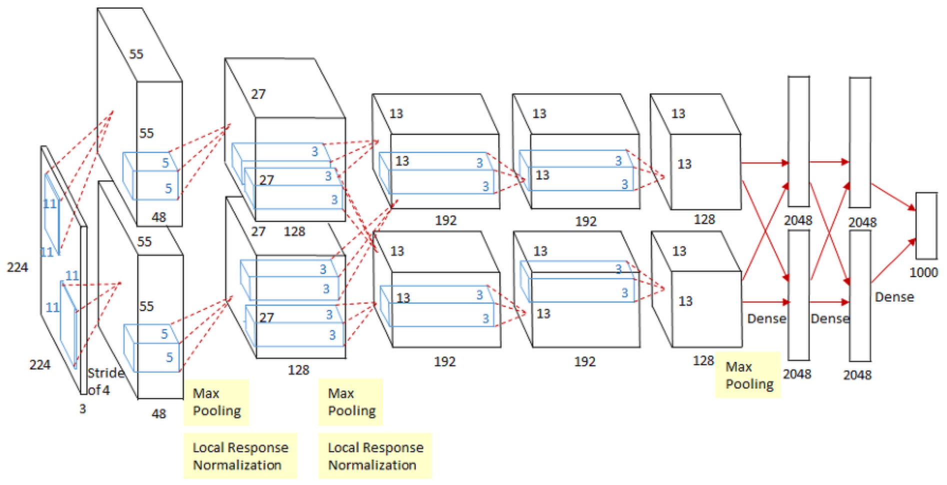


Figure 2.14: Architecture of AlexNet, original picture taken from [167].

weights, thus providing an additional benefit of low computational complexity. Despite this, VGG is characterized by having a huge number of parameters, e.g. 138 million in the case of VGG-19, representing its main limitation by making it computationally expensive and difficult to deploy on low resource systems.
As described in Figure 2.15, several configurations for VGGNet were presented in [289], that differ from each other in depth, i.e. from the number of weights layers. The main structure of VGGNets is hence characterized by having the feature learning part composed of a certain number of convolutional layers coupled with ReLU function and followed by a max-pooling layer and at the end 3 fully-connected layers with softmax as the output function.

ResNet *Residual Net* (ResNet) [118, 156, 315] explores the idea introduced by VGG: use a deeper structure with simple layers. Increasing only the number of layers of a CNN, gaining thus a deep network, is not sufficient, because it leads to worse results for the training and testing phases. In fact, using deeper plain networks increases the occurrence of the problem of vanishing/exploding gradients. The main breakthrough ResNet introduces to solve this kind of problem lies indeed in *Residual Blocks* (see Figure 2.16), which is composed of a certain number of convolutional layers and a skip/shortcut connection from the input to the output of the block. The idea behind this method is that the input $\mathbf{x}$ of a certain layer can be passed to the component some layers later either following the traditional path which involves convolutional layers and ReLU transform succession or going through an express way that directly passes $\mathbf{x}$ there. The final outcome $\mathbf{x}^{\text{out}}$ is thus determined by [120]:

$$\mathbf{x}^{\text{out}} = \mathcal{F}(\mathbf{x}) + \mathbf{x}, \tag{2.35}$$

where $\mathcal{F}$ represents the composition of the several layers of the Residual Block. It is easy to notice that, if we are facing the problem of vanishing gradients in the convolutional layers, we now always still have the identity $\mathbf{x}$ to transfer back to earlier layers.
Figure 2.16 (b) proposes a variant of the residual block, *Bottleneck Residual Block*, that utilises 1×1 convolutions to create a bottleneck. Hence, a 1×1 convolutional layer is added to the start and end of the block. The use of a bottleneck is thus increasing the number of layers but on the other hand, it reduces the number of parameters and matrix multiplications, while not degrading the performance of the network [120].

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 2.15: Different VGG configurations as presented in the original paper [289]. The depth of the nets increases as we move from left to right, i.e. from (A) to (E), since more layers (the bold ones) are added. The ReLU function is not mentioned for brevity but is added after each convolutional layer.

A comparison between several ResNet architectures, as proposed in the original paper [118], is described in Figure 2.17. As can be seen, the basic structure is characterized by the feature learning part, composed of a convolutional layer, a max-pooling layer, and then four residual or bottleneck blocks, followed by the classification part with an average pooling and a fully-connected layer. It is also useful to point out that the difference between the first two networks in Figure 2.17 and the others is the presence of the bottleneck blocks, which leads to deeper networks.

GoogleNet *GoogleNet* [299], known also as *Inception-V1*, was the winner of the ILSVRC competition in 2014. Figure 2.18 provides a schematic representation of its architecture, as presented in [299]. It can be noticed that conventional convolutional layers are substituted by *inception blocks*. Inception module, depicted in Figure 2.19, performs max-pooling and convolutions on the input arriving from the previous layer, with 3 different sizes of kernels or filters, specifically 1×1, 3×3, and 5×5. Hence, it encapsulates filters of different sizes in order to capture spatial information at different scales, i.e. at fine and coarse grain levels. The basic idea under the inception block can be summarized as split, transform, and merge, three operations that help in learning diverse types of variations present in the same category of images having different resolutions.
Then, to regulate the computations connected with bigger kernels, a bottleneck layer of 1×1 convolutional filter is also added before employing large-size kernels or after the pooling layer. Even if this lowers the number of parameters, it also drastically reduces the feature space in the next layer, leading sometimes to the loss of useful information [156]. The connection's density was also reduced by using a global average pooling layer, that takes a feature map of 7×7 and averages it to 1×1, decreasing the number of associated trainable parameters to 0. In this way, these

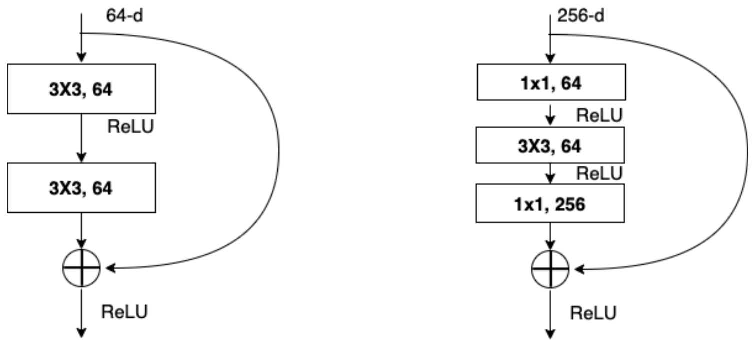
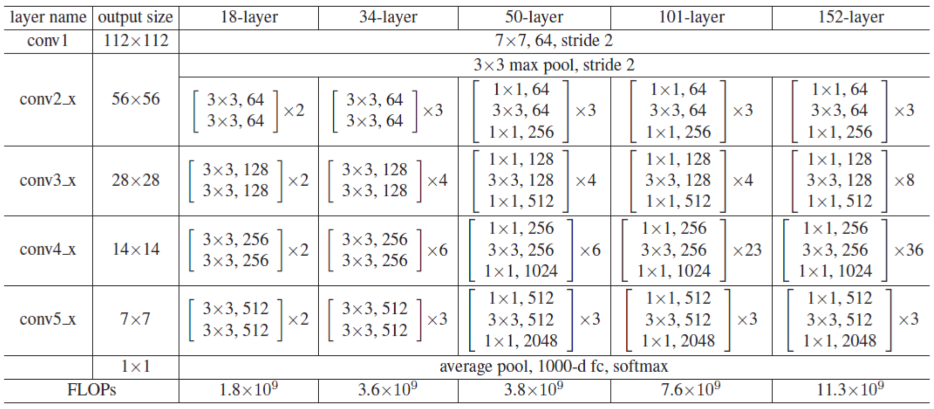


Figure 2.16: The basic residual block (left) and the proposed bottleneck design (right). Source: original paper [118].

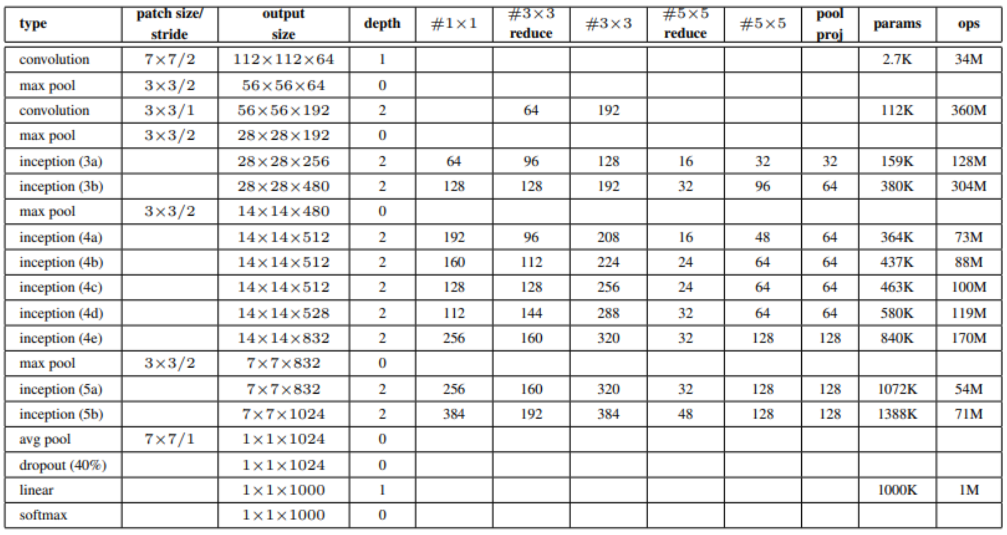


layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 2.17: The overall architecture for all ResNets, as presented in the original paper [118].

parameter tunings are causing a significant decrease in the number of weights from 138 million (as for VGG-19) to 4 million. Then, at the end of the architecture, there is the classification part, made of two fully-connected layers followed by the softmax classification function.
To regularise and prevent overfitting, the authors have also added an additional component, known as an *Auxiliary Classifier*. They essentially applied softmax to the outputs of two of the inception modules and computed an auxiliary loss over the same labels. As can be deduced, auxiliary classifiers are only utilized during training and then removed during inference. Hence, for training purposes, the total loss function is a weighted sum of the auxiliary loss and the real loss.

Inception Network In the improvement and development of CNNs, the inception network represents an important milestone. Compared to traditional CNNs, which just stacked convolution layers deeper and deeper, its complex and heavily engineered structure was able to push performance, both in terms of speed and accuracy. Several versions were created from the first one, GoogleNet: Inception-V2 and -V3 [301], Inception-V4 and Inception-ResNet [298], where each version is an iterative improvement over the previous one. The presence of large filters, as 5×5 in the classical inception module, causes the input dimension to decrease by a large margin, leading to a possible loss in accuracy. For this reason, in *Inception-V2* [301], the 5×5 convolution is replaced with 3×3 kernels, as described in Figure 2.20 (a). Another change introduced is factorization: the conversion



type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Figure 2.18: GoogleNet architecture [299].

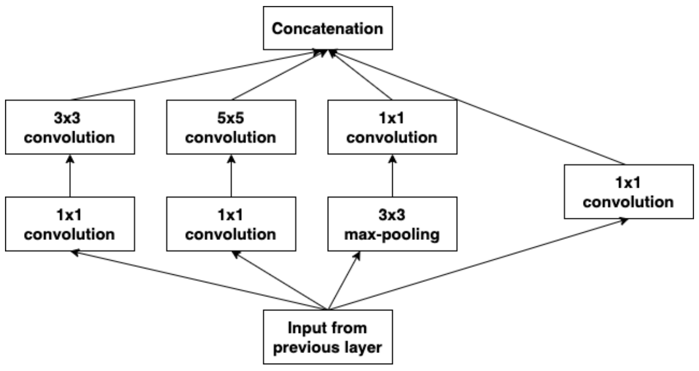
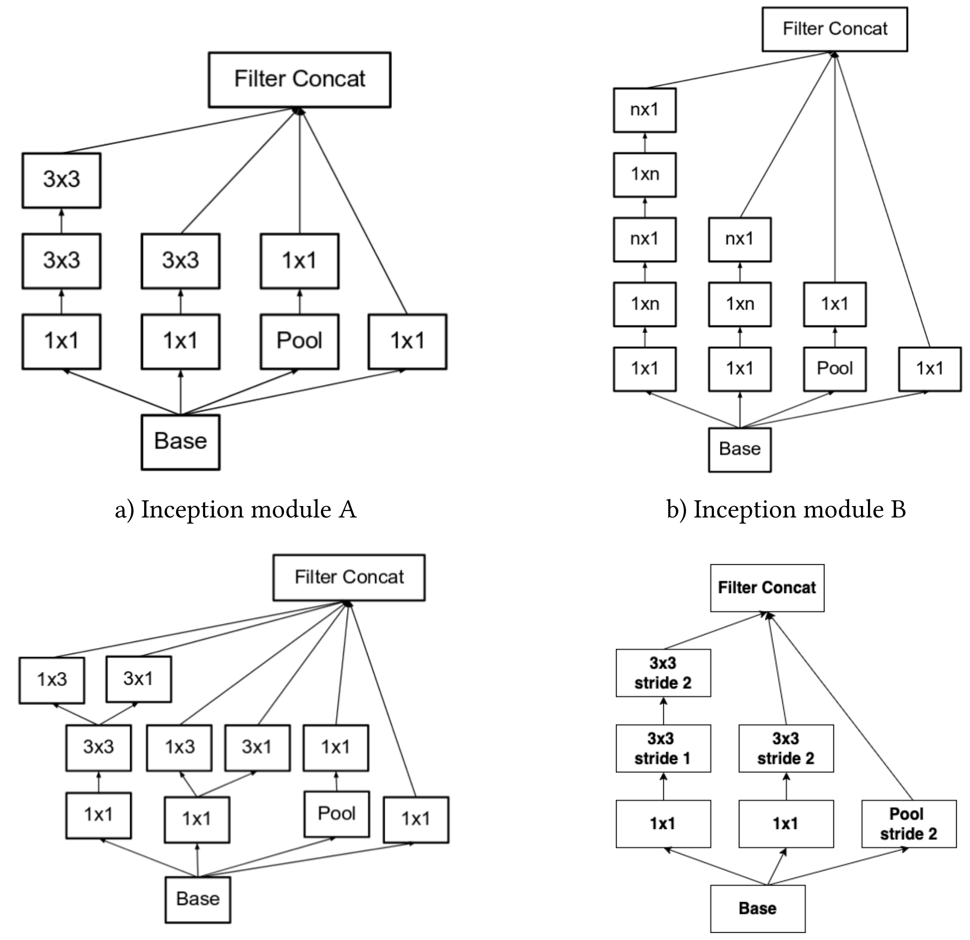


Figure 2.19: Basic architecture of the inception block [156].

of 3 × 3 filters into an asymmetric convolution 1 × 3 followed by a 3 × 1 convolution, that has been proven to reduce computational complexity of 33% [301] (see Figure 2.20 (b)). Then, in order to avoid too deeper modules with an excessive reduction in dimensions and thus loss of information, the filter banks were expanded, making the block wider instead of deeper (see Figure 2.20 (c)). The above three principles were thus used to build three different types of inception modules: A, B, and C, described in Figure 2.20. Hence, the overall schema for Inception-V2 is summarized in Figure 2.21, where the first layers of the net are referred to in [301] as the stem of the architecture, composed of three convolutional layers, a max-pooling layer, and other three convolutional layers.
Inception-V3 [301] represents an evolution of the previous version: it incorporates all of the upgrades of Inception-V2 but with some modifications. RMSprop [309, 42] was introduced as an optimizer, whereas, in the auxiliary classifiers, batch normalization or dropout is used to let the auxiliary loss have a bigger contribution.
Inception-V4 was introduced in combination with *Inception-ResNet* in 2016 [298]. It distinguishes itself from Inception-V3 for a more uniform structure, characterized by many different inception



a) Inception module A

b) Inception module B

c) Inception module C

d) Reduction Block

Figure 2.20: Modifications of the inception module in Inception-V2 and Inception-V3 (a, b and c) and structure of the reduction block (d), as presented in [301, 298].

modules[30], indicated in Figure 2.22 (a) with A, B and C. Each of these inception blocks combines the same features presented in the previous version (Figure 2.20) using, in each of it, different size filters and sequences of layers. At the beginning, before these blocks, there is also an initial set of layers, called the *stem layer*, with a similar structure to the one in Inception-V2. Inception-V4 introduced then specialized *Reduction Blocks* for the purpose of efficient size reduction of the feature maps by changing their width and height. In this way, the grid size is reduced efficiently whilst the activation dimension of the network filters is expanded. As depicted in Figure 2.20 (d), these modules are composed of two parallel blocks of convolution and pooling later concatenated.
In [298], it is explored also the possibility of using residual networks on the Inception model by proposing two sub-versions of Inception ResNet: Inception-ResNet V1 and Inception-ResNet V2. Hence, these CNNs are characterized by the presence of an Inception-ResNet module, compu-

[30]Since the purpose of this Section is to offer an overview of commonly used CNN architectures, we are not going to provide all the details about the structure of these inception modules, the stem, and the reduction layers, both for Inception-V4 and for Inception-ResNet. The interesting reader can refer to the original paper [298].

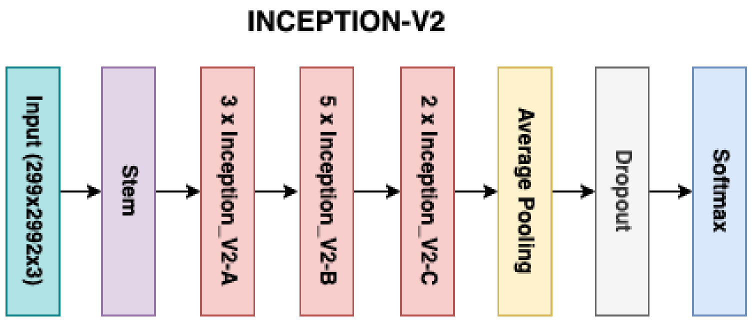


Figure 2.21: Overall schema for Inception-V2, as described in [298].

tationally less expensive than the original Inception blocks used in Inception V4 [298]. In each Inception-ResNet block, the pooling operation is replaced in favor of the residual connections (the identity mapping). Then, to ensure that the residual addition operation is feasible and thus that the input and output after convolution have the same dimensions, a 1×1 convolution is inserted to match the depth after the original convolution block operation. Pooling operations are not deleted at all from the CNN: they can still be found in the reduction blocks, that are similar to those introduced for Inception-V4. These two CNNs present a similar structure apart from the stem block: in Inception-ResNet V2 is the same as Inception-V4, while the one in Inception-ResNet V1 is close to the one in Inception-V3.

MobileNet *MobileNet* [138, 271] is TensorFlows first mobile computer vision model open-sourced by Google, designed specifically to be used in mobile applications. It is characterized by the use of *depthwise separable convolutions*, which significantly reduces the number of parameters resulting in lightweight deep neural networks. A depthwise separable convolution is made from two operations: a depthwise convolution, and a pointwise convolution. This type of convolution originated from the idea of spatial separable convolution, namely a filters depth and spatial dimension can be separated. Hence, given a 2-D filter matrix, it can be rewritten as the scalar product between two vectors:

$$K = \mathbf{u}_1 \cdot \mathbf{u}_2 = \mathbf{u}_1 \mathbf{u}_2^T, \qquad \text{with } K \in \mathbb{R}^{d^K_W \times d^K_H},\ \mathbf{u}_1 \in \mathbb{R}^{d^K_W \times 1},\ \mathbf{u}_2 \in \mathbb{R}^{d^K_H \times 1}. \tag{2.36}$$

It is easy to understand that this operation is reducing the number of parameters needed: we go from $d^K_W * d^K_H$ to $d^K_W + d^K_H$. However, this operation can be employed only if the kernel matrix can be separated into two smaller kernels, applied sequentially, but this is not always true. For this reason, depthwise separable convolutions were introduced to deal also with filters that cannot be factorized and to take into account also their depth, i.e. the number of channels. Similar to the spatial separable convolution, a depthwise separable convolution splits a kernel into two separate kernels that do two convolutions: the depthwise convolution and the pointwise convolution. In the first part, depthwise convolution, we apply to the input image $\mathbf{x}$, with shape $(d^{\text{in}}_W, d^{\text{in}}_H, d^{\text{in}}_C)$, a convolution operation without changing its depth. Hence, we are applying to the input d^{in}_C filters of size $d^K_W \times d^K_H \times 1$, where d^{in}_C represents the number of channels in the input and in this case also in the output. After this, there is the pointwise convolution, which aims at increasing the number of channels in the previous feature maps. It is so named because it uses a 1×1 kernel, or a kernel that iterates through every single point. Therefore, here we are using d^{out}_C kernel matrices, with shape $1 \times 1 \times d^{\text{in}}_C$, to get the final output.
We can understand the benefit of this factorization by calculating the number of multiplications needed for the original and this new convolution. In the standard case, given a filter of dimension

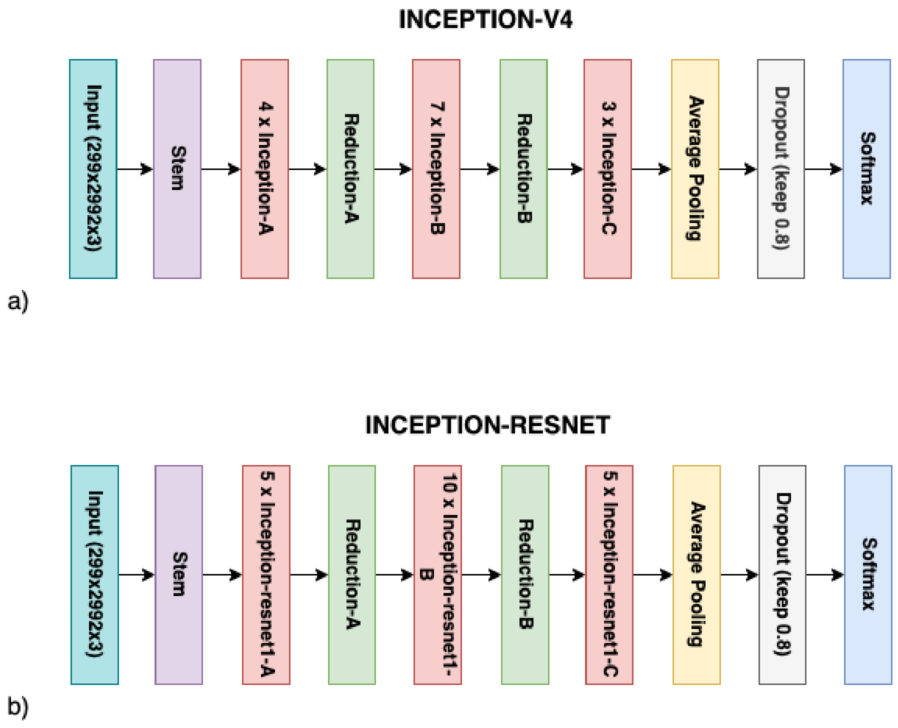


Figure 2.22: Overall schema for Inception-V4 (a) and for Inception-ResNet (b), as presented in [298].

$d^K_W \times d^K_H \times d^{\text{in}}_C \times d^{\text{out}}_C$, it moves $d^{\text{out}}_W * d^{\text{out}}_H$ times through the input. Hence, the computational cost of this operation is:

$$d^K_W * d^K_H * d^{\text{in}}_C * d^{\text{out}}_C * d^{\text{out}}_W * d^{\text{out}}_H,$$

where d^{out}_W and d^{out}_H are the width and height of the output feature maps respectively. Whereas, in the same setting, with above operations the computational cost is:

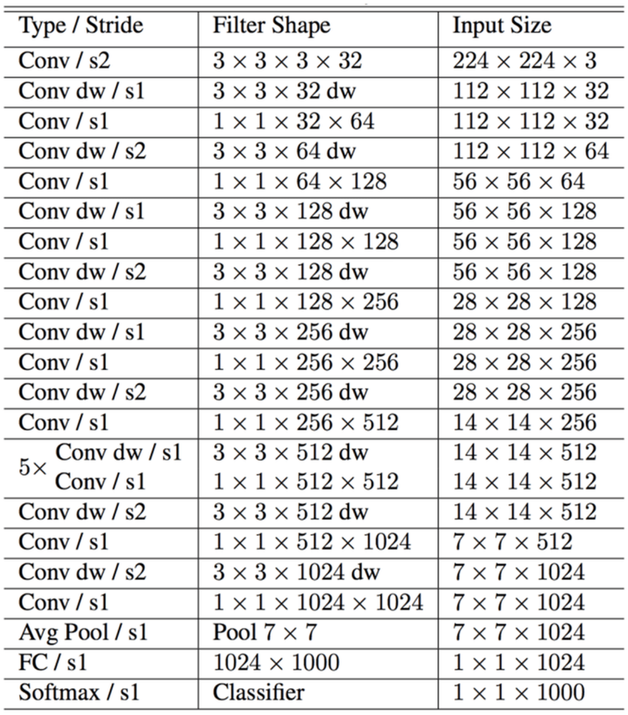
$$d^K_W * d^K_H * d^{\text{in}}_C * 1 * d^{\text{out}}_W * d^{\text{out}}_H + 1 * 1 * d^{\text{in}}_C * d^{\text{out}}_C * d^{\text{out}}_W * d^{\text{out}}_H,$$

where the first addendum takes care of the multiplication in the depthwise convolution, whilst the second of the pointwise convolution. The main difference between these two convolutions lies thus in the fact that the classical one is transforming the input d^{out}_C times, whereas depthwise separable is transforming it once in the depthwise step and elongating it to d^{out}_C channels in the pointwise part, leading to save computational power. The computation reduction is indeed:

$$\frac{d^K_W * d^K_H * d^{\text{in}}_C * 1 * d^{\text{out}}_W * d^{\text{out}}_H + 1 * 1 * d^{\text{in}}_C * d^{\text{out}}_C * d^{\text{out}}_W * d^{\text{out}}_H}{d^K_W * d^K_H * d^{\text{in}}_C * d^{\text{out}}_C * d^{\text{out}}_W * d^{\text{out}}_H} = \frac{1}{d^{\text{out}}_C} + \frac{1}{d^K_W * d^K_H}.$$

Figure 2.23 summarizes the overall architecture of MobileNet, where *Conv dw* indicates the depthwise convolution and *Conv* the classical one. It is also important to point out that Batch Normalization and ReLU are applied after each convolution.

ShuffleNet *ShuffleNet* [335, 201] was proposed and designed by Face++ team for mobile devices with limited computing power. In this architecture, the commonly used pointwise convolutions,



Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure 2.23: Architecture of MobileNet. Source: original paper [138].

namely convolution with 1×1 kernel matrices, are substituted by pointwise group convolutions, mitigated using a channel shuffle operation to reduce computation costs. Group convolution is a generalization of the aforementioned depthwise convolution, where we are not imposing having the same number of input and output channels. In fact, in this case, the input feature maps are divided into two or more groups in the channel dimension, and then convolution is performed separately on each group, as depicted in Figure 2.24. In a group-wise convolution, the input and output channels are thus not the same, because the number of output channels for each group does not have to equal the number of input channels in the group. In this way, since the weights are not shared between the groups and each convolution filter works on fewer input channels than before, the number of parameters needed is reduced. This kind of operation presents a side effect. Each feature map in the output derives from only a fraction of the inputs, without having interactions with other groups, limiting thus the networks ability to learn interesting things. In order to solve this problem, *channel shuffling* is introduced. Hence, as described in Figure 2.24, after performing grouped convolution, the output feature maps are rearranged along the channels dimension through the channel shuffle operation.

Starting from this, the basic building block of ShuffleNet is constructed. It is characterized by the presence of 1×1 bottleneck grouped convolution layer followed by channel shuffle. Then, there is a 3×3 depthwise convolution layer with batchnorm, where ReLU is dropped. The final layer of the block is composed of another 1×1 grouped convolution operation, which is used to expand the number of channels again in order to match the channels of the input. This is fundamental because the ShuffleNet block is characterized also by the presence of a residual connection.

The full architecture then starts with a regular 3×3 convolution with stride 2 followed by max pooling. Then there are three stages, each with 4 or 8 ShuffleNet blocks. In the end, there is global

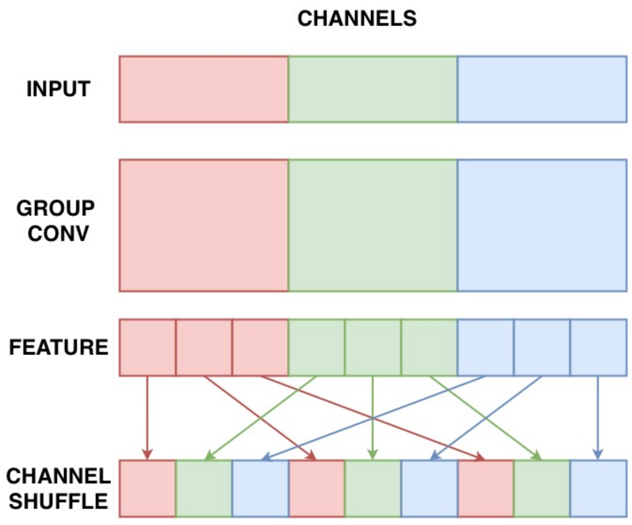


Figure 2.24: Representation of channel shuffle with group convolution: original paper [335].

average pooling and a fully-connected layer that does the classification. There exists also another version of this net, *ShuffleNet V2* [201], where an analysis on grouped convolution is performed. To have a lite model, the group convolutions are substituted with a new channel split operation, which sends half the channels through the residual branch, i.e. leaving them unchanged, and the other half through the ShuffleNet block branch. In the end, the outputs of these two groups are not summed as before but concatenated. The overall architecture of ShuffleNet V2 remains the same as the previous version, changing only the structure of the building blocks, leading to less computational effort.

SqueezeNet *SqueezeNet* [141] was one of the first designed lite models, able to achieve great performances: it has ImageNet accuracy similar to AlexNet, but with 50 times fewer parameters. Also in this case, the basic idea of this net is to substitute convolution layers with large kernels, common in CNNs, with small filters, such as 1×1 and 3×3. In order to obtain parameter reduction in the network, a *fire module* is introduced as the building block of the CNN. Figure 2.25 summarizes

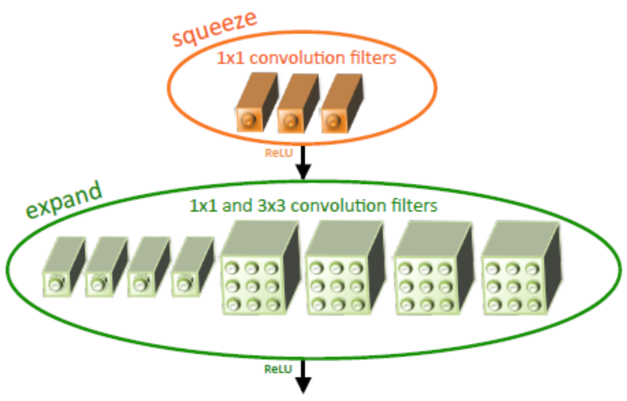


Figure 2.25: Organization of convolution filters in the Fire module, as presented in the original paper [141].

the fundamental operations of the fire module. Firstly, there is a *squeeze layer*, that consists of a 1×1 convolutional layer reducing the number of channels. The role of this layer is thus that of compressing data by reducing the number of parameters. This is then followed by an *expand block* composed of two parallel convolution layers: one with a 1×1 kernel, the other with a 3×3 kernel. The outputs of these two groups of layers are concatenated, leading thus to an increase in the number of channels again.
The overall architecture of SqueezeNet is then characterized by having convolutional and max-pooling layers at the beginning, followed by eight fire modules in succession, sometimes with max-pooling layers between them. The interesting feature of this net is that classification is not performed through fully-connected layers, but employing a convolution layer followed by global average pooling. A variant of SqueezeNet is represented by *SqueezeNext* [91]. The architectural improvements introduced are connected with the fire module. Now there is the introduction of the residual connection and there are two squeeze layers for the aim of channel reduction. After these bottlenecks, there is no longer a 3×3 convolution, that has been split up into two smaller convolutions, 3×1 and 1×3, to decrease again the number of parameters needed. This new module is thus decreasing, as pointed out, the number of channels, and thus of parameters, increasing, on the other hand, the depth of the network to have a general improvement in the model. In the end, there is again, as in the previous version, an expansion layer, composed of a 1×1 convolutional layer increasing the number of channels, to be the same as the output of the residual connection. In this case, also the classification part is changed, employing a fully-connected layer together with a bottleneck layer and a global average pooling to reduce spatial dimensions.

2.4 Object Detection

Object detection [59, 316, 337, 149, 338, 191, 334] consists of the methodologies to automatically detect and localize specific objects in images or videos. It thus aims at solving the task of locating the presence of objects with a bounding box and recognizing the classes of the located objects in an image. Hence, this kind of algorithm produces, as output, a list of object categories present in the image along with an axis-aligned bounding box indicating the position of every instance for each object category. As it can be understood, this is a more challenging problem than image classification, since in this case, when searching for the localization of an object, numerous candidates must also be processed in order to find a good candidate with good precision. When dealing with image recognition tasks, we also assume that there is only one major object in the image and hence the focus is on assigning the correct category to it, but generally, there can be multiple objects in a picture of interest. Object detection aims then at solving also this problem, providing the specific positions for each item in the image.
Connected to object detection, there is the problem of *image segmentation*[31] [334, 10, 60, 185, 256, 153, 117], that aims at recognizing and localizing semantic regions at pixel level. Hence, it focuses on detecting the pixel-level regions of each object instance in an image, called *masks*, providing an understanding on how to divide a picture into regions belonging to different semantic classes.
We introduce now the basic notions connected to this topic, useful to understand how the classification and localization of the objects are predicted and hence the subsequent discussion about the different types of architectures developed to solve this problem (see Section 2.4.7).

[31] We are not going into details on this topic, since in our project we have focused only on image recognition and object detection. However, due to its connection to some databases and architectures described, it was important to mention it to get a more complete overview.

2.4.1 Bounding Boxes

To localize objects we need to define *bounding boxes*, i.e. boxes that wrap around an object representing thus its bounds [334]. A usual way to represent them is by providing a vector with the coordinates of the abscissa q_1 and the ordinate q_2 that constitute its boundaries, i.e. the coordinates of the upper-left corner of the rectangle and the such coordinates of the lower-right corner: $(q_1^{\min}, q_2^{\min}, q_1^{\max}, q_2^{\max})$. Another common way used to represent boxs position and dimensions is to use the coordinates of the center c of the bounding box, its width w and height h, i.e. the vector (c_{q_1}, c_{q_2}, w, h). These two vectors are strictly related to each other, in fact we can define a function to convert between these two representations. For example, let $(q_1^{\min}, q_2^{\min}, q_1^{\max}, q_2^{\max})$ be the boundary coordinates, we can determine the center-size coordinates in this way [334]:

$$\begin{aligned} c_{q_1} &= \frac{q_1^{\min} + q_1^{\max}}{2}, \\ c_{q_2} &= \frac{q_2^{\min} + q_2^{\max}}{2}, \\ w &= q_1^{\max} - q_1^{\min}, \\ h &= q_2^{\max} - q_2^{\min}. \end{aligned} \tag{2.37}$$

Then, starting from these relations, it is easy to define the inverse function, that, given the center-size coordinates, provides the boundary coordinates.

2.4.2 Anchor Boxes

Usually, to predict bounding boxes for the objects of interest, object detection algorithms sample a large number of regions in the input image, determine if they contain the objects of interest, and then adjust the boundaries of the regions to make more accurate predictions. To create these regions, many schemes can be adopted: one of these is represented by the *anchor boxes*, first introduced in Faster R-CNN [249], and also known as *priors* [194]. This method corresponds to generating multiple bounding boxes with varying scales and aspect ratios centered on each pixel and using them to obtain the predicted bounding boxes.
Objects in pictures can occur at any position with any size and shape, thus, generally, we may have infinite possibilities for where and how an object can occur. Obviously, we cannot investigate all these options, and furthermore some can be excluded since they are simply improbable or uninteresting. We need thus to discretize the mathematical space of potential predictions into an acceptable number of possibilities, which are the anchor boxes. Hence, priors are these precalculated, fixed reference boxes, placed at every possible location in a feature map to account for variety in position and representing probable and approximate box predictions.
To generate multiple anchor boxes with different shapes, we need to introduce the notions of *scale* and *aspect ratio*. Let $\mathbf{x}$ be an input image with width w and height h and let w^B and h^B be the width and height of an anchor box. The aspect ratio $\alpha \in (0, 1]$ is basically the ratio between width and height of the prior, whereas the scale $\zeta > 0$ refers to the length or width in pixels of a box as a proportion of the total length or width in pixels of its containing image. They are thus defined as follows [334]:

$$\begin{aligned} \zeta^2 &= w^B * h^B, \\ \alpha &= \frac{w^B}{h^B}, \end{aligned} \tag{2.38}$$

from which we can obtain that the width and height of the anchor box are given by $w^B = \zeta\sqrt{\alpha}$ and $h^B = \zeta/\sqrt{\alpha}$. Therefore, if we want to generate multiple priors characterized by having different shapes, we need to set a series of scales $\zeta_1, \ldots, \zeta_n$ and of aspect ratios $\alpha_1, \ldots, \alpha_m$. In this way, we are generating $w * h * n * m$ reference boxes with each pixel as the center. This is unpractical, in fact usually we define a scale for all the feature maps in a layer and then provide a list of aspect ratios. Figure 2.26 provides an example of these default boxes for a central pixel and for a general one in which priors overshoot the edges of the feature map. In that case, priors are then clipped in order to be totally contained inside the image. In addition, since the first layers of a CNN are characterized by having larger feature maps, their priors have smaller scales, ideal for detecting smaller objects. Then, going deeper in the network, ζ will increase to let the model detect also bigger elements.

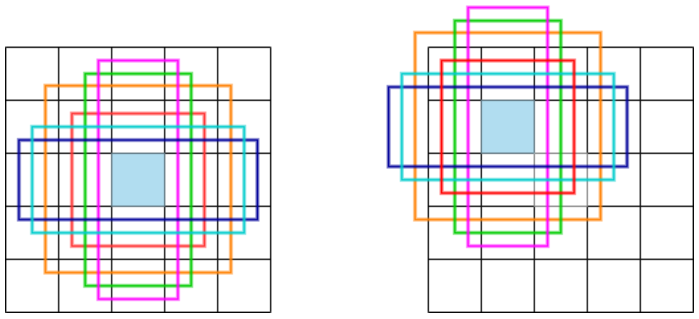


Figure 2.26: Example of priors around a central pixel (on the left) and for a general one in which priors overshoot the edges of the feature map (on the right). In both, there are 5 priors with aspect ratios 1, 2, 3, 1/2, 1/3 and areas of a square of side 0.55, and a 6th prior with aspect ratio 1 and of side 0.63.

As pointed out before, anchor boxes are introduced as an approximate starting point to find out how much they need to be adjusted to get a more exact prediction for a bounding box. Hence, in the end, each predicted box is a slight deviation from a prior. In order to account this, let $(\hat{c}_{q_1}, \hat{c}_{q_2}, \hat{w}, \hat{h})$ be the center-size coordinates of a predicted bounding box, and $(c^B_{q_1}, c^B_{q_2}, w^B, h^B)$ those of the anchor box with which the prediction was made. The *offsets* from a prior are thus defined as:

$$\begin{aligned} g_{c_{q_1}} &= \frac{\hat{c}_{q_1} - c^B_{q_1}}{w^B}, \\ g_{c_{q_2}} &= \frac{\hat{c}_{q_2} - c^B_{q_2}}{h^B}, \\ g_w &= \log\left(\frac{\hat{w}}{w^B}\right), \\ g_h &= \log\left(\frac{\hat{h}}{h^B}\right), \end{aligned} \tag{2.39}$$

where it can be noted that each of them is normalized by the corresponding dimension of the prior. These four offsets $(g_{c_{q_1}}, g_{c_{q_2}}, g_w, g_h)$ can thus be used also to encode the predicted bounding box's position and location.

2.4.3 Intersection over Union

Once the possible predicted bounding boxes and the category for each object in a picture are determined, we need to define a specific performance metric to take into account the accuracy of the prediction with respect to the expected label and the *ground-truth bounding box* of the object, i.e. the hand-labeled bounding box that encloses the targeted object. Hence, the overlapping area between the predicted box and the ground-truth box needs to be evaluated, for example using the *Intersection over Union metric*. The *Intersection over Union (IoU)* or *Jaccard Index* or *Jaccard Overlap* [334, 265, 83, 282] is an evaluation metric measuring the degree or extent to which two boxes overlap. It can thus be used to account for the detector accuracy by evaluating the overlapping area between the predicted bounding box e the related ground-truth. Given two sets $\mathcal{A}$ and $\mathcal{B}$, their IoU is the size of their intersection divided by the size of their union:

$$\text{IoU}(\mathcal{A}, \mathcal{B}) = \frac{\mathcal{A} \cap \mathcal{B}}{\mathcal{A} \cup \mathcal{B}}. \tag{2.40}$$

Its value is between 0 and 1: 0 means that two bounding boxes do not overlap at all, while 1 indicates that the two bounding boxes are equal. Hence, the higher values of IoU indicate the better predicted locations of the bounding boxes for the targeted objects. Usually, a value of 0.5 determines a good prediction and is fixed as IoU threshold to choose the box candidates to keep.

2.4.4 Labeling Anchor Boxes during Training

To train an object detection model, we need class and offset labels for each anchor box, where the former is the category of the object in the prior and the latter is the offset of the ground-truth bounding box relative to it. Hence, to label any generated anchor box, the ground-truth bounding boxes with the related classes of the objects in the pictures composing the training dataset should be exploited. In the following, we present an algorithmic procedure for assigning the closest ground-truth bounding boxes to anchor boxes [334].
Let $\mathbf{x}$ be an input image, $\mathcal{A}_1, \ldots, \mathcal{A}_{n_a}$ be the anchor boxes, and $\mathcal{B}_1, \ldots, \mathcal{B}_{n_b}$ be the ground-truth boxes, with $n_a \geq n_b$. We can define a matrix $\mathbf{M} \in \mathbb{R}^{n_a \times n_b}$, where each component m_{ij} represents the IoU between $\mathcal{A}_i$ and $\mathcal{B}_j$. In order to assign a ground-truth and thus a label to each anchor box, we need to follow these steps [334]:

1. Find the maximum value in $\mathbf{M}$ and denote with i_1 and j_1 the corresponding row and column indices. In this way, we are assigning the ground-truth box $\mathcal{B}_{j_1}$ to the prior $\mathcal{A}_{i_1}$. After the first assignment, all the elements in the i_1-th row and the j_1-th column in $\mathbf{M}$ are discarded.

2. Find now the maximum value among the remaining elements of $\mathbf{M}$, identified by indices i_2 and j_2 for the row and column, respectively. The anchor box $\mathcal{A}_{i_2}$ is thus coupled with the ground-truth $\mathcal{B}_{j_2}$ and, as in the previous step, we discard all the elements in the corresponding row and column of $\mathbf{M}$.

3. We proceed in this way until all the n_b ground-truth bounding boxes have been assigned to an anchor box, i.e. till all the elements in the n_b columns have been discarded.

4. For the remaining $n_a - n_b$ anchor boxes, we can assign a ground-truth box $\mathcal{B}_j$ to each anchor $\mathcal{A}_i$ by searching for the largest IoU between them throughout the i-th column of $\mathbf{M}$. $\mathcal{B}_j$ will be then linked to $\mathcal{A}_i$, only if the connected Jaccard index is greater than a predefined threshold.

5. Assign to each anchor box the label of the related ground-truth box and then determine the offsets between them using Equations (2.39). The priors that have not a link with a ground-truth bounding box are labeled with 0, that is the integer connected with the class "background", and are thus characterized also by 0s offsets.

In the last step, we have assigned the category background to anchor boxes without a linked ground-truth box. It is useful to point out that usually when preparing the dataset, this category is added to the set of class labels to take into account these cases. Priors whose classes are background are often referred to as *negative anchor boxes*, differentiating from the rest which are called *positive anchor boxes* [334].

2.4.5 Non-Maximum Suppression

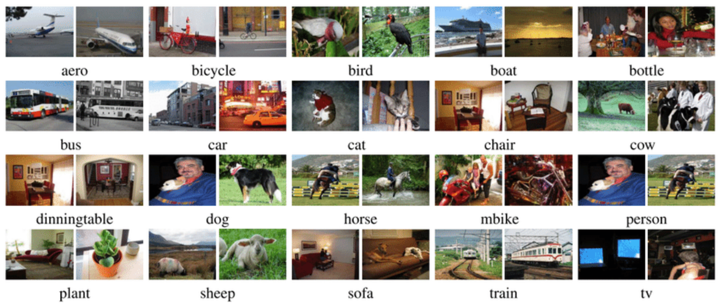
During the prediction phase, multiple highly overlapping anchor boxes, predict classes, and offsets are usually generated for the same object in a picture, providing thus redundant information. To solve the issue of duplicate proposals, it is usually employed a simple algorithmic approach called Non-Maximum Suppression (NMS) [258, 334, 28]. More technically speaking, for each predicted bounding box $\hat{\mathcal{B}}$, we evaluate the predicted likelihood for each class, where the largest value p denotes the predicted class for it. This value is usually referred to as the *confidence* or *score* of the predicted bounding box. All the predicted non-background bounding boxes are then sorted by confidence in descending order to generate a list $\mathbf{L}$. Now the sorted list is manipulated through these steps [334]:

1. Select the proposals with the highest confidence score, remove them from $\mathbf{L}$ and use them to form a basis $\mathcal{U}$.
2. Compare the bounding boxes in $\mathcal{U}$ with the remaining in $\mathbf{L}$ using the Jaccard index as an element of similarity. Hence, we calculate the IoU between each element in $\mathcal{U}$ and each element in $\mathbf{L}$ and then remove from $\mathbf{L}$ the proposals with IoU greater than a predefined threshold δ. In this way, we are keeping the predicted bounding boxes with the highest confidence — the one in $\mathcal{U}$ — but dropping others that are too similar to it but with non-maximum scores, namely the proposals discarded from $\mathbf{L}$.
3. Select the proposals with the second highest confidence from the remaining bounding boxes in $\mathbf{L}$. Remove them from this list and add them to the basis $\mathcal{U}$. Calculate again the IoU of the elements in $\mathcal{U}$ with all the proposals in $\mathbf{L}$ and suppress the boxes which have high IoU than threshold δ.
4. Repeat the above process until all the proposals in $\mathbf{L}$ have been used, i.e. have been discarded or used as a basis. In this way, the IoU of any pair of predicted bounding boxes in $\mathcal{U}$ is below the predefined threshold δ, and hence they are not too similar to each other.

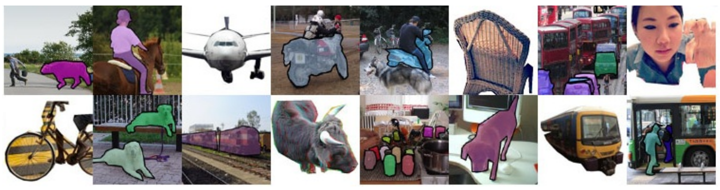
The bounding boxes in $\mathcal{U}$ represent thus the remaining proposal for the objects in a picture and are thus the ones that will be displayed in the output. At this point and after the application of the NMS process, there should be a single bounding box for each object in the image, but this is strictly connected with the choice of the IoU threshold. If it is too high there can be too many proposals for the same object, whereas, on the other hand, if it is too low we may end up missing proposals for objects. A value that is usually chosen is 0.6 [334].

2.4.6 Datasets

Datasets play a key role in the object detection task, especially in determining the final accuracy and hence the capability of the model in solving a problem [191, 338, 282]. Using a huge number of images can help in capturing a vast richness and diversity of objects, but can lead to computationally expensive tests. On the other hand, a small amount of data is easy to manage but can be connected to the low performance of the model. As described for the problem of image recognition in Section 2.3.1, there are basically two possible choices for the dataset: construct a custom set of data or use a benchmark one. The common features between them are the elements strictly necessary to construct this type of database. In addition to images, we need files, called *annotations* encoding information about the objects present in them with the corresponding label and ground-truth box for each item. In benchmark datasets, these annotations are provided, whereas in the custom case the images are hand-annotated.



a) PASCAL VOC [83, 269]



b) MS COCO [189]

Figure 2.27: Examples of images from (a) PASCAL VOC [83, 82, 269], and (b) MS COCO [189].

From the point of view of benchmark sets of data, there are four famous choices for object detection [191, 338, 282]: PASCAL VOC [83, 82], ImageNet [68] (see Section 2.3.1), MS COCO [189] and Open Images [168, 164]. **PASCAL VOC** (PASCAL Visual Object Classes) dataset [83, 82] consists of 11,530 images for training and validations with 27,450 annotations for regions of interest. Starting from only four categories in 2005, the dataset has increased to 20 categories that are common in everyday life (see Figure 2.27 (a)): person, bird, cat, cow, dog, horse, sheep, airplane, bicycle, boat, bus, car, motorbike, train, bottle, chair, dining table, potted plant, sofa, tv/monitor.
MS COCO (Microsoft COCO) dataset [189] was a response to the criticism connected with the pictures in ImageNet, where the objects to be detected are large and well centered, creating thus atypical scenarios for real-world cases. In this database, instead, there are complex everyday scenes with common objects in their natural context, closer to real-life — objects can be partially occluded, at a wide range of scales—, where fully-segmented instances are provided for each object for a more accurate detector evaluation. Hence, MS COCO can be used not only to solve the problem of object detection through bounding boxes but in particular that of object segmentation. It is composed of 300,000 fully segmented images, subdivided into 80 categories, some examples of which are shown in Figure 2.27 (b). Another famous database is then represented in Figure 2.28 by **Open Images** [168, 164], currently the largest publicly available object detection dataset. It differs from ImageNet and MS COCO not merely in terms of the significantly increased number of classes, images, bounding box annotations, and instance segmentation mask annotations, but also in the annotation process. In the former, instances of all classes in the dataset are exhaustively annotated, whereas for Open Images a classifier was applied to each image and only those labels with sufficiently high scores were sent for human verification and then annotated in case of human-confirmed positive labels [19].

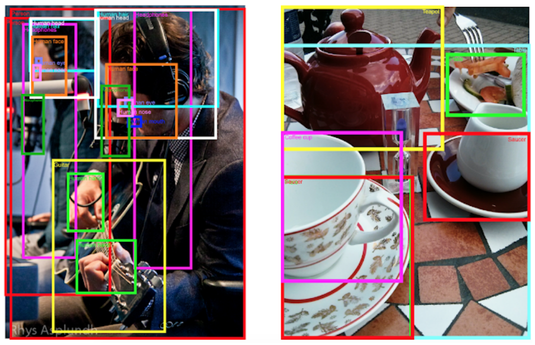
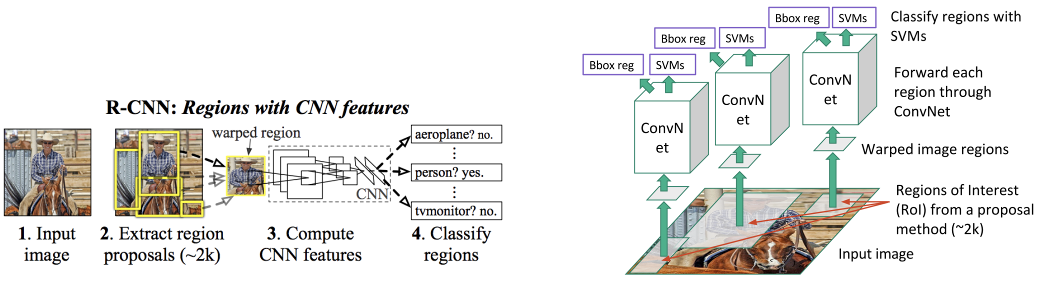


Figure 2.28: Example of annotated images from Open Images dataset [168, 164].

2.4.7 Object Detectors: Architectures for Object Detection

A huge number of architectures have been developed to deal with the problem of object detection [149, 338, 191, 294, 282]. They can generally be categorized into two classes: *Single-stage object detectors* [194, 246, 277, 247] and *Two-stage object detectors* [95, 61, 96, 94, 249, 117]. The main difference lies in the fact that the latter firstly generate some candidate object proposals and then classify them into specific classes, whereas one-stage methods simultaneously extract and classify all the object proposals. This results in having higher detection accuracy but slower detection speed for the two-stage methods, and a much faster detection speed and comparable detection accuracy for the single-stage detectors. We are now briefly introducing these two different types of object detectors by providing some examples of existing architectures.

Two-stage Methods for Object Detection In the two-stage methods [149, 338, 191], object detection is treated as a multistage process: firstly, given an input image, some proposals of possible objects are extracted, then they are classified into specific object categories. Among the various two-stage object detectors developed, the series of R-CNN, including R-CNN [96, 95], Fast R-CNN [94], Faster R-CNN [249], and Mask R-CNN [117], is very representative of this category.
To distinguish different objects in an image, we need a method to detect the region of interest, where the targeted items can be found. A naive approach to solve this problem, called *exhaustive search*, consists in selecting different parts of the image, and using a CNN to classify the presence of the object within that region. Obviously, this leads to a computationally infeasible technique since we need to pick a huge number of regions to cover all the possible positions and scales objects may have. The **Region-based Convolutional Network (R-CNN)** [96, 95] represents a method to bypass this problem. It introduces a new approach to detect region proposals: the *selective search method* [311]. To generate the proposal boxes for the targeted objects, it starts by initializing small regions in the picture and then merges them using a greedy algorithm exploiting a hierarchical grouping, where the detected regions are merged according to a variety of color spaces and similarity metrics. Hence, trough this procedure just 2000 regions are extracted from the image, representing thus the *region proposals* or *Region of Interest (RoI)* [96]. As depicted in Figure 2.29, these 2000 candidate proposals are warped into a square and then fed into a CNN to extract a fixed-length (4096-dimensional) feature vector from each of these regions. These features are then classified using Singular Vector Machines (SVMs) and, in case the presence of an object is detected, a linear regressor is used to tighten and adjust the bounding box of the objects, giving in output the four offset values described in Equation (2.39) defining the predicted box. In this way, a R-CNN tries to mimic the final stages of classification in CNNs, where fully-connected layers are used to output a score and category for


R-CNN: Regions with CNN features
warped region
aeroplane? no.
person? yes.
tvmonitor? no.
CNN
1. Input image
2. Extract region proposals (~2k)
3. Compute CNN features
4. Classify regions
Bbox reg
SVMs
ConvNet
Classify regions with SVMs
Forward each region through ConvNet
Warped image regions
Regions of Interest (RoI) from a proposal method (~2k)
Input image

a) R-CNN b) Close-up on the detection procedure

Figure 2.29: Schema of the R-CNN architecture (a) and of the class and location detection procedure for the region of interests (b). Source: original papers [96, 95].

the image under consideration, that in this case corresponds to the several region proposals. Even if this method is very intuitive, it presents a lot of problems. First of all, the computation of CNN features of different proposals are not shared: for each of the 2000 object proposals in an image, a CNN forward pass has to be performed, leading to a time-consuming training procedure. Then, the candidate regions are generated employing the selective search algorithm, which is a fixed method not involved in backpropagation. For this reason, this represents a weak point, because, since it is not affected by the learning phase, it can lead to bad candidate region proposals.
The drawbacks of R-CNN were solved by the same authors by building a faster object detection algorithm, called **Fast R-CNN** [94]. The approach is similar to R-CNN with the difference that the 2000 region proposals are not fed into the CNN every time, but the CNN operation is done only once per image, providing a faster algorithm. Hence, as described in Figure 2.30, the entire image is

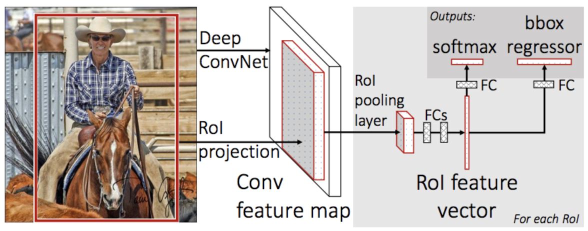


Figure 2.30: Overall schema of Fast R-CNN taken from the original paper [94].

used as: (i) input for the CNN, that extracts features from it before having the region proposals; (ii) starting point to identify the region proposals through the selective search method. Once the convolutional feature maps are obtained, they are scaled to get a valid pre-defined size region of interests through a *Region of Interest pooling layer*. Hence, for every detected RoI, it takes a section of the input feature map that corresponds to it and divides it into a predefined number of equal-sized sections. The output of the RoI layer is then obtained by finding the maximum value in each section. At this point, the convolutional feature maps have thus been warped into fixed-size spatial regions, that are fed into two fully connected layers. To be precise, the classification part is made of two sibling branches with fully connected operations responsible for object classification and box regression. Therefore, the output of the box regression will be the offset values for the bounding box, whereas a softmax layer is employed for the class prediction. Since now Fast R-CNN has two sibling outputs for object classification and box regression, the loss $\mathcal{L}$ should reflect this property, becoming the joint of a classification loss $\mathcal{L}_{\text{cls}}$ and a regression loss $\mathcal{L}_{\text{loc}}$. Hence, a multi-task loss is defined for each labeled RoI as follows [94, 149]:

$$\mathcal{L}_{|\text{RoI},i}(\hat{\mathbf{y}}_{\text{cls}}, \mathbf{y}_{\text{cls}}, \hat{\mathbf{y}}_{\text{loc}}, \mathbf{y}_{\text{loc}}) = \mathcal{L}_{\text{cls}|\text{RoI},i}(\hat{\mathbf{y}}_{\text{cls}}, \mathbf{y}_{\text{cls}}) + \lambda[\mathbf{y}_{\text{cls}} \geq 1]\mathcal{L}_{\text{loc}|\text{RoI},i}(\hat{\mathbf{y}}_{\text{loc}}, \mathbf{y}_{\text{loc}}), \quad \text{for } i = 1, \dots, n_{\text{RoI}}, \tag{2.41}$$

where n_{RoI} is the total number of region proposals; $\hat{\mathbf{y}}_{\text{cls}}$ represents the predicted output of the object detector, i.e. $\hat{\mathbf{y}}_{\text{cls}} = (\hat{p}_0, \dots, \hat{p}_{n_{\text{class}}})$ is the predicted discrete probability distribution (per RoI) over all the n_{class}+1, since we added also category 0 as background; whereas $\hat{\mathbf{y}}_{\text{loc}}$ are the four predicted offsets $\hat{\mathbf{y}}_{\text{loc}} = (\hat{y}_{\text{loc},q_1}, \hat{y}_{\text{loc},q_2}, \hat{y}_{\text{loc},w}, \hat{y}_{\text{loc},h})$. Then, $\mathbf{y} = (\mathbf{y}_{\text{cls}}, \mathbf{y}_{\text{loc}})$ is the total output of the object detector, where $\mathbf{y}_{\text{cls}}$ and $\mathbf{y}_{\text{loc}}$ are the expected classification and localization outputs respectively. As stated in [94], $\mathcal{L}_{\text{cls}}$ is the log loss (1.18) the log loss for the true class $\mathbf{y}_{\text{cls}}$:

$$\mathcal{L}_{\text{cls}}(\hat{\mathbf{y}}_{\text{cls}}, \mathbf{y}_{\text{cls}}) = -\log(\hat{p}_{\mathbf{y}_{\text{cls}}}). \tag{2.42}$$

The bounding-box regression loss $\mathcal{L}_{\text{loc}}$ is determined by:

$$\mathcal{L}_{\text{loc}}(\hat{\mathbf{y}}_{\text{loc}}, \mathbf{y}_{\text{loc}}) = \sum_{j \in \{q_1, q_2, w, h\}} smooth_{L^1}(\hat{y}_{\text{loc},j} - y_{\text{loc},j}), \tag{2.43}$$

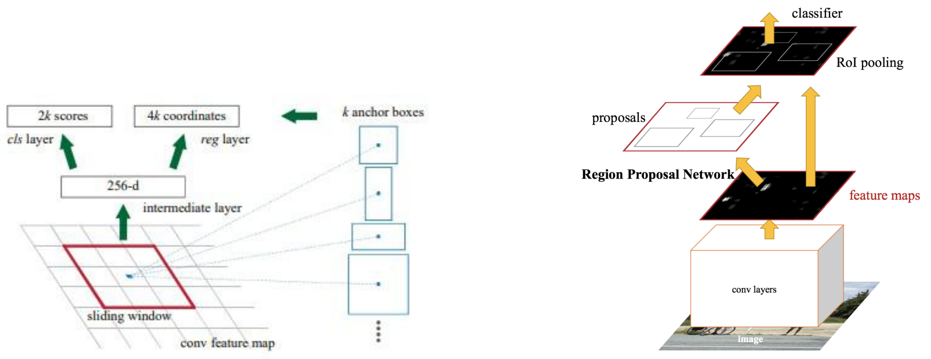
where $smooth_{L^1}$ is a robust L^1 loss less sensitive to outliers, defined in [94] as:

$$smooth_{L^1} = \begin{cases} 0.5x^2, & \text{if } |x| < 1, \\ |x| - 0.5, & \text{otherwise.} \end{cases} \tag{2.44}$$

The classification and localization losses are then balanced through the hyper-parameter λ and using the Iverson bracket indicator function[32] $[\mathbf{y}_{\text{cls}} \geq 1]$ [144, 160] to evaluate when $\mathbf{y}_{\text{cls}}$ is equal 0. Namely, when we have the background class, there is no bounding box, and thus the RoI is not contributing to the regression loss.
Even if Fast R-CNN solved the problem of the long training phase connected with the multiple CNN forward steps and several SVM's, there was still an open issue with the presence of the selective search algorithm, that it proved to be a time-consuming procedure for generating region proposals. This problem was overcome by the last evolution of this network: **Faster R-CNN** [249], where a **Region Proposal Network (RPN)** is introduced to generate region proposals. As described in Figure 2.31 (a), to construct these candidates, a small network is slid over a convolutional feature map, e.g. the output of a CNN. Hence, RPN is implemented in a fully convolutional way: a 3×3 convolutional layer, followed by two sibling 1×1 convolutional layers for box regression and box classification. To be more precise, for each sliding window location, multiple region proposals are simultaneously predicted based on k anchors of different aspect ratios and scales. For example, in the original implementation of Faster R-CNN in [249], three different aspect ratios of $\{1:2, 1:1, 2:1\}$ and three different scales of $\{0.5, 1, 2\}$ are used, providing thus $k = 9$ different anchor boxes at each sliding window.
After this step, the two parallel 1×1 convolutional layers are applied, and thus, for each of these anchor boxes, two different types of predictions are provided: the binary classification and the bounding box regression adjustment. For the classification layer, there are $2k$ outputs for each sliding window, where 2 is related to the fact of having a binary classification, i.e. the score of being

[32]We recall that the Iverson brackets are a generalization of the Kronecker delta. Given a statement $\mathcal{S}$, it is thus defined as:

$$[\mathcal{S}] = \begin{cases} 1 & \text{if } \mathcal{S} \text{ is true,} \\ 0 & \text{otherwise.} \end{cases} \tag{2.45}$$



a) Region Proposal Network b) Faster R-CNN

Figure 2.31: Structure of the Region Proposal Network (a) and of Faster R-CNN (b), as presented in [249].

background (not an object) and the score of being foreground (an actual object). On the other hand, the box regression layer has $4k$ outputs for each sliding window, representing the 4 offsets value that needs to be applied to the center of the proposal to better fit the object it is predicting. For training this network, first, binary class labels need to be assigned to identify whether an anchor contains an object or background. In [249], two conditions are used to assign a positive label to an anchor: having the highest IoU with a ground-truth box or having an IoU overlap higher than 0.7 with any ground-truth box. The negative label is then assigned to all the anchors with an IoU lower than 0.3 for all ground truth boxes. On the other hand, all the anchors having neither positive nor negative label does not contribute to training. Now, a multitask loss is then introduced [249] to take into account the classification and box localization predictions for each RoI:

$$\mathcal{L}(\{\hat{\mathrm{y}}_{\mathrm{cls}}\}, \{\hat{\mathrm{y}}_{\mathrm{loc}}\}) = \frac{1}{N_{\mathrm{cls}}} \sum_{i=1}^{n_{\mathrm{RoI}}} \mathcal{L}_{\mathrm{cls}}(\hat{\mathrm{y}}_{\mathrm{cls},i}, \mathrm{y}_{\mathrm{cls}}) + \lambda \frac{1}{N_{\mathrm{loc}}} \sum_{i=1}^{n_{\mathrm{RoI}}} [\mathrm{y}_{\mathrm{cls}} \geq 1] \mathcal{L}_{\mathrm{loc}}(\hat{\mathrm{y}}_{\mathrm{loc}}, \mathrm{y}_{\mathrm{loc}}), \tag{2.46}$$

where the classification loss $\mathcal{L}_{\mathrm{cls}}$ and regression loss $\mathcal{L}_{\mathrm{loc}}$ are the same introduced for Fast R-CNN. In this case, $\hat{y}_{\mathrm{cls},i}$ represents the predicted probability of anchor i being an object, whilst $\mathrm{y}_{\mathrm{cls}}$ is the ground truth label (binary) of whether anchor i is an object, i.e. is 1 when we have a positive prior. Hence, in particular, $\mathcal{L}_{\mathrm{cls}}$ corresponds with the log loss over two classes, object versus non-object:

$$\mathcal{L}_{\mathrm{cls}}(\hat{\mathrm{y}}_{\mathrm{cls},i}, \mathrm{y}_{\mathrm{cls}}) = -\mathrm{y}_{\mathrm{cls}} \log(\hat{\mathrm{y}}_{\mathrm{cls},i}) - (1 - \mathrm{y}_{\mathrm{cls}}) \log(1 - \hat{\mathrm{y}}_{\mathrm{cls},i}). \tag{2.47}$$

Then, the index i refers then to the index of an anchor box, N_{cls} and N_{loc} are the terms to, respectively, normalize classification loss and location loss, representing the batch size (e.g. 256 [249]) and the number of anchor locations (e.g. about 2400 [249]).
Therefore, as depicted in Figure 2.30 (b), Faster R-CNN integrates proposal generation, proposal classification, and proposal regression into a unified network, and can thus be subdivided into two modules: RPN to extract candidate object proposals, avoiding the selective search method and leading to accelerated training and testing processes and improved performances; and Fast R-CNN detector to classify these proposals into the specific categories and predict more accurate proposal locations. RPN and Fast R-CNN are then not trained independently, since they share the same base network. Following [249], a possible way to train them is using *alternating training*: namely, RPN is firstly trained generating some proposals, that are then used to train Fast R-CNN. Otherwise, another approach is represented by *approximate joint training*, where RPN and Fast R-CNN are seen

as a unified network. In this way, the total loss is given by the joint of the RPN loss and Fast R-CNN loss, and, at each iteration of the training process, the proposals generated by RPN are treated as fixed proposals when training Fast R-CNN detector, i.e. the derivative of proposals coordinates are ignored.
The detectors proposed deal only with the problem of object detection by predicting locations for the objects in a picture and the related class label. **Mask R-CNN**[33] [117] has been proposed to solve the problem of instance segmentation: detect precisely the pixels that correspond to an object and label them. Mask R-CNN incorporates thus instance segmentation and object detection into

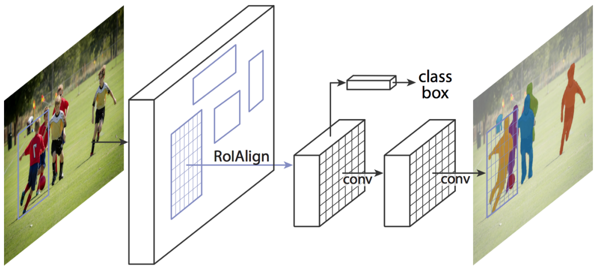


Figure 2.32: The Mask R-CNN framework. Image from the original paper [117].

a unified framework based on Faster R-CNN architecture. Specifically, as depicted in Figure 2.32, it replaces the RoI pooling layer with the *Region of Interest alignment layer*, which makes use of bilinear interpolation to preserve the spatial information on the feature maps, hence more suitable for pixel-level prediction. In particular, these feature maps are used not only for class and bounding box prediction for each RoI but also to detect the pixel-level position of the object, i.e. its mask, using an additional fully convolutional network. In this case, the loss will also take into account this new task, by adding a mask loss to the loss described for Faster R-CNN. The multitask loss of Mask R-CNN on each sampled RoI align is thus the joint of classification loss, regression loss, and mask loss:

$$\mathcal{L} = \mathcal{L}_{\text{loc}} + \mathcal{L}_{\text{cls}} + \mathcal{L}_{\text{mask}}, \tag{2.48}$$

where $\mathcal{L}_{\text{mask}}$ is the average binary cross-entropy loss defined in Equation (1.18), only including k-th mask if the RoI Align is associated with the ground truth class k:

$$\mathcal{L}_{\text{mask}} = -\frac{1}{m^2} \sum_{1 \le i,j \le m} \left[-y_{ij} \log(\hat{y}^k_{ij}) - (1 - y_{ij}) \log(1 - \hat{y}^k_{ij}). \right], \tag{2.49}$$

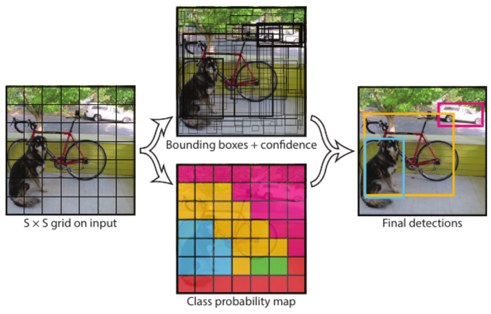
where we are considering having masks with dimension $m \times m$, one for each class n_{class}.

One-stage Methods for Object Detection Even if Faster R-CNN brought large benefits as an accelerated training process with improved detection accuracy with respect to previous detector's model attempts, the presence of two different components for classification and detection (RPN and Fast R-CNN) represents a bottleneck in real-time applications due to the time needed to handle these two different parts. One-stage networks, based on global regression and classification, aim to overcome this difficulty by simultaneously predicting object category and object location [149, 338, 191]. In this way, by mapping straightly from image pixels to bounding box coordinates and class probabilities, one-stage frameworks can reduce time expense, having much faster detection speed,

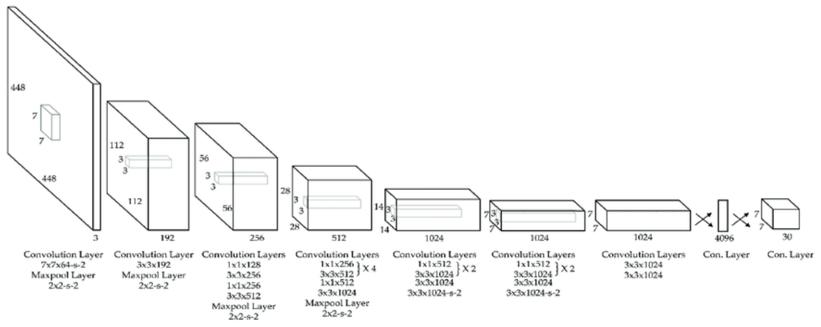
[33]We are not going into details of image segmentation and related architectures because it is outside the scope of this thesis. The interested readers can refer to [117, 334, 185, 60, 10, 256, 153].

but with comparable detection accuracy. Among the one-stage methods, we focus on YOLO [246, 247] and SSD [194].

YOLO (You Only Look Once) [246, 247, 248] is a unified object detector, that treats the problem of object detection as a regression problem from image pixels to spatially separated bounding boxes and associated class probabilities, as described in Figure 2.33 (a). Unlike region-based approaches



a) Schematic representation of YOLO [246].



b) YOLO architecture [246, 163]

Figure 2.33: Schematic representation of the process of localization and detection in YOLO (a) and of its architecture (b) [246, 247, 163].

discussed for the two-stage methods, YOLO uses features from an entire image globally and not from a local region. More technically speaking, the input image is divided into an $S \times S$ grid, being $S \geq 1$ a constant integer value, where each grid cell is responsible for predicting the object centered in it, thus providing as output B bounding boxes with objectness scores and n_{class} conditional class probabilities. Therefore, the prediction for each bounding box corresponds with a vector $(\hat{c}_{q_1}, \hat{c}_{q_1}, \hat{w}, \hat{h}, \hat{o}_{\text{cs}})$, where the first four components are the center-size coordinates of the bounding box and $\hat{o}_{\text{cs}}$ is the corresponding confidence objectness score, defined as [246, 338]:

$$\hat{o}_{\text{cs}} = \Pr(Object) * \text{IoU}_{\text{pred}}^{\text{truth}}. \tag{2.50}$$

which indicates how likely there exist objects, $\Pr(Object) \geq 0$, and, if an object exists in that cell, it shows the confidence of its prediction $\text{IoU}_{\text{pred}}^{\text{truth}}$, i.e. the IoU between the predicted box and the ground truth. These confidence scores are thus reflecting how confident the model is in its predictions about the presence of an object and its accuracy on those predictions. In addition to this, at the same time, n_{class} conditional class probabilities $\hat{p}_k = \Pr(Class_k \mid Object)$ are also predicted for each grid cell. It should be noticed that these probabilities are conditioned on the grid cell containing an object and thus only the contribution from the grid cell containing an object is calculated. During testing time, class-specific confidence scores for each box are achieved by multiplying the individual box confidence predictions with the conditional class probabilities in this way [246, 338]:

$$\hat{o}_{\text{cs}} * \hat{p}_k = \Pr(Object) * \text{IoU}_{\text{pred}}^{\text{truth}} * \Pr(Class_k \mid Object) = \Pr(Class_k) * \text{IoU}_{\text{pred}}^{\text{truth}}, \tag{2.51}$$

that encodes both the probability of class-specific objects in the box and the accuracy of the box prediction, namely the fitness between the predicted box and the object.
When training the object detector, we need to take into account both localization errors and classification accuracy predictions. Hence, as done for the two-stage methods, a multi-part loss function balancing the two parts needs to be introduced [246, 338]:

$$\begin{aligned}\mathcal{L}(\hat{Y}_{\text{cls}}, \hat{Y}_{\text{loc}}) = \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(c_{q_1}^i - \hat{c}_{q_1}^i)^2 + (c_{q_2}^i - \hat{c}_{q_2}^i)^2 \right] + \\ + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(c_{q_1}^i - \hat{c}_{q_1}^i)^2 + (c_{q_2}^i - \hat{c}_{q_2}^i)^2 \right] + \\ + \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} (p^i - \hat{p}^i)^2 + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} (p^i - \hat{p}^i)^2 + \\ + \sum_{i=o}^{S^2} \mathbb{1}_{i}^{\text{noobj}} \sum_{k=1}^{n_{\text{classes}}} (y_k^i - \hat{y}_k^i)^2,\end{aligned} \tag{2.52}$$

where the predicted localization outputs are $\hat{Y}_{\text{loc}} = [\hat{\mathbf{y}}_{\text{loc}}^1, \dots, \hat{\mathbf{y}}_{\text{loc}}^{S^2}]$ with $\hat{\mathbf{y}}_{\text{loc}}^i = (\hat{c}_{q_1}^i, \hat{c}_{q_2}^i.\hat{w}^i, \hat{h}^i)$, for . Similarly, $\mathbf{y}_{\text{loc}}^i = (c_{q_1}^i, c_{q_2}^i.w^i, h^i)$, for $i = 1, \dots, S^2$, represents the expected localization outputs for each grid cell. Then, $\mathbf{y}_{\text{cls}}^j = (y_1^j, \dots, y_{n_{\text{class}}}^j)$ and $\hat{\mathbf{y}}_{\text{cls}}^j = (\hat{y}_1^j, \dots, \hat{y}_{n_{\text{class}}}^j)$, for $j = 1, \dots, S^2$, are respectively the expected and predicted classification outputs, i.e. the expected and predicted probabilities that an object in a grid cell belongs to a certain category. Then, $\mathbb{1}_i^{\text{noobj}}$ denotes if an object appears in cell i, whilst $\mathbb{1}_{ij}^{\text{noobj}}$ highlights that the j-th bounding box predictor in cell i is responsible for that prediction.
Note that the first terms in the loss take into account the bounding box coordinates predictions, the next two the confidence scores, whereas the last one the classification predictions. The two parameters, λ_{coord} and λ_{noobj} accomplish the need to weigh the multiple tasks, localization and classification. In the original paper [246], these values are set to 5 and 0.5 respectively, i.e. the loss from bounding box coordinate predictions is increased, whilst the loss from confidence predictions for boxes that don't contain objects is decreased. We need to point out also that the loss function penalizes classification errors only when an object is present in the grid cell under consideration, and, in the same way, the box predictor penalizes localization errors when the highest IoU of any predictor in that grid cell is achieved.
Figure 2.33 (b) shows the overall architecture of YOLO, which basically consists of 24 convolutional layers and 2 fully-connected layers, where some convolutional layers construct ensembles of inception modules with 1×1 reduction layers followed by 3×3 convolutional layers. YOLOv2 [247] represents an improved version of YOLO which adopts several impressive strategies, such as Batch Normalization, anchor boxes learned via dimension cluster, and multi-scale training, to improve its performance. In [247], YOLO9000 is also introduced. It can detect over 9000 object categories in real-time using a joint optimization method to train simultaneously on an ImageNet classification dataset and a COCO detection dataset with WordTree in order to combine data from multiple sources. In this way, YOLO9000 can be used to perform weakly supervised detection, i.e. detecting object classes that do not have bounding box annotations.

As can be understood from the previous discussion, YOLO has many benefits. Since the region proposal generation stage is completely dropped, this detector is using a smaller set of candidate regions, only 98 per image, with respect to the 2000 proposals of the selective search method. On the other hand, YOLO makes more localization errors than Faster R-CNN, due to the coarse division of bounding box location, scale, and aspect ratio that leads the detector to fail in localizing some objects, especially small ones. To overcome the difficulties, **SSD** (Single Shot Detector) was introduced in [194]. Given a specific feature map, instead of using fixed grids as in YOLO, SSD,

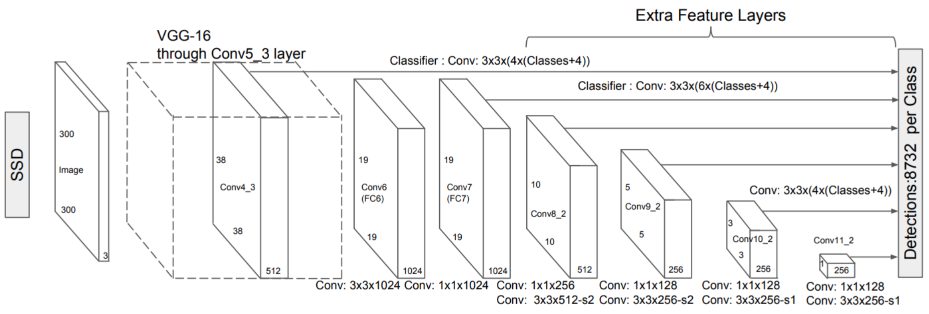


Figure 2.34: SSD architecture. Source: original paper [194].

inspired by MultiBox [79], RPN [249] and multi-scale representation [18], takes advantage of a set of default anchor boxes with different aspect ratios and scales in order to discretize the output space of bounding boxes. As described in Figure 2.34, the architecture of SSD can be divided mainly in three parts: (i) a *base net*, i.e. a CNN as VGG-16 [289], the one used in the original paper [194], providing the low-level feature maps; (ii) some *auxiliary convolutional layers*, namely extra feature layers added to extract higher-level feature maps; (iii) two *predictors*, responsible for localizing and identifying objects in these feature maps. Hence, given in input a RGB image[34] $\mathbf{x}$, it is initially processed by a CNN to extract low-level features of the objects. As discussed in Section 2.2, the last layers of a CNN are specialized in the classification and labeling of objects in a picture. Thus, since now additional steps are needed to extract the region of interests to be classified, the fully-connected layers at the end of the CNN are converted into convolutional layers[35] by exploiting the equivalence existing between these two types of layers (see Section 2.2.5). In particular, these converted convolutional layers are numerous and large in size, and thus need to be modified in order to reduce their number and the size of each filter. A useful way to do this is by subsampling parameters, e.g. picking every m-th parameter along a particular dimension, as happens in decimation. The modified CNN obtained is then followed by the auxiliary layers, which are simple convolutional blocks providing additional feature maps each progressively smaller than the last.

At this point, to define the region proposals, we need to introduce priors. As stated in [194], they are applied to several low- and high-level feature maps: referring to the notation used in Figure 2.34, the output of an intermediate layer of the base net, $conv4\_3$, the output of the modified base net, $conv7$, the output of each auxiliary convolutional block, $conv8\_2$, $conv9\_2$, $conv10\_2$, $conv11\_2$. Following the original paper [194], we use different anchor boxes, with the convention that larger feature maps, i.e. $conv4\_3$, $conv7$, have priors with smaller scales and are therefore ideal for detecting smaller objects. Table 2.1 summarizes all the priors' scales and aspect ratios used for the several feature

[34]The size of the input image is $300 \times 300 \times 3$ for SSD-300 and $512 \times 512 \times 3$ for SSD-512, as stated in the original paper [194].

[35]Following [194], we are converting all the fully-connected layers in the classification part, $FC6$, $FC7$, into convolutional layers, $conv6$, $conv7$, except for the last layer, $FC8$, which is completely tossed away, since it is the one providing the final classification vector. The labels of the layers refer to the notation used in the original paper [194] and present in Figure 2.34.

maps. It can also be pointed out that we are using 8732 priors in total, where this number is obtained by summing up the number of reference boxes defined in each pixel of each feature map. Now, given these priors, for each of them, hence at each location in each feature map, we want to predict the offsets for a predicted bounding box and n_{class} scores for it, namely the probability an object in a region proposal belongs to a precise class of the dataset (including also the background category). In

Table 2.1: Priors used in the original implementation. As can be seen there are a total of 8732 priors defined for the SSD300.

Feature map from	Feature map dimensions	Prior scale	Aspect ratios	Number of priors per positions	Total Number of priors
*conv*4_3	38×38	0.1	1:1, 2:1, 1:2 + an extra prior	4	5776
*conv*7	19×19	0.2	1:1, 2:1, 1:2, 3:1, 1:3 + an extra prior	6	2166
*conv*8_2	10×10	0.375	1:1, 2:1, 1:2, 3:1, 1:3 + an extra prior	6	600
*conv*9_2	5×5	0.55	1:1, 2:1, 1:2, 3:1, 1:3 + an extra prior	6	150
*conv*10_2	3×3	0.725	1:1, 2:1, 1:2 + an extra prior	4	36
*conv*11_2	1×1	0.9	1:1, 2:1, 1:2 + an extra prior	4	4
Grand Total	-	-	-	-	8732 priors

order to solve this task, two predictors are inserted at the end of SSD: one for localization predictions and one for class predictions. In both cases, this predictor corresponds to a convolutional layer with a 3×3 kernel. For the localization case we have $4 \times n^{(\ell)}_{\text{priors}}$ channels, where 4 is the number of the offsets and $n^{(\ell)}_{\text{priors}}$ the number of priors for each location in that feature map ℓ, whereas for the class classifier the filter has $n_{\text{class}} \times n^{(\ell)}_{\text{priors}}$ channels, since we are predicting a set of n_{class} scores for each prior at each location in the ℓ-th feature map. Hence, since we have in total 8732 priors, the final outputs will be reshaped in the form 8732×4 for the localization layer and $8732 \times n_{\text{class}}$ for the classification layer.
To train the model for solving these two different tasks (bounding box localizations and class scores), we need to introduce a loss function $\mathcal{L}$, called **Multibox loss** [194], that extended the one presented in [79, 300] to handle multiple object categories. It thus takes into account both types of predictions through a weighted sum of the two different losses connected:

$$\mathcal{L} = \frac{1}{N_{\text{pos}}} \left(\mathcal{L}_{\text{cls}} + \alpha \mathcal{L}_{\text{loc}} \right), \tag{2.53}$$

where α is a learnable parameter, set to 1 in the original paper [194], whereas N_{pos} is the number of positive matches, about which we will talk about in more detail now. To understand how the localization loss $\mathcal{L}_{\text{loc}}$ and the classification loss $\mathcal{L}_{\text{cls}}$ are defined, we need to set up a way to identify when we have positive and negative matches between priors and ground truth boxes. Hence, first of all, the IoU between each anchor box and the ground truth bounding boxes are computed. Then, all the priors with IoU greater than 0.5 are positive matches, i.e. they contain an object, whereas, if IoU < 0.5, we have negative matches, with background as a related label. Now, the localization loss is computed only on the positive matches, since there are no ground truth coordinates for the negative ones. Hence, in order to understand how accurately we regress positively matched predicted boxes to the corresponding ground truth coordinates, we use as $\mathcal{L}_{\text{loc}}$ the averaged smooth L^1 function, introduced in Equation (2.44):

$$\mathcal{L}_{\text{loc}}(\{\hat{\mathbf{y}}_{\text{loc}}\}, \{\mathbf{y}_{\text{loc}}\}) = \frac{1}{N_{\text{pos}}} \sum_{i \in \mathcal{P}} \sum_{m \in C} [IoU \geq 0.5] smooth_{L^1}(\hat{y}^i_{\text{loc},m} - g^i_{\text{loc},m}), \tag{2.54}$$

where n_{pos} is the number of positive region proposals, i.e. the cardinality of the set $\mathcal{P}$ containing all the indexes of these positive priors, whereas $C = \{c_{q_1}, c_{q_2}, w, h\}$ are the coordinates of the offsets. It can be pointed out that we are not using the coordinates of the ground truth box, but the coordinates of the offset with respect to the prior i under consideration, i.e. a vector g_{loc} defined as in Equation (2.39). Then, $[IoU \geq 0.5]$ represents the Iverson bracket indicator for matching the i-th default box to the ground truth box, defined in Equation (2.45).
To define the classification loss, we need before to introduce the notion of *Hard Negative Mining* [194, 36]. As discussed above, we have associated a ground truth box and the corresponding ground truth label to each prior, if the match between them is positive. Now, after the matching step, a great number of default boxes will be negative and thus characterized by having a 0 label, namely they do not contain an object. This happens especially when the number of possible priors is large. Hence, this introduces a significant imbalance between the positive and negative training examples and thus leads to a model that is less likely to detect objects because, more often than not, it is taught to detect the background class. In order to overcome this problem, we need to limit the number of negative matches that will be evaluated in the loss function by using only the *hard negatives*, namely those predictions where the model found it hardest to recognize that there are no objects. In particular, these correspond to the negative examples with the highest confidence loss for each default box. The confidence loss is thus simply the sum of the Cross-Entropy losses among the positive and hard negative matches, i.e. the softmax loss over multiple classes confidences defined in Equation (2.30):

$$\begin{aligned}\mathcal{L}_{\text{cls}}(\{\hat{\mathbf{y}}_{\text{cls}}\}, \{\mathbf{y}_{\text{cls}}\}) = &- \sum_{i \in \mathcal{P}} \sum_{k=1}^{n_{\text{class}}} [IoU \geq 0.5] y^i_{\text{cls},k} \log \left(\frac{\exp(\hat{y}^i_{\text{cls},k})}{\sum_{m=1}^{n_{\text{class}}} \exp(\hat{y}^i_{\text{cls},m})} \right) \\ &- \sum_{j \in \mathcal{HN}} \log \left(\frac{\exp(\hat{y}^j_{\text{cls},0})}{\sum_{m=1}^{n_{\text{class}}} \exp(\hat{y}^i_{\text{cls},m})} \right),\end{aligned} \tag{2.55}$$

where $\mathcal{HN}$ represents the set of indexes that corresponds to the hard negative samples.
Once the SSD has been trained, it can be applied to new images to solve the problem of object detection. As stated before, we have 8732 boxes, thus it is needed to define a criterion to eliminate some of these proposals, especially the one that corresponds to the same object. Therefore, we need to discard boxes that are overlapping, which can lead to the detection of more objects than present, since some of them are counted more than once. In order to do this, NMS, described in Section 2.4.5, is applied, producing in this way a single box for each object in the image as final detection.
Despite its speed and accuracy, SSD presents some drawbacks connected with the detection of smaller objects. The presence of shallow layers may not generate enough high-level features to do prediction for small objects. For this reason, there is a need of introducing data augmentation and thus to have a large number of data for training purposes.

CHAPTER 3 A Reduced Order Approach for Artificial Neural Networks

3.1 Introduction

The previous chapter has introduced the field of computer vision by presenting two applications: image recognition and object detection. The core of the aforementioned discussion was in particular Convolutional Neural Networks (CNNs), a family of models designed to solve these tasks. Even if CNNs are widely used for several applications in the academic and industrial fields, most research on CNNs does not consider the possible limitations that can be encountered running these models on embedded systems or more in general on low-cost hardware. In particular, when it comes to having a practical application in embedded devices, deep neural networks should have fast and efficient architectures, providing real-time predictions. Typical benchmark CNNs presented in Section 2.3.2, such as ResNet and VGGNet, are characterized by millions of parameters, leading to several implications when running models in practice, especially for the possible long training time [99, 299] and for the required storage space. For this reason, it emerges the need to develop lightweight versions of the aforementioned CNNs or manually designed CNNs for this purpose. When dealing with the development of highly optimized architectures or convolutional layers, we have to take into account the necessity of having resulting models as accurate as the benchmark ones, but requiring less computational effort and less storage space.
In recent years, several methods have been employed to create lightweight Artificial Neural Networks (ANNs), in particular CNNs and object detectors. A possible approach consists in performing model compression on an already trained model to decrease the memory footprint, using for example input resizing and network pruning [110, 196, 192, 92, 29], low-rank matrix, and tensor factorization [266, 336, 227], parameter quantization [56, 69]. Therefore, starting from the image recognition and object detection architectures presented in Section 2.3.2 and in Section 2.4.7, we can construct the compressed versions of them by employing the aforementioned techniques. On the other hand, a branch of research in this area has focused on manually designing convolutional layers, and thus CNN architectures, for memory-constrained systems. As described in Section 2.3.2, some example are represented by MobileNet [138], ShuffleNet [335], and SqueezeNet [141]. These hardware-efficient CNNs can then be employed as the backbone for object detector's implementation, leading in this way to light-weight deep neural networks with reasonable performances [270, 138, 219, 296, 186, 324, 148].
In our works [210, 211, 212], we have proposed another type of approach than those listed here. The core of our idea lies in well-established techniques, widely employed in the context of Reduced Order Modeling (ROM) [21, 22, 23, 261, 267, 260, 306, 262, 325, 235]: Proper Orthogonal Decomposition (POD) [126, 22, 306, 262] and Active Subspaces (AS) [51, 53]. Our reduced network is thus constructed from the original model structure by retaining only a certain number of layers, responsible to detect the important features of the objects. The remaining part is then substituted with a reduction layer, compressing the high-dimensional feature maps into low-dimensional ones, followed by an input-output mapping, providing the final prediction of the network. The resulting model is thus characterized by a decreased number of parameters, chosen through a smart selection procedure,

allowing us to reduce the required resources and the computing time to infer the model.
In this chapter we will thus describe our general approach for the reduction of ANNs [210, 211, 212]. Section 3.3 will deal exactly with the explanation of the reduction technique proposed for a general ANN, employing the numerical tools, such as AS, POD, and Polynomial Chaos Expansion (PCE), introduced in Section 3.2.1. In Section 3.4 we will then apply this dimensionality reduction method to a CNN, and in particular to VGG-16 to solve the image recognition task for CIFAR-10 and for a custom dataset, connected with the collaboration with *Electrolux Professional*. Section 3.5 extends then the proposed reduction procedure to more complex architectures, like the one that deals with the problem of object detection. We will then describe also in this case a practical application of this method to SSD-type architectures, employing PASCAL VOC as the dataset.

3.2 Numerical Tools

We introduce in this section all the techniques employed for the reduction of the network, to make it easier to understand the framework in Section 3.3.

3.2.1 Dimensionality Reduction Techniques

The subsection is devoted to an algorithmic overview of the reduction methods tested within this contribution, the Active Subspaces (AS) property and the Proper Orthogonal Decomposition (POD). Widely employed in the ROM [21, 22, 23, 126, 261, 267, 260, 306] community, such techniques are used to reduce the dimensionality of the output for the intermediate layer, e.g. high-dimensional convolutive features, but we postpone to the next sections the details. We just specify that, even if in this thesis, such as in [210], we have focused on AS and POD, the proposed framework is generic, allowing in principle to replace these two with other reduction techniques.

Active Subspaces

Active Subspaces [51, 53] method is a reduction tool used to identify important directions in the parameter space by exploiting the gradients of a function of interest. Such information allows applying a rotational transformation to the domain in order to obtain, in the end, an approximation of the original function in a lower dimension. As discussed in [308, 306, 66, 105, 52, 262, 67, 307], its application has been proven successful in several parametrized engineering models.
We now briefly review the process of finding active subspaces of a scalar function g, depending on the inputs $\boldsymbol{\mu}$. Let $\boldsymbol{\mu} = [\mu_1 \dots \mu_n]^T \in \mathbb{R}^n$ represent a n-dimensional variable characterized by a probability density function $\rho(\boldsymbol{\mu})$, and let $g : \mathbb{R}^n \to \mathbb{R}$ be the function of interest. We are assuming here that g is scalar and continuous, but there exists also a vector-valued extension [254, 330]. Starting from this, we can construct an uncentered covariance matrix $\mathbf{C}$ of the gradient of g by considering the average of the outer product of the gradient with itself:

$$\mathbf{C} = \mathbb{E}[\nabla g(\boldsymbol{\mu}) \nabla g(\boldsymbol{\mu})^T] = \int (\nabla_{\boldsymbol{\mu}} g)(\nabla_{\boldsymbol{\mu}} g)^T \rho \, d\boldsymbol{\mu}, \tag{3.1}$$

where the symbol $\mathbb{E}[\cdot]$ denotes the expected value, and $\nabla_{\boldsymbol{\mu}} g \equiv \nabla g(\boldsymbol{\mu})$. We suppose that the gradients are computed during the simulation, otherwise, if not provided, they can be approximated with different techniques such as local linear models, global models, finite difference, or Gaussian process [3, 321], for example.
It can be noted that $\mathbf{C}$ is real and symmetric, hence it admits the following eigenvalue decomposition [136]:

$$\mathbf{C} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^T, \qquad \boldsymbol{\Lambda} = \text{diag}(\lambda_1, \dots, \lambda_n), \quad \lambda_1 \geq \dots \geq \lambda_n \geq 0, \tag{3.2}$$

where $\mathbf{V}$ is the $n \times n$ orthogonal matrix whose columns $\{\mathbf{v}^1, \ldots, \mathbf{v}^n\}$ are the normalized eigenvectors of C, whereas $\mathbf{\Lambda}$ is a diagonal matrix containing the corresponding non-negative eigenvalues λ_i, for $i = 1, \ldots, n$, arranged in descending order.
Let now n_{AS} be an integer[36], such that $n_{\mathrm{AS}} < n$, we can decompose the two matrices $\mathbf{V}$ and $\mathbf{\Lambda}$ as:

$$\mathbf{\Lambda} = \begin{bmatrix} \mathbf{\Lambda}_1 & \\ & \mathbf{\Lambda}_2 \end{bmatrix}, \qquad \mathbf{V} = [\mathbf{V}_1 \ \ \mathbf{V}_2], \qquad \mathbf{V}_1 \in \mathbb{R}^{n \times n_{\mathrm{AS}}}, \ \ \mathbf{V}_2 \in \mathbb{R}^{n \times (n - n_{\mathrm{AS}})}. \tag{3.3}$$

The space spanned by $\mathbf{V}_1$ columns is called the *active subspace* of dimension $n_{\mathrm{AS}} < n$, whereas the *inactive subspace* is described as the range of the remaining eigenvectors in $\mathbf{V}_2$. Once we have defined these spaces, the input $\boldsymbol{\mu} \in \mathbb{R}^n$ can be reduced to a low-dimensional vector $\tilde{\boldsymbol{\mu}}_1 \in \mathbb{R}^{n_{\mathrm{AS}}}$ using $\mathbf{V}_1$ as projection map. To be more precise, any $\boldsymbol{\mu} \in \mathbb{R}^n$ can be expressed as the sum of two addends, involving the active and inactive subspaces, using the decomposition in Equation (3.3) and the properties of $\mathbf{V}$:

$$\boldsymbol{\mu} = \mathbf{V}\mathbf{V}^T\boldsymbol{\mu} = \mathbf{V}_1\mathbf{V}_1^T\boldsymbol{\mu} + \mathbf{V}_2\mathbf{V}_2^T\boldsymbol{\mu} = \mathbf{V}_1\tilde{\boldsymbol{\mu}}_1 + \mathbf{V}_2\tilde{\boldsymbol{\mu}}_2, \tag{3.4}$$

where the two new variables $\tilde{\boldsymbol{\mu}}_1$ and $\tilde{\boldsymbol{\mu}}_2$ are the *active* and *inactive variable* respectively:

$$\tilde{\boldsymbol{\mu}}_1 = \mathbf{V}_1^T\boldsymbol{\mu} \in \mathbb{R}^{n_{\mathrm{AS}}}, \qquad \tilde{\boldsymbol{\mu}}_2 = \mathbf{V}_2^T\boldsymbol{\mu} \in \mathbb{R}^{n - n_{\mathrm{AS}}}. \tag{3.5}$$

For the actual computations of the AS we have used the open-source Python package called ATHENA [255], and in particular the *Frequent Directions method* [90, 57], as it will be described in Section 3.3.2.

Proper Orthogonal Decomposition

In this section, we are going to describe the Proper Orthogonal Decomposition (POD) approach of ROM [126], commonly employed for decreasing the number of degrees of freedom of a parametric system in various applications [38, 39, 200, 64, 66, 65, 306, 262, 123, 76, 236, 286, 127].
Let $\mathbf{S} = [\mathbf{u}^1 \ldots \mathbf{u}^{n_S}]$ be a matrix composed of n_S full order system outputs $\mathbf{u}^i \in \mathbb{R}^n$, called *snapshots matrix*. The aim of POD is to describe these collected solutions as a linear combination of a few main structures, the POD modes, and thus project them onto a low dimensional space spanned by these modes. To calculate the POD modes, we need to compute the Singular Value Decomposition (SVD) of the snapshots matrix S:

$$\mathbf{S} = \mathbf{\Psi}\mathbf{\Sigma}\mathbf{\Theta}^T, \tag{3.6}$$

where the left-singular vectors, i.e. the columns of the unitary matrix $\mathbf{\Psi}$, are the POD modes, and the diagonal matrix Σ contains the corresponding singular values in decreasing order.
Let now n_{POD} be an integer, such that $n_{\mathrm{POD}} < n$. We can define a projection matrix from a space of dimension n to one of dimension n_{POD} by selecting the first modes, i.e. the first n_{POD} column of $\mathbf{\Psi}$. In this way, by discarding the last $n - n_{\mathrm{POD}}$ columns, we are retaining only the most energetic modes, defining a reduced space into which we project the high-fidelity solutions. Technically speaking, let $\mathbf{\Psi}_{n_{\mathrm{POD}}}$ be the aforementioned projection matrix. By considering the multiplication between $\mathbf{\Psi}_{n_{\mathrm{POD}}}$ and the snapshots matrix §, we can obtain a reduced version of S:

$$\mathbf{S}^{\mathrm{POD}} = \mathbf{\Psi}_{n_{\mathrm{POD}}}^T\mathbf{S}. \tag{3.7}$$

where the columns of $\mathbf{S}^{\mathrm{POD}}$ represent the reduced snapshot $\tilde{\mathbf{u}}^i \in \mathbb{R}^{n_{\mathrm{POD}}}$, with $\tilde{\mathbf{u}}^i = \mathbf{\Psi}_{n_{\mathrm{POD}}}^T\mathbf{u}^i$.

[36]As described in [57], a possible way to choose the reduction parameter n_{AS} is represented by the *number of active neurons* in layer l, $n_{l,AS}$. Given Λ, $n_{l,AS}$, for any layer index $1 \leq l \leq L+1$, is defined in the following way:

$$n_{l,AS} = argmin\left\{i : \frac{\lambda_1 + \cdots + \lambda_i}{\lambda_1 + \cdots + \lambda_{n_l}} \geq 1 - \epsilon\right\}$$

where $\epsilon > 0$ is a defined threshold.

3.2.2 Input–output Mapping

Once the outputs of the intermediate layer are dimensionally reduced, we need to correlate the latter to the final output of the original network, e.g. the belonging classes in an image identification problem. As described in [210, 57], an input–output mapping is thus built starting from the output of the reduction layer $\mathbf{z}$, employing for approximating this map two methods: the Polynomial Chaos Expansion (PCE) [326, 320] and fully-connected Feedforward Neural Networks (FNNs) [85].

Polynomial Chaos Expansion

The Polynomial Chaos Expansion theory was initially proposed by Wiener in [320], showing that a real-valued random variable $X : \mathbb{R}^R \longrightarrow \mathbb{R}$ can be decomposed in the following way:

$$X(\boldsymbol{\xi}) = \sum_{j=0}^{\infty} c_j \phi_j(\boldsymbol{\xi}), \tag{3.8}$$

hence as an infinite sum of multivariate orthogonal polynomials ϕ_j weighted by unknown deterministic coefficients c_j [146]. The vector $\boldsymbol{\xi} = (\xi_1, \dots, \xi_R)$ represents then a multi-dimensional random vector, where each element is associated with uncertain input parameters.
Starting from Equation (3.8), we can derive a finite approximation of this infinite sum by truncating it at the $(P+1)$-th term, with P being a finite integer:

$$X(\boldsymbol{\xi}) \approx \sum_{j=0}^{P} c_j \phi_j(\boldsymbol{\xi}), \tag{3.9}$$

where the number of unknown coefficients in this summation is given by $P + 1 = \frac{(p+R)!}{p!R!}$ [89], and p indicates the degree of the polynomial we are considering in the R-dimensional space.
Assuming that the parameters $\xi_1, \dots, \xi_R$ are independent, we can decompose $\phi_j(\boldsymbol{\xi})$ into products of one-dimensional functions [88]:

$$\phi_j(\boldsymbol{\xi}) = \phi_j(\xi_1, \dots, \xi_R) = \prod_{k=1}^{R} \phi_k^{d_k}(\xi_k), \quad j = 0, \dots, P, \quad d_k = 0, \dots, p, \quad s.t. \sum_{k=1}^{R} d_k \leq p. \tag{3.10}$$

Now, in order to determine the PCE, we need to find out the polynomial chaos expansion coefficients c_j for $j = 0, \dots, P$, and the one-dimensional orthogonal polynomial $\phi_k^{d_k}$, $k = 1, \dots, R$, of degree d_k. Based on the work of Askey and Wilson [15], we can provide the orthogonal polynomials for different distributions. One of the possible choices is represented by the Gaussian distribution with the related Hermite polynomials. On the other hand, the estimation of the coefficients of PCE can be carried out in different ways [293]: following a projection method based on the orthogonality of the polynomials or following a regression method, that is the one we are going to describe.
To determine the coefficients c_j, we need thus to solve the following minimization problem:

$$\mathbf{c} = \operatorname*{arg\,min}_{\mathbf{c}^* \in \mathbb{R}^P} \frac{1}{n_{\text{PCE}}} \sum_{i=1}^{n_{\text{PCE}}} \left(\hat{X} - \sum_{j=0}^{P} c_j^* \phi_j(\boldsymbol{\xi}^i) \right), \tag{3.11}$$

where n_{PCE} indicates the total number of realizations of the input vector we are considering, whereas $\hat{X}$ represents the real output of the model. In order to solve Equation (3.11), we need to consider the matrix $\boldsymbol{\Phi}$ defined as:

$$\boldsymbol{\Phi} = \begin{pmatrix} \phi_0(\boldsymbol{\xi}^1) & \phi_1(\boldsymbol{\xi}^1) & \cdots & \phi_P(\boldsymbol{\xi}^1) \\ \phi_0(\boldsymbol{\xi}^2) & \phi_1(\boldsymbol{\xi}^2) & \cdots & \phi_P(\boldsymbol{\xi}^2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\boldsymbol{\xi}^{n_{\text{PCE}}}) & \phi_1(\boldsymbol{\xi}^{n_{\text{PCE}}}) & \cdots & \phi_P(\boldsymbol{\xi}^{n_{\text{PCE}}}) \end{pmatrix}. \tag{3.12}$$

Thus, the solution of Equation (3.11) is computed by a least-square optimization [35]:

$$\mathbf{c} = (\mathbf{\Phi}^T \mathbf{\Phi})^{-1} \mathbf{\Phi}^T \hat{X}, \tag{3.13}$$

where, if the matrix $\mathbf{\Phi}^T \mathbf{\Phi}$ is ill-conditioned, as it may happen, the singular value decomposition method should be employed.

Fully-Connected Feedforward Neural Network

A fully-connected FNN is an ANN characterized by having forward fully connections, i.e. each neuron in a layer is connected with all the neurons in the next layer. Since a detailed description of this type of ANNs has been carried out in Section 1.5, we recall here only the most important expressions connected with this model.
Let $\tilde{\mathbf{x}} \in \mathbb{R}^{n_{\text{in}}}$ be the input vector and M the total number of hidden layers of the FNN. The output vector $\mathbf{y}^{\text{FNN}} \in \mathbb{R}^{n_{\text{out}}}$ is hence obtained using Equation (1.14):

$$\begin{aligned} y_j^{\text{FNN}} &= \sigma\left(\sum_{i=1}^{n_M} w_{ji}^{(M+1)} \tilde{x}_i^{(M)}\right) = \sigma\left(\sum_{i=1}^{n_M} w_{ji}^{(M+1)} \left(\sigma\left(\sum_{q=1}^{n_{M-1}} w_{iq}^{(M)} \tilde{x}_q^{(M-1)}\right)\right)\right) = \cdots = \\ &= \sigma\left(\sum_{i=1}^{n_M} w_{ji}^{(M+1)} \left(\sigma\left(\sum_{q=1}^{n_{M-1}} w_{iq}^{(M)} \left(\sigma\left(\dots\left(\sigma\left(\sum_{k=1}^{n_{in}} w_{sk}^{(1)} \tilde{x}_k\right)\right)\right)\right)\right)\right)\right), \qquad j = 1, \dots, n_{\text{out}}, \end{aligned} \tag{3.14}$$

where n_m, $m = 1, \dots, M$, represents the number of neurons in layer m, whereas n_{in} and n_{out} are the neurons in the input and output layers respectively, $W^m = (w_{ki}^{(m)})_{ki}$, $k = 1, \dots, n_m$, $i = 1, \dots, n_{m-1}$ indicates the weight matrix related to layer m. To find the optimal values of the weights leading to optimal performances, the backpropagation algorithm should then be employed, as discussed in Section 1.5.2.

3.3 Reduced Artificial Neural Networks

Starting from the idea explored in [57], we propose a general framework to construct a reduced version of a ANN, based on the techniques described in Section 3.2. We provide thus in this section the rigorous description of the proposed reduction method for a generic ANN [210, 57, 212], on which we only make the assumption on its depth, i.e. on the number of layers. Let $\mathcal{ANN} : \mathbb{R}^{n_{\text{in}}} \longrightarrow \mathbb{R}^{n_{\text{out}}}$ be the original ANN composed by L hidden layers[37] and then consider the train dataset $\mathcal{D}_{\text{train}} = \{\mathbf{x}^{(0),j}, \mathbf{y}^j\}_{j=1}^{n_{\text{train}}}$ made of n_{train} input samples and corresponding expected outputs. We denote with $\{\hat{\mathbf{y}}^j\}_{j=1}^{n_{\text{train}}}$ the predicted outputs of the $\mathcal{ANN}$ for each element of $\mathcal{D}_{\text{train}}$. As discussed in Section 1.4, an ANN can be described as composition of functions $f_j : \mathbb{R}^{n_{j-1}} \longrightarrow \mathbb{R}^{n_j}$ for $j = 1, \dots, L+1$, representing the different layers of the network, e.g. convolutional, fully connected, batch-normalization, ReLU, pooling layers:

$$\mathcal{ANN} = f_{L+1} \circ f_L \circ \cdots \circ f_1. \tag{3.15}$$

Our method starts from the original structure of the ANN and provides a reduced version of it following the procedure described in Algorithm 4 composed of the following three steps [210, 212]:

1. *Network Splitting* (Section 3.3.1): This first step deal with the detection of the information we want to retain and discard from the original model. The $\mathcal{ANN}$ is thus split into two parts, a pre-model and a post-model, determined by the cut-off layer l.

[37]In this Chapter, we are employing the same notation introduced in Section 1.4. Hence, also in this case, if we have an ANN with L hidden layers, the net is composed of $L + 1$ layers.

2. *Reduction Layer* (Section 3.3.2): The second step deals with the dimensionality reduction of the pre-model output, which usually lies in a high-dimensional space. Two ROM methods, AS and POD, are employed for this purpose.
3. *Input-Output Mapping* (Section 3.3.3): Once the pre-model output has been reduced, we need a mapping to link the output of the reduction layer with the final output of the network. In this case, PCE and fully-connected FNNs have been used to construct this input–output map.

We are now going to describe in detail each of these modules by employing the numerical tools introduced in Section 3.2.

Algorithm 4 Pseudo-code for the construction of the reduced Artificial Neural Network

```
Inputs:
  • a dataset with $n_{\text{train}}$ input samples $\mathcal{D}_{\text{train}} = \{\mathbf{x}^{(0),j}, \mathbf{y}^j\}_{j=1}^{n_{\text{train}}}$;
  • an artificial neural network $\mathcal{ANN}$;
  • $\{\hat{\mathbf{y}}^j\}_{j=1}^{n_{\text{train}}}$ real output of the $\mathcal{ANN}$;
  • reduced dimension $r$;
  • index of the cut-off layer $l$.
1: $\mathcal{ANN}^l_{\text{pre}}, \mathcal{ANN}^l_{\text{post}} = \text{splitting\_net}(\mathcal{ANN}, l)$;
2: $\mathbf{x}^{(l)} = \mathcal{ANN}^l_{\text{pre}}(\mathbf{x}^{(0)})$;
3: $\mathbf{z} = \text{reduce}(\mathbf{x}^{(l)}, r)$;
4: $\tilde{\mathbf{y}} = \text{input\_output\_map}(\mathbf{z}, \hat{\mathbf{y}})$;
5: Training of the constructed reduced net.
Output: Reduced Net $\mathcal{ANN}^{\text{red}}$.
```

3.3.1 Splitting Network

Let the index l denote the *cut-off layer*, namely the layer at which we are cutting the net. The original network $\mathcal{ANN} : \mathbb{R}^{n_{\text{in}}} \to \mathbb{R}^{n_{\text{out}}}$ is thus split in two different parts such that the first l layers constitutes the *pre-model* while the last $L + 1 - l$ layers form the so-called *post-model*. Considering the description of the network as a composition of functions (3.15), we can formally define the pre- and the post-model as:

$$\mathcal{ANN}^l_{\text{pre}} = f_l \circ f_{l-1} \circ \cdots \circ f_1, \qquad \mathcal{ANN}^l_{\text{post}} = f_{L+1} \circ f_L \circ \cdots \circ f_{l+1}. \tag{3.16}$$

Therefore, the original model can be rewritten as a composition of the post-model with the pre-model:

$$\mathcal{ANN}(\mathbf{x}^{(0)}) = \mathcal{ANN}^l_{\text{post}}(\mathcal{ANN}^l_{\text{pre}}(\mathbf{x}^{(0)})), \tag{3.17}$$

for any $1 \le l \le L$ and for any $\mathbf{x}^{(0)} \in \mathcal{D}_{\text{train}}$. The reduction of the network effectively happens by copying the pre-model from the original net and approximating the post-model with the algorithm we are going to describe. It is important to specify that the cut-off layer l is the only parameter of this initial step, and it plays an important role in the final outcome. This index indeed defines how many layers of the original network are kept in the reduced architecture, by simply checking how much information of the original network we are discarding. As done in [57], it is chosen empirically based on considerations about the network and the dataset at hand, balancing the final accuracy and the compression ratio.

3.3.2 Reduction Layer

Fixed a cut-off layer l, consider the pre-model $\mathcal{ANN}^l_{\text{pre}}$ defined in Equation (3.17), and compute its output at input $\mathbf{x}^{(0)} \in \mathcal{D}_{\text{train}}$:

$$\mathbf{x}^{(l)} = \mathcal{ANN}^l_{\text{pre}}(\mathbf{x}^{(0)}). \tag{3.18}$$

As introduced previously, the output of the pre-model $\mathbf{x}^{(l)}$ usually lies in a high-dimensional space, thus we aim to project it onto a low-dimensional space of dimension r, with $r < n_l$, using the two reduction techniques introduced in Section 3.2.1:

- **Active Subspaces**: as described in [57, 210, 212] and in Section 3.2.1, we consider as function of interest g_l defined as the composition of the post-model with a chosen loss function for the problem at hand:
$$g_l(\mathbf{x}^{(l)}) = \text{loss}(\mathcal{ANN}^l_{\text{post}}(\mathbf{x}^{(l)})), \tag{3.19}$$
in order to extract the most important directions and determine the projection matrix $\boldsymbol{W}_{\text{proj}}$ used to reduce the pre-model output. Given the pre-model outputs $\{\mathbf{x}^{i,(l)}\}_{i=1}^{n_{\text{train}}}$ computed for each sample of $\mathcal{D}_{\text{train}}$, the empirical covariance matrix $\hat{\mathbf{C}}$ is computed by considering a discrete version of Equation (3.1):
$$\hat{\mathbf{C}} = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \nabla g_l(\mathbf{x}^{i,(l)}) \nabla g_l(\mathbf{x}^{i,(l)})^T. \tag{3.20}$$
Since computing the eigenvalue decomposition of $\hat{\mathbf{C}}$ is computational expensive, to save computational cost, we have employed the memory saving *Frequent Direction method*[38] [90, 57], implemented inside ATHENA [255]. Algorithm 5 describes how the frequent direction

Algorithm 5 A pseudo-code for the computation of the Active Subspace using the frequent direction algorithm

Inputs:
- dataset with n_{train} input samples $\mathcal{D}_{\text{train}} = \{\mathbf{x}^{j,(0)}\}_{j=1}^{n_{\text{train}}}$;
- the pre-model $\mathcal{ANN}^l_{\text{pre}}(\cdot)$;
- a subroutine for computing $\nabla g_l(\mathbf{x})$;
- the dimension of the truncated singular value decomposition r.

```
1: Select first $r$ samples from $\mathcal{D}_{\text{train}}$: $\mathcal{D}^{\text{red}}_{\text{train}} = \{\mathbf{x}^{i,(0)}\}_{i=1}^{r}$;
2: Compute the pre-model output for each sample in $\mathcal{D}^{\text{red}}_{\text{train}}$: $\mathbf{x}^{i,(l)} = \mathcal{ANN}^l_{\text{pre}}(\mathbf{x}^{i,(0)})$;
3: Define $\mathbf{G}_{\text{red}} = [\nabla g_l(\mathbf{x}^{1,(l)}), \dots, \nabla g_l(\mathbf{x}^{r,(l)})]$;
4: for $t = r+1, \dots, n_{\text{train}}$ do
5:     Compute the SVD: $\mathbf{G}_{\text{red}} = \hat{\mathbf{V}}_{\text{red}} \hat{\mathbf{\Lambda}}_{\text{red}} \hat{\mathbf{U}}^T_{\text{red}}$;
6:     Update: $\mathbf{G}_{\text{red}} = \hat{\mathbf{V}}_{\text{red}} \sqrt{\hat{\mathbf{\Lambda}}^2_{\text{red}} - \hat{\lambda}^2_r}$;
7:     Take a new sample: $\mathbf{x}^{t,(l)} = \mathcal{ANN}^l_{\text{pre}}(\mathbf{x}^{t,(0)})$;
8:     Substitute last column of $\hat{\mathbf{G}}_{\text{red}}$ with the gradient of the new sample: $\hat{\mathbf{G}}_{\text{red}}(:, r) = \nabla g_l(\mathbf{x}^{t,(l)})$.
9: end for
```

Output: The projection matrix $\hat{\mathbf{V}}_{\text{red}} \in \mathbb{R}^{n_l \times r}$.

[38]For more details on the Frequent Direction Methods the interest reader can refer to Appendix A.

method is applied for computing the active subspace. Hence, we define the matrix $\hat{\mathbf{G}}$ as:

$$\hat{\mathbf{G}} = [\nabla g_l(\mathbf{x}^{1,(l)}), \dots, \nabla g_l(\mathbf{x}^{n_{\text{train}},(l)})], \tag{3.21}$$

and compute its SVD:

$$\hat{\mathbf{G}} = \hat{\mathbf{V}}\hat{\mathbf{\Lambda}}\hat{\mathbf{U}}^T \in \mathbb{R}^{n_l \times n_{\text{train}}} \qquad \text{with } \hat{\mathbf{\Lambda}} = \text{diag}(\hat{\lambda}_1, \dots, \hat{\lambda}_{n_l}). \tag{3.22}$$

We can point out that the left singular vectors in $\hat{\mathbf{V}}$ approximate the eigenvectors of $\hat{\mathbf{C}}$, whereas the eigenvalues of $\hat{\mathbf{C}}$ are approximated by the singular values of $\hat{\mathbf{G}}$, contained in $\hat{\mathbf{\Lambda}}$, i.e $\mathbf{\Lambda} \approx \hat{\mathbf{\Lambda}}^2$. Then, since we are interesting in computing the r dominant singular value components, instead of computing the complete SVD of $\hat{\mathbf{G}}$ we store only a reduced version of it $\hat{\mathbf{G}}_{\text{red}} \in \mathbb{R}^{n_l \times r}$. This reduced matrix is initialized using the first r columns of $\hat{\mathbf{G}}$, i.e. the gradients of the first r pre-model outputs, and then updated in the following way:

$$\hat{\mathbf{G}}_{\text{red}} = \hat{\mathbf{V}}_{\text{red}}\sqrt{\hat{\mathbf{\Lambda}}^2_{\text{red}} - \hat{\lambda}^2_r}, \tag{3.23}$$

where now the last column of $\hat{\mathbf{G}}_{\text{red}}$ is zero and is then replaced with the gradient vector of a new sample. At the end of the procedure, when we have considered all the samples in $\mathcal{D}_{\text{train}}$, we obtain the projection matrix $\boldsymbol{W}_{\text{proj}}$, corresponding to $\hat{\mathbf{V}}_{\text{red}}$, the output of the frequent direction method summarized in Algorithm 5.

- **Proper Orthogonal Decomposition**: as discussed in [210, 212] and in section 3.2.1, the projection matrix $\boldsymbol{W}_{\text{proj}}$ is computed by exploiting the SVD of the snapshot matrix S, composed by the pre-model outputs for each sample in$\mathcal{D}_{\text{train}}$. Hence, given the reduction parameter r, we can define the projection matrix by considering the first r POD modes.

Therefore, considering the output of the pre-model $\mathbf{x}^{(l)}$, and denoting with $\boldsymbol{W}_{\text{proj}}$ the projection method, obtained using the AS or the POD, we can derive a reduced version $\mathbf{z}$ of $\mathbf{x}^{(l)}$ as:

$$\mathbf{z} = \boldsymbol{W}^T_{\text{proj}}\mathbf{x}^{(l)}, \qquad \text{with } \mathbf{z} \in \mathbb{R}^r, \tag{3.24}$$

with r being the reduction parameter, i.e. the dimension of the low-dimensional space onto which we are projecting our full solution. From the previous relation, it can also be deduced that the reduction layer consists of a simple matrix multiplication between the output of the pre-model $\mathbf{x}^{(l)}$ and the projection matrix $\boldsymbol{W}_{\text{proj}}$, giving in output a reduced tensor $\mathbf{z}$.

3.3.3 Input-Output Mapping

After the pre-model output has been dimensionally reduced and the reduced solution $\mathbf{z}$ has been obtained, the last step of the method deals with the construction of the mapping to correlate $\mathbf{z}$ with the final output of the original network, namely the predicted output $\hat{\mathbf{y}}$. Two different techniques, introduced in Section 3.2.2, have been employed to find that map:

- the **Polynomial Chaos Expansion** introduced in section 3.2.2. As described in equation (3.8), the final output of the network $\hat{\mathbf{y}} = \mathcal{ANN}(\mathbf{x}^{(0)}) \in \mathbb{R}^{n_{\text{out}}}$, i.e. the true response of the model, can be approximated in the following way:

$$\tilde{\mathbf{y}} \approx \sum_{|\boldsymbol{\alpha}|=0}^{p} \mathbf{c}_{\boldsymbol{\alpha}}\phi_{\boldsymbol{\alpha}}(\mathbf{z}), \qquad |\boldsymbol{\alpha}| = \alpha_1 + \dots + \alpha_r, \tag{3.25}$$

where $\phi_{\boldsymbol{\alpha}}(\mathbf{z})$ are the multivariate polynomial functions chosen based on the probability density function ρ associated with $\mathbf{z}$. Therefore, the estimation of coefficients $\mathbf{c}_\alpha$ is carried out by solving the minimization problem (3.11):

$$\min_{c_\alpha} \frac{1}{n_{\text{train}}} \sum_{j=1}^{n_{\text{train}}} \left\| \hat{\mathbf{y}}^j - \sum_{|\boldsymbol{\alpha}|=0}^{p} \mathbf{c}_{\boldsymbol{\alpha}} \phi_{\boldsymbol{\alpha}}(\mathbf{z}^j) \right\|^2 . \tag{3.26}$$

- a **Fully-connected FNN** described in section 3.2.2 and more detailed in Section 1.5. In this case, the output of the reduction layer $\mathbf{z}$ coincides with the network input, and by using Equation (3.14) we obtain that the final output $\tilde{\mathbf{y}}$ of the reduced net[39] is determined by:

$$\tilde{y}_j = \sum_{i=1}^{n_1} w_{ji}^{(2)} z_i^{(1)} = \sum_{i=1}^{n_1} w_{ji}^{(2)} \sigma \left(\sum_{m=1}^{r} w_{im}^{(1)} z_m \right), \qquad j = 1, \dots, n_{\text{out}}, \tag{3.27}$$

  where n_{out} corresponds to the number of neurons in the output layer, e.g. to the categories that compose the dataset under consideration for an image recognition problem. As activation function σ, a possible choice [210, 212] is represented by the *Softplus* function:

$$\text{Softplus}(\mathbf{x}) = \frac{1}{\beta} \log(1 + \exp(\beta \mathbf{x})). \tag{3.28}$$

  Note that in a CNN the classification part is characterized by the presence of a fully-connected FNN, as described in Section 2.2 and in Figure 2.1. Hence, this justifies the introduction of this kind of structure at the end of our reduction network for classification.

3.3.4 Training Phase

Once the reduced version of the network under consideration is constructed, we need to train it. Following [57], for the training phase of the reduced ANN the technique of *knowledge distillation* [130], is used. A knowledge distillation framework contains a large pre-trained *teacher model*, our full network, and a small *student model*, in our case $\mathcal{ANN}^{\text{red}}$. Therefore, the main goal is to train efficiently the student network under the guidance of the teacher network in order to gain a comparable or even superior performance.
Let $\hat{\mathbf{y}}$ be a vector of *logits*, i.e. the output of the last layer in a deep neural network. The probability p_i that the input belongs to the i-th class is given by the softmax function

$$p_i = \frac{exp(\hat{y}_i)}{\sum_{j=0}^{n_{\text{class}}} exp(\hat{y}_j)}. \tag{3.29}$$

As described in [130], a temperature factor T needs to be introduced to control the importance of each target

$$p_i = \frac{exp(\hat{y}_i/T)}{\sum_{j=0}^{n_{\text{class}}} exp(\hat{y}_j/T)}, \tag{3.30}$$

where if $T \to \infty$ all classes have the same probability, whereas if $T \to 0$ the targets p_i become one-hot labels.
First of all, we need to define the *distillation loss*, that matches the logits between the teacher model and the student model. As done in [57], the *response-based knowledge* is used to transfer

[39]Note that in this case the number of hidden layers is set to 1 since, as discussed in Section 3.4.2, we notice that one hidden layer is enough to gain a good level of accuracy (see for example Table 3.3).

the knowledge from the teacher to the student by mimicking the final prediction of the full net. Therefore, in this case the distillation loss [103, 130] is given by:

$$L_D(p(\hat{\mathbf{y}}_t, T), p(\hat{\mathbf{y}}_s, T)) = \mathcal{L}_{\mathrm{KL}}(p(\hat{\mathbf{y}}_t, T), p(\hat{\mathbf{y}}_s, T)), \tag{3.31}$$

where $\hat{\mathbf{y}}_t$ and $\hat{\mathbf{y}}_s$ indicate the logits of the teacher and student networks[40], respectively, whereas $\mathcal{L}_{\mathrm{KL}}$ represents the Kullback-Leibler (KL) divergence loss [157]:

$$\mathcal{L}_{\mathrm{KL}}((p(\hat{\mathbf{y}}_s, T), p(\hat{\mathbf{y}}_t, T)) = T^2 \sum_j p_j(\hat{y}_{t,j}, T) \log \frac{p_j(\hat{y}_{t,j}, T)}{p_j(\hat{y}_{s,j}, T)}. \tag{3.32}$$

The *student loss* is defined as the cross-entropy loss between the ground truth label and the logits of the student network [103]:

$$L_S(\hat{\mathbf{y}}, p(\hat{\mathbf{y}}_s, T)) = \mathcal{L}_{\mathrm{CE}}(\mathbf{y}, p(\hat{\mathbf{y}}_s, T)), \tag{3.33}$$

where $\mathbf{y}$ is a ground truth vector, characterized by having only the component corresponding to the ground truth label on the training sample set to 1 and the others are 0. Then, $\mathcal{L}_{\mathrm{CE}}$ represents the cross entropy loss

$$\mathcal{L}_{\mathrm{CE}}(\mathbf{y}, p(\hat{\mathbf{y}}_s, T)) = \sum_i -y_i \log(p_i(\hat{y}_{s,i}, T)). \tag{3.34}$$

As can be observed, both losses, eq. (3.31) and eq. (3.33), use the same logits of the student model but with different temperatures: $T = \tau > 1$ in the distillation loss, and $T = 1$ in the student loss. Finally, the final loss is a weighted sum between the distillation loss and the student loss:

$$L(\mathbf{x}^{(0)}, W) = \lambda L_D(p(\hat{\mathbf{y}}_t, T = \tau), p(\hat{\mathbf{y}}_s, T = \tau)) + (1 - \lambda) L_S(\mathbf{y}, p(\hat{\mathbf{y}}_s, T = 1)), \tag{3.35}$$

where λ is the regularization parameter, $\mathbf{x}^{(0)}$ is an input vector of the training set, and W are the parameters of the student model.

3.4 A Reduced Approach for Convolutional Neural Networks

This section is dedicated to the application of the dimensionality reduction method proposed to CNNs. First, we provide in Section 3.4.1 an overview of some of the existing techniques to enhance the efficiency of CNNs regarding memory footprint and computation time [259], including the one proposed in Section 3.3. Then, in Section 3.4.2 we describe the practical application of our method to a CNN, VGG-16, to solve the problem of image recognition for the CIFAR-10 dataset and a custom one, namely a dataset constructed for the collaboration with *Electrolux Professional.*

3.4.1 Reduction Strategies for Convolutional Neural Networks

The application of Deep Neural Networks in several engineering fields and in particular in embedded systems with particular space constraints has led in recent years to the need for methods to compress and speed up the inference in CNNs. Among the several existing techniques developed for this purpose [259], we are now going to briefly describe some of them, such as *network pruning*, *low-rank factorization*, *parameter quantization*, *manual architecture design*, and *neural architecture search.*

[40] Note that in our case $\hat{\mathbf{y}}_s$ corresponds to the output of the reduced network $\tilde{\mathbf{y}}$.

Network Pruning The main goal of network pruning approaches is to prune redundant channels in the weight matrices of a trained net by setting a substantial number of these parameters to zero [110, 196, 259]. As a consequence, we achieve the property of parameter sparsity to enhance resource-efficiency for a CNN, leading to an accelerated and compressed model. There exist different sparsity constraints to select which channels have to be pruned, such as weight magnitude criterion [86, 111, 183], based on the absolute value (or magnitude) of weights, and gradient magnitude pruning [176, 27], determined from the product between gradients over a minibatch of training data and the corresponding weight of each parameter. Then, we can distinguish between two different approaches depending on the structure to be pruned: *unstructured pruning* and *structured pruning*. In the unstructured case, individual weights, regardless of their location in a weight tensor, are set to zero[110, 111, 176, 114]. Among the approaches belonging to this category, we mention the *optimal brain damage algorithm* [176] and the *optimal brain surgeon algorithm* [114], where, starting from a pre-trained network, weights that cause the least increase in loss function are pruned. Even if we are increasing the sparsity of the tensors, there are some drawbacks connected with this method: it yields practical efficiency improvements only for very high sparsity and, since most frameworks and hardware cannot accelerate sparse matrices computation, this does not have an impact on the final cost of the network. To overcome this problem is thus needed an approach that alters directly the very architecture of the network, and this is represented by structured pruning [318, 183, 205, 199]. In this case, chosen weight structures are set to zero, namely weights of entire convolution filters or of a kernel row/column are removed instead of just pruning connections. This leads to networks that are lighter to store, due to fewer parameters, and that require fewer computations and less memory during runtime. On the other hand, great attention has to be placed when pruning an entire kernel or removing a row/column, because this affects the ensuing layer and the output dimensions. Then, although all these pruning methods achieve a high level of parameter sparsity in the model, they are usually iterative procedures that require a fine-tuning of the parameters, leading thus to computationally expensive methods for deep neural networks and for some applications.

Low-rank Factorization To estimate the informative parameters of deep CNNs special matrix structures can be introduced, in order to provide a low-rank factorization for them [266, 227]. A possible approach could be to replace a convolutional layer with several smaller convolutional layers applied sequentially [229]. This leads to a final model characterized by a much lower total computational cost since the number of parameters involved in the filter tensors is reduced. A particular example of this was described in Section 2.3.2 for InceptionNet [299, 301, 298], where some 3×3 convolutional layers are converted into an asymmetric convolution 1×3 followed by a 3×1 kernel. On the other hand, there exist techniques that focus on the last fully connected layers, which also require a large amount of memory [227]. In this case, the parameters are stacked in the weight matrix, to which the tensorization is applied. As can be understood, these approaches have the benefit of reducing the number of parameters and matrix multiplication, thus the computational complexity. However, a decent amount of retraining is needed to achieve convergence when compared to the original model, and there are no strong guarantees that using this low-rank factorization leads to accurate models.

Parameter Quantization This technique concerns reducing the number of bits used for the representation of the weights and the activation functions by substituting the floating points with integers inside the network [56, 259, 55, 245]. In this way, the created network requires less memory to be stored and for computing predictions. Furthermore, on certain hardware, representations using fewer bits facilitate faster computations. For instance, to quantize inputs for each convolution layer, the activation function is replaced by a quantization function (e.g. logical XNOR), converting each parameter to values in $\{-1, 1\}$. In this way, a sophisticated CNN becomes a simple logic circuit.

Despite these benefits, using such approaches can lead also to a significant loss in accuracy, as there is no guarantee that approximating parameters while sacrificing precision for a compressed representation will not compromise the final results. Furthermore, training such discrete-valued CNNs can be very challenging as they cannot be directly optimized using gradient-based methods, but some modifications need to be introduced in order to reduce precision computations during backpropagation and hence to facilitate low-resource training [56, 259].

Manual Architecture Design Instead of modifying existing architectures to make them more efficient, another approach could be to develop manually the design of new architectures that are inherently resource-efficient. Typically these architectures employed particular layers and functions, designed specifically with the purpose of dimensionality reduction. An example is represented by global average pooling [188], which helps in the transition between feature learning and classification parts, i.e. from the output tensor of a convolutional layer to the input vector for a fully-connected layer. It basically reduces the spatial dimensions of each channel into a single feature by averaging over all values within a channel. As already mentioned in Section 2.3.2 for InceptionNet [299, 301, 298], many architectures employ also 1×1 convolution to reduce weights in the CNN architecture [299, 118]. Other particular types of convolution have already been introduced in Section 2.3.2 since they were manually designed to build new CNN architectures: the depthwise separable convolution, characterizing MobileNet [138, 271]; channel shuffle with grouped convolution, a key ingredient in the building block of ShuffleNet [335, 201]; fire modules with squeeze and expand layers, the core of SqueezeNet [141]. Despite the great reduction achieved, the main drawback connected with this kind of methodology is the need to study and manually design a layer that satisfies the requirement of reducing the parameters of a CNN, while keeping good performances.

Neural Architecture Search A recently emerging approach in this field is represented by Neural Architecture Search (NAS) [259, 342, 40, 302], that tries to discover automatically a good CNN structure. To achieve this goal, we define a discrete space of possible architectures in which we subsequently search for an architecture, optimizing in the meantime an objective function, such as the validation error. A possible procedure for this purpose is represented by using a heavily over-parameterized model where each layer contains several parallel paths, where each of these represents a different architectural block [40]. Some probability parameters are then introduced after each layer to drive the selection of the modules during the training phase. In this way, after backpropagation, the selected CNN architecture is represented by the most probable path through all the layers. Alternatively, a resource-efficient model, called EfficientNet, can be constructed by steps [302]: starting from a small model and then enlarging it following a principled compound scaling approach which simultaneously increases the number of layers, the number of channels, and the spatial resolution.
Although this approach leads to the discovery of resource-efficient architectures, the evaluation of the validation error represents a time-consuming step since it requires a full training run of the whole net. Then, the space of architectures is characterized typically by exponential size in the number of layers, leading thus to the need for a careful design to facilitate an efficient search within that space.

Reduced Convolutional Neural Network As described, most of the existing methods are only deleting and changing model parameters directly without changing the network architectures, or are designing manually layers and network structures for the purpose of dimensionality reduction. Our proposed approach, described in Section 3.3 and in [210] for a general ANN, differs from the one listed here, because, starting from an (already trained) ANN, and in particular from a CNN, we have

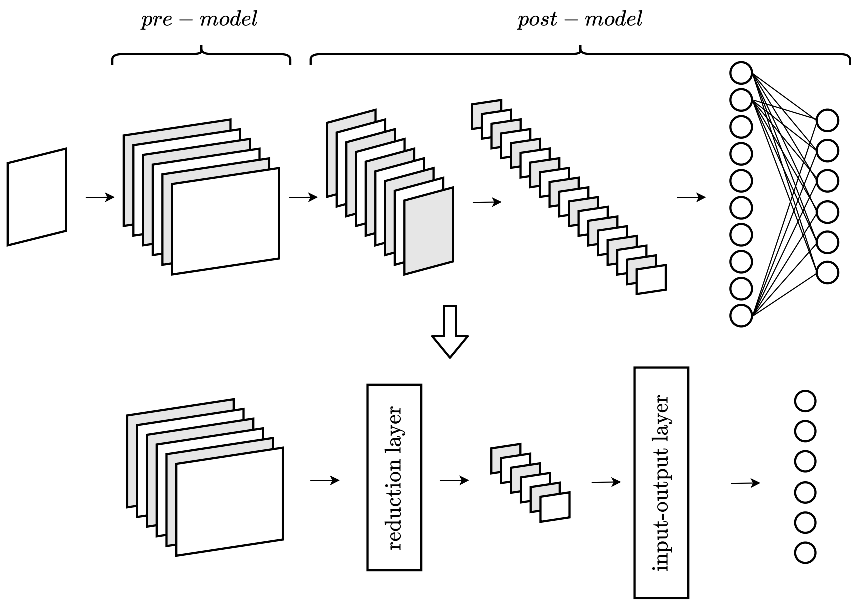


Figure 3.1: Graphical representation of the reduction method proposed for a CNN.

constructed its reduced version by modifying the whole architecture of the original net. The reduced network is thus constructed by retaining a certain number of layers of the original CNN and replacing the remaining ones with an input-output mapping. In this way, we are splitting the net into two parts connected by the reduced method, which helps in reducing the typically large dimensions of the intermediate layers by keeping only the most important information. We are performing a smart selection of the main parameters of the network, allowing us to reduce the required resources and the computing time to infer the model, as will be discussed in Section 3.4.2. Figure 3.1 presents then a graphical representation of the reduction method proposed in [210] and summarized in Algorithm 4.

3.4.2 Practical Application on VGG-16

In this section, we are going to revise the results obtained by applying the different reduced methods proposed to a CNN [210]. In particular, we will present a comparison between the results obtained with a reduced and full version of a CNN in terms of final accuracy, memory allocation, and speed of the procedure.

As stated before, the problem to be solved is the running of this net within an embedded system with

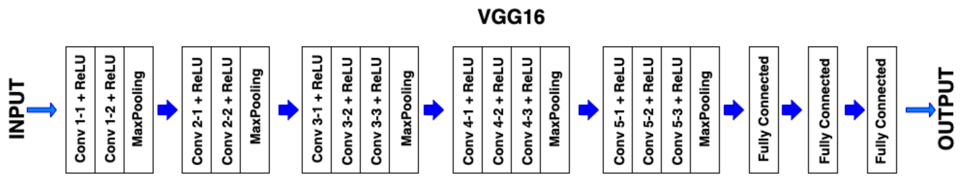


Figure 3.2: Graphical representation of VGG-16 architecture.

particular memory constraints to perform the image recognition task. Among the several existing CNN architectures presented in Section 2.3.2, we have chosen as testing model VGG-16 [289].

Table 3.1: Results obtained with CIFAR-10 dataset.

Network	Accuracy		Storage (Mb)			Time	
VGG-16	77.98%		56.15			46 h	
	Epoch 0	Epoch 10	Pre-M	AS/POD	PCE/FNN	Init	Train
AS+PCE (5)	13.52%	82.01%	2.12	3.12	0.05	43 min	4.5 h
AS+FNN (5)	33.06%	80.43%	2.12	3.12	0.0047	5 h	4.5 h
POD+FNN (5)	62.16%	80.24%	2.12	3.12	0.0047	79 min	5 h
AS+PCE (6)	14.42%	84.69%	4.37	3.12	0.05	49 min	5.5 h
AS+FNN (6)	33.76%	82.13%	4.37	3.12	0.0047	5 h	4.5 h
POD+FNN (6)	63.84%	83.93%	4.37	3.12	0.0047	83 min	5 h
AS+PCE (7)	4.25%	85.60%	6.62	0.78	0.05	35 min	5.5 h
AS+FNN (7)	75.66%	86.03%	6.62	0.78	0.0047	1.5 h	5 h
POD+FNN (7)	80.17%	87.45%	6.62	0.78	0.0047	12 min	5 h

Table 3.2: Results obtained with the custom dataset.

Network	Accuracy		Storage (Mb)			Time	
VGG-16	95.65%		56.14			22 min	
	Epoch 0	Epoch 10	Pre-M	AS/POD	PCE/FNN	Init	Train
AS+PCE (5)	29.03%	95.21%	2.12	3.12	0.02	2 min	10 min
AS+FNN (5)	94.63%	94.92%	2.12	3.12	0.0021	12.5 min	12 min
POD+FNN (5)	96.52%	96.66%	2.12	3.12	0.0021	28 sec	11.5 min
AS+PCE (6)	29.75%	95.79%	4.37	3.12	0.02	2.5 min	10 min
AS+FNN (6)	94.92%	95.36%	4.37	3.12	0.0021	12.5 min	12.5 min
POD+FNN (6)	96.23%	96.37%	4.37	3.12	0.0021	33 sec	13 min
AS+PCE (7)	28.59%	94.05%	6.62	0.78	0.02	1.5 min	11 min
AS+FNN (7)	94.34%	94.63%	6.62	0.78	0.0021	4.5 min	13 min
POD+FNN (7)	96.37%	96.52%	6.62	0.78	0.0021	33 sec	14 min

Its structure, depicted in Figure 3.2, is composed of 13 convolutional blocks alternating with 5 max-pooling layers, representing the feature learning part, and 3 fully connected layers, responsible for the final classification.
The implementation of the technique proposed is carried out using PyTorch [237] as the development environment, while for the actual computation of the active subspaces we use the open-source Python package ATHENA [255]. The original network and its reduced versions have been trained and tested on two different datasets: a benchmark one represented by CIFAR-10 dataset [166] (see Section 2.3.1), and a custom one. This last set of data was connected with the practical application in a professional appliance we had to do for *Electrolux Professional*. It is thus composed of 3448 32×32 color images organized in 4 classes: 3 non-overlapping classes and a mixed one, characterized by pictures with objects of different categories present at the same time.
First of all, the original network VGG-16 has been trained on each of the different datasets presented for 60 epochs[41]. Table 3.1 and Table 3.2 show the levels of accuracy of VGG-16 at the end of the training phase, 77.98% for the CIFAR-10 and 95.65% for the custom dataset, which represent the values we want to achieve or exceed with the reduced networks. We report also the results obtained with different reduced versions of VGG-16 constructed following the steps of algorithm 4 and using

[41]We have chosen 60 (and then 10 for the reduced net) to be the number of epochs for the training phase as a trade-off between the final accuracy and the time needed. For this reason, we kept the same value in the two different cases we are considering to have a fair comparison.

three cut-off layers[42] l: 5, 6, and 7, as done in [57]. We remark that in the case of dimensionality reduction with the AS technique, we employed the *Frequent Direction method* [90] implemented inside ATHENA to compute the AS. The dimension of the reduced space r onto which we are projecting the high-dimensional features —the output of the pre-model— is set to 50 both for AS and for POD in analogy with the structural analysis considerations presented in [57].
Table 3.3 is then reporting different results obtained by training the reduced net for 10 epochs

Table 3.3: Results obtained for the reduced net POD+FNN (7) trained on CIFAR-10 with different structures for the FNN.

			Hidden layers			
			1	**2**	**3**	**4**
Hidden neurons	**10**	Epoch 0	81.39%	67.92%	75.52%	81.57%
		Epoch 10	87.89%	87.59%	87.46%	87.26%
		Storage FNN (Mb)	0.0024	0.0028	0.0032	0.0036
	20	Epoch 0	80.17%	80.05%	79.97%	78.28%
		Epoch 10	87.45%	87.13%	87.42%	86.68%
		Storage FNN (Mb)	0.0047	0.0063	0.0079	0.0095
	30	Epoch 0	77.57%	80.36%	80.43%	76.26%
		Epoch 10	86.92%	86.25%	86.30%	85.25%
		Storage FNN (Mb)	0.0070	0.0106	0.0141	0.0177
	40	Epoch 0	71.24%	70.38%	69.31%	68.15%
		Epoch 10	85.04%	84.60%	84.18%	83.64%
		Storage FNN (Mb)	0.0093	0.0156	0.0219	0.0281

using different FNN architectures, i.e. a different number of hidden layers and also hidden neurons, which are kept constant in each hidden layer of the net. In particular, each of the FNNs used has been trained for 500 epochs during the initialization process, thus before the re-training step of the whole net reported in the tables. The comparison in Table 3.3 is then made in terms of memory allocation for the FNN and accuracy of the corresponding reduced net at epoch 0, i.e. after its initialization, and at epoch 10, i.e. after the re-training of the whole reduced net. It can be noticed that increasing the depth and width of the network is not leading to a remarkable gain in accuracy. For this reason, based on considerations about the final accuracy and the allocation in memory of the FNN, the structures chosen for testing the reduction method using a FNN as input-output map are the following:

- **CIFAR-10**: 50 input neurons, 10 output neurons, and one hidden layer with 20 hidden neurons.
- **Custom Dataset**: 50 input neurons, 4 output neurons, and one hidden layer with 10 hidden neurons.

Starting from this, we have constructed different reduced networks, i.e. AS+PCE, AS+FNN, POD+FNN, using three different cut-off layers: 5,6, and 7. These reduced models have then been re-trained for 10 epochs using the aforementioned datasets. Table 3.1 and Table 3.2 show a comparison between the full and reduced VGG-16 in terms of accuracy (before and after the final training), memory storage, and time needed for the initialization and the training processes. From these data, we can

[42]As explained in [57] and in the correspondent implementation the indices 5, 6 and 7 correspond to the indices of the convolutional layers in a list where only convolutional and linear layers are taken into consideration as possible cut-off layers. Therefore, taking into account the whole net with all the different layers, these correspond to 11, 13, and 16.

also point out that the results provided are consistent between the two different sets of data.
Regarding the problem of memory constraints in an embedded system, it can be noticed that the storage required for the proposed nets is decreased compared to that of the original VGG-16 in both cases (see Table 3.1 and Table 3.2). In fact, the checkpoint file[43] stored for the full net occupies 56.14 Mb, whereas that of its reduced versions less than 10 Mb. Hence, if we compute the compression ratio related to each of these nets:

$$\text{compression\_ratio} = 1 - \frac{\text{compress\_size}}{\text{uncompress\_size}} \tag{3.36}$$

we obtain that the saving in memory is around 90% in all the cases. It can also be highlighted that the use of a FNN instead of the PCE is saving space in memory of two orders of magnitude: 10^{-4} against 10^{-2}.
From the point of view of final accuracy, it can be observed that the proposed reduced CNN achieved a similar level (in most cases also greater) as the original VGG-16 but with the benefits of requiring much smaller storage. Furthermore, it can be observed that increasing the cut-off layer index l leads to nets with higher accuracy since we are retaining more information (i.e. features) from the original VGG-16, but on the other hand, there is a smaller compression ratio. For this reason, as pointed out before, the choice of index l should be done carefully as a trade-off between the final accuracy and the level of reduction, but also taking into account the field of application.
Table 3.1 shows also that the reduced net POD+FNN leads also to another gain in terms of time, as it does not require additional training with the whole dataset after the initialization, i.e. at epoch 0. From the table, it can indeed be noticed that, before the re-training of the whole reduced network, its accuracy is already acceptable and for index 7 is also quite high. The immediate consequence of this is the saving of the time needed to gain a performing network, which is in the order of 5 hours. These considerations are true and consistent also using a different set of data, as the custom dataset under consideration. In fact, Table 3.2 reports how for the three choices of l the POD+FNN net has an accuracy greater than the original VGG-16 even after the initialization process.

3.5 Reduction Strategies for Object Detectors

This section will deal with reduction strategies for object detectors. As introduced for CNNs, there exists a great interest in using these models in the industrial world to solve complex tasks, as that of object detection introduced in Section 2.4. Hence, the necessity of running these deep neural networks in embedded systems highlights some drawbacks connected with most of these architectures. Since object detectors are characterized by having millions of parameters, this results in huge computational effort from the point of view of the training time [99, 299] and a great amount of space for storing and running them.
In the last years, many techniques have been developed to obtain light-weight versions of this kind of models. Some approaches employ the methods described for CNNs in Section 3.4.1. Hence, they directly adopt the original object detection architecture employing, for example, input resizing and network pruning [110, 92, 29]. Other methods consist in modifying the structure of the object detectors, retaining their original image classifier, but based on well-established hardware-efficient CNNs as a backbone. To lower network complexity, there is thus the introduction of highly optimized base nets and convolutional feature layers, such as the one previously described in Section 2.3.2: MobileNet [138], ShuffleNet [335], SqueezeNet [141]. The employment of these optimized CNNs in

[43]Note that in both cases (CIFAR-10 and custom dataset) the checkpoint file requires 56 Mb of memory, but if you need to store additional information (on the architecture of the net, training epochs, loss,...) the required allocation is around 220 Mb.

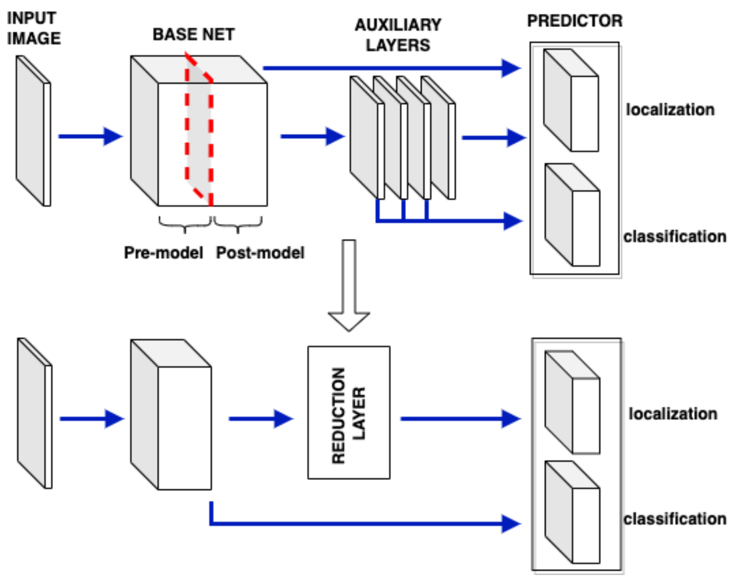


Figure 3.3: Graphical representation of the reduction method proposed for an object detector.

commonly used object detector's architectures, e.g. YOLO, SSD, Faster R-CNN, has the benefit of providing light-weight deep neural networks with reasonable performances [270, 138, 219, 296, 186, 324, 148, 323].
Unlike what was done in the aforementioned methods, to face the compute-intensive and memory-intensive issues, we have extended and adapted the dimensionality reduction method presented in Section 3.3 and in [210, 57] to SSD type architectures. Starting from the original structure, we compress the network by retaining only a certain number of layers of the base net and substituting the remaining part with this reduction layer. The dimensionality reduction operation is carried out using POD, which is thus responsible for decreasing the number of hyperparameters of the model. In this way, the resulting network requires less space to be stored and in fine-tuning time. In the next sections, we are now going to provide a detailed description of the proposed procedure for this case and its practical application on SSD300, using PASCAL VOC as the dataset.

3.5.1 Reduction of SSD-Type Object Detectors

In this section we provide the detailed description of the proposed reduced technique for SSD-type object detectors [211], that is summarized in Figure 3.3 and in Algorithm 6. This framework is extending the one proposed in Section 3.3 and in [210] in order to include also more complex networks, such as the ones that solve the problem of object detection [294, 339, 191].
Let $\mathcal{Obj\_Det}$ be an object detector composed by N hidden layers. Following [211], the only assumption we are making on $\mathcal{Obj\_Det}$ is that we have a network composed by: *i,*) a base net, a CNN extracting the low-level features, as the one introduced in Section 2.3.2; *ii,*) some additional convolutional layers responsible for capturing the high-level features; *iii,*) a predictor, that will output the predicted class and localization, e.g. the coordinates of the bounding box, for each object in the image. Therefore, we have that:

$$\mathcal{Obj\_Det} = [\mathcal{basenet}, \mathcal{auxlayers}, \mathcal{predictor}]. \tag{3.37}$$

Algorithm 6 Pseudo-code for the construction of the reduced object detector [211].

Inputs:

- train dataset $\mathcal{D}_{\text{train}} = \{\mathbf{x}^{(0),j}, \mathbf{y}^j_{\text{loc}}, \mathbf{y}^j_{\text{cls}}\}_{j=1}^{N_{\text{train}}}$,
- $\mathit{Obj\_Det} = [\mathit{basenet}, \mathit{auxlayers}, \mathit{predictor}]$,
- reduced dimension r,
- index of the cut-off layer l,
- a test dataset $\mathcal{D}_{\text{test}} = \{\mathbf{x}^i, \mathbf{y}^i_{\text{loc}}, \mathbf{y}^i_{\text{cls}}\}_{i=1}^{N_{\text{test}}}$.

Output: Reduced Object Detector $\mathit{Obj\_Det}^{\text{red}}$

```
1: $\mathit{basenet}^l_{\text{pre}}, \mathit{basenet}^l_{\text{post}} = \text{splitting\_net}(\mathit{basenet}, l)$
2: $\mathbf{x}^{(l)} = \mathit{basenet}^l_{\text{pre}}(\mathbf{x}^{(0)})$
3: $\mathbf{z} = \text{reduce}(\mathbf{x}^{(l)}, r)$
4: $\hat{\mathbf{y}}_{\text{loc}}, \hat{\mathbf{y}}_{\text{cls}} = \mathit{predictor}(\mathbf{x}^{(l)}, \mathbf{z})$
5: Training of the constructed reduced net using $\mathcal{D}_{\text{test}}$.
```

Note that, as described in Section 2.4.7, an example of architecture satisfying this assumption is SSD. It is also important to point out that we do not need to have a pre-trained object detector on the dataset of interest, but we only need to have the status of this deep neural network after its initialization.
As discussed in Section 1.4, an object detector can then be seen as a map between its inputs and outputs, $\mathit{Obj\_Det} : \mathbb{R}^{n_{\text{in}}} \longrightarrow \mathbb{R}^{n_{\text{out}}}$, where in this case $n_{\text{out}} = n_{\text{class}} \times 4$, since we have two different outputs: a tensor $\hat{\mathbf{y}}_{\text{cls}}$ representing the predicted probability for each n_{class} of the dataset and a tensor $\hat{\mathbf{y}}_{\text{loc}}$ containing the four offsets values. Let then $\mathcal{D}_{\text{train}} = \{\mathbf{x}^{(0),j}, \mathbf{y}^j_{\text{loc}}, \mathbf{y}^j_{\text{cls}}\}_{j=1}^{N_{\text{train}}}$ be the training dataset, made of n_{train} input samples and corresponding expected classification and localization outputs. Following the procedure described in Section 3.3, our reduction method operates directly on the original structure of an object detector, starting from its status after initialization without loading weights from a previous train with the dataset of interest, to provide a reduced version. Algorithm 6 presents an algorithmic representation of our method, that is composed of the following three steps [211]:

1. *Network Splitting* (Section 3.5.2): As described in Section 3.3.1, in this first step we are defining the pre-model and a post-model, by fixing a cut-off layer l. In this way, we are also deciding how much information of the original network, and in particular of the base net, we want to retain.

2. *Reduction Layer* (Section 3.3.2): The base net is usually followed by an additional structure responsible for the detection of the high-level features of objects, giving thus a whole understanding of the picture. The second step performs thus the dimensionality reduction of the pre-model output, substituting the auxiliary layers with a reduction layer, employing POD, described in Section 3.2.1, as ROM method.

3. *Predictor* (Section 3.5.4): After we have projected the high-dimensional pre-model output into a low-dimensional space, the obtained features need to be employed for the classification and localization tasks. In this case, a predictor with the same architecture as that of the original network is used.

After the reduced version of the object detector has been initialized, we need to train it. Despite what is done for a general ANN and in particular for a CNN, we have not used the loss introduced in Section 3.3.4 based on the knowledge distillation technique. Using such a framework for object detection tasks has been proven to be non-trivial and sub-optimal [182]. Based on the observations in [182], the original and reduced models present different levels of precision, since they are characterized by different prediction responses and different ways to rank their predicted bounding boxes. Hence, imitating all the feature maps of the teacher network is not a good idea to improve the accuracy of the reduced object detector. For this reason, we have used as loss function the one designed for the object detector under consideration. Further improvements can then be introduced in new versions of our proposed reduced method. Chapter 4 will briefly explain how we can develop a method for distilling knowledge from a full object detector, based on the techniques introduced in [182, 45].
In the following sections, we are now going to provide some details on the three steps needed to apply our reduction method of [210, 212] to an object detector, by employing the methods described in Section 3.3 and in Section 3.2.

3.5.2 Network Splitting

Let l be the fixed cut-off layer and $\mathcal{basenet}$ the CNN base net, composed of L hidden layers. As done in [210, 57, 212], the base net, i.e. the first part of $\mathcal{Obj\_Det}$, is splitted in two different blocks based on l: the *pre-model* and the *post-model*. In particular, $\mathcal{basenet}$ can be described as as compositions of L functions $f_j : \mathbb{R}^{n_{j-1}} \to \mathbb{R}^{n_j}$, for $j = 1, \dots, L+1$, representing the different layers of the network (see Section 1.4):

$$\mathcal{basenet} = f_{L+1} \circ f_L \circ \cdots \circ f_1. \tag{3.38}$$

Equation (3.38) can now be rewritten employing the pre- and post-model:

$$\mathcal{basenet}(\mathbf{x}^{(0)}) \equiv \mathcal{basenet}^l_{\text{post}}(\mathcal{basenet}^l_{\text{pre}}(\mathbf{x}^{(0)})), \tag{3.39}$$

where $\mathbf{x}^{(0)} \in \mathbb{R}^{n_{\text{in}}}$ is an input image and the pre- and post-models are defined by:

$$\begin{aligned} \mathcal{basenet}^l_{\text{pre}} &= f_l \circ f_{l-1} \circ \cdots \circ f_1, \\ \mathcal{basenet}^l_{\text{post}} &= f_L \circ f_{L-1} \circ \cdots \circ f_{l+1}. \end{aligned} \tag{3.40}$$

Also here, as discussed in Section 3.3, the cut-off layer l should be chosen carefully, due to its important role in the final outcome, controlling how many layers of the original net, i.e. information, we are discarding. In this case, following [210, 211], the choice of l was driven by considerations about the network and the dataset at hand, balancing the final accuracy and the compression ratio.

3.5.3 Reduction Layer

Once we have defined the pre-model for the base net $\mathcal{basenet}^{(l)}_{\text{pre}}$, we can compute its output $\{\mathbf{x}^{(l),i}\}_{i=1}^{N_{\text{train}}}$ for each sample in $\mathcal{D}_{\text{train}}$:

$$\mathbf{x}^{(l)} = \mathcal{basenet}^{(l)}_{\text{pre}}(\mathbf{x}^{(0)}). \tag{3.41}$$

In particular, since $\mathbf{x}^{(l)}$ usually lies in a high-dimensional space, we aim at projecting it onto a low-dimensional one by retaining only the most important directions and thus information. We are thus introducing a linear layer performing dimensionality reduction on the pre-model output

$\mathbf{x}^{(l)}$ in place of the auxiliary layers, as described in Figure 3.3 and in algorithm 6. Based on the method presented in [211] and described in Section 3.3, to decrease the huge number of parameters characterizing the feature maps output of the pre-model, we are exploiting the POD approach introduced in Section 3.2.1.
Therefore, once we have computed the pre-model outputs for each element of $\mathcal{D}_{\text{train}}$, we unroll them as columns of a matrix $\mathbf{S} = [\mathbf{x}^{(l),1}, \dots, \mathbf{x}^{(l),N_{\text{train}}}]$, on which we apply the POD. Following Section 3.2.1, we can decompose $\mathbf{S}$ using the SVD in order to determine the POD modes. Given then the reduction parameter r, i.e. the dimension of the space onto which we want to project the output of the pre-model, we can determine the project matrix $\boldsymbol{W}_{\text{proj}}$ and, hence, the reduced version of $\mathbf{x}^{(l)}$:

$$\mathbf{z}^i = \boldsymbol{W}_{\text{proj}}^T \mathbf{x}^{(l),i}, \quad \text{for } \ i = 1, \dots, N_{\text{train}}. \tag{3.42}$$

After this reduction layer, we have then in hand the reduced representation of the pre-model output for any image in our dataset.

3.5.4 Predictor

The last block in the reduced framework is represented by the predictor, that deals with connecting the reduced features $\{\mathbf{z}^i\}_{i=1}^{N_{\text{train}}}$ to the expected outputs $\{\mathbf{y}^i_{\text{loc}}, \mathbf{y}^i_{\text{cls}}\}_{i=1}^{N_{\text{train}}}$. This mapping is realized by using the original classifier, composed of two siblings classifiers, one for localization prediction and one for class prediction. Due to the structural changes made in the previous parts of the object detector, we need to modify the inputs for the predictor. We highlight indeed that, typically, architectures for object detection use the output of several layers — e.g. base net convolutional layers, auxiliary layers — to predict the final output. In the reduced network, instead, the only inputs are represented by the output of the pre-model $\mathbf{x}^{(l)}$ and its reduced version $\mathbf{z}$ obtained through the reduction layer.
Since the number of inputs has been modified, this will affect the total number of priors of the object detector. As discussed in Section 2.4.2, several priors with different shapes are used for each selected input, to find the correct place and sizes (width and height) of the predicted bounding boxes. In particular larger feature maps — e.g. base net convolutional layers output — have priors with smaller resolutions and are therefore ideal for detecting minor details of the image, whereas smaller feature maps — e.g. auxiliary layers output — will be responsible for detecting high-level features, like objects' shapes. Now, in the proposed reduced framework, since we have decreased the number of inputs for the predictor, there will be also fewer priors, that have to be chosen properly. Hence, we have performed an empirical analysis in order to find the right scale parameter, which controls the priors' resolution, to gain comparable results with the original network.

3.5.5 Numerical Results

As discussed in [211], we have conducted experiments using SSD300 [195] with VGG-16 [289] as base net. To evaluate the ability of the full and reduced nets to detect objects in pictures, we have then trained and tested the original and compressed models on different datasets: the PASCAL VOC [81], and a custom one, linked with the collaboration with *Electrolux Professional.* The latter is composed of 247 images — 197 used for training purposes and 50 for testing— subdivided into three classes. Hence, to simulate the custom case subject to a non-disclosure agreement, we have also created another database with similar features to the custom one. Therefore, starting from VOC2007, a subset of PASCAL VOC, we have extracted a smaller dataset made up of 300 images subdivided into two categories: cats and dogs.
First of all, we have trained and tested the original net SSD300 with the cat-dog dataset, the *Electrolux Professional* dataset, and the whole PASCAL VOC. The results obtained are displayed in the first

Table 3.4: Results obtained with the cat-dog dataset.

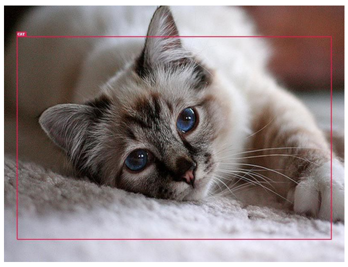
Network	mAP	Accuracy Cat	Accuracy Dog	Storage (Mb)	Training Time
SSD300	80.8%	76.3%	85.4%	91.09	61 min
SSD300_red	59%	50.3%	67.7%	77.45	36 min

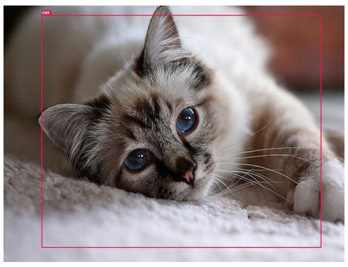
Table 3.5: Results obtained with *Electrolux Professional* dataset.

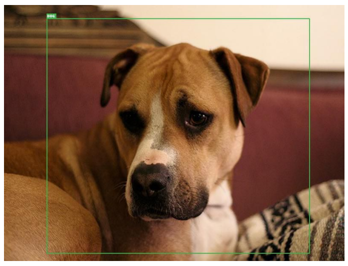
Network	mAP	Accuracy Class 1	Accuracy Class 2	Accuracy Class 3	Storage (Mb)	Training Time
SSD300	85.4%	74.5 %	91.1%	90.7%	92.6	2.5 h
SSD300_red	85.8%	82.4%	92.4%	82.7%	77.59	1.5 h

columns of Table 3.4, Table 3.5, and Table 3.6. In particular, for the cat-dog database, Figure 3.4 provides some examples of expected outputs from our reduced SSD300, namely outputs of the original model.
We have then applied the reduction method described in Section 3.5.1 to the original architecture of SSD300, where we have not used the status of the network after training, but after initialization, i.e. with all the weights randomly initialized except those of the VGG-16, pre-trained on ImageNet [68]. Hence, the net has been cut at layer 16, i.e. cut-off index 7, of VGG-16, and then the output of the pre-model, having 369664 parameters, has been projected onto a reduced space of dimension $r = 50$, in analogy with [210, 57]. As highlighted in section 3.5.1, the inputs to the predictor part are different from the full model. In the case of SSD300, these correspond to[44]: the output of the third layer in the fourth convolutional block $conv4\_3$, the final output of the net $conv7$, the outputs of each block of the auxiliary layers $conv8\_2$, $conv9\_2$, $conv10\_2$, $conv11\_2$. In our reduced network the only inputs used for predictions are the output of the pre-model, i.e. the output of layer $conv3\_3$, and its reduced version, the output of the reduction layer. In this way, the number of priors we are taking into account is less than before: 5782 instead of 8732 for the full net. We have then adapted the scaling factor to our case, choosing two different scales: 0.1 and 0.9, i.e. one that takes into account smaller objects (10% of the picture) and one for bigger objects (90% of the picture). Despite these changes connected with inputs and default boxes, the predictor block remains the same as the original SSD300. Once the reduced SSD300 has been initialized, it has then been trained with the aforementioned datasets. We can also mention that to train SSD300 we have employed Stochastic Gradient Descent (SGD) as suggested in the original paper [194], whereas for our reduced model we have seen empirically that Adam was a better choice.
Starting with the cat-dog case, it can be seen in Table 3.4 that, after a 500 epochs of training, the net has an overall Mean Average Precision (mAP) of 70.2%. In particular, it reaches an accuracy of 86.6% for dogs and 53.8% for cats. Table 3.4 summarizes also the results obtained with our reduced method, by training the reduced SSD300 for 500 epochs. It can be observed that the mAP has decreased by 11% compared to the original net: the accuracy for the categories dogs and cats are now 67.7% and 50.3% respectively. Figure 3.4 shows some output images proving how the results obtained with the full and the reduced net are comparable.
Similarly, SSD300 and the proposed reduced version have been trained and tested on the *Electrolux Professional* dataset. Table 3.5 presents an overall of the results obtained in this custom case. Starting from a full SSD300, we have constructed a new network performing as the original one, with similar

[44]The labels of the layers refer to the one presented in Section 2.4.7, and in particular in Figure 2.34.


(a) SSD300


(b) Reduced SSD300


(c) SSD300

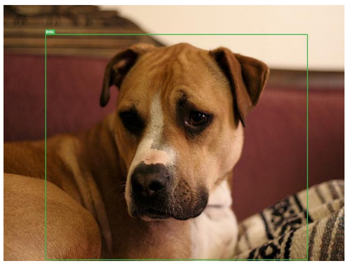

(d) Reduced SSD300

Figure 3.4: Comparison of the results obtained using the original SSD300 and its reduced version on two test images.

Table 3.6: Results obtained with PASCAL VOC.

Network	mAP	Storage (Mb)	Training Time
SSD300	77.8%	100.23	66 h
SSD300_red	39%	78.26	39 h

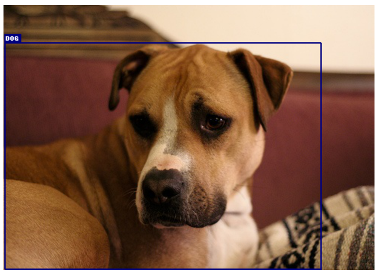
levels of accuracy for each class composing the dataset. Even if the cat-dog dataset was constructed to simulate the *Electrolux Professional* dataset, this slight difference in the achieved accuracy can be connected to the fact that the latter is composed of a smaller variety of images, strictly related to the final application.
The last test case employs the whole PASCAL VOC dataset, hence a huge database compared with the ones used in the previous cases. Table 3.6 summarizes the results obtained in this framework after a 500 epochs training. As can be observed, here there is a higher decrease, of 50%, in the mAP of the model with respect to the previous cases. In Table 3.7 we present also a comparison on the level of accuracy achieved for each class in the dataset using the original and reduced SSD300. Figure 3.5 presents then some examples of output images obtained with the two object detectors. As can be observed, the reduced network is less accurate than the full SSD300, with some outliers predictions, but, despite this, the results are in general comparable.
From the point of view of the memory storage, we can compute the compression ratio using Equation (3.36). We have thus a reduction of 15% for the cat-dog dataset, 16% for the custom case, and 22% for PASCAL VOC. We highlight that such compression is not remarkable as the one shown for the image recognition case in Section 3.4 and in [210] since in this case the larger dimension of the pre-model output has led to a huge matrix for storing the POD modes, limiting a lot the potential space gain.
Beside the slight space reduction, our proposed approach is characterized by a halving of the training

Table 3.7: Accuracy obtained for each class of the PASCAL VOC dataset with SSD300 and our reduced version.

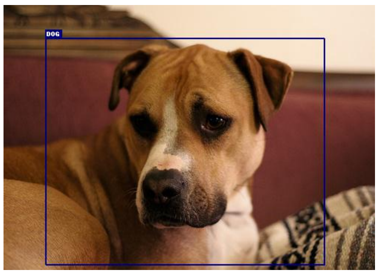
	Accuracy	
Category	**SSD300**	**SSD300_red**
aeroplane	79%	59%
bicycle	85%	45%
bird	78%	30%
boat	71%	31%
bottle	48%	18%
bus	87%	50%
car	86%	64%
cat	89%	46%
chair	58%	16%
cow	83%	46%
dining table	78%	30%
dog	86%	37%
horse	88%	55%
motorbike	85%	48%
person	79%	34%
pottedplant	54%	22%
sheep	78%	44%
sofa	79%	30%
train	86%	52%
tvmonitor	76%	32%

and inference time, allowing thus to accelerate the fine-tuning (performed also in the full network to optimize the auxiliary and classifier layers) of the network in the reduced version. Assuming that the weights of base net layers are already pre-trained[45], the minor number of hyperparameters results indeed in a faster optimization task, as described in Table 3.4, in Table 3.5 and in Table 3.6. Reduction of CNNs becomes then not only a technique to compress the architecture dimension, but also to accelerate the learning and inference steps of the studied network.
To conclude, while the reduction in time and space is comparable regardless of the dataset, the accuracy is much greater when the total number of classes used is low. In the latter context, our approach seems to demonstrate the ability to automatically isolate the most important convolutional features. Keeping the same reduced architecture for the three different datasets, characterized by 2, 3 and 20 classes respectively, we have thus noticed how our proposed method is able to automatically detect redundant and superfluous information, leading, in the end, to a drastic difference in the mAP. It is therefore clear from the results shown in Table 3.4, in Table 3.5 and in Table 3.6 that there is a trend between the compression and the accuracy of the reduced model linked to the complexity of the problem, which is roughly quantified by the number of image classes of the dataset under consideration.

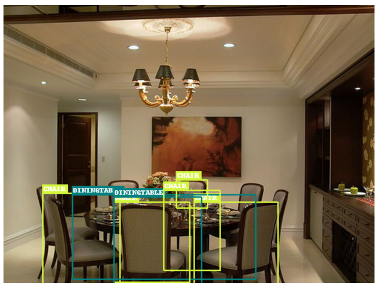
[45]The presence of pre-trained weights is mandatory for building the POD space.



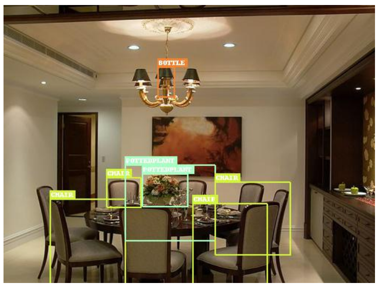
(a) SSD300



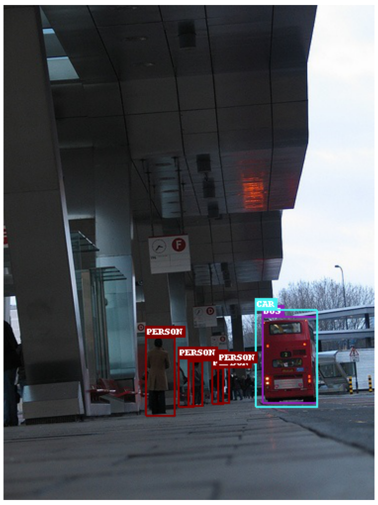
(b) Reduced SSD300



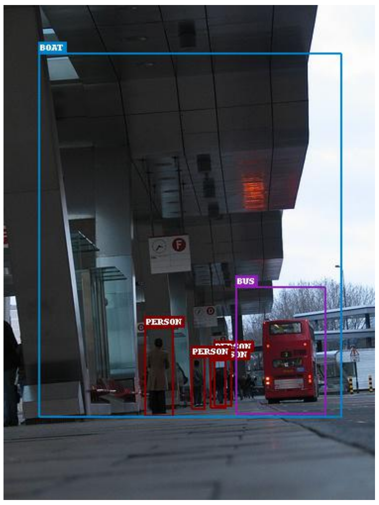
(c) SSD300



(d) Reduced SSD300



(e) SSD300



(f) Reduced SSD300

Figure 3.5: Comparison of the results obtained using the original SSD300 and its reduced version on three test images from PASCAL VOC.

CHAPTER 4 Conclusions and Future Perspectives

In this thesis we proposed a generic dimensionality reduction framework for ANNs, aiming at constructing a reduced version of the net under consideration. The need to develop such a procedure arises from the collaboration with *Electrolux Professional* in the framework of an industrial doctoral grant agreement. The final goal of this procedure coincides thus with having a model feasible to be placed inside a professional appliance, and in particular in an embedded system with limited hardware. Moving the inference phase from servers with great computing power to embedded devices with memory and power constraints represents a challenging problem in these engineering fields, which should be addressed by designing light-weight network models and methodologies.

Moved by the computational complexity of deep learning models, characterized by a huge number of tunable parameters, we have thus constructed a reduced version of the original network, by reducing the number of layers at the expense of a minimal error in the final prediction. Such a reduction is performed by splitting the full ANN into two parts, namely the pre- and post-model, identified by a cut-off layer index. The choice of this parameter represents a critical point in the procedure because it determines the amount of information we are retaining and discarding from the original model. At the moment, it is based on empirical analysis of the storage reduction and accuracy level achieved. Future works will better explore this trade-off between the complexity of the problem and the compression of the network, aiming hopefully to find an automatic way to compute the cut-off layer depending on the problem at hand.

The reduction occurs then by replacing the post-model with a linear reduction layer, involving ROM techniques and a response surface, creating an input-output mapping between the low-dimensional version of the pre-model feature maps and the final output. We have thus analyzed different approaches to perform compression of the original ANN by combining two dimensionality reduction methods, such as AS and POD, and input-output mappings, as PCE and a fully-connected FNN. The experiments conducted show indeed that different combinations of these techniques can lead to differences in the final accuracy.

First of all, we have conducted numerical experiments on CNNs to tackle the image recognition task. In this case, our proposed reduction framework produces a light-weight version of the original network, reduced in the number of layers and parameters, keeping a good level of accuracy. Furthermore, the combination of POD with FNN leads also to a decrease in the training time, which makes the proposed framework better than the inspiring method proposed in [57]. In many contexts, the learning procedure represents indeed the real bottleneck of the CNNs use. Therefore, extending our approach to reduce the architecture dimension during the training, and not only once it is finished, could induce hopefully a remarkable speedup in the optimization step. Further investigations will be carried on in this direction.

We have then tried to extend this methodology to more complex architectures, such as the one solving object detection tasks. In this case, we are assuming that the structure of the networks has to be of SSD-type, i.e. composed of a base net, some auxiliary layers, and two siblings predictor, one for classification and one for localization. Our proposed approach consists thus of a (linear) reduction layer, derived from POD, connecting the selected layers of the base net (pre-model) with the original predictor block. Comparing the original network with its reduced version, faster training and a decreased required space are shown exploiting our framework. Therefore, this allows for a faster and less computational expensive learning process. Consequently, this also leads to a reduced inference time, and therefore to the possibility of having real-time predictions once the reduced model is applied in an embedded device placed in a professional appliance.

The obtained results shows also an intuitive trend between the reduction and the accuracy with respect to the complexity of the problem, in this case roughly quantified by the number of image classes. Keeping indeed the same reduced architecture for the several datasets employed (2, 3 and 20 classes), we have seen a drastic difference in the average accuracy, which demonstrates the ability of the proposed method to automatically detect redundant and superfluous information. Hence, in the *Electrolux Professional* test case, where the dataset was characterized by little variety in the images and by few categories, the reduction manages to obtain good results as in the case of image recognition. In the other two cases, where we have a dataset with few classes but more diversified images and a very large dataset, the accuracy results are slightly different. In the first case, we have a slight decrease in the value of the average accuracy, while in the other a halving of it.

Besides these merits, the proposed reduction technique might be not appropriate in the case of object detection due to the slight reduction in the storage coupled with a decreased accuracy level. The practical application to SSD-type architectures has indeed demonstrated that dealing with high-dimensional features in the pre-model, can vanish the compression of the network. The bigger dimension of the POD modes, representing the pre-model features, is thus requiring more amount of space to be stored than the simple case of image recognition. The experiments conducted have shown that in some cases the reduction achieved is not satisfying as previously, potentially reaching the space needed by the full network. To overcome this limit, a possible solution is represented by the employment of hyper reduction technique or POD variants to minimize the quantity of data that has to be saved. Furthermore, in the image recognition framework, the knowledge distillation technique is introduced in the training phase to let the reduced model acquire knowledge from the original network and achieve comparable performances. In this case, it was not possible to apply this method since it is not so trivial to extend it for object detection. Another possible improvement could thus be the implementation of a knowledge distillation framework for object detection. Based on the state of the art [182, 45], a feasible approach corresponds to the introduction of a particular cross-entropy loss to handle the bounding box regression and, in particular, to incorporate the position-wise prediction difference as guidance for feature imitation from the full model. These further improvements will be part of an ongoing master project carried out with S. Zanin, N. Demo, and G. Rozza.

Another interesting topic connected with CNNs is the extension of the notion of discrete filter, characterizing these architectures, to a continuous one. The aim of this project is the application of CNN filters to unstructured data, by approximating the continuous filter with a trainable function constructed through a FNN. Employing such modules instead of the classical ones can have several benefits:

- They can be exploited in the case of not complete images, e.g. characterized by the presence of holes or bad quality regions, or, in a general perspective, to sparse incomplete data. This can thus be useful also for the training of image reconstructors, aiming at restoring the quality of images.
- The possible application to non-structured data leads to a filter able to capture non-trivial relationships also in an unstructured context.
- They can be used to find latent space representations, namely the most important information enclosed in input data, for complex manifolds by the use of continuous Autoencoders (AEs), exploiting continuous filters. Hence, using the knowledge contained in the latent space, it is possible to reproduce, by only knowing the time of the snapshot, phenomena, like the complete liquid-gas state.

- They can be applied for solving parametric problems for a finite set of data, to obtain a generalization of the solution. Hence, a continuous convolution network, containing the convolution and the transpose convolution operation, can be employed in the context of ROM to reconstruct the solution based on the finite input data.
- They can be included in a generative adversarial network to construct a continuous version of it, with the aim of enlarging, for example, the dataset for computational fluid dynamics simulations.

In this way, deep learning architectures, based on CNNs, coupled with continuous filters can hence be exploited to solve problems in different domain settings, such as the unstructured continuous one characterizing the aforementioned Navier-Stokes, or the liquid-gas phase problem. These topics are part of an ongoing internship project carried out with D. Coscia, N. Demo, G. Stabile, and G. Rozza [54].

APPENDIX A Frequent Direction Method

In recent years, the enormous amount of data available has led to the necessity of developing methods to handle them. The presence of limited memory available at any given time represents then an additional space constraint reinforcing this need. The *data streaming paradigm* [221, 93] deals exactly with computations on a huge dataset, where data can be stored in a matrix. A common approach to manage all this information is performing a low-rank approximation for such matrices by employing the truncated SVD [90]. However, the presence of these large matrices, connected with having this large stream of data, renders standard SVD algorithms infeasible. Therefore, a possible solution to this problem is represented by *matrix sketching approaches*. Given a large matrix A, the key idea is to compute a significantly smaller matrix B, such that $A \approx B$, to be used instead of A in computations without too much loss of precision.
There exist four main matrix sketching approaches [90] that can be followed to obtain a smaller version of the original matrix A. The first one generates a sparser reproduction of A, that can thus be stored more efficiently and provide faster calculations [1, 11, 73]. The second technique consists of randomly combining matrix rows, relying on subspace embedding methods and strong concentration of measure phenomena. The third approach relies on finding a small subset of matrix rows (or columns) approximating the entire matrix [32, 33, 71]. A fourth sketching procedure coincides then with the *Frequent Directions (FD)* method [90], the approach we are interesting in and we have employed in Chapter 3, and in particular in Section 3.3.2, to compute the AS. It is a deterministic approach, that draws on the matrix sketching and the item frequency approximation problems. Hence, before diving deep into the FD method, we should introduce the item frequency estimation problem.

Algorithm 7 Frequent Items Algorithm [217]

Inputs:

- stream of data A in a domain $[d]$;
- a number $k < d$.

```
1:  Set $T = \emptyset$ and a counter $m = 0$;
2:  for all $i \in [d]$ do:
3:      $m = m + 1$;
4:      if $i \in T$ then:
5:          $c_i = c_i + 1$;
6:      else if $| T |< k$ then:
7:          $T = T \cup \{i\}$ and $c_i = 1$;
8:      else $\forall$ j∈T:
9:          $c_j = c_j - 1$;
10:         if $c_j = 0$ then:
11:             $T = T \setminus \{j\}$.
12:         end if
13:     end if
14: end for
```

Output: Approximate frequencies $\hat{c}_i$ for each $i \in [d]$.

In many applications, an interesting problem to deal with is finding the frequency with which an object of interest occurs. A possible solution to this task is represented by the *item frequency approximation algorithm* [217, 152, 90, 331], also known as *frequent items algorithm.* Algorithm 7 summarizes the procedure proposed by Misra and Gries in [217]. Given a stream of n elements $A = \{a_1, \dots, a_n\}$, where each $a_i \in [d]$ and $[d]$ be a domain containing d elements $\{1, \dots, d\}$, we want to compute the number of times an item j appears in the stream, namely its frequency. Let now c_j represents this frequency of item j, defined as $c_j = \mid \{a_i \in A \mid a_i = j\} \mid$. A trivial approach consists in keeping a counter for each item, leading thus to an expensive algorithm. In [217], a method storing only $k < d$ items, i.e. k counters, is proposed. Hence, we define a stored items set T, which can contain at most k elements, keeping track of the items for which we have a counter. At the beginning, T is initialized to the empty set and will then be updated as the counter is incremented. The algorithm counts items by checking if an object has already been encountered and is therefore already in T. Hence, if the new item belongs to the stored items T, the corresponding counter is increased, otherwise, if the maximum cardinality of T has not been reached, the item is allocated in T and its counter is set to one. In the case all the k counters map to a value of at least one, we decrease all counts by the same amount until one has a zero value. At the end of the procedure, the final values c_j obtained give the approximate frequencies for each $j \in [d]$, where for items not stored in T, c_j is set to 0.
This procedure can then be generalized to the case where $A \in \mathbb{R}^{n \times d}$ is a matrix given to the algorithm as a stream of its rows [90]. First of all, we constrain the rows of A to be indicator vectors, i.e. $a_i \in \{e_1, \dots, e_d\}$ with e_j the standard j-th basis vector. In particular, we have that if the i-the element in the stream is j, then the i-th row of A coincides with the j-th unitary vector, i.e. $a_i = e_j$. In this way, the frequency of item j can be expressed as $c_j = \|Ae_j\|^2$. By applying the frequent items algorithm 7 on A, we can construct a matrix $B \in \mathbb{R}^{k \times d}$, describing a low rank approximation of A. In fact, once we have obtained the frequencies $\hat{c}_j$ for each item $j \in [d]$, we can set the corresponding row in B equal to $\hat{c}_j^{1/2} \cdot e_j$ whenever $\hat{c}_j > 0$. The resulting matrix B is thus characterized by having $rank(B) = k$ and $\|Be_j\|^2 = \hat{c}_j$.

Algorithm 8 Frequent Directions Algorithm [90]

Inputs:

- matrix $A \in \mathbb{R}^{n \times d}$;
- a number $k < d$.

```
1: Set $B = 0^{k \times d}$;
2: for $i = 1, \dots, n$ do:
3:     $B_k = a_i$;
4:     $\text{svd}(B) = U \Sigma V^T$;
5:     $B = \sqrt{\Sigma^2 - \sigma_k^2 I_k} \cdot V^T$.
6: end for
```

Output: Sketch matrix $B \in \mathbb{R}^{k \times d}$.

An extension of this result to general matrices is represented by the *FD* method [90, 187], summarized in Algorithm 8. In this case, a sketch matrix $B \in \mathbb{R}^{k \times d}$, initialized to the zero matrix, is updated every time a new row from the input matrix A is added. At each step of the procedure, a new row from A replaces the last row of B, which is then nullified by rotating and shrinking orthogonal vectors by roughly the same amount, as described in line 5. At the end, we have thus as output a sketch matrix B, representing a good low approximation of A, confirmed by the goodness estimation results presented in [90, 187].

Bibliography

[1] D. Achlioptas and F. Mcsherry. "Fast Computation of Low-Rank Matrix Approximations". In: *J. ACM* 54.2 (2007), 9–es. DOI: 10.1145/1219092.1219097.

[2] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. "A learning algorithm for boltzmann machines". In: *Cognitive Science* 9.1 (1985), pp. 147–169. DOI: https://doi.org/10.1016/S0364-0213(85)80012-4.

[3] K. Ahnert and M. Abel. "Numerical differentiation of experimental data: local versus global methods". In: *Computer Physics Communications* 177 (Nov. 2007), pp. 764–774. DOI: 10.1016/j.cpc.2007.03.009.

[4] I. Aizenberg, N. N. Aizenberg, and J. P. Vandewalle. *Multi-valued and universal binary neurons: Theory, learning and applications*. Springer Science & Business Media, 2000. DOI: 10.1007/978-1-4757-3115-6.

[5] A. Ajit, K. Acharya, and A. Samanta. "A review of convolutional neural networks". In: *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*. IEEE. 2020, pp. 1–5. DOI: 10.1109/ic-ETITE47903.2020.049.

[6] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamara, M. A. Fadhel, M. Al-Amidie, and L. Farhan. "Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions". In: *Journal of big Data* 8.1 (2021), pp. 1–74. DOI: 10.1186/s40537-021-00444-8.

[7] D. J. Amit. *Modeling Brain Function: The World of Attractor Neural Networks*. Cambridge University Press, 1989. DOI: 10.1017/CBO9780511623257.

[8] J. A. Anderson. "A simple neural network generating an interactive memory". In: *Mathematical biosciences* 14.3-4 (1972), pp. 197–220. DOI: 10.1016/0025-5564(72)90075-2.

[9] M. Anthony. *Discrete mathematics of neural networks: selected topics*. SIAM, 2001. DOI: 10.1137/1.9780898718539.

[10] A. Arnab and P. H. S. Torr. "Pixelwise Instance Segmentation with a Dynamically Instantiated Network". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 879–888. DOI: 10.1109/CVPR.2017.100.

[11] S. Arora, E. Hazan, and S. Kale. "A fast random sampling algorithm for sparsifying matrices". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2006, pp. 272–279. DOI: 10.1007/11830924_26.

[12] M. Arozullah and A. Namphol. "A data compression system using neural network based architecture". In: *1990 IJCNN International Joint Conference on Neural Networks*. Vol. 1. 1990, pp. 531–536. DOI: 10.1109/IJCNN.1990.137619.

[13] S. Arridge, P. Maass, O. Öktem, and C.-B. Schönlieb. "Solving inverse problems using data-driven models". In: *Acta Numerica* 28 (2019), pp. 1–174. DOI: 10.1017/S0962492919000059.

[14] *Artificial intelligence : principles and applications / edited by Masoud Yazdani*. Chapman and Hall, 1986.

[15] R. Askey and J. A. Wilson. “Some basic hypergeometric orthogonal polynomials that generalize Jacobi polynomials”. In: *Memoirs of the American Mathematical Society* 54.319 (1985). DOI: 10.1090/memo/0319.

[16] M. Avanzo, L. Wei, J. Stancanello, M. Vallières, A. Rao, O. Morin, S. A. Mattonen, and I. El Naqa. “Machine and deep learning methods for radiomics”. In: *Medical Physics* 47.5 (2020), e185–e202. DOI: 10.1002/mp.13678.

[17] A. Barron. “Universal approximation bounds for superpositions of a sigmoidal function”. In: *IEEE Transactions on Information Theory* 39.3 (1993), pp. 930–945. DOI: 10.1109/18.256500.

[18] S. Bell, C. L. Zitnick, K. Bala, and R. Girshick. “Inside-Outside Net: Detecting Objects in Context with Skip Pooling and Recurrent Neural Networks”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 2874–2883. DOI: 10.1109/CVPR.2016.314.

[19] R. Benenson, S. Popov, and V. Ferrari. “Large-scale interactive object segmentation with human annotators”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, 2019, pp. 11692–11701. DOI: 10.1109/CVPR.2019.01197.

[20] Y. Bengio. “Deep learning of representations for unsupervised and transfer learning”. In: *Proceedings of ICML workshop on unsupervised and transfer learning*. JMLR Workshop and Conference Proceedings. 2012, pp. 17–36.

[21] P. Benner, S. Grivet-Talocia, A. Quarteroni, G. Rozza, W. Schilders, and L. M. Silveira. *Model Order Reduction: Volume 1: System- and Data-Driven Methods and Algorithms*. De Gruyter, 2021. DOI: 10.1515/9783110498967.

[22] P. Benner, W. Schilders, S. Grivet-Talocia, A. Quarteroni, G. Rozza, and L. Miguel Silveira. *Model Order Reduction: Volume 2: Snapshot-Based Methods and Algorithms*. De Gruyter, 2020. DOI: 10.1515/9783110671490.

[23] P. Benner, W. Schilders, S. Grivet-Talocia, A. Quarteroni, G. Rozza, and L. Miguel Silveira. *Model Order Reduction: Volume 3: Applications*. De Gruyter, 2020. DOI: 10.1515/9783110499001.

[24] M. Benning, E. Celledoni, M. J. Ehrhardt, B. Owren, and C.-B. Schönlieb. “Deep learning as optimal control problems: Models and numerical methods”. In: *Journal of Computational Dynamics* 6.2 (2019), pp. 171–198. DOI: 10.3934/jcd.2019009.

[25] C. M. Bishop. “Pattern recognition and feed-forward networks”. In: *The MIT encyclopedia of the cognitive sciences*. Vol. 13. 2. MIT Press, 1999.

[26] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[27] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag. “What is the state of neural network pruning?” In: *Proceedings of machine learning and systems* 2 (2020), pp. 129–146.

[28] N. Bodla, B. Singh, R. Chellappa, and L. S. Davis. “Soft-NMS Improving Object Detection with One Line of Code”. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 5562–5570. DOI: 10.1109/ICCV.2017.593.

[29] M. Bonnaerens, M. Freiberger, and J. Dambre. “Anchor pruning for object detection”. In: *arXiv preprint arXiv:2104.00432* (2021).

[30] L. Bottou. “Online learning and stochastic approximations”. In: *On-line learning in neural networks* 17.9 (1998), pp. 9–42. DOI: 10.1017/CBO9780511569920.003.

[31] L. Bottou, F. E. Curtis, and J. Nocedal. “Optimization Methods for Large-Scale Machine Learning”. In: *SIAM Review* 60.2 (2018), pp. 223–311. DOI: 10.1137/16M1080173.

[32] C. Boutsidis, P. Drineas, and M. Magdon-Ismail. “Near Optimal Column-Based Matrix Reconstruction”. In: *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. 2011, pp. 305–314. DOI: 10.1109/FOCS.2011.21.

[33] C. Boutsidis, M. W. Mahoney, and P. Drineas. “An improved approximation algorithm for the column subset selection problem”. In: *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*. SIAM. 2009, pp. 968–977. DOI: 10.1137/1.9781611973068.105.

[34] C. Bowles, L. Chen, R. Guerrero, P. Bentley, R. Gunn, A. Hammers, D. A. Dickie, M. V. Hernández, J. Wardlaw, and D. Rueckert. “Gan augmentation: Augmenting training data using generative adversarial networks”. In: *arXiv preprint arXiv:1810.10863* (2018).

[35] O. Bretscher. *Linear Algebra with Applications*. Pearson Education, 2013.

[36] M. Bucher, S. Herbin, and F. Jurie. “Hard negative mining for metric learning based zero-shot classification”. In: *European Conference on Computer Vision, ECCV 2016 Workshops*. Springer. 2016, pp. 524–531. DOI: 10.1007/978-3-319-49409-8_45.

[37] M. Buckland and F. Gey. “The relationship between recall and precision”. In: *Journal of the American society for information science* 45.1 (1994), pp. 12–19. DOI: 10.1002/(SICI)1097-4571(199401)45:1<12::AID-ASI2>3.0.CO;2-L.

[38] T. Bui-Thanh, M. Damodaran, and K. Willcox. “Proper orthogonal decomposition extensions for parametric applications in compressible aerodynamics”. In: *21st AIAA Applied Aerodynamics Conference*. 2003, p. 4213. DOI: 10.2514/6.2003-4213.

[39] T. Bui-Thanh, M. Damodaran, and K. Willcox. “Aerodynamic Data Reconstruction and Inverse Design using Proper Orthogonal Decomposition”. In: *AIAA journal* 42.8 (2004), pp. 1505–1516. DOI: 10.2514/1.2159.

[40] H. Cai, L. Zhu, and S. Han. “Proxylessnas: Direct neural architecture search on target task and hardware”. In: *5th International Conference on Learning Representations (ICLR 2019)* (2019).

[41] R. y Cajal. *Textura del sistema nervioso del hombre y de los vertebrados : estudios sobre el plan estructural y composición histológica de los centros nerviosos adicionados de consideraciones fisiológicas fundadas en los nuevos descubrimientos. Volumen II.* Imprenta y Librería de Nicolás Moya, Madrid, 1904.

[42] O. Calin. *Deep Learning Architectures: A Mathematical Approach*. Springer Series in the Data Sciences. Springer International Publishing, 2020. DOI: 10.1007/978-3-030-36721-3.

[43] A. Cauchy. “Methode generale pour la resolution des systemes d’equations simultanees”. In: *C.R. Acad. Sci. Paris* 25 (1847), pp. 536–538.

[44] E. Celledoni, M. J. Ehrhardt, C. Etmann, R. I. McLachlan, B. Owren, C.-B. Schönlieb, and F. Sherry. “Structure-preserving deep learning”. In: *European Journal of Applied Mathematics* 32.5 (2021), pp. 888–936. DOI: 10.1017/S0956792521000139.

[45] G. Chen, W. Choi, X. Yu, T. Han, and M. Chandraker. “Learning Efficient Object Detection Models with Knowledge Distillation”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Curran Associates Inc., 2017, pp. 742–751. DOI: 10.5555/3294771.3294842.

[46] W. Chen, Q. Wang, J. S. Hesthaven, and C. Zhang. “Physics-informed machine learning for reduced-order modeling of nonlinear problems”. In: *Journal of Computational Physics* 446 (2021), p. 110666. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2021.110666.

[47] F. Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

[48] D. C. Cirean, U. Meier, L. M. Gambardella, and J. Schmidhuber. "Deep, Big, Simple Neural Nets for Handwritten Digit Recognition". In: *Neural Computation* 22.12 (2010), pp. 3207–3220. DOI: 10.1162/neco_a_00052.

[49] T. J. Cleophas and A. H. Zwinderman. *Machine Learning in Medicine-Cookbook*. Springer, 2014. DOI: 10.1007/978-3-319-04181-0.

[50] M. A. Cohen and S. Grossberg. "Absolute stability of global pattern formation and parallel memory storage by competitive neural networks". In: *IEEE transactions on systems, man, and cybernetics* 5 (1983), pp. 815–826. DOI: 10.1109/TSMC.1983.6313075.

[51] P. G. Constantine. *Active subspaces: Emerging ideas for dimension reduction in parameter studies*. Vol. 2. SIAM Spotlights. SIAM, 2015. DOI: 10.1137/1.9781611973860.

[52] P. G. Constantine and A. Doostan. "Time-dependent global sensitivity analysis with active subspaces for a lithium ion battery model". In: *Statistical Analysis and Data Mining: The ASA Data Science Journal* 10.5 (2017), pp. 243–262. DOI: 10.1002/sam.11347.

[53] P. G. Constantine, E. Dow, and Q. Wang. "Active Subspace Methods in Theory and Practice: Applications to Kriging Surfaces". In: *SIAM Journal on Scientific Computing* 36.4 (2014), A1500–A1524. ISSN: 1095-7197. DOI: 10.1137/130916138.

[54] D. Coscia, L. Meneghetti, N. Demo, G. Stabile, and G. Rozza. "A continuous trainable filter for convolution with unstructured data". 2022.

[55] M. Courbariaux, Y. Bengio, and J.-P. David. "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 3123–3131.

[56] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1". In: *arXiv preprint arXiv:1602.02830* (2016).

[57] C. Cui, K. Zhang, T. Daulbaev, J. Gusak, I. Oseledets, and Z. Zhang. "Active subspace of neural networks: Structural analysis and universal attacks". In: *SIAM Journal on Mathematics of Data Science* 2.4 (2020), pp. 1096–1122. DOI: 10.1137/19M1296070.

[58] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314. DOI: 10.1007/BF02551274.

[59] B. Cyganek. *Object detection and recognition in digital images: theory and practice*. John Wiley & Sons, 2013.

[60] J. Dai, K. He, and J. Sun. "Instance-Aware Semantic Segmentation via Multi-task Network Cascades". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 3150–3158. DOI: 10.1109/CVPR.2016.343.

[61] J. Dai, Y. Li, K. He, and J. Sun. "R-FCN: Object Detection via Region-based Fully Convolutional Networks". In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett. Vol. 29. Curran Associates, Inc., 2016.

[62] M. P. Deisenroth, A. A. Faisal, and C. S. Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

[63] N. Demo, M. Strazzullo, and G. Rozza. "An extended physics informed neural network for preliminary analysis of parametric optimal control problems". In: *arXiv preprint arXiv:2110.13530* (2021).

[64] N. Demo, M. Tezzele, G. Gustin, G. Lavini, and G. Rozza. “Shape Optimization by means of Proper Orthogonal Decomposition and Dynamic Mode Decomposition”. In: *Technology and Science for the Ships of the Future: Proceedings of NAV 2018: 19th International Conference on Ship & Maritime Research*. IOS Press, 2018, pp. 212–219. DOI: 10.3233/978-1-61499-870-9-212.

[65] N. Demo, M. Tezzele, A. Mola, and G. Rozza. “A complete data-driven framework for the efficient solution of parametric shape design and optimisation in naval engineering problems”. In: *VIII International Conference on Computational Methods in Marine Engineering*. 2019.

[66] N. Demo, M. Tezzele, A. Mola, and G. Rozza. “An efficient shape parametrisation by free-form deformation enhanced by active subspace for hull hydrodynamic ship design problems in open source environment”. In: *The 28th International Ocean and Polar Engineering Conference*. 2018.

[67] N. Demo, M. Tezzele, and G. Rozza. “A supervised learning approach involving active subspaces for an efficient genetic algorithm in high-dimensional optimization problems”. In: *SIAM Journal on Scientific Computing* 43.3 (2021), B831–B853. DOI: 10.1137/20M1345219.

[68] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[69] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li. “GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework”. In: *Neural Networks: the official journal of the International Neural Network Society* 100 (2018), pp. 49–58. DOI: 10.1016/j.neunet.2018.01.010.

[70] L. Deng and Y. Liu. *Deep learning in natural language processing*. Springer, 2018. DOI: 10.1007/978-981-10-5209-5.

[71] A. Deshpande and S. Vempala. “Adaptive sampling and fast low-rank matrix approximation”. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2006, pp. 292–303. DOI: 10.1007/11830924_28.

[72] J. J. DiCarlo, D. Zoccolan, and N. C. Rust. “How does the brain solve visual object recognition?” In: *Neuron* 73.3 (2012), pp. 415–434. DOI: 10.1016/j.neuron.2012.01.010.

[73] P. Drineas and A. Zouzias. “A note on element-wise matrix sparsification via a matrix-valued Bernstein inequality”. In: *Information Processing Letters* 111.8 (2011), pp. 385–389. DOI: 10.1016/j.ipl.2011.01.010.

[74] J. Duchi, E. Hazan, and Y. Singer. “Adaptive subgradient methods for online learning and stochastic optimization.” In: *Journal of machine learning research* 12.7 (2011), pp. 2121–2159.

[75] C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia. “Incorporating Second-Order Functional Knowledge for Better Option Pricing”. In: *Proceedings of the 13th International Conference on Neural Information Processing Systems*. NIPS’00. Denver, CO: MIT Press, 2000, pp. 451–457.

[76] S. Dutta, M. W. Farthing, E. Perracchione, G. Savant, and M. Putti. “A greedy non-intrusive reduced order model for shallow water equations”. In: *Journal of Computational Physics* 439 (2021), p. 110378. DOI: 10.1016/j.jcp.2021.110378.

[77] J. L. Elman. “Finding structure in time”. In: *Cognitive science* 14.2 (1990), pp. 179–211. DOI: 10.1207/s15516709cog1402_1.

[78] D. Erhan, A. Courville, Y. Bengio, and P. Vincent. “Why Does Unsupervised Pre-training Help Deep Learning?” In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics.* Ed. by Y. W. Teh and M. Titterington. Vol. 9. Proceedings of Machine Learning Research. PMLR, 2010, pp. 201–208.

[79] D. Erhan, C. Szegedy, A. Toshev, and D. Anguelov. “Scalable Object Detection Using Deep Neural Networks”. In: (2014), pp. 2155–2162. DOI: 10.1109/CVPR.2014.276.

[80] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean. “A guide to deep learning in healthcare”. In: *Nature medicine* 25.1 (2019), pp. 24–29. DOI: 10.1038/s41591-018-0316-z.

[81] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. “The Pascal Visual Object Classes (VOC) Challenge”. In: *International Journal of Computer Vision* 88.2 (June 2010), pp. 303–338. DOI: 10.1007/s11263-009-0275-4.

[82] M. Everingham, S. Eslami, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. “The pascal visual object classes challenge: A retrospective”. In: *International journal of computer vision* 111.1 (2015), pp. 98–136. DOI: 10.1007/s11263-014-0733-5.

[83] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. “The pascal visual object classes (voc) challenge”. In: *International journal of computer vision* 88.2 (2010), pp. 303–338. DOI: 10.1007/s11263-009-0275-4.

[84] B. Farley and W. Clark. “Simulation of self-organizing systems by digital computer”. In: *Transactions of the IRE Professional Group on Information Theory* 4.4 (1954), pp. 76–84. DOI: 10.1109/TIT.1954.1057468.

[85] T. L. Fine. *Feedforward neural network methodology.* Springer Science & Business Media, 2006.

[86] T. Gale, E. Elsen, and S. Hooker. “The state of sparsity in deep neural networks”. In: *arXiv preprint arXiv:1902.09574* (2019).

[87] M. W. Gardner and S. Dorling. “Artificial neural networks (the multilayer perceptron)a review of applications in the atmospheric sciences”. In: *Atmospheric environment* 32.14-15 (1998), pp. 2627–2636. DOI: 10.1016/S1352-2310(97)00447-0.

[88] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for computer algebra.* Springer Science & Business Media, 1992. DOI: 10.1007/b102438.

[89] R. G. Ghanem and P. D. Spanos. *Stochastic finite elements: a spectral approach.* Courier Corporation, 2003.

[90] M. Ghashami, E. Liberty, J. M. Phillips, and D. P. Woodruff. “Frequent Directions: Simple and Deterministic Matrix Sketching”. In: *SIAM Journal on Computing* 45.5 (2016), pp. 1762–1792. DOI: 10.1137/15M1009718.

[91] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer. “SqueezeNext: Hardware-Aware Neural Network Design”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW).* 2018, pp. 1719–171909. DOI: 10.1109/CVPRW.2018.00215.

[92] S. Ghosh, S. K. K. Srinivasa, P. Amon, A. Hutter, and A. Kaup. “Deep Network Pruning for Object Detection”. In: *2019 IEEE International Conference on Image Processing (ICIP).* 2019, pp. 3915–3919. DOI: 10.1109/ICIP.2019.8803505.

[93] P. B. Gibbons and Y. Matias. "Synopsis Data Structures for Massive Data Sets". In: *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '99. Baltimore, Maryland, USA: Society for Industrial and Applied Mathematics, 1999, pp. 909–910. ISBN: 0898714346. DOI: 10.5555/314500.315083.

[94] R. Girshick. "Fast R-CNN". In: *International Conference on Computer Vision (ICCV)*. 2015.

[95] R. Girshick, J. Donahue, T. Darrell, and J. Malik. "Region-Based Convolutional Networks for Accurate Object Detection and Segmentation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.1 (2016), pp. 142–158. DOI: 10.1109/TPAMI.2015.2437384.

[96] R. Girshick, J. Donahue, T. Darrell, and J. Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2014.

[97] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[98] X. Glorot, A. Bordes, and Y. Bengio. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by G. Gordon, D. Dunson, and M. Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, 2011, pp. 315–323.

[99] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[100] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger. Vol. 27. Curran Associates, Inc., 2014.

[101] L. A. Goodman. "On the Exact Variance of Products". In: *Journal of the American Statistical Association* 55.292 (1960), pp. 708–713. DOI: 10.1080/01621459.1960.10483369.

[102] M. Gordon and M. Kochen. "Recall-precision trade-off: A derivation". In: *Journal of the American Society for Information Science* 40.3 (1989), pp. 145–151. DOI: https://doi.org/10.1002/(SICI)1097-4571(198905)40:3<145::AID-ASI1>3.0.CO;2-I.

[103] J. Gou, B. Yu, S. J. Maybank, and D. Tao. "Knowledge distillation: A survey". In: *International Journal of Computer Vision* 129.6 (2021), pp. 1789–1819. DOI: 10.1007/s11263-021-01453-z.

[104] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks". In: *Proceedings of the 23rd International Conference on Machine Learning*. 2006, pp. 369–376. DOI: 10.1145/1143844.1143891.

[105] Z. J. Grey and P. G. Constantine. "Active subspaces of airfoil shape parameterizations". In: *AIAA Journal* 56.5 (2018), pp. 2003–2017. DOI: 10.2514/1.J056054.

[106] S. Grossberg. "Contour Enhancement, Short Term Memory, and Constancies in Reverberating Neural Networks". In: *Studies in Applied Mathematics* 52.3 (1973), pp. 213–257. DOI: 10.1002/sapm1973523213.

[107] P. Grother. "NIST Special Database 19 Handprinted Forms and Characters Database". In: World Wide Web-Internet and Web Information Systems, 1995. DOI: 10.18434/T4H01C.

[108] J. Hadamard. *Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées*. Académie des sciences. Mémoires. Imprimerie nationale, 1908.

[109] A. Halevy, P. Norvig, and F. Pereira. “The unreasonable effectiveness of data”. In: *IEEE intelligent systems* 24.2 (2009), pp. 8–12. DOI: 10.1109/MIS.2009.36.

[110] S. Han, H. Mao, and W. J. Dally. “Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding”. In: *4th International Conference on Learning Representations*. 2016.

[111] S. Han, J. Pool, J. Tran, and W. J. Dally. “Learning Both Weights and Connections for Efficient Neural Networks”. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’15. Montreal, Canada: MIT Press, 2015, pp. 1135–1143.

[112] M. Harrison. *Machine Learning Pocket Reference: Working with Structured Data in Python*. O’Reilly Media, 2019. ISBN: 9781492047513. URL: https://books.google.it/books?id=RoirDwAAQBAJ.

[113] D. Hartmann and H. Van der Auweraer. “Digital twins”. In: *Progress in Industrial Mathematics: Success Stories*. Ed. by C. M., P. C., and Q. P. Springer, 2021, pp. 3–17. DOI: 10.1007/978-3-030-61844-5_1.

[114] B. Hassibi and D. Stork. “Second order derivatives for network pruning: Optimal Brain Surgeon”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Hanson, J. Cowan, and C. Giles. Vol. 5. Morgan-Kaufmann, 1992.

[115] S. Haykin. *Neural Networks and Learning Machines*. Vol. 10. Neural networks and learning machines. Prentice Hall, 2009. ISBN: 9780131471399.

[116] S. Haykin. *Neural Networks: A Comprehensive Foundation*. 2nd. USA: Prentice Hall PTR, 1998. DOI: 10.5555/521706.

[117] K. He, G. Gkioxari, P. Dollár, and R. Girshick. “Mask R-CNN”. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2980–2988. DOI: 10.1109/ICCV.2017.322.

[118] K. He, X. Zhang, S. Ren, and J. Sun. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[119] K. He, X. Zhang, S. Ren, and J. Sun. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.

[120] K. He, X. Zhang, S. Ren, and J. Sun. “Identity mappings in deep residual networks”. In: *European conference on computer vision, ECCV 2016*. Springer. 2016, pp. 630–645. DOI: 10.1007/978-3-319-46493-0_38.

[121] D. O. Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[122] M. W. Hess, A. Quaini, and G. Rozza. “A comparison of reduced-order modeling approaches using artificial neural networks for PDEs with bifurcating solutions”. In: *ETNA - Electronic Transactions on Numerical Analysis* 56 (2022), pp. 52–65. DOI: 10.1553/etna_vol56s52.

[123] M. W. Hess, A. Quaini, and G. Rozza. “A Data-Driven Surrogate Modeling Approach for Time-Dependent Incompressible Navier-Stokes Equations with Dynamic Mode Decomposition and Manifold Interpolation”. 2022.

[124] M. W. Hess, A. Quaini, and G. Rozza. “Data-Driven Enhanced Model Reduction for Bifurcating Models in Computational Fluid Dynamics”. 2022.

[125] J. Hesthaven and S. Ubbiali. “Non-intrusive reduced order modeling of nonlinear problems using neural networks”. In: *Journal of Computational Physics* 363 (2018), pp. 55–78. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2018.02.037.

[126] J. S. Hesthaven, G. Rozza, and B. Stamm. *Certified Reduced Basis Methods for Parametrized Partial Differential Equations*. 1st ed. Springer Briefs in Mathematics. Switzerland: Springer, 2015, p. 135. DOI: 10.1007/978-3-319-22470-1.

[127] S. Hijazi, G. Stabile, A. Mola, and G. Rozza. "Data-driven POD-Galerkin reduced order model for turbulent flows". In: *Journal of Computational Physics* 416 (2020), p. 109513. DOI: 10.1016/j.jcp.2020.109513.

[128] G. E. Hinton and R. R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks". In: *Science* 313.5786 (2006), pp. 504–507. DOI: 10.1126/science.1127647.

[129] G. Hinton and T. J. Sejnowski. *Unsupervised learning: foundations of neural computation*. MIT press, 1999. DOI: 10.7551/mitpress/7011.001.0001.

[130] G. Hinton, O. Vinyals, and J. Dean. "Distilling the Knowledge in a Neural Network". In: *NIPS Deep Learning and Representation Learning Workshop*. 2015.

[131] G. E. Hinton and R. Zemel. "Autoencoders, Minimum Description Length and Helmholtz Free Energy". In: *Advances in Neural Information Processing Systems*. Ed. by J. Cowan, G. Tesauro, and J. Alspector. Vol. 6. Morgan-Kaufmann, 1994, pp. 3–10.

[132] S. Hochreiter. "The vanishing gradient problem during learning recurrent neural nets and problem solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116. DOI: 10.1142/S0218488598000094.

[133] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies". In: *A Field Guide to Dynamical Recurrent Neural Networks*. Ed. by S. C. Kremer and J. F. Kolen. IEEE Press, 2001.

[134] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[135] J. J. Hopfield. "Neural networks and physical systems with emergent collective computational abilities". In: *Proceedings of the national academy of sciences* 79.8 (1982), pp. 2554–2558. DOI: 10.1073/pnas.79.8.2554.

[136] R. A. Horn and C. R. Johnson. *Matrix analysis*. Cambridge University Press, Cambridge, 1985. DOI: 10.1017/CBO9780511810817.

[137] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366. DOI: 10.1016/0893-6080(89)90020-8.

[138] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[139] Y. Huang and Y. Chen. "Survey of State-of-Art Autonomous Driving Technologies with Deep Learning". In: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 2020, pp. 221–228. DOI: 10.1109/QRS-C51114.2020.00045.

[140] D. H. Hubel and T. N. Wiesel. "Effects of monocular deprivation in kittens". In: *Naunyn-Schmiedebergs Archiv für Experimentelle Pathologie und Pharmakologie* 248.6 (1964), pp. 492–497. DOI: 10.1007/BF00348878.

[141] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size". In: *arXiv preprint arXiv:1602.07360* (2016).

[142] E. Ising. “Contribution to the Theory of Ferromagnetism”. In: *Zeitschrift für Physik volume* 31 (1925), pp. 253–258. DOI: 10.1007/BF02980577.

[143] A. Ivakhnenko, A. Ivakhnenko, V. Lapa, V. LAPA, V. Lapa, and R. McDonough. *Cybernetics and Forecasting Techniques*. Modern analytic and computational methods in science and mathematics. American Elsevier Publishing Company, 1967.

[144] K. E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.

[145] J. Janai, F. Güney, A. Behl, and A. Geiger. “Computer Vision for Autonomous Vehicles: Problems, Datasets and State-of-the-Art”. In: *Foundations and Trends in Computer Graphics and Vision* 12 (2020), pp. 1–308. DOI: 10.1561/0600000079.

[146] C. Janya-Anurak. *Framework for analysis and identification of nonlinear distributed parameter systems using Bayesian uncertainty quantification based on Generalized Polynomial Chaos*. Vol. 31. KIT Scientific Publishing, 2017. DOI: 10.5445/KSP/1000066940.

[147] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. “What is the best multi-stage architecture for object recognition?” In: *2009 IEEE 12th International Conference on Computer Vision*. 2009, pp. 2146–2153. DOI: 10.1109/ICCV.2009.5459469.

[148] L. Jiang, W. Nie, J. Zhu, X. Gao, and B. Lei. “Lightweight object detection network model suitable for indoor mobile robots”. In: *Journal of Mechanical Science and Technology* (2022). DOI: 10.1007/s12206-022-0138-2.

[149] X. Jiang, A. Hadid, Y. Pang, E. Granger, and X. Feng. *Deep Learning in object detection and recognition*. Springer, 2019. DOI: 10.1007/978-981-10-5152-4.

[150] L. P. Kaelbling, M. L. Littman, and A. W. Moore. “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285. DOI: 10.1613/jair.301.

[151] U. Kamath, J. Liu, and J. Whitaker. *Deep learning for NLP and speech recognition*. Vol. 84. Springer, 2019. DOI: 10.1007/978-3-030-14596-5.

[152] R. M. Karp, S. Shenker, and C. H. Papadimitriou. “A Simple Algorithm for Finding Frequent Elements in Streams and Bags”. In: *ACM Trans. Database Syst.* 28.1 (2003), pp. 51–55. DOI: 10.1145/762471.762473.

[153] R. Ke, A. Bugeau, N. Papadakis, P. Schuetz, and C.-B. Schönlieb. *A multi-task U-net for segmentation with lazy labels*. 2020.

[154] N. S. Keskar and R. Socher. *Improving Generalization Performance by Switching from Adam to SGD*. 2017. eprint: 1712.07628.

[155] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi. “A survey of the recent architectures of deep convolutional neural networks”. In: *Artificial Intelligence Review* 53.8 (2020), pp. 5455–5516. DOI: 10.1007/s10462-020-09825-6.

[156] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi. “A survey of the recent architectures of deep convolutional neural networks”. In: *Artificial Intelligence Review* 53.8 (2020), pp. 5455–5516. DOI: 10.1007/s10462-020-09825-6.

[157] T. Kim, J. Oh, N. Y. Kim, S. Cho, and S.-Y. Yun. “Comparing Kullback-Leibler Divergence and Mean Squared Error Loss in Knowledge Distillation”. In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*. Ed. by Z.-H. Zhou. International Joint Conferences on Artificial Intelligence Organization, 2021, pp. 2628–2635. DOI: 10.24963/ijcai.2021/362.

[158] D. P. Kingma and J. Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Y. Bengio and Y. LeCun. 2015.

[159] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671.

[160] D. E. Knuth. “Two Notes on Notation”. In: *The American Mathematical Monthly* 99.5 (1992), pp. 403–422. DOI: 10.2307/2325085. (Visited on 06/14/2022).

[161] T. Kohonen. “Correlation Matrix Memories”. In: *IEEE Transactions on Computers* C-21.4 (1972), pp. 353–359. DOI: 10.1109/TC.1972.5008975.

[162] T. Kolarik and G. Rudorfer. “Time Series Forecasting Using Neural Networks”. In: *Proceedings of the International Conference on APL: The Language and Its Applications: The Language and Its Applications*. APL ’94. Antwerp, Belgium: Association for Computing Machinery, 1994, pp. 86–94. ISBN: 0897916751. DOI: 10.1145/190271.190290.

[163] C. Koylu, C. Zhao, and W. Shao. “Deep Neural Networks and Kernel Density Estimation for Detecting Human Activity Patterns from Geo-Tagged Images: A Case Study of Birdwatching on Flickr”. In: *ISPRS International Journal of Geo-Information* 8 (Jan. 2019), p. 45. DOI: 10.3390/ijgi8010045.

[164] I. Krasin, T. Duerig, N. Alldrin, V. Ferrari, S. Abu-El-Haija, A. Kuznetsova, H. Rom, J. Uijlings, S. Popov, S. Kamali, M. Malloci, J. Pont-Tuset, A. Veit, S. Belongie, V. Gomes, A. Gupta, C. Sun, G. Chechik, D. Cai, Z. Feng, D. Narayanan, and K. Murphy. “OpenImages: A public dataset for large-scale multi-label and multi-class image classification.” In: *Dataset available from https://storage.googleapis.com/openimages/web/index.html* (2017).

[165] D. Kriesel. *A Brief Introduction to Neural Networks*. Zeta2. 2007. URL: http://www.dkriesel.com/science/neural_networks.

[166] A. Krizhevsky and G. Hinton. “Learning multiple layers of features from tiny images”. In: *Master’s thesis, Department of Computer Science, University of Toronto* (2009).

[167] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[168] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari. “The Open Images Dataset V4: Unified Image Classification, Object Detection, and Visual Relationship Detection at Scale”. In: *International Journal of Computer Vision* 128 (Mar. 2020). DOI: 10.1007/s11263-020-01316-z.

[169] M. Lapin, M. Hein, and B. Schiele. “Top-k Multiclass SVM”. In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015.

[170] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[171] Y. LeCun. “Generalization and network design strategies”. In: *Connectionism in perspective* 19.143-155 (1989), p. 18.

[172] Y. LeCun and Y. Bengio. “Convolutional networks for images, speech, and time series”. In: *The handbook of brain theory and neural networks*. Vol. 3361. 10. MIT Press, 1995.

[173] Y. LeCun, Y. Bengio, and G. Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539.

[174] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. “Handwritten digit recognition with a back-propagation network”. In: *Advances in neural information processing systems* 2 (1989).

[175] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[176] Y. LeCun, J. Denker, and S. Solla. “Optimal brain damage”. In: *Advances in neural information processing systems* 2 (1989), pp. 598–605.

[177] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski. “A theoretical framework for back-propagation”. In: *Proceedings of the 1988 connectionist models summer school*. Vol. 1. 1988, pp. 21–28.

[178] C.-Y. Lee, P. W. Gallagher, and Z. Tu. “Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree”. In: *Artificial intelligence and statistics*. PMLR. 2016, pp. 464–472.

[179] K. Lee and K. T. Carlberg. “Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders”. In: *Journal of Computational Physics* 404 (2020), p. 108973. DOI: 10.1016/j.jcp.2019.108973.

[180] S. Leijnen and F. v. Veen. “The neural network zoo”. In: *Multidisciplinary Digital Publishing Institute Proceedings*. Vol. 47. 1. 2020, p. 9. DOI: 10.3390/proceedings2020047009.

[181] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. “Multilayer feedforward networks with a nonpolynomial activation function can approximate any function”. In: *Neural Networks* 6.6 (1993), pp. 861–867. DOI: 10.1016/S0893-6080(05)80131-5.

[182] G. Li, X. Li, Y. Wang, S. Zhang, Y. Wu, and D. Liang. “Knowledge Distillation for Object Detection via Rank Mimicking and Prediction-guided Feature Imitation”. In: *AAAI*. 2022.

[183] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. “Pruning Filters for Efficient ConvNets”. In: 2017.

[184] Q. Li and S. Hao. “An optimal control approach to deep learning and applications to discrete-weight neural networks”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 2985–2994.

[185] Y. Li, H. Qi, J. Dai, X. Ji, and Y. Wei. “Fully Convolutional Instance-Aware Semantic Segmentation”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 4438–4446. DOI: 10.1109/CVPR.2017.472.

[186] Y. Li and C. Lv. “SS-YOLO: An Object Detection Algorithm based on YOLOv3 and ShuffleNet”. In: *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. Vol. 1. 2020, pp. 769–772. DOI: 10.1109/ITNEC48623.2020.9085091.

[187] E. Liberty. “Simple and deterministic matrix sketching”. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2013, pp. 581–588.

[188] M. Lin, Q. Chen, and S. Yan. *Network In Network*. 2013. DOI: 10.48550/ARXIV.1312.4400.

[189] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. “Microsoft coco: Common objects in context”. In: *European conference on computer vision*. Springer. 2014, pp. 740–755. DOI: 10.1007/978-3-319-10602-1_48.

[190] J. Liu, A. I. Aviles-Rivero, H. Ji, and C.-B. Schönlieb. “Rethinking medical image reconstruction via shape prior, going deeper and faster: Deep joint indirect registration and reconstruction”. In: *Medical Image Analysis* 68 (2021), p. 101930. DOI: 10.1016/j.media.2020.101930.

[191] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, and M. Pietikäinen. “Deep learning for generic object detection: A survey”. In: *International journal of computer vision* 128.2 (2020), pp. 261–318. DOI: 10.1007/s11263-019-01247-4.

[192] L. Liu, L. Deng, X. Hu, M. Zhu, G. Li, Y. Ding, and Y. Xie. "Dynamic Sparse Graph for Efficient Deep Learning". In: *International Conference on Learning Representations*. 2019.

[193] L.-P. Liu, T. G. Dietterich, N. Li, and Z.-H. Zhou. "Transductive Optimization of Top k Precision". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. IJCAI'16. New York, New York, USA: AAAI Press, 2016, pp. 1781–1787. DOI: 10.5555/3060832.3060870.

[194] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. "SSD: Single Shot MultiBox Detector". In: *Computer Vision – ECCV 2016*. Ed. by B. Leibe, J. Matas, N. Sebe, and M. Welling. Cham: Springer International Publishing, 2016, pp. 21–37. DOI: 10.1007/978-3-319-46448-0_2.

[195] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. "SSD: Single Shot MultiBox Detector". In: *Computer Vision – ECCV 2016*. Springer International Publishing, 2016, pp. 21–37. DOI: 10.1007/978-3-319-46448-0_2.

[196] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. "Rethinking the Value of Network Pruning". In: *International Conference on Learning Representations*. 2019.

[197] D. Livingstone. *Artificial Neural Networks: Methods and Applications*. Jan. 2009. DOI: 10.1007/978-1-60327-101-1.

[198] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis. "DeepXDE: A deep learning library for solving differential equations". In: *SIAM Review* 63.1 (2021), pp. 208–228. DOI: 10.1137/19M1274067.

[199] J.-H. Luo, J. Wu, and W. Lin. "ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 5068–5076. DOI: 10.1109/ICCV.2017.541.

[200] H. V. Ly and H. T. Tran. "Modeling and control of physical processes using proper orthogonal decomposition". In: *Mathematical and computer modelling* 33.1-3 (2001), pp. 223–236. DOI: 10.1016/S0895-7177(00)00240-5.

[201] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018.

[202] W. Ma and J. Lu. *An Equivalence of Fully Connected Layer and Convolutional Layer*. 2017.

[203] K. Maharana, S. Mondal, and B. Nemade. "A Review: Data Pre-Processing and Data Augmentation Techniques". In: *Global Transitions Proceedings* (2022). DOI: 10.1016/j.gltp.2022.04.020.

[204] Z. Mao, A. D. Jagtap, and G. E. Karniadakis. "Physics-informed neural networks for high-speed flows". In: *Computer Methods in Applied Mechanics and Engineering* 360 (2020), p. 112789. DOI: 10.1016/j.cma.2019.112789.

[205] Z. Mariet and S. Sra. "Diversity networks: Neural network compression using determinantal point processes". In: *International Conference of Learning Representation (ICLR)* (2016).

[206] D. Marr. *Vision: A computational investigation into the human representation and processing of visual information*. MIT press, 2010.

[207] S. Marso and M. El Merouani. "Predicting financial distress using hybrid feedforward neural network with cuckoo search algorithm". In: *Procedia Computer Science* 170 (2020), pp. 1134–1140. DOI: 10.1016/j.procs.2020.03.054.

[208] W. S. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/BF02478259.

[209] L. R. Medsker and L. Jain. “Recurrent neural networks”. In: *Design and Applications* 5 (2001), pp. 64–67. DOI: 10.1201/9781420049176.

[210] L. Meneghetti, N. Demo, and G. Rozza. “A Dimensionality Reduction Approach for Convolutional Neural Networks”. In: *arXiv preprint* (2021). arXiv: 2110.09163 [cs.LG].

[211] L. Meneghetti, N. Demo, and G. Rozza. “A Proper Orthogonal Decomposition approach for parameters reduction of Single Shot Detector networks”. In: *arXiv preprint* (2022). arXiv: 2207.13551 [cs.LG].

[212] L. Meneghetti, N. Shah, M. Girfoglio, N. Demo, M. Tezzele, A. Lario, G. Stabile, and G. Rozza. “A Deep Learning Approach to Improve ROMs”. In: *Advanced Reduced Order Methods and Applications in Computational Fluid Dynamics*. Society for Industrial & Applied Mathematics, 2022.

[213] H. N. Mhaskar. “Approximation properties of a multilayered feedforward artificial neural network”. In: *Advances in Computational Mathematics* 1.1 (1993), pp. 61–80. DOI: 10.1007/BF02070821.

[214] G. A. Miller. “WordNet: a lexical database for English”. In: *Communications of the ACM* 38.11 (1995), pp. 39–41. DOI: 10.1145/219717.219748.

[215] M. Minsky and S. A. Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017. DOI: 10.7551/mitpress/11301.001.0001.

[216] S. Mishra and R. Molinaro. “Estimates on the generalization error of Physics-Informed Neural Networks for approximating PDEs”. In: *IMA Journal of Numerical Analysis* (2022). DOI: 10.1093/imanum/drab093.

[217] J. Misra and D. Gries. “Finding repeated elements”. In: *Science of Computer Programming* 2.2 (1982), pp. 143–152. DOI: 10.1016/0167-6423(82)90012-0.

[218] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[219] T. Mudumbi, N. Bian, Y. Zhang, and F. Hazoume. “An Approach Combined the Faster RCNN and Mobilenet for Logo Detection”. In: *Journal of Physics: Conference Series* 1284.1 (2019). DOI: 10.1088/1742-6596/1284/1/012072.

[220] B. Müller, J. Reinhardt, and M. T. Strickland. *Neural networks: an introduction*. Springer Science & Business Media, 1995. DOI: 10.1007/978-3-642-57760-4.

[221] S. Muthukrishnan et al. “Data streams: Algorithms and applications”. In: *Foundations and Trendső in Theoretical Computer Science* 1.2 (2005), pp. 117–236. DOI: 10.1561/0400000002.

[222] V. Nair and G. E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Haifa, Israel: Omnipress, 2010, pp. 807–814.

[223] K. Nakano. “Associatron-A Model of Associative Memory”. In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-2.3 (1972), pp. 380–388. DOI: 10.1109/TSMC.1972.4309133.

[224] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh, and K. Shaalan. “Speech Recognition Using Deep Neural Networks: A Systematic Review”. In: *IEEE Access* 7 (2019), pp. 19143–19165. DOI: 10.1109/ACCESS.2019.2896880.

[225] Y. Nesterov. “A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$”. In: vol. 269. 1983, pp. 543–547.

[226] K. Noda, H. Arie, Y. Suga, and T. Ogata. “Multimodal integration learning of robot behavior using deep neural networks”. In: *Robotics and Autonomous Systems* 62.6 (2014), pp. 721–736. DOI: 10.1016/j.robot.2014.03.003.

[227] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov. “Tensorizing Neural Networks”. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’15. MIT Press, 2015, pp. 442–450.

[228] D. L. Olson and D. Delen. *Advanced data mining techniques*. Springer Science & Business Media, 2008. DOI: 10.1007/978-3-540-76917-0.

[229] I. Oseledets. “Tensor-Train Decomposition”. In: *SIAM J. Scientific Computing* 33 (Jan. 2011), pp. 2295–2317. DOI: 10.1137/090752286.

[230] M. van Otterlo and M. Wiering. “Reinforcement Learning and Markov Decision Processes”. In: *Reinforcement Learning: State-of-the-Art*. Ed. by M. Wiering and M. van Otterlo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–42. DOI: 10.1007/978-3-642-27645-3_1.

[231] S. E. Palmer. *Vision science: Photons to phenomenology*. MIT press, 1999.

[232] S. J. Pan and Q. Yang. “A survey on transfer learning”. In: *IEEE Transactions on knowledge and data engineering* 22.10 (2009), pp. 1345–1359. DOI: doi:10.1109/TKDE.2009.191.

[233] D. Papapicco. “A neural network framework for reduced order modelling of non-linear hyperbolic equations in computational fluid dynamics”. In: *Master’s thesis, Politecnico di Torino* (2021).

[234] D. Papapicco, N. Demo, M. Girfoglio, G. Stabile, and G. Rozza. “The Neural Network shifted-proper orthogonal decomposition: A machine learning approach for non-linear reduction of hyperbolic equations”. In: *Computer Methods in Applied Mechanics and Engineering* 392 (2022), p. 114687. DOI: 10.1016/j.cma.2022.114687. arXiv: 2108.06558 [math.NA].

[235] D. Pasetto, A. Guadagnini, and M. Putti. “A reduced-order model for Monte Carlo simulations of stochastic groundwater flow”. In: *Computational Geosciences* 18.2 (2014), pp. 157–169. DOI: 10.1007/s10596-013-9389-4.

[236] D. Pasetto, A. Guadagnini, and M. Putti. “POD-based Monte Carlo approach for the solution of regional scale groundwater flow driven by randomly distributed recharge”. In: *Advances in Water Resources* 34.11 (2011), pp. 1450–1463. ISSN: 0309-1708. DOI: 10.1016/j.advwatres.2011.07.003.

[237] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035.

[238] P. Petersen and F. Voigtlaender. “Equivalence of approximation by convolutional neural networks and fully-connected networks”. In: *Proceedings of the American Mathematical Society* 148.4 (2020), pp. 1567–1581. DOI: 10.1090/proc/14789.

[239] F. Pichi, F. Ballarin, G. Rozza, and J. S. Hesthaven. “An artificial neural network approach to bifurcating phenomena in computational fluid dynamics”. 2021.

[240] H. A. Pierson and M. S. Gashler. “Deep learning in robotics: a review of recent research”. In: *Advanced Robotics* 31.16 (2017), pp. 821–835. DOI: 10.1080/01691864.2017.1365009.

[241] A. Pinkus. “Approximation theory of the MLP model in neural networks”. In: *Acta Numerica* 8 (1999), pp. 143–195. DOI: 10.1017/S0962492900002919.

[242] B. T. Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *Ussr computational mathematics and mathematical physics* 4.5 (1964), pp. 1–17. DOI: 10.1016/0041-5553(64)90137-5.

[243] M. Raissi, P. Perdikaris, and G. E. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707. DOI: 10.1016/j.jcp.2018.10.045.

[244] M. Raissi, A. Yazdani, and G. E. Karniadakis. “Hidden fluid mechanics: A Navier-Stokes informed deep learning framework for assimilating flow visualization data”. In: *arXiv preprint arXiv:1808.04327* (2018).

[245] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. “Xnor-net: Imagenet classification using binary convolutional neural networks”. In: *European conference on computer vision, ECCV 2016*. Vol. 9908. Springer. 2016, pp. 525–542. DOI: 10.1007/978-3-319-46493-0_32.

[246] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. “You Only Look Once: Unified, Real-Time Object Detection”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[247] J. Redmon and A. Farhadi. “YOLO9000: Better, Faster, Stronger”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 6517–6525. DOI: 10.1109/CVPR.2017.690.

[248] J. Redmon and A. Farhadi. “YOLOv3: An incremental improvement”. In: *arXiv preprint* (2018). eprint: 1804.02767.

[249] S. Ren, K. He, R. Girshick, and J. Sun. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015.

[250] B. D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996. DOI: 10.1017/CBO9780511812651.

[251] N. Rochester, J. Holland, L. Haibt, and W. Duda. “Tests on a cell assembly theory of the action of the brain, using a large digital computer”. In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 80–93. DOI: 10.1109/TIT.1956.1056810.

[252] R. Rojas. *Neural Networks: A Systematic Introduction*. Berlin, Heidelberg: Springer-Verlag, 1996.

[253] R. Rojas. “The Backpropagation Algorithm”. In: *Neural Networks*. Springer, 1996, pp. 149–182.

[254] F. Romor, M. Tezzele, A. Lario, and G. Rozza. “Kernel-based Active Subspaces with application to CFD parametric problems using Discontinuous Galerkin method”. In: *arXiv preprint arXiv:2008.12083* (2020).

[255] F. Romor, M. Tezzele, and G. Rozza. “ATHENA: Advanced Techniques for High dimensional parameter spaces to Enhance Numerical Analysis”. In: *Software Impacts* 10 (2021), p. 100133. ISSN: 2665-9638. DOI: 10.1016/j.simpa.2021.100133.

[256] O. Ronneberger, P. Fischer, and T. Brox. “U-net: Convolutional networks for biomedical image segmentation”. In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241. DOI: 10.1007/978-3-319-24574-4_28.

[257] F. Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain”. In: *Psychological review* 65.6 (1958), p. 386. DOI: 10.1037/h0042519.

[258] A. Rosenfeld and M. Thurston. “Edge and Curve Detection for Visual Scene Analysis”. In: *IEEE Transactions on Computers* C-20.5 (1971), pp. 562–569. DOI: 10.1109/T-C.1971.223290.

[259] W. Roth, G. Schindler, M. Zöhrer, L. Pfeifenberger, R. Peharz, S. Tschiatschek, H. Fröning, F. Pernkopf, and Z. Ghahramani. “Resource-efficient neural networks for embedded systems”. In: *arXiv preprint arXiv:2001.03048* (2020).

[260] G. Rozza, M. Malik, N. Demo, M. Tezzele, M. Girfoglio, G. Stabile, and A. Mola. “Advances in reduced order methods for parametric industrial problems in computational fluid dynamics”. In: *Proceedings of the 6th European Conference on Computational Mechanics: Solids, Structures and Coupled Problems, ECCM 2018 and 7th European Conference on Computational Fluid Dynamics, ECFD 2018*. 2020, pp. 59–76.

[261] G. Rozza, M. Hess, G. Stabile, M. Tezzele, and F. Ballarin. In: *Handbook on Model Reduction*. Ed. by P. Benner, S. Grivet-Talocia, A. Quarteroni, G. Rozza, W. H. A. Schilders, and L. M. Silveira. 2020. Chap. Basic Ideas and Tools for Projection-Based Model Reduction of Parametric Partial Differential Equations. DOI: 10.1515/9783110671490-001.

[262] G. Rozza, G. Stabile, and F. Ballarin. *Advanced Reduced Order Methods and Applications in Computational Fluid Dynamics*. Society for Industrial & Applied Mathematics, 2022.

[263] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0.

[264] D. E. Rumelhart and J. L. McClelland, eds. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986.

[265] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. “Imagenet large scale visual recognition challenge”. In: *International journal of computer vision* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[266] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran. “Low-rank matrix factorization for deep neural network training with high-dimensional output targets”. In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 6655–6659. DOI: 10.1109/ICASSP.2013.6638949.

[267] F. Salmoiraghi, F. Ballarin, G. Corsi, A. Mola, M. Tezzele, and G. Rozza. “Advances in geometrical parametrization and reduced order models and methods for computational fluid dynamics problems in applied sciences and engineering: overview and perspectives”. In: *Proceedings of the ECCOMAS Congress 2016, VII European Conference on Computational Methods in Applied Sciences and Engineering*. Ed. by M. Papadrakakis, V. Papadopoulos, G. Stefanou, and V. Plevris. 2016. DOI: 10.7712/100016.1867.8680.

[268] O. San, R. Maulik, and M. Ahmed. “An artificial neural network framework for reduced order modeling of transient flows”. In: *Communications in Nonlinear Science and Numerical Simulation* 77 (2019), pp. 271–287. ISSN: 1007-5704. DOI: 10.1016/j.cnsns.2019.04.025.

[269] M. San Biagio, S. Martelli, M. Crocco, M. Cristani, and V. Murino. “Encoding structural similarity by cross-covariance tensors for image classification”. In: *International Journal of Pattern Recognition and Artificial Intelligence* 28 (Sept. 2014), p. 19. DOI: 10.1142/S0218001414600088.

[270] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4510–4520. DOI: 10.1109/CVPR.2018.00474.

[271] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[272] A. Scagliotti. “Deep Learning Approximation of Diffeomorphisms via Linear-Control Systems”. In: *Math. Control Re. Fields, to appear* (2022). DOI: 10.48550/ARXIV.2110.12393.

[273] D. Scherer, A. Müller, and S. Behnke. “Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition”. In: *Artificial Neural Networks – ICANN 2010*. Ed. by K. Diamantaras, W. Duch, and L. S. Iliadis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 92–101. DOI: 10.1007/978-3-642-15825-4_10.

[274] W. Schilders, ed. *Mathematics: Key Enabling Technology for Scientific Machine Learning*. 2021.

[275] J. Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural networks* 61 (2015), pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003.

[276] T. J. Sejnowski. *The Deep Learning Revolution*. Cambridge, MA: MIT Press, 2018.

[277] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. “Overfeat: Integrated recognition, localization and detection using convolutional networks”. In: Publisher Copyright: Ī 2014 International Conference on Learning Representations, ICLR. All rights reserved.; 2nd International Conference on Learning Representations, ICLR 2014 ; Conference date: 14-04-2014 Through 16-04-2014. 2014.

[278] R. Setiono and G. Lu. “Image compression using a feedforward neural network”. In: *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN’94)*. Vol. 7. 1994, 4761–4765 vol.7. DOI: 10.1109/ICNN.1994.375045.

[279] N. V. Shah, M. Girfoglio, P. Quintela, G. Rozza, A. Lengomin, F. Ballarin, and P. Barral. *Finite element based model order reduction for parametrized one-way coupled steady state linear thermomechanical problems*. 2021. arXiv: 2111.08534 [math.NA].

[280] U. Shaham, A. Cloninger, and R. R. Coifman. “Provable approximation properties for deep neural networks”. In: *Applied and Computational Harmonic Analysis* 44.3 (2018), pp. 537–557. ISSN: 1063-5203. DOI: https://doi.org/10.1016/j.acha.2016.04.003. URL: https://www.sciencedirect.com/science/article/pii/S1063520316300033.

[281] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014. DOI: 10.1017/CBO9781107298019.

[282] R. Shanmugamani and S. Moore. *Deep Learning for Computer Vision: Expert Techniques to Train Advanced Neural Networks Using TensorFlow and Keras*. Packt Publishing, 2018.

[283] L. Shao, F. Zhu, and X. Li. “Transfer Learning for Visual Categorization: A Survey”. In: *IEEE Transactions on Neural Networks and Learning Systems* 26.5 (2015), pp. 1019–1034. DOI: 10.1109/TNNLS.2014.2330900.

[284] Y. Shin. “On the Convergence of Physics Informed Neural Networks for Linear Second-Order Elliptic and Parabolic Type PDEs”. In: *Communications in Computational Physics* 28 (June 2020), pp. 2042–2074. DOI: 10.4208/cicp.OA-2020-0193.

[285] C. Shorten and T. M. Khoshgoftaar. “A survey on image data augmentation for deep learning”. In: *Journal of big data* 6.1 (2019), pp. 1–48. DOI: 10.1186/s40537-019-0197-0.

[286] A. Siade, M. Putti, and W. Yeh. “Snapshot selection for groundwater model reduction using proper orthogonal decomposition”. In: *Water Resources Research - WATER RESOUR RES* 46 (Aug. 2010). DOI: 10.1029/2009WR008792.

[287] P. Siena, M. Girfoglio, F. Ballarin, and G. Rozza. “Data-driven reduced order modelling for patient-specific hemodynamics of coronary artery bypass grafts with physical and geometrical parameters”. In: *arXiv preprint arXiv:2203.13682* (2022).

[288] P. Siena, M. Girfoglio, and G. Rozza. “Fast and accurate numerical simulations for the study of coronary artery bypass grafts by artificial neural network”. In: *arXiv preprint arXiv:2201.01804* (2022).

[289] K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Y. Bengio and Y. LeCun. 2015.

[290] Y. Singh and A. S. Chauhan. “NEURAL NETWORKS IN DATA MINING.” In: *Journal of Theoretical & Applied Information Technology* 5.1 (2009). DOI: 10.9790/3021-04360106.

[291] D. F. Specht et al. “A general regression neural network”. In: *IEEE transactions on neural networks* 2.6 (1991), pp. 568–576. DOI: 10.1109/72.97934.

[292] W. Su, Y. Yuan, and M. Zhu. “A Relationship between the Average Precision and the Area Under the ROC Curve”. In: *Proceedings of the 2015 International Conference on The Theory of Information Retrieval*. ICTIR ’15. Northampton, Massachusetts, USA: Association for Computing Machinery, 2015, pp. 349–352. DOI: 10.1145/2808194.2809481.

[293] B. Sudret. “Global sensitivity analysis using polynomial chaos expansions”. In: *Reliability engineering & system safety* 93.7 (2008), pp. 964–979. DOI: 10.1016/j.ress.2007.04.002.

[294] F. Sultana, A. Sufian, and P. Dutta. “A Review of Object Detection Models Based on Convolutional Neural Network”. In: 2020, pp. 1–16. ISBN: 978-981-15-4287-9. DOI: 10.1007/978-981-15-4288-6_1.

[295] C. Sun, A. Shrivastava, S. Singh, and A. Gupta. “Revisiting unreasonable effectiveness of data in deep learning era”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 843–852.

[296] Y. Sun, C. Wang, and L. Qu. “An Object Detection Network for Embedded System”. In: *2019 IEEE International Conferences on Ubiquitous Computing Communications (IUCC) and Data Science and Computational Intelligence (DSCI) and Smart Computing, Networking and Services (SmartCNS)*. 2019, pp. 506–512. DOI: 10.1109/IUCC/DSCI/SmartCNS.2019.00110.

[297] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. “On the importance of initialization and momentum in deep learning”. In: *Proceedings of the 30th International conference on machine learning*. Ed. by S. Dasgupta and D. McAllester. Vol. 28. Proceedings of Machine Learning Research 3. PMLR, 2013, pp. 1139–1147.

[298] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. “Inception-v4, inception-resnet and the impact of residual connections on learning”. In: *Thirty-first AAAI conference on artificial intelligence*. 2017.

[299] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[300] C. Szegedy, S. Reed, D. Erhan, and D. Anguelov. *Scalable, high-quality object detection*. Tech. rep. arXiv, 2015. URL: http://arxiv.org/abs/1412.1441.

[301] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. “Rethinking the inception architecture for computer vision”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826. DOI: 10.1109/CVPR.2016.308.

[302] M. Tan and Q. Le. “Efficientnet: Rethinking model scaling for convolutional neural networks”. In: *International conference on machine learning*. PMLR. 2019, pp. 6105–6114.

[303] Z. Tang and P. A. Fishwick. “Feedforward neural nets as models for time series forecasting”. In: *ORSA journal on computing* 5.4 (1993), pp. 374–385. DOI: 10.1287/ijoc.5.4.374.

[304] W. Taylor. “Electrical simulation of some nervous system functional activities”. In: *IEEE Transactions on Information Theory - TIT* (Jan. 1956).

[305] A. Tewari and P. L. Bartlett. “On the Consistency of Multiclass Classification Methods”. In: *Learning Theory*. Ed. by P. Auer and R. Meir. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 143–157. DOI: 10.1007/11503415_10.

[306] M. Tezzele, N. Demo, A. Mola, and G. Rozza. “An integrated data-driven computational pipeline with model order reduction for industrial and applied mathematics”. In: *Novel Mathematics Inspired by Industrial Challenges*. Ed. by M. Günther and W. Schilders. Mathematics in Industry X. Springer International Publishing, 2022. DOI: 10.1007/978-3-030-96173-2_7.

[307] M. Tezzele, N. Demo, G. Stabile, A. Mola, and G. Rozza. “Enhancing CFD predictions in shape design problems by model and parameter space reduction”. In: *Advanced Modeling and Simulation in Engineering Sciences* 7.1 (2020), p. 40. DOI: 10.1186/s40323-020-00177-y.

[308] M. Tezzele, F. Salmoiraghi, A. Mola, and G. Rozza. “Dimension reduction in heterogeneous parametric spaces with application to naval engineering shape design problems”. In: *Advanced Modeling and Simulation in Engineering Sciences* 5.1 (2018), p. 25. DOI: 10.1186/s40323-018-0118-3.

[309] T. Tieleman, G. Hinton, et al. “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude”. In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[310] S. Trenn. “Multilayer Perceptrons: Approximation Order and Necessary Number of Hidden Units”. In: *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 19 (June 2008), pp. 836–44. DOI: 10.1109/TNN.2007.912306.

[311] J. R. Uijlings, K. E. Van De Sande, T. Gevers, and A. W. Smeulders. “Selective search for object recognition”. In: *International journal of computer vision* 104.2 (2013), pp. 154–171. DOI: 10.1007/s11263-013-0620-5.

[312] A. M. Uttley. “The classification of signals in the nervous system”. In: *Electroencephalography and clinical neurophysiology* 6.3 (1954), pp. 479–494. DOI: 10.1016/0013-4694(54)90064-4.

[313] J. E. Van Timmeren, D. Cester, S. Tanadini-Lang, H. Alkadhi, and B. Baessler. “Radiomics in medical imaginghow-to guide and critical reflection”. In: *Insights into imaging* 11.1 (2020), pp. 1–16. DOI: 10.1186/s13244-020-00887-2.

[314] K. R. Varshney. “Trustworthy Machine Learning”. In: *Chappaqua, NY, USA: Independently Published* (2022).

[315] H. Wang and B. Raj. *On the Origin of Deep Learning*. 2017.

[316] X. Wang. “Deep learning in object recognition, detection, and segmentation”. In: *Foundations and Trends in Signal Processing* 8.4 (2016), pp. 217–382.

[317] W. Weinan. “A Proposal on Machine Learning via Dynamical Systems”. English (US). In: *Communications in Mathematics and Statistics* 5.1 (2017), pp. 1–11. DOI: 10.1007/s40304-017-0103-z.

[318] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. “Learning Structured Sparsity in Deep Neural Networks”. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS’16. Barcelona, Spain: Curran Associates Inc., 2016, pp. 2082–2090. ISBN: 9781510838819.

[319] P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University, 1975.

[320] N. Wiener. “The Homogeneous Chaos”. In: *American Journal of Mathematics* 60.4 (1938), pp. 897–936. DOI: 10.2307/2371268.

[321] C. K. Williams and C. E. Rasmussen. *Gaussian processes for machine learning*. Vol. 2. 3. MIT press Cambridge, MA, 2006.

[322] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht. “The Marginal Value of Adaptive Gradient Methods in Machine Learning”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 4151–4161.

[323] A. Womg, M. J. Shafiee, F. Li, and B. Chwyl. “Tiny SSD: A Tiny Single-Shot Detection Deep Convolutional Neural Network for Real-Time Embedded Object Detection”. In: May 2018, pp. 95–101. DOI: 10.1109/CRV.2018.00023.

[324] B. Wu, F. Iandola, P. Jin, and K. Keutzer. “SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving”. In: July 2017, pp. 446–454. DOI: 10.1109/CVPRW.2017.60.

[325] C.-A. Xia, D. Pasetto, B. X. Hu, M. Putti, and A. Guadagnini. “Integration of moment equations in a reduced-order modeling strategy for Monte Carlo simulations of groundwater flow”. In: *Journal of Hydrology* 590 (2020), p. 125257. ISSN: 0022-1694. DOI: 10.1016/j.jhydrol.2020.125257.

[326] D. Xiu and G. E. Karniadakis. “The Wiener–Askey polynomial chaos for stochastic differential equations”. In: *SIAM journal on scientific computing* 24.2 (2002), pp. 619–644. DOI: 10.1137/S1064827501387826.

[327] D. Yarotsky. “Error bounds for approximations with deep ReLU networks”. In: *Neural Networks* 94 (2017), pp. 103–114. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2017.07.002. URL: https://www.sciencedirect.com/science/article/pii/S0893608017301545.

[328] T. Young, D. Hazarika, S. Poria, and E. Cambria. “Recent trends in deep learning based natural language processing”. In: *IEEE Computational intelligenCe magazine* 13.3 (2018), pp. 55–75.

[329] I. Zafar, G. Tzanidou, R. Burton, N. Patel, and L. Araujo. *Hands-on convolutional neural networks with TensorFlow: Solve computer vision problems with modeling in TensorFlow and Python*. Packt Publishing Ltd, 2018.

[330] O. Zahm, P. G. Constantine, C. Prieur, and Y. M. Marzouk. “Gradient-based dimension reduction of multivariate vector-valued functions”. In: *SIAM Journal on Scientific Computing* 42.1 (2020), A534–A558. DOI: 10.1137/18M1221837.

[331] M. J. Zaki and W. Meira Jr. *Data Mining and Machine Learning: Fundamental Concepts and Algorithms*. Cambridge University Press, 2020.

[332] M. Zancanaro, M. Mrosek, G. Stabile, C. Othmer, and G. Rozza. “Hybrid neural network reduced order modelling for turbulent flows with geometric parameters”. In: *Fluids* 6.8 (2021), p. 296. DOI: 10.3390/fluids6080296.

[333] M. D. Zeiler and R. Fergus. “Visualizing and understanding convolutional networks”. In: *European conference on computer vision, ECCV 2014.* Springer International Publishing. 2014, pp. 818–833. DOI: 10.1007/978-3-319-10590-1_53.

[334] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. “Dive into Deep Learning”. In: *arXiv preprint arXiv:2106.11342* (2021).

[335] X. Zhang, X. Zhou, M. Lin, and J. Sun. “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), pp. 6848–6856. DOI: 10.1109/CVPR.2018.00716.

[336] X. Zhang, J. Zou, K. He, and J. Sun. “Accelerating very deep convolutional networks for classification and detection”. In: *IEEE transactions on pattern analysis and machine intelligence* 38.10 (2015), pp. 1943–1955. DOI: 10.1109/TPAMI.2015.2502579.

[337] X. Zhang, Y.-H. Yang, Z. Han, H. Wang, and C. Gao. “Object Class Detection: A Survey”. In: *ACM Comput. Surv.* 46.1 (2013). DOI: 10.1145/2522968.2522978.

[338] Z.-Q. Zhao, P. Zheng, S.-T. Xu, and X. Wu. “Object Detection With Deep Learning: A Review”. In: *IEEE transactions on neural networks and learning systems* 30.11 (2019), pp. 3212–3232. DOI: 10.1109/tnnls.2018.2876865.

[339] Z.-Q. Zhao, P. Zheng, S.-T. Xu, and X. Wu. “Object Detection With Deep Learning: A Review”. In: *IEEE transactions on neural networks and learning systems* 30.11 (2019), pp. 3212–3232. DOI: 10.1109/tnnls.2018.2876865.

[340] Z. Zhong, L. Zheng, G. Kang, S. Li, and Y. Yang. “Random erasing data augmentation”. In: *Proceedings of the AAAI conference on artificial intelligence.* Vol. 34. 07. 2020, pp. 13001–13008. DOI: 10.1609/aaai.v34i07.7000.

[341] Y. Zhou and R. Chellappa. “Computation of optical flow using a neural network”. In: *IEEE 1988 International Conference on Neural Networks.* Vol. 2. 1988, pp. 71–78. DOI: 10.1109/ICNN.1988.23914.

[342] B. Zoph and Q. V. Le. “Neural architecture search with reinforcement learning”. In: *5th International Conference on Learning Representations (ICLR 2017)* (2017).

Symbols

$\mathcal{ANN}$ Artificial Neural Network seen as a function between inputs and outputs. 9, 85, 134

α Aspect ratio anchor boxes. 66

$\mathbf{b}$ Tensor representing the bias of an ANN. 8, 14, 15

$\mathbf{b}^{(\ell)}$ Tensor representing the bias for the ℓ-th layer of an ANN. 13, 41

$d_C^{(\ell)}$ Number of channels of the output tensor from layer ℓ in a Convolutional Neural Network. 38, 41

$\mathcal{CNN}$ Convolutional Neural Network seen as a function between inputs and outputs. 37, 38

$\mathcal{D}$ Generic dataset composed of pairs input-output, i.e. $\{\mathbf{x}^i,\mathbf{y}^i\}_{i=0}^{n_{\text{samples}}}$. 7, 8, 38, 49

$\mathcal{D}_{\textbf{test}}$ Testing dataset. 7, 8, 11, 26

$\mathcal{D}_{\textbf{train}}$ Training dataset. 7, 8, 11, 50, 85, 87, 88, 99, 100

$f_{\textbf{conv}}$ Convolution operation. 42, 43

$f_{\textbf{flatten}}$ Flatten function. 48, 49

$f_{\textbf{pool}}$ Pooling function. 45

$f_{\textbf{pool}}^{(\ell)}$ Pooling function relative to pooling layer ℓ. 46, 47

f_ℓ Function associated to layer ℓ in an Artificial Neural Network, mapping the output of layer $\ell - 1$ into a tensor in n_ℓ. 9, 38

$K^{(\ell)}$ Kernel or filter tensor related to the ℓ-th layer in a CNN. 40

$\mathbf{h}$ Signal propagated by the propagation function in an ANN, i.e. the input for the activation function. 6

$d_H^{(\ell)}$ Height of the output tensor from layer ℓ in a Convolutional Neural Network. 38, 41, 46

d_H^K Height of the kernel in a convolutional layer. 40, 49

$\mathbf{K}$ Kernel or filter for a convolutional layer in a Convolutional Neural Network. 39, 40, 49

l Index of the cut-off layer. 85–87, 98, 99, 134

ℓ Layer ℓ in an Artificial Neural Network. 9, 38, 40–42, 46, 49, 79, 133, 134

$\mathcal{L}$ Loss function. 14, 15, 17, 19, 25, 50, 72, 79

$\mathcal{L}_{\mathbf{cls}}$ Loss function for classification. 73, 74, 79

$\mathcal{L}_{\mathbf{loc}}$ Loss function for box prediction (localization). 73, 74, 79, 80

$n_{\mathbf{class}}$ Number of classes in a dataset, i.e. number of neurons in the output layer of an Convolutional Neural Network. 50, 51, 73, 75, 76, 79, 98

n_ℓ Number of neurons composing layer ℓ of an Artificial Neural Network. 9, 42, 133

L Number of hidden layers in an Artificial Neural Network. 8, 12, 38

$n_{\mathbf{in}}$ Number of neurons in the input layer of an Artificial Neural Network. 8

$n_{\mathbf{out}}$ Number of neurons in the output layer of an Artificial Neural Network. 8, 38

$\mathcal{N}$ Set of neurons for an ANN. 5, 7, 9

n_K Number of filters/kernels in a convolutional layer. 40

$\mathcal{Obj\_Det}$ Object Detector seen as a function between inputs and outputs. 97, 99

$\tilde{p}$ Padding parameter of a convolutional layer. ix, 42

$\mathcal{ANN}^{l}_{\mathbf{pre}}$ Pre-model obtained by cutting the $\mathcal{ANN}$ at cut-off layer l. 87

$\boldsymbol{W}_{\mathbf{proj}}$ Projection matrix for the reduction layer. 87, 88, 100

r Reduction parameter. 87, 88

ζ Scale anchor boxes. 66, 67

σ Activation function. 3, 4, 6, 9, 12, 16, 26, 28, 30, 43

s Stride parameter of a convolutional layer. ix, 40–42

W Weight matrix. 3, 5, 8, 14, 42

$W^{(\ell)}$ Weight matrix for layer ℓ. 13, 49

$d^{(\ell)}_{\mathcal{W}}$ Width of the output tensor from layer ℓ in a Convolutional Neural Network. 38, 41, 46

$d^{K}_{\mathcal{W}}$ Width of the kernel in a convolutional layer. 40, 49

$\mathbf{x}$ Tensor representing the input for a neuron or an Artificial Neural Network. 3, 7–9, 11, 15, 37, 56, 61, 78, 133

$\mathbf{x}^{(\ell)}$ Tensor representing the output for layer ℓ in an Artificial Neural Network. 9, 13, 38, 44

$\mathbf{x}^{(l)}$ Tensor representing the output of the pre-model. 87, 88, 100

$\mathbf{y}_{\mathbf{cls}}$ Tensor representing the expected classification output of an object detector. 73

$\hat{\mathbf{y}}_{\mathbf{cls}}$ Tensor representing the predicted classification output of an object detector. 73, 98

$\mathbf{y}$ Tensor representing the expected output from a neuron or an Artificial Neural Network. 7, 8, 15, 26, 133

$\mathbf{y}_{\mathbf{loc}}$ Tensor representing the expected localization output of an object detector. 73

$\hat{\mathbf{y}}_{\mathbf{loc}}$ Tensor representing the predicted localization output of an object detector. 73, 98

$\hat{\mathbf{y}}$ Tensor representing the predicted output of a neuron or an Artificial Neural Network. 3, 8, 15, 26, 37, 38, 48, 49, 88

$\tilde{\mathbf{y}}$ Tensor representing the output of the reduction network $\mathcal{ANN}^{\mathrm{red}}$. 89

$\mathbf{z}$ Tensor representing the output of the reduction layer. 84, 88

Acronyms

AE Autoencoder. 11, 106

ANN Artificial Neural Network. ix, xiii, xv, xvi, 1–3, 5, 7–11, 14, 26, 27, 35–37, 42, 43, 81, 82, 85, 92, 99, 105, 133–135

AP Average Precision. 50, 51

AS Active Subspaces. v, xv, xvi, 81, 82, 86, 88, 95, 105, 109

CE Cross Entropy. 14

CNN Convolutional Neural Network. v, ix, xiii–xvi, 2, 10, 35–44, 46, 48–56, 58, 60, 61, 64, 67, 71–73, 78, 81, 82, 89–93, 96, 97, 99, 103, 105–107, 133, 134

FD Frequent Directions. 109, 110

FNN Feedforward Neural Network. v, xv, xvi, 3, 8–14, 18, 20–23, 26, 27, 37, 38, 48, 49, 84–86, 89, 95, 96, 105, 106

i.i.d. independent and identically distributed. 19–21, 27–31

IoU Intersection over Union. 68, 69, 74, 76, 77, 79, 80

MAE Mean Absolute Error. 14

mAP Mean Average Precision. 51, 101–103

MSE Mean-Squared Error. 13, 26

NMS Non-Maximum Suppression. 69, 80

PCE Polynomial Chaos Expansion. v, xv, xvi, 82, 84, 86, 96, 105

POD Proper Orthogonal Decomposition. v, xiii, xv, xvi, 81–83, 86, 88, 95, 97, 98, 100, 102, 105, 106

RNN Recurrent Neural Network. 2, 10

RoI Region of Interest. 71–75

ROM Reduced Order Modeling. v, xiii, xv, 81–83, 86, 98, 105, 107

RPN Region Proposal Network. 73–75, 78

SGD Stochastic Gradient Descent. 19, 20, 22, 24, 36, 101

SVD Singular Value Decomposition. 83, 87, 88, 100, 109