Università degli Studi di Modena e Reggio Emilia

Dottorato di Ricerca in Matematica
in convenzione con l'Universtità degli Studi di Ferrara e l'Università degli Studi di Parma
Ciclo XXXII

A Framework for the Analysis of Machine Learning Systems

Candidato:
Matteo Spallanzani

Relatore:
Prof. Dr. Marko Bertogna

Correlatore:
Prof. Dr. Gian Paolo Leonardi

Coordinatore del Corso di Dottorato:
Prof. Dr. Cristian Giardinà

Prefazione

La storia della filosofia e della scienza è una storia di previsioni ed errori. Il fare previsioni si fonda sulla comprensione degli schemi del nostro ambiente, ed è fondamentale per comportamenti intelligenti come i processi decisionali e la pianificazione. Possiamo conoscere l'ambiente solamente attraverso strumenti di misura, siano essi sensi naturali o sensori artificiali, che ci forniscono dati. È proprio in questi dati che speriamo di trovare schemi. Non c'è modo di sapere se disponiamo di dati sufficienti per scoprire gli schemi che vorremmo conoscere. A volte non abbiamo dati sufficienti. A volte, inconsciamente, scartiamo dati utili. Altre volte, più maliziosamente, ignoriamo consapevolmente quelle misurazioni che potrebbero entrare in conflitto con gli schemi che vogliamo vedere nei dati. Altre volte ci concentriamo su schemi irrilevanti, destinati a fallire nel momento in cui raccoglieremo più dati. Il determinismo è stata la tendenza a preferire schemi descritti da relazioni funzionali chiamate *modelli deterministici*. All'inizio del secolo scorso, la statistica è emersa come il campo scientifico incaricato di formalizzare e organizzare le tecniche utilizzate da fisici, chimici, biologi e altri scienziati per scoprire gli schemi nei loro dati. Molti insiemi di misurazioni non sono compatibili con l'ipotesi deterministica. Pertanto, gli statistici hanno accettato l'esistenza di schemi probabilistici e hanno sviluppato strumenti più raffinati per quantificare l'incertezza intrinseca alle previsioni degli scienziati, i *modelli stocastici*.

L'introduzione delle macchine calcolatrici ha modificato profondamente la nostra interazione con i dati. Innanzitutto, i processori hanno automatizzato e accelerato i calcoli matematici. In secondo luogo, lo sviluppo della tecnologia delle memorie ha permesso di registrare quantità sempre maggiori di dati, superando di gran lunga quella degli archivi di periodi storici precedenti. La combinazione di questi fattori ha reso le domande "È possibile automatizzare l'estrazione di schemi dai dati?" ed "È possibile replicare comportamenti intelligenti?" sia significative che accessibili. Queste domande sono le basi dell'*apprendimento automatico* e dell'*intelligenza artificiale* e, come vedremo, sono intimamente correlate.

Il grado di miniaturizzazione dei moderni dispositivi elettronici consente di posizionarli in diversi ambienti. L'utilizzo degli schemi di questi ambienti è fondamentale per applicazioni economiche e sociali. Ad esempio, il monitoraggio in tempo reale dei macchinari industriali ne consente una manutenzione più efficace, riducendo i costi operativi. Un altro esempio è quello dei sensori biomedicali intelligenti in grado di monitorare pazienti con patologie cardiache: prevedere gli attacchi di cuore in tempo potrebbe salvar loro la vita. L'apprendimento automatico studia modelli e algoritmi in grado di trovare schemi in questi fenomeni. Durante l'ultimo decennio del secolo scorso, sono emerse tre tendenze tecnologiche. Innanzitutto, l'evoluzione della tecnologia Internet ha permesso di supportare ratei e volumi di scambio di dati sempre maggiori. In secondo luogo, la miniaturizzazione dei transistor ha permesso di integrare computer in piccoli sistemi a prezzi convenienti, collegando questi sistemi a Internet (*Internet of things*, IoT). Infine, la progettazione delle architetture dei calcolatori ha visto un cambio di paradigma verso sistemi paralleli, che ottengono capacità di processamento utilizzando multipli core più lenti anziché un'unica unità di elaborazione veloce.

Il tipico prodedimento per applicare algoritmi statistici ai dati raccolti da *computer edge* (ovvero computer con risorse di calcolo limitate che sono posizionati direttamente nell'ambiente) si basa su una connessione Internet: il computer con risorse limitate (il client) acquisisce dati e li trasmette a un computer o ad un insieme di computer (il server) in grado di elaborarli; dopodiché, l'output dell'algoritmo statistico viene ritrasmesso al client, che implementa la funzionalità desiderata usando questo risultato. Questa dipendenza da una connessione Internet ha potenziali ripercussioni sulle prestazioni del sistema. Ad esempio, la connessione potrebbe essere inaffidabile (i.e., i pacchetti di informazioni potrebbero andare persi attraverso la rete), necessitando la ritrasmissione e degradando la latenza della risposta. Altre volte la connessione potrebbe avere una larghezza di banda limitata (i.e., potrebbe essere inviata solo una determinata quantità di informazioni alla volta), necessitando la serializzazione della comunicazione e degradando nuovamente la latenza della risposta. In alcuni scenari, una connessione Internet potrebbe non essere disponibile. Inoltre, i computer edge vengono generalmente collocati su sistemi alimentati da batterie. L'energia di queste batterie può essere rapidamente scaricata da programmi caratterizzati da un elevato carico computazionale, dall'uso inefficiente delle gerarchie di memoria dei computer e dalle antenne utilizzate per le comunicazioni. I vincoli di latenza per le applicazioni e i vincoli energetici per i dispositivi rappresentano una limitazione significativa al dispiegamento di sistemi di apprendimento automatico su computer edge.

Quali caratteristiche rendono un determinato sistema di apprendimento

automatico preferibile ad altri? Quali sono le proprietà che un tale sistema dovrebbe soddisfare per essere dispiegato su un computer edge? Come cambia questa idoneità quando cambiano i vincoli? Stando alle nostre conoscenze, queste domande non sono mai state esplorate simultaneamente. Sosteniamo che fare scelte solide e progettare efficacemente sistemi di apprendimento automatico richiedano l'integrazione di concetti da diverse discipline. In particolare, dovremmo considerare tre aspetti:

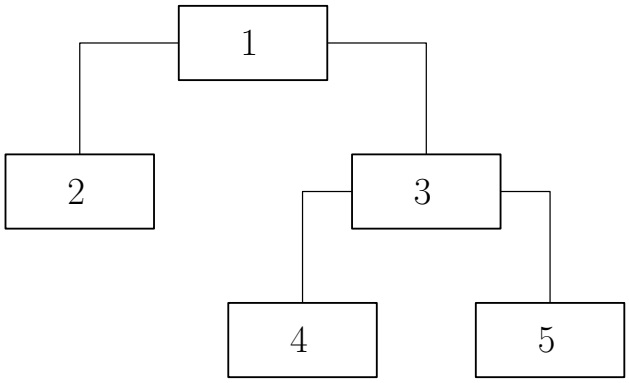
(i) **proprietà modellistiche**: le entità e le relazioni che il sistema può approssimare e la qualità di queste approssimazioni espressa in termini della struttura del sistema stesso;

(ii) **proprietà algoritmiche**: i costi computazionali seriale/parallelo (complessità di lavoro / complessità di passo) e il costo di memoria (complessità spaziale) degli algoritmi utilizzati dal sistema;

(iii) **proprietà architetturali**: le caratteristiche che un computer deve o dovrebbe soddisfare per eseguire i programmi del sistema.

Nel Capitolo 1, definiamo i concetti di insieme di misurazioni, intelligenza artificiale e sistema di apprendimento automatico. Riassumiamo anche alcuni concetti rilevanti della teoria degli algoritmi e delle architetture dei calcolatori su cui baseremo la nostra analisi. Nel Capitolo 2, definiamo il problema dell'apprendimento sulle *bag*, che può essere ricondotto a un normale problema di apprendimento in cui lo spazio di input è uno spazio di distribuzioni di probabilità. Durante il primo anno del corso di dottorato, abbiamo formalizzato e analizzato nel dettaglio un nuovo metodo per classificare bag di vettori, il metodo *fingerprint*, sviluppato durante una collaborazione con Tetra Pak. Questo sistema è stato progettato per risolvere un problema di strategia commerciale, in cui l'obiettivo era l'accuratezza statistica piuttosto che l'efficienza computazionale ed esemplifica bene i costi inerenti ad algoritmi statistici complessi. Nel Capitolo 3, introduciamo le *reti neurali artificiali* e discutiamo le loro proprietà computazionali. Questi sistemi hanno strutture modulari che supportano un efficiente algoritmo di apprendimento basato sul gradiente e le loro primitive computazionali si adattano meglio alle architetture *single instruction, multiple data* (istruzione singola, dati multipli) meglio di altri sistemi. Nel Capitolo 4, descriviamo le applicazioni avanzate delle reti neurali artificiali per illustrare le loro proprietà di modellazione. Introduciamo alcuni problemi di visione artificiale e le *reti neurali convoluzionali*, che sono *benchmark* popolari per gli algoritmi di apprendimento sviluppati per le reti neurali artificiali. Rispetto ad altri sistemi di apprendimento automatico, le strutture modulari delle reti neurali artificiali consentono di riutilizzare parte

di un programma pre-addestrato su un compito diverso, riducendo dunque i tempi di modellazione. Durante il secondo anno del corso di dottorato, abbiamo collaborato con Maserati per sviluppare un modello di analisi dati in grado di identificare le ragioni oggettive alla base delle valutazioni soggettive delle automobili; il modello che abbiamo ideato sfrutta esattamente questa proprietà di modularità. Nel Capitolo 5, descriviamo le *reti neurali quantizzate*, reti neurali artificiali che possono essere eseguite in modo estremamente efficiente su acceleratori hardware dedicati ma che pongono anche problemi matematici molto impegnativi. In particolare, eliminano l'ipotesi di differenziabilità delle reti neurali artificiali classiche, ostacolando l'applicazione di algoritmi di apprendimento basati sul gradiente. Analizziamo poi le proprietà di approssimazione di questi sistemi di apprendimento automatico, dimostrando che sono in teoria potenti come le reti neurali artificiali classiche. Usando tecniche di analisi funzionale, mostriamo che aggiungere rumore agli argomenti di funzioni non differenziabili recupera (almeno parzialmente) la differenziabilità. Descriviamo infine l'algoritmo di apprendimento denominato *additive noise annealing* (annichilimento del rumore additivo) che abbiamo sviluppato sulla base di questa analisi, riportando risultati allo stato dell'arte su benchmark di classificazione d'immagini.



Non è necessario leggere i capitoli nell'ordine in cui appaiono. Il Capitolo 1 è un prerequisito sia per il Capitolo 2 che per il Capitolo 3, che possono essere letti indipendentemente; il Capitolo 3 è un prerequisito sia per il Capitolo 4 che per il Capitolo 5, che possono anch'essi essere letti indipendentemente.

Preface

The history of philosophy and science is a story of predictions and mistakes. Making predictions is about getting insights into the patterns of our environment, and it is fundamental to intelligent behaviours like decision-making and planning. We can access the environment only via media, be it natural senses or artificial sensors, that provide us with data. It is indeed in this data that we hope to find patterns. There is no way to know whether we have sufficient data to derive the knowledge we would like to have. Sometimes we do not have sufficient data. Sometimes we unconsciously discard useful data. Other times, more maliciously, we consciously ignore those measurements that could conflict with the pattern we want to see in the data. Other times we focus on irrelevant patterns, doomed to fail as long as we will collect more data. Determinism has been a tendency towards preferring patterns described by functional relationships called *deterministic models*. At the beginning of the past century, statistics emerged as the scientific field in charge of formalising and organising the techniques used by physicists, chemists, biologists and other scientists to derive patterns from their data. Many data sets are not compatible with the deterministic hypothesis. Therefore, statisticians have accepted the existence of probabilistic patterns and developed more refined tools to quantify the uncertainty inherent to scientists' predictions, *stochastic models*.

The introduction of computing machines has altered our interaction with data to a great extent. First, processing units have automated and accelerated mathematical computations. Second, the development of storage technologies has allowed recording ever-increasing quantities of data, far exceeding that of the archives from previous historical periods. The combination of these factors has made the questions "Is it possible to automate the extraction of patterns from data?" and "Is it possible to replicate intelligent behaviours?" both relevant and accessible. These questions are the foundations of *machine learning* and *artificial intelligence* and, as we will see, they are intimately related.

The degree of miniaturisation of modern electronic devices allows deploy-

ing them in diverse environments. Exploiting the patterns of these environments is critical for economic and social applications. For example, real-time monitoring of industrial machinery allows more effective maintenance, ultimately reducing operating costs. Another example is that of smart biomedical sensors that can monitor patients with heart conditions: predicting heart attacks on time could save their lives. Machine learning studies models and algorithms that are able to capture the patterns in these phenomena. During the last decade of the past century, three technological trends have emerged. First, the evolution of Internet technology has allowed supporting ever-increasing rates and volumes of data exchange on computer networks. Second, the miniaturisation of transistors has allowed integrating computers into small systems at affordable prices, connecting these systems to the Internet (*Internet of things*, IoT). Third, computer architectures design has undergone a paradigm shift towards parallel systems, which achieve throughput by using multiple slower cores instead of a single fast processing unit.

The typical pipeline that applies statistical algorithms to data collected by *edge computers* (i.e., resource-constrained computers which are deployed directly into the environment) relies on an Internet connection: the resource-constrained computer (the client) acquires data and transmits it to a computer or a computer cluster (the server) which is capable of processing it; then, the output of the statistical algorithm is transmitted back to the client, which implements the desired functionality on this result. This dependency on an Internet connection has potential drawbacks on the performances of the system. For example, the connection might be unreliable (i.e., packages of information might get lost through the network), requiring retransmission and degrading latency performances. Other times the connection might have limited bandwidth (i.e., only a certain amount of information might be sent at a time), requiring serialization of communication and again degrading latency performances. In certain scenarios, an Internet connection might simply not be available. Moreover, edge computers are usually deployed on battery-powered systems. The energy of these batteries can be quickly drained by computationally intensive programs, by the inefficient use of the computers' memory hierarchies and by the antennas used for communications. The latency constraints for the applications and the energy constraints for the devices pose a significant limitation to the deployment of machine learning systems on edge computers.

What makes a given machine learning system more attractive than competitors? Which are the properties that such a system should satisfy to be deployed on an edge computer? How does this suitability change when constraints change? To our knowledge, these questions have never been explored jointly. We argue that solid choices and effective design of machine learning

systems require integrating concepts from diverse disciplines. In particular, we should consider three aspects:

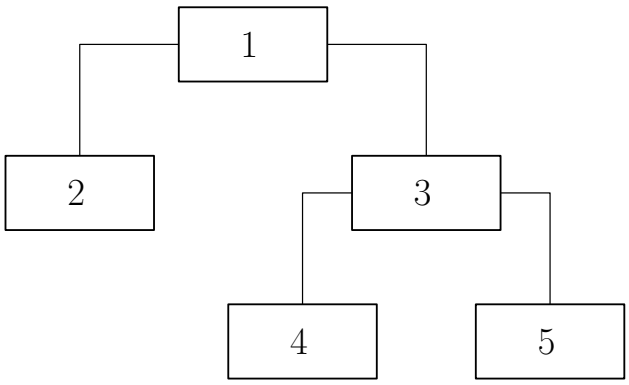
(i) **modelling properties**: the entities and relationships that the system can approximate, and the quality of these approximations in terms of the structure of the system;

(ii) **algorithmic properties**: the serial/parallel computational costs (work complexity / step complexity) and the memory cost (space complexity) of the algorithms used by the system;

(iii) **architectural properties**: the characteristics that a computer must or should satisfy to run the system's programs.

In Chapter 1, we define the concepts of data set, artificial intelligence and machine learning system. We also summarise the some relevant concepts of algorithms theory and computer architectures on which we will ground our analysis. In Chapter 2, we define the problem of learning on bags, which can be brought back to an ordinary learning problem where the input space is a space of probability distributions. During the first year of the PhD course, we better formalised and analysed a new method to classify bags of vectors, the *fingerprint* method, which was developed during a collaboration with Tetra Pak. This system was designed to solve a business intelligence problem, where the goal was statistical accuracy more than computational efficiency, and it well exemplifies the costs inherent to complex statistical algorithms. In Chapter 3, we introduce *artificial neural networks* and discuss their computational properties. They have modular structures which support efficient gradient-based learning algorithm, and their computational primitives fit efficient *single instruction, multiple data* computer architectures better than other systems. In Chapter 4, we describe advanced applications of artificial neural networks to illustrate their modelling properties. We introduce computer vision tasks and *convolutional neural networks*, which are popular benchmarks for learning algorithms targetting artificial neural networks. With respect to other machine learning systems, the modular structures of artificial neural networks allow reusing part of a pre-trained program on a different task, therefore reducing modelling time. During the second year of the PhD course, we collaborated with Maserati to develop a data analysis model that could find the objective reasons behind subjective evaluations of automobiles; the model we devised leverages exactly this modularity property. In Chapter 5, we describe *quantized neural networks*, artificial neural networks that can be run extremely efficiently on dedicated hardware accelerators but which also pose very challenging mathematical problems. In particular, they

drop the differentiability requirement of classical artificial neural networks, hampering the application of gradient-based learning algorithms. We analyse the approximation properties of these machine learning systems, proving that they are in principle as powerful as classical artificial neural networks. Using tools from functional analysis, we show that adding noise to the arguments of non-differentiable functions partially recovers differentiability. We describe the *additive noise annealing* learning algorithm that we developed on top of this analysis, reporting *state-of-the-art* benchmark results on image classification tasks.



It is not necessary to read the chapters in the order they appear. Chapter 1 is a prerequisite for both Chapter 2 and Chapter 3, which can be read independently; Chapter 3 is a prerequisite for both Chapter 4 and Chapter 5, which can also be read independently.

Contents

Chapter 1

Introduction

Knowledge is achieved through data and the processes used to manipulate this data. *Data sets* are collections of observations gathered using measurement instruments: essentially, they are numbers providing a simplified and objective description of certain physical phenomena. It is hoped that the patterns contained in these data sets can provide a sufficiently detailed picture of the functional relationships or the probability distributions governing the phenomena. This knowledge can then be used for scientific or economic purposes. From a scientific perspective, it is interesting to get insights into the processes used to extract information from data. From an economic perspective, the sheer amount of available data makes it desirable to detect patterns automatically. This desire for understanding and automating the extraction of statistical patterns is the motivation behind *artificial intelligence* and *machine learning*. Learning algorithms are computational models of the abstract entities and processes used in models of learning systems. Any implementation of an algorithm in a given formal language is a program. Programs are the only descriptions that can be translated into machine code and executed on digital computers.

In this chapter, we will summarise the fundamental concepts of measure theory and probability theory needed to introduce the concept of data set. We will then introduce the concepts of artificial intelligence and machine learning system. Finally, we will recall concepts from algorithms theory and computer architectures that we will use in our analysis.

1.1 Mathematical background

In the following, we will consider as axioms the definitions of set and subset. Mathematics is a language to study sets and operations defined on these sets.

In classical set theory [1], a **function** between two sets A, B is defined as a **relationship**

$$G \subset A \times B \tag{1.1}$$

that satisfies the *non-ambiguity property*

$$\forall\, (a_1, b_1), (a_2, b_2) \in G\,, b_1 \neq b_2 \implies a_1 \neq a_2\,. \tag{1.2}$$

Due to this property, one can define a non-ambiguous **rule**

$$\begin{aligned} f \,:\, & A \to B \\ & a \mapsto f(a) \coloneqq b \,|\, (a, b) \in G\,. \end{aligned}$$

In such cases the relationship (1.1) is also called the **graph** of f, and is denoted by $\mathcal{G}(f)$. To distinguish the two definitions of function, we will refer to the former by *relationship-form function* (or **graph**) and to the latter by *rule-form function* (or, simply, **function**). Let now $f_1, f_2 : A \to B$ be two functions. When there exists $a \in A$ such that $f_1(a) \neq f_2(a)$, we say that f_1 *and* f_2 *are different*, and we write $f_1 \neq f_2$. Let now $B = \{0, 1\}$ be a binary set. Note that the choice of the values 0 and 1 is arbitrary; the only requirement is for B to contain two clearly distinguished elements. Let

$$E \subseteq A \tag{1.3}$$

be a given subset of A. The function

$$\begin{aligned} \chi_E \,:\, & A \to B \\ & a \mapsto \chi_E(a) \coloneqq \begin{cases} 0\,, & \text{if } a \notin E \\ 1\,, & \text{if } a \in E\,. \end{cases} \end{aligned}$$

is called the **characteristic function** of E. We say that an element $a \in A$ *satisfies property* χ_E whenever $\chi_E(a) = 1$. For every subset E there exists a unique characteristic function χ_E associated with it. Conversely, we observe that for each function $\chi_E : A \to B$ there exists a unique subset $E \subseteq A$ such that χ_E is its associated characteristic function. In fact, given a characteristic function χ_E, there exists a unique subset

$$E \coloneqq \{a \in A \,|\, \chi_E(a) = 1\}\,. \tag{1.4}$$

The collection of all possible subsets of A is called the **power set** of A, and is denoted by $\mathcal{P}(A)$. The collection of all possible characteristic functions on A is also called the **power set** of A, but is denoted by 2^A.

Let A be a set of non-logical objects [2]. We denote by the symbol $\mathcal{A}$ a **carrier set** which can be either A or some set of logical objects built upon it, such as 2^A. Let $n \geq 1$ be a positive integer. An n**-ary operation** on $\mathcal{A}$ is a function

$$\begin{aligned} \star : \quad & (\mathcal{A})^n \to \mathcal{A} \\ & (a_1, a_2, \ldots, a_n) \mapsto \star((a_1, a_2, \ldots, a_n)) \,. \end{aligned} \tag{1.5}$$

For simplicity, the notation $\star((a_1, a_2, \ldots, a_n))$ is usually shortened to $\star(a_1, a_2, \ldots, a_n)$. Let

$$\star_1, \star_2, \ldots, \star_K$$

denote **operations** on $\mathcal{A}$. The tuple

$$T_{\mathcal{A}} \coloneqq (\mathcal{A}, \star_1, \star_2, \ldots, \star_K) \tag{1.6}$$

is called an **algebraic structure** on $\mathcal{A}$ [3]. Consider a characteristic function $\chi_E \in 2^A$; the unary operation

$$\begin{aligned} - \, : \, & 2^A \to 2^A \\ & \chi_E \mapsto \overline{\chi}_E \,, \end{aligned} \tag{1.7}$$

where

$$\overline{\chi}_E(a) \coloneqq \begin{cases} 0, & \text{if } \chi_E(a) = 1 \\ 1, & \text{if } \chi_E(a) = 0 \,, \end{cases} \quad \forall a \in A \,,$$

is called the **negation**. Consider characteristic functions $\chi_{E_1}, \chi_{E_2} \in 2^A$; the binary operations

$$\begin{aligned} * : \quad & 2^A \times 2^A \to 2^A \\ & (\chi_{E_1}, \chi_{E_2}) \mapsto \chi_{E_1} * \chi_{E_2} \,, \end{aligned} \tag{1.8}$$

$$\begin{aligned} \star : \quad & 2^A \times 2^A \to 2^A \\ & (\chi_{E_1}, \chi_{E_2}) \mapsto \chi_{E_1} \star \chi_{E_2} \,, \end{aligned} \tag{1.9}$$

where

$$\begin{aligned} (\chi_{E_1} * \chi_{E_2})(a) &\coloneqq \chi_{E_1}(a) * \chi_{E_2}(a) \\ (\chi_{E_1} \star \chi_{E_2})(a) &\coloneqq \overline{\overline{\chi_{E_1}} * \overline{\chi_{E_2}}} \end{aligned}$$

are called the **meet** and **join** respectively. These operations can be mapped to the classical counterparts of set theory, **complement** (^{c}), **intersection**

($\cap$) and **union** ($\cup$). By defining these two sets of operations, we have also defined two structures,

$$T_{2^A} \coloneqq (\quad 2^A, {}^{-}, *, \star)\,, \tag{1.10}$$
$$T_{\mathcal{P}(A)} \coloneqq (\mathcal{P}(A),\ {}^{c}, \cap, \cup)\,, \tag{1.11}$$

called the **power set algebras** over A. The advantage of (1.10) is that it makes the link with the physical implementation more apparent (e.g., by thinking to two-states relays). When the goal is building machines that can automatically elaborate data and improve their decisions, computability is a fundamental requirement. With this preliminary discussion, we wanted to highlight the fact that two mathematical structures can be equivalent for modelling purposes but not for computability purposes, one of them being *nearer to physical reality* that the other. The process of mathematical modelling consists of a sequence of transformations from more realistic models to more abstract ones: at each step of this sequence, some details of the richer model are dropped, and a simpler structure replaces the previous one. The reason for the success of a model lies in the irrelevance of the details that have been dropped during the modelling process; equivalently, the reason for the failure of a model lies in the importance of these details. We should always be explicit about our design choices to better understand why our models fail.

Let $\Sigma \subseteq \mathcal{P}(A)$ be a given collection of subsets of A. Let us define on Σ the unary complement operation ^{c} and the ∞-ary operation

$$\begin{aligned} \cup_{n=1}^{\infty} &: (\mathcal{P}(A))^{\infty} \to \mathcal{P}(A) \\ &\{E_n\}_{n\in\mathbb{N}} \mapsto \cup_{n=1}^{\infty} E_n \coloneqq \{a \in A \,|\, \exists\, \tilde{n}\,, a \in E_{\tilde{n}}\}\,, \end{aligned} \tag{1.12}$$

called **countable union**. The tuple

$$T_{\Sigma} \coloneqq (\Sigma, {}^{c}\,, \cup_{n=1}^{\infty}) \tag{1.13}$$

is a structure called a **σ-algebra** when the following properties are satisfied:

(i) $A \in \Sigma$;

(ii) $E \in \Sigma \implies E^c \in \Sigma$;

(iii) $\forall\, \{E_n\}_{n\in\mathbb{N}} \,|\, E_n \in \Sigma\,, \forall\, n \in \mathbb{N} \implies \cup_{n=1}^{\infty} E_n \in \Sigma$.

When any of these properties is not satisfied by Σ, one can replace Σ in (1.13) with the *σ-algebra generated by* Σ, $\mathcal{A} = \sigma(\Sigma)$ [4]. The pair

$$(A, T_{\mathcal{A}}) \tag{1.14}$$

is called a **measurable space**. Let $T_{\mathcal{B}} = (\mathcal{B}, ^c, \cup_{n=1}^{\infty})$ denote a second σ-algebra and $(B, T_{\mathcal{B}})$ the corresponding measurable space. A function $f : A \to B$ satisfying the *measurability property*

$$\forall\, B \in \mathcal{B}\,, f^{-1}(B) \in \mathcal{A} \tag{1.15}$$

is called a **measurable function**. Let $\{E_n\}_{n\in\mathbb{N}}$ be an infinite collection of elements in $\mathcal{A}$. $\{E_n\}_{n\in\mathbb{N}}$ is said to be **mutually disjoint** if $E_{n_1} \cap E_{n_2} = \emptyset\,, \forall\, n_1 \neq n_2$. A function

$$\begin{aligned} \mu_{\mathcal{A}} \,:\, & \mathcal{A} \to [0, +\infty] \\ & E \mapsto \mu_{\mathcal{A}}(E) \end{aligned} \tag{1.16}$$

that satisfies the properties

(i) $\exists\, E \in \mathcal{A} \,|\, \mu_{\mathcal{A}}(E) < +\infty$;

(ii) $\mu_{\mathcal{A}}(\cup_{n=1}^{\infty} E_n) = \sum_{n=1}^{\infty} \mu_{\mathcal{A}}(E_n)$ for every collection $\{E_n\}_{n\in\mathbb{N}}$ of mutually disjoint sets;

is called a **measure** over $\mathcal{A}$. The graph

$$G(\mu_{\mathcal{A}}) \coloneqq \{(E, \mu_{\mathcal{A}}(E)) \,|\, E \in \mathcal{A}\} \tag{1.17}$$

is called the measure **distribution** over $\mathcal{A}$. The tuple

$$(A, T_{\mathcal{A}}, \mu_{\mathcal{A}}) \tag{1.18}$$

is called a **measure space**. Let $(A, T_{\mathcal{A}}, \mu_{\mathcal{A}}), (B, T_{\mathcal{B}})$ and $f : A \to B$ be a given measure space, a measurable space and a measurable function, respectively. The **pushforward** of $\mu_{\mathcal{A}}$ through f is the measure

$$\begin{aligned} f_*\mu_{\mathcal{A}} \,:\, & \mathcal{B} \to [0, \infty] \\ & B \mapsto \mu_{\mathcal{A}}(f^{-1}(B))\,. \end{aligned} \tag{1.19}$$

Measure theory is a fundamental tool of modern probability theory [5]. If property (i) is replaced by the stronger *normalisation property*

(i) $\mu_{\mathcal{A}}(A) = 1$,

the function (1.16) is called a **probability measure**, whereas its graph (1.17) is called a **probability distribution**. The measure space (1.18) is called a **probability space**, and measurable functions are called **random variables** (RV).

Let $B = \{b_1, b_2, \dots, b_n, \dots\}$ denote a discrete set (i.e., finite or countable). Let $\mathcal{B} = \sigma(\{\{b\} \,|\, b \in B\})$ denote the carrier set of the σ-algebra generated by the singletons. It follows that $\mathcal{B} = \mathcal{P}(B)$. Let

$$\begin{aligned} \lambda \,:\, \mathcal{B} &\to [0, +\infty] \\ E &\mapsto \#(E) \end{aligned} \tag{1.20}$$

denote the (σ-finite) measure called the **counting measure**. Any function

$$\begin{aligned} p \,:\, B &\to [0, 1] \\ b &\mapsto p(b) \end{aligned} \tag{1.21}$$

that satisfies the *normalisation property*

$$\sum_{n=1}^{\infty} p(b_n) = 1$$

is called a **probability mass function** (PMF) over B. Using the integral operation defined with respect to the counting measure (1.20), a PMF induces a probability measure

$$\begin{aligned} \mu_{\mathcal{B}} \,:\, \mathcal{B} &\to [0, 1] \\ E &\mapsto \int_E p(b)\lambda(db) = \sum_{b \in E} p(b)\,. \end{aligned} \tag{1.22}$$

Let $C = \mathbb{R}^n$ for some $n \in \mathbb{N}$; therefore, C has the cardinality of the continuum. Let $T_{\mathcal{C}}$ be the *Borel σ-algebra* over C (i.e., the σ-algebra generated by the set of open subsets of $\mathbb{R}$) and let λ denote the usual *Lebesgue measure* over $\mathcal{C}$. The Radon-Nykodym theorem implies that every measure $\mu_{\mathcal{C}}$ over $\mathcal{C}$ which is absolutely continuous with respect to λ (i.e., $\lambda(E) = 0 \implies \mu_{\mathcal{C}}(E) = 0$) can be represented in integral form:

$$\mu_{\mathcal{C}}(E) = \int_E p(c)\lambda(dc)\,,\ \forall E \in \mathcal{C}\,. \tag{1.23}$$

The function $p : C \to \mathbb{R}_0^+$ is called the **probability density function** (PDF) of the measure $\mu_{\mathcal{C}}$ with respect to λ; it is a special case of the Radon-Nykodym derivative, and as such is also denoted by the symbol $d\mu_{\mathcal{C}}/d\lambda$.

Machine learning models are often designed (and analysed) as functions

$$\Phi \,:\, X \to Y$$

between continuous sets X, Y but, as we will see in Section 1.2, data sets are constrained to take values in finite subsets of these sets. To make the link

between discrete and continuous probability spaces more explicit, we need to introduce the concept of *generalised probability density function*. Let $(B, T_{\mathcal{B}})$ and $(C, T_{\mathcal{C}})$ be measurable spaces, where $C = \mathbb{R}^n$ is a continuous set, and $B \subset C$ is a discrete subset. Let $\mu_{\mathcal{B}}$ denote the probability measure over $\mathcal{B}$ associated with a probability mass function (1.21), and let λ denote the Lebesgue measure over $\mathcal{C}$. Let

$$\begin{aligned} I \,:\, B &\to C \\ b &\to b \end{aligned} \tag{1.24}$$

denote the **immersion** of B into C. The pushforward measure $\mu_{\mathcal{C}} \coloneqq I_* \mu_{\mathcal{B}}$ is not absolutely continuous with respect to λ; therefore, a PDF as the one defined in (1.23) does not exist. Let $\tilde{b} \in B$ denote a given point. Suppose $\mu_{\mathcal{B}}$ has associated PMF of the form

$$p(b) = \begin{cases} 0, & \text{if } b \neq \tilde{b} \\ 1, & \text{if } b = \tilde{b}\,; \end{cases}$$

i.e., it is concentrated on the point $\tilde{b} \in B$. The pushforward measure $\mu_{\mathcal{C}}$ can be defined as follows:

$$\begin{aligned} \delta_{\tilde{b}} \,:\, \mathcal{C} &\to [0, +\infty] \\ E &\mapsto \begin{cases} 0, & \text{if } \tilde{b} \notin E \\ 1, & \text{if } \tilde{b} \in E\,. \end{cases} \end{aligned} \tag{1.25}$$

This measure is called the **simple Dirac's measure** on $\mathcal{C}$ associated with $\tilde{b} \in C$. Borrowing the formalism from Dirac himself [6], we can define this measure using integral formalism:

$$\begin{aligned} \mu_{\mathcal{C}}(E) &= \int_E \delta_{\tilde{b}}(c) \lambda(dc) \\ &= \int \delta_{\tilde{b}}(c) \chi_E(c) \lambda(dc) \\ &= \chi_E(\tilde{b})\,. \end{aligned} \tag{1.26}$$

From a formal perspective, the notation $\delta_{\tilde{b}}(c)$ is meaningless since $\delta_{\tilde{b}}$ is not a function; but the formal analogies with the integral expressions (1.22) and (1.23) make its meaning clear. The representation of $\mu_{\mathcal{C}}$ given by (1.26) has the advantage of generalising the concept of PDF also to pushforward measures $\mu_{\mathcal{C}} = I_* \mu_{\mathcal{B}}$. Given a PMF (1.21) defined on B and the immersion

(1.24), one can define the associated **Dirac measure** on $\mathcal{C}$

$$\begin{aligned} \sum_{b\in B} p(b)\delta_b \,:\, \mathcal{C} &\to [0,1] \\ E &\mapsto \sum_{b\in B} p(b)\delta_b(E)\,, \end{aligned} \tag{1.27}$$

where the δ_b are simple Dirac's measures (1.25). A specific but relevant example of Dirac's measures are *empirical measures*. Let $N \in \mathbb{N}$ denote a positive integer. When a probability mass function (1.21) satisfies the property

$$\forall\, b \in B\,, \exists\, i \in \{0, 1, \ldots, N\} \,|\, p(b) = i/N\,, \tag{1.28}$$

the corresponding Dirac's measure (1.27) is called an **empirical measure**.

When joint information is available about two sets X and Y, a common necessity is to know how the information about Y changes depending on the available information about X. This is the *conditioning* problem. Let $(\Omega, T_{\mathcal{O}}, \mu_{\mathcal{O}})$ denote a probability space. Let $(X, T_{\mathcal{X}})$ and $(Y, T_{\mathcal{Y}})$ denote measurable spaces, where $\mathcal{X}, \mathcal{Y}$ are the carrier sets of the respective σ-algebras. Let $(X \times Y, T_{\mathcal{XY}})$ denote their product space; here, $\mathcal{XY} \coloneqq \sigma(\mathcal{X} \times \mathcal{Y})$. Let $f : \Omega \to X \times Y$ denote a measurable function. The pushforward $\mu \coloneqq f_*\mu_{\mathcal{O}}$ is called the **joint probability measure** over $\mathcal{X} \times \mathcal{Y}$. Let

$$\begin{aligned} \pi_X \,:\, X \times Y &\to X \\ (x, y) &\mapsto x \end{aligned} \tag{1.29}$$

denote the **projection** of $X \times Y$ over X. This projection is a measurable function since $\pi_X^{-1}(E_X) = E_X \times Y \in \mathcal{XY}$ for all $E_X \in \mathcal{X}$. Therefore, we can define the pushforward $\mu_{\mathcal{X}} \coloneqq \pi_{X*}\mu$, called the **marginal probability measure** of μ over $\mathcal{X}$. Given $x \in X$, we define

$$\pi_X^{-1}(x) \coloneqq \{x\} \times Y \tag{1.30}$$

as the **fibre** of $X \times Y$ associated with x. We define

$$\{x\} \times \mathcal{Y} \coloneqq \{\{x\} \times E_Y \,|\, E_Y \in \mathcal{Y}\} \tag{1.31}$$

to be the **fibre σ-algebra** associated with x; this is a **sub-σ-algebra** of $\mathcal{XY}$. We observe that (1.31) is isomorphic to $T_{\mathcal{Y}}$ for each $x \in X$. For any $x \in X$, a probability measure

$$\begin{aligned} \mu_x \,:\, \{x\} \times \mathcal{Y} &\to [0,1] \\ \{x\} \times E_Y &\mapsto \mu_x(\{x\} \times E_Y)\,, \end{aligned} \tag{1.32}$$

on the fibre σ-algebra is called a **fibre probability measure**. Due to x being fixed, it is more common to interpret fibre probability measures as simple probability measures on $\mathcal{Y}$:

$$\mu_x : \mathcal{Y} \to [0,1] \tag{1.33}$$
$$E_Y \mapsto \mu_x(E_Y) \tag{1.34}$$

is called the **conditional probability measure** on $\mathcal{Y}$ given x. Let

$$\mu_{Y|X} : X \times \mathcal{Y} \to [0,1] \tag{1.35}$$
$$(x, E_Y) \mapsto \mu_x(E_Y)$$

denote a function such that, for any fixed $E_Y \in \mathcal{Y}$, it is measurable with respect to $\mathcal{X}$. When μ admits a representation in terms of (1.35),

$$\mu : \quad \mathcal{X}\mathcal{Y} \to [0,1] \tag{1.36}$$
$$E_X \times E_Y \mapsto \int_{E_X} \mu_{Y|X}(x, E_Y)\mu_{\mathcal{X}}(dx)\,,$$

we say that μ *satisfies the disintegration property*. We call

$$(\mu_{\mathcal{X}}, \{\mu_x\}_{x \in X}) \tag{1.37}$$

the **disintegration** of μ. This is guaranteed whenever μ satisfies the hypothesis of the disintegration theorem [7]. In ML research, this is very often an implicit assumption. Another common (and implicit) assumption is that conditional measures (1.33) also have either an associated PMF (when Y is discrete) or a PDF, either classical or generalised (when Y is continuous). These objects are called **conditional probability mass functions** (conditional PMF) or **conditional probability density functions** (conditional PDF) respectively and are denoted by either the symbol $p(y|x)$ or by the symbol $p_x(y)$.

1.2 Patterns live in data sets

The physical world is described by some unknowable states set Ω. Measuring instruments are media that provide a simplified and objective description of the physical world [8]. A measuring instrument is characterised by some conventional (finite) set of symbols

$$X = \{x_1, x_2, \ldots, x_S\} \tag{1.38}$$

called the **measuring scale** of the instrument. Whenever a measuring instrument interacts with the physical world's state $\omega \in \Omega$ a symbol $x \in X$ from the scale is returned. This process can be modelled defining a RV

$$\begin{aligned} x \,:\, \Omega &\to X \\ \omega &\to x(\omega)\,, \end{aligned} \tag{1.39}$$

called the **measuring function** associated with the instrument having scale X. The output of any evaluation $x(\omega)$ is called a **measurement** or **sample**. For example, a ruler is a straight stick on which S equally spaced signs marked by symbols in $X = \{x_1, x_2, \ldots, x_S\}$ are placed, left to right. When another straight object is placed onto a ruler so that its leftmost extreme coincides with the leftmost sign on the ruler, some of the signs on the underlying ruler are covered and cannot be seen anymore. The rightmost covered sign is marked by some symbol $x(\omega) \in X$, which is the measurement of the object. Another example of measuring instruments are analog-to-digital converters (ADCs). An ADC is composed by a probe immersed in the physical environment (whose properties include an electromagnetic field), a capacitor connected to the probe via a switched conductor wire, reference voltages v^+, v^-, an integrator circuit and a clock; the ADC also includes a scale $X = \{x_1, x_2, \ldots, x_{2^B}\}$, encoded in a table using entries represented by B bits each. Suppose that the capacitor has zero charge in it and that the switch on the conductor wire is open. At a given instant, the switch is closed: the electric current induced on the wire by the external electromagnetic field accumulates charge in the capacitor as long as the switch is closed. After a fixed number 2^B of clock oscillations, the switch on the wire is opened to prevent the current from continuing charging up the capacitor, and the reference voltage of inverse polarity acts removing charge from it. The switch is left open until all the charge has been removed from the capacitor. This equalisation process requires a number $s \in \{0, 1, \ldots, 2^B\}$ of clock oscillations to complete, and the ADC returns the corresponding measurement $x(\omega) \in X$.

Humans can be thought of as measuring instruments as well. A semantic is defined by a vocabulary of concepts represented by the symbols $X = \{x_1, x_2, \ldots, x_S\}$, which is in turn defined by some linguistic convention amongst humans. Suppose that a person is given a photograph and a marker pen, and is then asked to write on the back of the photograph the symbol $x_s \in X$ corresponding to the concept that she thinks best describes the content of the photograph. She looks at the photograph and, based on her past experiences and current information, takes her decision. This process is equivalent to applying a measuring instrument to collect a measurement $x(\omega) \in X$.

Suppose to have a measuring instrument with measuring scale (1.38) and measuring function (1.39). Let $N \geq 1$ be a positive integer. Suppose to take N measurements of the physical world using the given instrument; this process is called a **measuring process** or **sampling process** of length N. The output of this process is a finite sequence

$$\mathfrak{D} = \{x(\omega^{(1)}), x(\omega^{(2)}), \ldots, x(\omega^{(N)})\} \tag{1.40}$$

called a **data set**. $\mathfrak{D}$ is a particular kind of set called a **multi-set** [9]: different measurements $x(\omega^{(i_1)}), x(\omega^{(i_2)}) \in \mathfrak{D}$, $i_1 \neq i_2$ could be instances of the same symbol $x \in X$. With abuse of notation we will write

$$\mathfrak{D} = \{x^{(1)}, x^{(2)}, \ldots, x^{(N)}\} \tag{1.41}$$

to denote (1.40). Since measuring scales (1.38) are finite sets, it is natural to associate with a data set (1.41) the PMF

$$\begin{aligned} p \,:\, & X \to [0,1] \\ & x \to \frac{\#\{x(\omega) \in \mathfrak{D} \,|\, x(\omega) = x\}}{N}\,; \end{aligned} \tag{1.42}$$

here, # is the cardinality function. As we discussed earlier in this section, it is possible to interpret X as a subset of some continuous set $\bar{X}$, and define a Dirac's measure (1.27) associated with $\mathfrak{D}$:

$$\begin{aligned} \mu_{\bar{\mathcal{X}}} \,:\, & \bar{\mathcal{X}} \to [0,1] \\ & E \mapsto \sum_{x \in X} p(x)\delta_x(E)\,. \end{aligned} \tag{1.43}$$

Observe that the PMF (1.42) is such that $\mu_{\bar{\mathcal{X}}}$ is an empirical measure.

Labelled data sets are particular instances of data sets whose points are pairs of measurements. Let $X = \{x_1, x_2, \ldots, x_{S_X}\}$ and $Y = \{y_1, y_2, \ldots, y_{S_Y}\}$ denote the measuring scales (1.38) of two different measuring instruments. Let x and y denote the associated measuring functions (1.39). The **labelled data set** resulting from a sampling process of length N is the sequence of joint measurements

$$\begin{aligned} \mathfrak{D} &= \{(x(\omega^{(1)}), y(\omega^{(1)})), (x(\omega^{(2)}), y(\omega^{(2)})), \ldots, (x(\omega^{(N)}), y(\omega^{(N)}))\} \\ &= \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(N)}, y^{(N)})\}\,. \end{aligned} \tag{1.44}$$

The first components $x^{(i)}$ of the measurements are called the **inputs** (or, in statistical terminology, the *explanatory variables*), whereas y_i is called the **label** (or, in statistical terminology, the *response variables*). In artificial

intelligence applications, the samples from labelled data sets are interpreted as descriptions of actions $y^{(i)}$ taken in response to states $x^{(i)}$. Therefore, the concept of conditional distribution is important in these applications. To this end, the empirical measure associated with the labelled data set (1.44) can be disintegrated as follows:

$$\mu_{\mathcal{X}} = \frac{1}{N}\sum_{i=1}^{N} \delta_{x^{(i)}}\,, \tag{1.45}$$

$$\mu_x = \begin{cases} \frac{1}{K(x)}\sum_{i=1}^{N} \delta^x_{x^{(i)}}\delta_{y^{(i)}}, & \text{if } K(x) \geq 1 \\ ?, & \text{if } K(x) = 0\,, \end{cases} \tag{1.46}$$

where

$$\begin{aligned} \delta_{x^{(i)}} &: \mathcal{X} \to [0,1] \\ \delta_{y^{(i)}} &: \mathcal{Y} \to [0,1] \end{aligned}$$

are simple Dirac's measures (1.25), $K(x) = \sum_{i=1}^{N} \delta^x_{x^{(i)}}$ and

$$\delta^x_{x^{(i)}} = \begin{cases} 0, & \text{if } x^{(i)} \neq x \\ 1, & \text{if } x^{(i)} = x\,, \end{cases}$$

denotes Kronecker's delta. The fact that μ_x remains undefined whenever $K(x) = 0$ is a consequence of the fact that it is defined $\mu_{\mathcal{X}}$-almost everywhere. A labelled data set (1.44) is said to be a **function-form data set** if it satisfies the non-ambiguity property

$$\forall\, (x^{(i_1)}, y^{(i_1)}), (x^{(i_2)}, y^{(i_2)}) \in \mathfrak{D}\,, y^{(i_1)} \neq y^{(i_2)} \implies x^{(i_1)} \neq x^{(i_2)}\,;$$

for example, this is the case of data sets whose pairs have inputs $x^{(i)}$ which are all different from one another. In such cases, the fiber measures μ_x take the form

$$\mu_x = \delta_{f(x)}\,, \tag{1.47}$$

where $f(x) \in Y$ is the unique element such that $(x, f(x)) \in \mathfrak{D}$. This model takes its name from the fact that it is compatible with the (usually implicit) assumption that there exists an underlying functional relationship $f : X \to Y$ in the data.

As a final note about data sets, we briefly discuss the problem of **multiple labellers**. To improve performance of ML systems it is important to collect as much data as possible. The labels $y^{(i)}$ are usually assigned to inputs $x^{(i)}$ by human experts. This process is expensive and time-consuming, and is

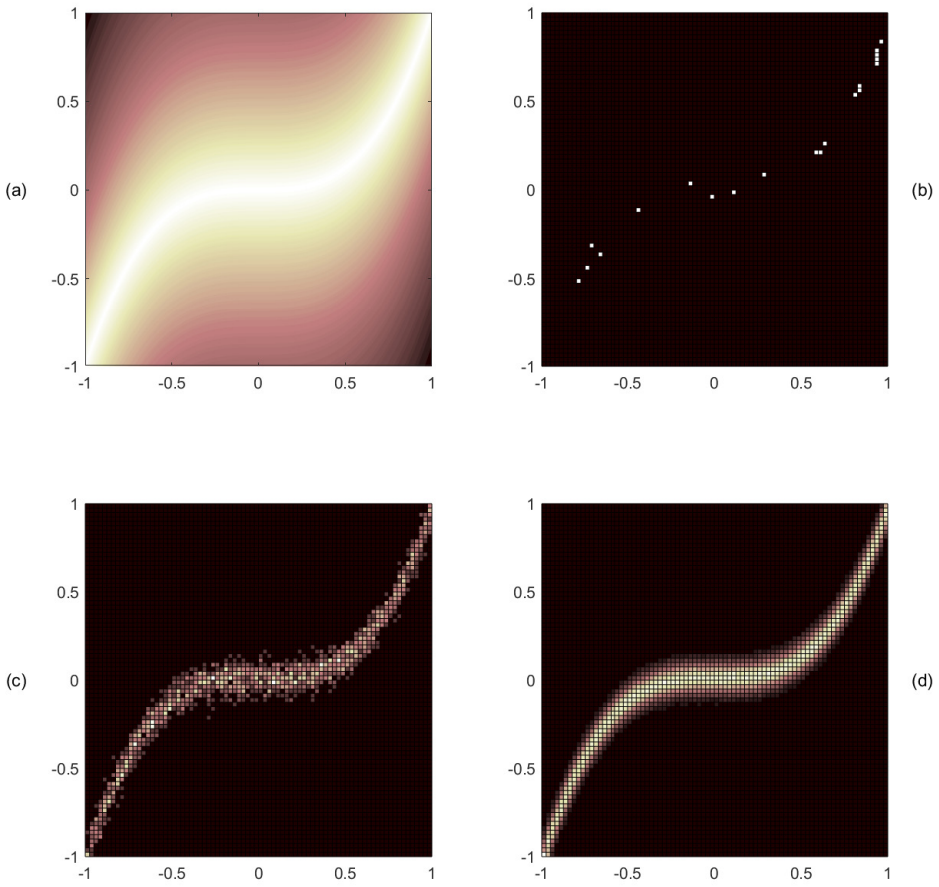


Figure 1.1: An example probability measure over the space $[-1, 1] \times \mathbb{R}$ (a) which admits disintegration

$$p(x) = \frac{1}{2}\,,$$
$$p_x(y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-x^3)^2}{2\sigma^2}}\,.$$

Measuring instruments with scales $X = Y = \{-1.000, -0.975, ..., 0.975, 1.000\}$ are used to collect samples from this process: $N = 20$ (b), $N = 2000$ (c) and $N = 200000$ (d). When just a few measurements are taken, it is likely to have a function-form data set.

performed by specialised *data annotation companies*. An interesting solution is asking non-experts to assign labels. The drawback of this solution is usually the poorer quality of the statistics on certain parts of the domain $X \times Y$. For example, suppose we want to classify plants species Y from photographs X. A person might be an expert in tropical plants, whereas a second one might be an expert in distinguishing evergreen plants. The first annotator is likely to generate reliable labels for pictures $X^{(1)} \subset X$ depicting tropical plants and poor labels for pictures $X^{(2)} \subset X$ of evergreen plants. The situation is likely to be reversed for the second annotator. To model this, suppose we have $u \in \{1, 2, \dots, U\}$ annotators. Each one is presented with a subset $X^{(u)} \subset X$ of pictures. Each of them will generate a data set

$$\mathfrak{D}^{(u)} = \{(x^{(u,1)}, y^{(u,1)}), (x^{(u,2)}, y^{(u,2)}), \dots, (x^{(u,N_u)}, y^{(u,N_u)})\},$$

with associated measure of the form (1.43). It might happen that aggregating this information in the data set

$$\begin{aligned}\mathfrak{D} &= \bigcup_{u=1}^{U} \mathfrak{D}^{(u)} \\ &= \{(x^{(1,1)}, y^{(1,1)}), \dots, (x^{(U,N_U))}, y^{(U,N_U)})\},\end{aligned}$$

and removing information about the labeller $u \in \{1, 2, \dots, U\}$ deteriorates the quality of the statistics on regions $X^* \subset X$ about which none of the annotators has sufficient expertise. Nevertheless, the statistical information about $X \times Y$ is expected to increase with the number U of annotators, since the biases of one annotator can be balanced by the expertise of others.

1.3 Artificial intelligence and machine learning

Alan Turing provided a quantitative definition of intelligence [10], replacing the question "Can machines think?" (involving the subjective definition of *thought*) with the less ambiguous "Can a machine fool a human into thinking it is another human when performing a given task?". This last question is called the **Turing test**. Informally speaking, Turing defined a system to be intelligent if a human examiner, interpreted as a measurement instrument to test whether the system's decision-making is human-like, cannot distinguish between the decision-making process performed by the system and the decision-making performed by a human.

Despite the rich philosophical debate around artificial intelligence [11], we were not able to find a formal definition. Hereafter we propose a formulation

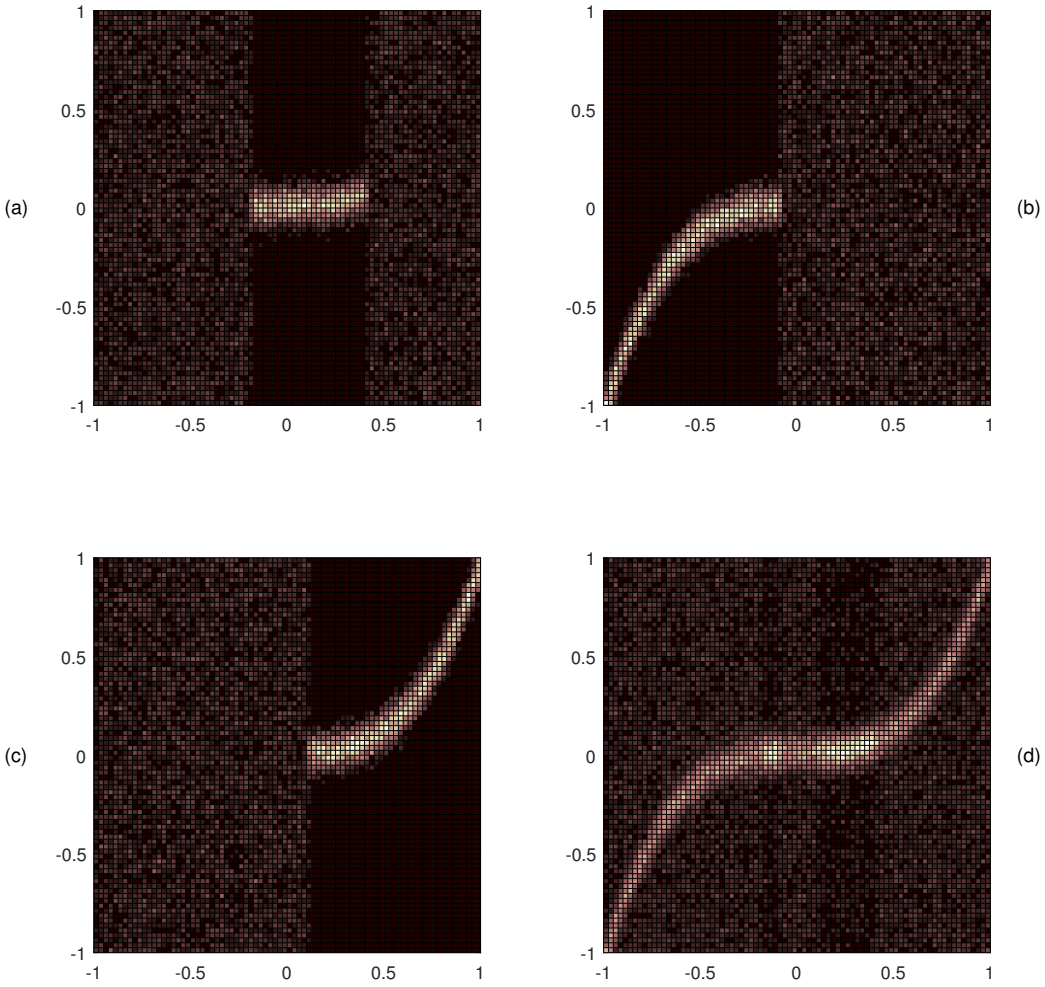


Figure 1.2: Example of data sets collected applying different measuring instruments to the process of Figure 1.2a: (a) a tool returning random measurements when the variable x is outside a specific range, (b) a tool returning random measurements when the variable x is higher than a specific threshold, (c) a tool returning random measurements when the variable x is lower that a specific threshold. Aggregating all the measurements into a unique data set (d) gives a clearer picture of the underlying phenomenon.

in terms of probability measures which we think will clarify its connection to machine learning. Suppose that both the system and the human take inputs from a set X and must choose an action from another set Y. Let $(X \times Y, T_{\mathcal{XY}}, \mu)$ be a product probability space associated with the system, where μ admits a disintegration $(\mu_{\mathcal{X}}, \{\mu_x\}_{x \in X})$. The marginal probability measure $\mu_{\mathcal{X}}$ depends only on the environment: the system has no control

over it. Therefore, the system is characterised by the class of conditional probability measures

$$\{\mu_x\}_{x\in X}\,, \tag{1.48}$$

which is called the **policy** of the system. Let us denote by M_Y the family of all probability measures over $\mathcal{Y}$, and define a *divergence* on this space [12]:

$$\begin{aligned} d\,&: M_Y \times M_Y \to [0,+\infty] \\ &(\mu_{\mathcal{Y}}^{(1)}, \mu_{\mathcal{Y}}^{(2)}) \mapsto d(\mu_{\mathcal{Y}}^{(1)}, \mu_{\mathcal{Y}}^{(2)})\,; \end{aligned}$$

a divergence is a function satisfying the following properties:

(i) $d(\mu_{\mathcal{Y}}^{(1)}, \mu_{\mathcal{Y}}^{(2)}) \geq 0\,,\ \forall\, \mu_{\mathcal{Y}}^{(1)}, \mu_{\mathcal{Y}}^{(2)} \in M_Y$;

(ii) $d(\mu_{\mathcal{Y}}^{(1)}, \mu_{\mathcal{Y}}^{(2)}) = 0$ if and only if $\mu_{\mathcal{Y}}^{(1)} = \mu_{\mathcal{Y}}^{(2)}$.

Note that every metric is a divergence. Let $u \in \{1, 2, \ldots, U\}$ index the set of human examiners. Let $M_Y(u,x) \subseteq M_Y$ denote the set of probability measures over $\mathcal{Y}$ conditioned on x that are evaluated as human-like by examiner u. The examiner's role is to evaluate the quantity

$$d(\mu_x, M_Y(u,x)) \coloneqq \inf_{\mu_{\mathcal{Y}} \in M_Y(u,x)} d(\mu_x, \mu_{\mathcal{Y}})\,.$$

The system is said to *pass the Turing test for examiner u on input x* if

$$d(\mu_x, M_Y(u,x)) = 0\,.$$

The system is said to *pass the Turing test for examiner u* whenever

$$d(\mu_x, M_Y(u,x)) = 0$$

almost everywhere with respect to $\mu_{\mathcal{X}}$. In such cases, the system is an **artificial intelligence** (AI).

AI systems can automate tasks that typically require human intervention, ranging from language translation to vehicle driving. Their impact on the economy and society can hardly be underestimated. The Turing test does not provide any prescription about the implementation of the system with policy (1.48), apart from it being a digital computer. For example, it does not require that the policy evolves in time. The chess program and computer *Deep Blue* were designed to implement a static policy based on a tree-search algorithm [13, 14]. Updating the policy required redesigning the software, or even the hardware. Such kind of products requires expensive and time-consuming development cycles.

Machine learning (ML) is the branch of AI concerned with computer systems that can update their policy in time using their past experience alone, without external intervention. The definition of a learning-capable system is operational [15]. Let τ denote a given task, that the system can solve by observing states in some set X and choosing actions from some set Y. Denote by

$$\begin{aligned} M_{Y|X} &\coloneqq \{\{\mu_x\}_{x\in X}\} \\ &\subseteq (M_Y)^X \end{aligned} \tag{1.49}$$

the space of all possible policies called the **policy space** over Y. We suppose that the performance of the system can be assessed evaluating its policy using some operator P. A requirement on P is that it defines a partial order $\leq$ on the policy space. This property allows comparing different policies $\{\mu_x^{(t_1)}\}_{x\in X}, \{\mu_x^{(t_2)}\}_{x\in X} \in M_{Y|X}$. The system can update its policy from $\{\mu_x^{(T)}\}_{x\in X}$ to $\{\mu_x^{(T+1)}\}_{x\in X}$ using past experience. When the modification of the policy is such that $\{\nu_x^{(T+1)}\}_{x\in X} \geq \{\nu_x^{(t)}\}_{x\in X}$, $\forall t \leq T$, we say that *the system has learnt from its experience*. For example, if the task is playing chess, the set X represents the set of possible configurations of the chessboard, the set Y represents the possible moves, and the operator P computes the Elo rating [16] of the system. A chess-playing system that can modify its policy using the data points acquired during its games and whose Elo rating improves after the updates is learning to play chess.

A **machine learning system** must have three components:

(i) an **experience** $\mathfrak{D}$,

(ii) a **program space** $M \subseteq M_{Y|X}$,

(iii) a **learning algorithm** α.

The experience is given as a data set (1.41) or a labelled data set (1.44). In its simplest form, the program space is a parametric family

$$M \coloneqq \left\{\{\mu_x = \delta_{\Phi(\theta,x)}\}_{x\in X}\right\}_{\theta\in\Theta} .$$

Note that there is a one-to-one correspondence between these program spaces and the families

$$M \coloneqq \{\Phi(\theta,\cdot) : X \to Y\}_{\theta\in\Theta} \tag{1.50}$$

of parametric functions. For this reason, we call this kind of program spaces **function-form** program spaces. *Ensemble systems* have richer program

spaces,

$$M \coloneqq \left\{ \{\mu_x = \frac{1}{H} \sum_{h=1}^{H} \delta_{\Phi^{(h)}(\theta^{(h)},x)}\}_{x \in X} \right\}_{\theta=(\theta^{(1)},\theta^{(2)},\dots,\theta^{(H)}) \in \Theta},$$

which have a one-to-one correspondence with collections

$$M \coloneqq \{\{\Phi^{(h)}(\theta^{(h)}, \cdot) : X \to Y\}_{h=1,2,\dots,H}\}_{\theta=(\theta^{(1)},\theta^{(2)},\dots,\theta^{(H)}) \in \Theta} \tag{1.51}$$

of H programs that might differ for the parameter $\theta^{(h)}$ or even for the functions $\Phi^{(h)}$. We call this kind of program spaces **ensemble-form** program spaces. The learning algorithm α is used to select the program from M. In the case of function-form program spaces, the learning algorithms are initialised selecting an initial program $\Phi^{(0)}(\theta^{(0)}, \cdot) \in M$. Then, the learning algorithm uses the experience $\mathfrak{D}$ (or a subset of its) to update the program to $\Phi^{(t)}(\theta^{(t)}, \cdot)$. There are two possibilities to update the program:

(i) *static code*, where $\Phi^{(t)} \equiv \Phi^{(0)}$: the program is updated changing only the parameters $\theta^{(t)}$;

(ii) *dynamic code*, where also the function $\Phi^{(t)}$ can change over time.

In its simplest form, the learning problem is formulated as **supervised learning**. We suppose that the data set $\mathfrak{D}$ satisfies the function-form property (1.47). The goal of supervised learning is to minimise the divergence

$$d(\delta_{\Phi(\theta,x)}, \delta_{f(x)}) \tag{1.52}$$

between the data set fibre measure and the system conditional policy for as many points $x \in X$ as possible. In some applications, like those of computer vision (which we will describe in Chapter 4), the data set measure is credited as human-like since the data points are obtained by asking humans to attach semantic labels to the inputs. Therefore, from the perspective of artificial intelligence, a machine learning system which learns such a data set measure could actually be considered an AI.

1.4 Computational aspects of learning systems

Abstract models of learning systems provide insights into the entities they can approximate and the quality of these approximations. Nevertheless, physical

computers have limited memory, limited number representation capabilities (e.g., 32-bits integers, 32-bits floating-point numbers) and limited compute throughput. These aspects are fundamental for practical applications. The goal of this section is to outline the steps of the analysis that should be performed when deciding to solve a given task using a machine learning system.

Let

$$f : X \to Y \,, \tag{1.53}$$

be some abstract function. The transition from the abstract function to an algorithmic model requires two steps: defining *data structures* for X and Y, and defining the *algorithm* that implements the desired function f [17, 18]. A **data structure** is a tuple

$$(\mathtt{X}, \mathtt{S}, \mathtt{O} = (\mathtt{o}_1, \mathtt{o}_2, \ldots, \mathtt{o}_{N_0}), \mathtt{T}) \,. \tag{1.54}$$

$\mathtt{X}$ is the set of **instances**, and

$$\mathtt{S} : \mathtt{X} \to \mathbb{N}$$

is the function assigning an integer *space cost* to each instance, modelling the memory space occupied by the instance in terms of abstract units of memory; in particular, we partition instances in *elementary instances*

$$\mathtt{X}^1 \coloneqq \{\mathtt{x} \in \mathtt{X} \,|\, \mathtt{S}(\mathtt{x}) = 1\} \,,$$

and *derived instances* $\mathtt{x} \in \mathtt{X} \setminus \mathtt{X}^1$. $\mathtt{O}$ is the set of **operations**

$$\mathtt{o}_i : (\mathtt{X}^1)^{n_i} \to C_i$$

defined on elementary instances; n_i is the arity of the i-th operation and C_i its codomain, which can be both $\mathtt{X}$ (the operation is then said to be *internal*) or another set. The function

$$\mathtt{T} : \mathtt{O} \to \mathbb{N}$$

assigns an integer *time cost* to each operation, modelling the execution time of the operation in term of abstract units of time.

Let $\mathtt{X}$ and $\mathtt{Y}$ be the instance sets of the data structures associated with the domain and the codomain of the abstract function (1.53). An **algorithm** $\mathtt{f}$ is a finite sequence of unambiguous operations that must be performed to transform instances $\mathtt{x} \in \mathtt{X}$ into an instance $\mathtt{y} \in \mathtt{Y}$. In particular, for any given instance $\mathtt{x} \in \mathtt{X}$ there is an *instance-specific* version of the algorithm

$$\mathtt{f}(\mathtt{x}) \coloneqq \{\mathtt{o}^{(1)}, \mathtt{o}^{(2)}, \ldots, \mathtt{o}^{(N_{\mathtt{f}(\mathtt{x})})}\} \,, \tag{1.55}$$

where the operations $\mathtt{o}^{(i)}$ operate on appropriate data structures. When the sequence of operations is independent of the instance

$$\mathtt{f}(\mathtt{x}) \equiv \mathtt{f}\,,\ \forall\, \mathtt{x} \in \mathtt{X}\,, \tag{1.56}$$

we say that the algorithm is a **data-independent algorithm**. This happens when there are no data-dependent **branching points** in the algorithm. Algorithms can be analysed according to two criteria: **space cost** and **time cost**. Let N denote a positive integer and define

$$\mathtt{X}(N) \coloneqq \{\mathtt{x} \in \mathtt{X} \,|\, \mathtt{S}(\mathtt{x}) = N\} \tag{1.57}$$

to be the space of all the instances with space cost N. We denote by $\mathtt{S}(\mathtt{f}(\mathtt{x}), \mathtt{o}^{(t)})$ the total space cost of the instances of the data structures used by the instance-specific algorithm $\mathtt{f}(\mathtt{x})$ at instruction $\mathtt{o}^{(t)}$. The **space cost** of the algorithm is defined as the function

$$S(N) \coloneqq \sup_{\mathtt{x} \in \mathtt{X}(N)} \left\{ \max_{t=1,2,\ldots,N_{\mathtt{f}(\mathtt{x})}} \{\mathtt{S}(\mathtt{f}(\mathtt{x}), \mathtt{o}^{(t)})\} \right\}. \tag{1.58}$$

The **time cost** of the algorithm is defined as the function

$$T(N) \coloneqq \sup_{\mathtt{x} \in \mathtt{X}(N)} \left\{ \sum_{t=1}^{N_{\mathtt{f}(\mathtt{x})}} \mathtt{T}(\mathtt{o}^{(t)}) \right\}. \tag{1.59}$$

Note that both the space cost and the time cost measure the costs of the algorithm in the worst-case scenarios. Let $g(N)$ be some non-negative function of N, and let $f(N)$ denote either the space cost or the time cost of the algorithm $\mathtt{f}$. The algorithm is said to be a $O(g(N))$ whenever

$$\exists\, C > 0,\ \bar{N} \in \mathbb{N} \,|\, f(N) \leq C g(N)\,, \forall\, N \geq \bar{N}\,. \tag{1.60}$$

The algorithm is said to be a $\Omega(g(N))$ whenever

$$\exists\, c > 0,\ \bar{N} \in \mathbb{N} \,|\, c g(N) \leq f(N)\,, \forall\, N \geq \bar{N}\,. \tag{1.61}$$

The algorithm is said to be a $\Theta(g(N))$ whenever

$$\exists\, c, C > 0,\ \bar{N} \in \mathbb{N} \,|\, c g(N) \leq f(N) \leq C g(N)\,, \forall\, N \geq \bar{N}\,. \tag{1.62}$$

These definitions correspond to asymptotic upper bounds, lower bounds and tight bounds respectively. When $S(N) = \Theta(g(N))$, the function $g(N)$ is said to be the **space complexity** of the algorithm. When $T(N) = \Theta(g(N))$, the

function $g(N)$ is said to be the **time complexity** of the algorithm. There can be multiple algorithms `f` that implement a given function (1.53). The difference between these algorithms lies in their space costs and/or in their time costs. For example, the time complexity of the *insertion sort* algorithm is $O(N^2)$, whereas that of the *merge sort* algorithm is $O(N \log(N))$: the algorithms are **functionally equivalent** but merge sort has a better time cost. From the perspective of machine learning systems, two models which have equivalent modelling properties could greatly differ for the complexities of their learning and inference algorithms. In modern data analysis scenarios, data sets often contain millions or billions of measurements: the difference between an $O(N^2)$ algorithm and an $O(N)$ algorithm (where N is the data set size) can determine the difference between an impossible problem and a feasible solution.

The space cost and time cost functions are arbitrary and can be redefined depending on the level of analysis required. The only technical requirement we enforce is that they must define a *reasonable model of computation* [19]. For example, consider the problem of performing arithmetic operations on integers represented in the binary base. We consider 1-bit integers as the elementary instances and N-bits integers as derived instances. If we assign a unitary space cost to individual bits, N-bits numbers have space cost N. If we assign a unitary time cost to the (binary) operation of addition involving two bits, the addition between two N-bits integers can be realised by an algorithm which time cost is proportional to N. Consider now the problem of sorting sequences of integers. The concept of sorting requires the idea of an order relationship on the set of integers: therefore, the data structure used to represent integers must include a (binary) comparison operation, usually realised as the sequence of subtraction and a comparison with zero. Then, at the sequence level, we must model the operation of exchanging the positions of two components. To analyse the problem of sorting without the details associated with the representations of integers, we can consider integers as elementary instances (i.e., sequences of length one) and sequences of length greater than one as derived instances. We assign to each element of the sequence a unitary space cost; consequently, the space cost of sequences of length N is N. We assign unitary time cost to the operations of comparing two integers (this operation is defined on the data structure of integers) and unitary time cost to the operation of exchanging two components of the sequence (this operation is defined on the data structure of sequences of integers). If for some reason we need to model explicitly the fact that integers have a binary representation, it is sufficient to assign higher space cost to elementary sequence instances and higher time cost to the comparison operation between their elements.

We have seen that algorithms theory provides better quantitative tools to assess the practicality of machine learning systems. Still, the space cost of instances and the time cost of elementary operations are abstractions that pose no limitation to the number of computations that can be carried out or to the memory space that can be occupied. These hypotheses are still too unrealistic when evaluating the deployment of machine learning programs on real-world computers. Therefore, to clarify our analysis, we need to introduce some concepts from computer architectures [20]. Before an algorithm can be executed on an electronic computer, two steps are required. First, the algorithm must be described using a specific formal language (for example, the C programming language [21]) in what is called a **program**. Second, this program must be translated into a valid **executable** for the target hardware, which includes both operands and machine code instructions. Each computer architecture is characterised by a specific set of instructions that can be executed, called the **instruction set architecture** (ISA). The translation from a program in a low-level programming language to an executable is performed by a toolchain of programs:

- a *compiler* transforms the program into assembly code;
- an *assembler* translates assembly code into machine code;
- a *linker* inserts into this machine code the pointers to the other pieces of machine code which is necessary for execution (e.g., library functions).

We will not delve into the details of formal languages, compilers and linkers into this thesis since they are out of the scope of our discussion. Computers

Figure 1.3: A von Neumann computer architecture: the main memory is connected to the processing unit via a bus. The processing unit is divided in register file (R), control unit (CU) and datapath (D).

are usually built according to the von Neumann architecture, consisting of the **main memory** and the **processing unit**, connected via a **bus**, as in Figure 1.3. The main memory holds both the **operands** and the **instructions** of a program. Computations can only happen in the processing unit,

and the required operands must be moved to/from a local memory in the processing unit (the *register file*) from/to the main memory using specific instructions. Data can be written to the main memory (for example, the characters digited on a keyboard) or read from the main memory (for example, the characters printed on a sheet of paper by a printer) also using **input/output** (I/O) peripherals. We remark that storage devices like hard disk drives (HDDs) or solid-state drives (SSDs), commonly referred to as *memory* in everyday terminology, are not considered memory from a computer architectures point of view, but are an I/O peripheral built specifically to store large amounts of data. A computer is governed by an operational frequency, commonly referred to as the **clock frequency** of the system; a single oscillation of the clock is called a **clock cycle**. A processing unit is usually divided in

- the already mentioned **register file**, a small local memory where operands are moved from/to the main memory,
- the **control unit**, circuitry dedicated to *fetching* instructions from memory and *decoding* them, and
- the **datapath**, circuitry dedicated to executing instructions on operands which are stored in the register file.

. In a never-ending cycle, the processing unit fetches (i.e., moves) an instruction from the main memory to the control unit, the control unit decodes the instruction (i.e., it configures the datapath to perform the instruction) and the datapath executes the configured instruction on the appropriate operands of the register file. Examples of the circuitry of the datapath are arithmetic logic units (ALUs) and floating-point units (FPUs) supporting integer and floating-point arithmetic respectively: this circuitry implements arithmetic algorithms on data structures which are sequences of (usually) 32 or 64 bits, which are physically implemented as **registers**. The required number of clock cycles to execute a given instruction is called the **latency** of the instruction. Remember that the ISA must include specific instructions to move data from/to the main memory to/from the register file in the processing unit. Since the processing unit and the memory are physically separate components, these **memory operations** have (relatively) high latencies which are usually not taken into account by abstract computational models. Moreover, these operations also have (relatively) high energy cost. A typical solution to reduce the latencies and energy costs of memory operations is creating **memory hierarchies**. Memory hierarchies are sets of memories connected by dedicated and specialised buses, arranged in a pyramidal fashion from the main memory to the processing unit, as in Figure 1.4.

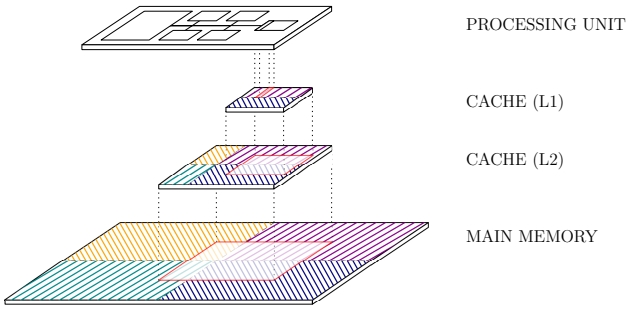


Figure 1.4: A simple memory hierarchy consisting of two intermediate memories. Each memory in the hierarchy is a copy of neighbouring memory locations of the memory underneath it.

Every memory communicates only with the one immediately below and immediately above it. Moving towards the processing unit, each component of the hierarchy has less capacity but also less latency. The levels of the hierarchy between the main memory and the processing unit are implemented as **caches**. When the processing unit reads data (operands and/or instructions) from memory, every cache creates a copy of a portion of neighbouring cells of the memory level below it, until the processing unit can read the data from the nearest cache. If the processing unit needs to read a second piece of data from a memory location which is near to the previous one (*spatial* or *temporal locality* of the operands), it is likely that this second piece of data will already be in the cache, reducing the latency of the read. When the processing unit needs to write the result of an operation back to memory, the process is applied in reverse order. To minimise the number and size of these memory transactions, it is desirable that operands and instructions are located in neighbouring regions on the physical memory. Therefore, programs with data-dependent branching instructions are likely to generate memory transactions patterns which are inefficient from the hardware perspective. The number of instructions executed per unit of time by a processing unit is called the **throughput**. In the second half of the past century, the throughput was increased by increasing the clock frequency of the computer: given a fixed ISA, more clock cycles per second translates into more instructions per second. However, increasing the frequency also increased the heat generated by the computer chips beyond the limit of the available cooling systems, and led to prohibitive energy consumptions. This phenomenon has been called the *power wall* and has motivated a paradigm shift in computer architectures design during the last two decades: parallelisation. According to Flynn's taxonomy, traditional von Neumann architectures are **single instruction, single data** (SISD) architectures: instructions must be executed sequen-

tially, and just on one or two operands at a time, depending on the arity of the instruction. Parallel computers extend the von Neumann architecture to include multiple control units and datapaths. Attaching multiple datapaths

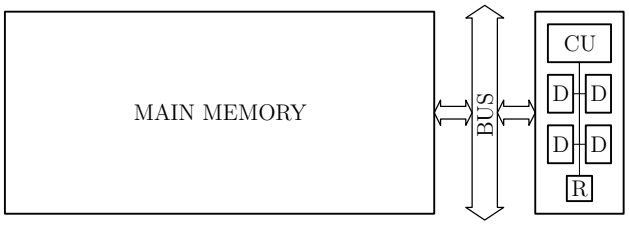


Figure 1.5: A SIMD computer architecture. Observe the multiple datapaths inside the processing unit.

to the same control unit as in Figure 1.5 allows executing the same instruction on multiple data; in Flynn's taxonomy, these architectures are called **single instruction, multiple data** (SIMD). The problem with SIMD architectures is that exactly the same instruction must be executed on all the different operands: this does not allow executing simultaneously multiple threads that have different code (*divergent threads*). Therefore, algorithms with data-dependent branching instructions are not optimal for these architectures. When also the control units are replicated, we have the so-called **multiple instructions, multiple data** (MIMD) architectures, capable of executing parallel programs with divergent threads; see Figure 1.8.

Energy-efficient computer architectures are computer architectures that consume less energy with respect to a set of given benchmark architectures. For example, datapaths implementing floating-point instructions require more energy than datapaths implementing only integer and fixed-point instructions. Therefore, it is desirable to avoid using complex fixed-point instructions when energy is a constraint of the application. As another example, memory operations that traverse the full memory hierarchy have higher latencies and higher energy costs than accessing data in caches. Therefore, reducing the number of accesses to the main memory is another desirable property. Given a cache memory of fixed size, programs whose operands can be represented using less than 32 or 64 bits could store more operands than classical programs into it. However, in this case, we need to design specialised instruction to operate with smaller operands, making the control units of the processing units more complex. Every piece of software (i.e., every program) can be implemented in hardware. At this point, we need to consider that, like every industrial manufacturing process, the production of dedicated hardware is expensive and requires amortising its cost on high production volumes. Therefore, before designing a computer architecture,

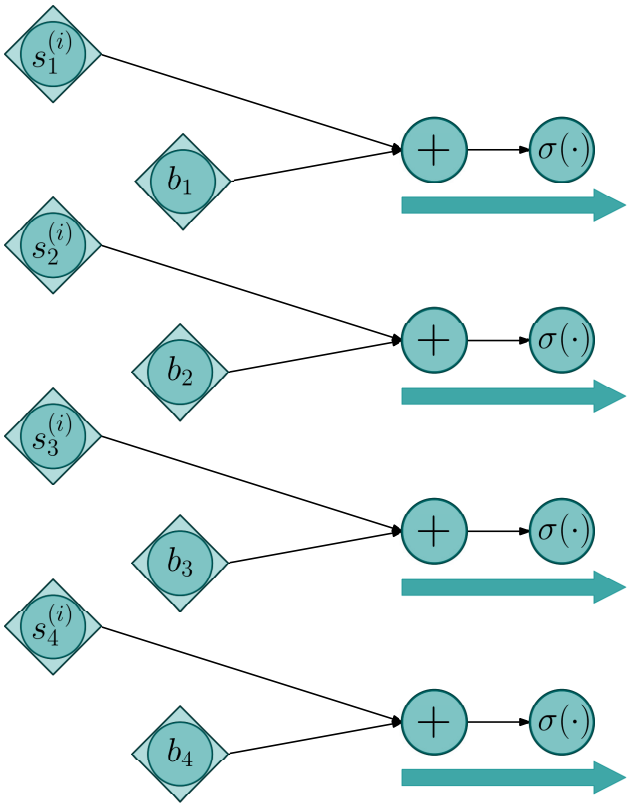


Figure 1.6: Example of a *Hadamard* (i.e., element-wise) operation. It takes two vectors, $\mathbf{s}$ and $\mathbf{b}$, and computes

$$\sigma(s_i + b_i)\,,\ i = 1, 2, 3, 4\,.$$

Each datapath (identified by a coloured arrow) executes the same sequence of instructions (*thread*, represented by a sequence of coloured circles). The operations do not depend on the input instances, and every datapath performs the same computation as the others. These operations are optimal for SIMD architectures since they can be performed on multiple operands at the same time.

it is more convenient to identify **computational primitives**, operations that are used by algorithms of widespread adoption. These computational primitives can then be implemented as dedicated instructions. Integrating these instructions directly into the ISA of a general-purpose processing unit would add unnecessary complex circuitry to its datapath since only a limited number of programs would use the instructions and energy would get wasted during normal execution regimes: it is more efficient to build a dedicated

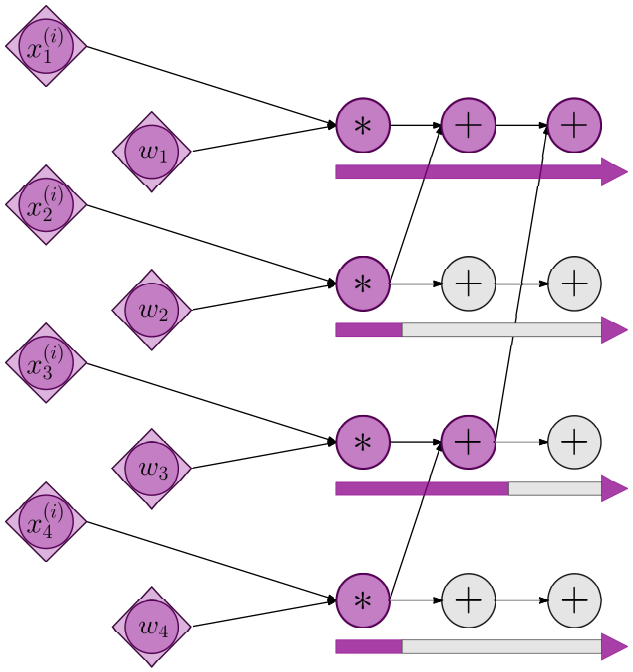


Figure 1.7: Example of a *reduce* operation. Reduce operations are extremely common in practice since they underlie tensor contraction operations. In this case, the operation performed is the dot product between two vectors:

$$\mathbf{x} \cdot \mathbf{w} = \sum_{i=1}^{4} x_i w_i \,.$$

Each datapath (identified by a coloured arrow) executes the same sequence of instructions (*thread*, represented by a sequence of coloured circles). The nature of the sum operation is such that it must be serialised and requires some datapaths to remain *idling* (i.e., these datapaths will not process operands during the operations represented by the grey circles) while some of them continue processing (coloured circles). Although this does not fully leverage the computational capabilities of SIMD processors, the instructions executed by each datapath do not depend on data; this independence of the operations from the data allows designing efficient access patterns to memory hierarchies.

piece of hardware that acts as a co-processor. The computers designed to implement specific computational primitives are called **application-specific integrated circuits** (ASICs). For example, *digital signal processors* (DSPs) are used to filter the inputs to many real-world signal processing systems.

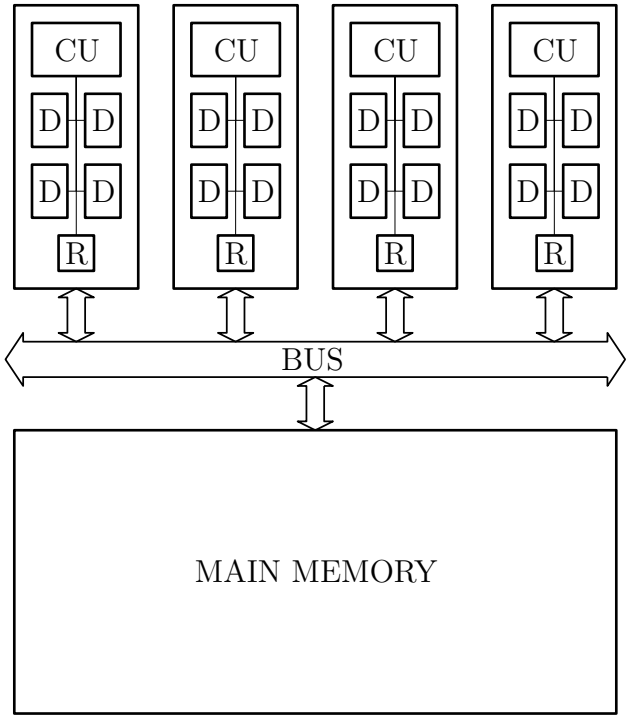


Figure 1.8: A MIMD architecture featuring multiple *cores* with the same architecture. In this example, each core has a SIMD architecture; therefore, it is advisable to schedule only sets of non-divergent threads on each processing unit. As an example, *general-purpose graphics processing units* (GPGPUs) fall into this category.

Hardware accelerators are ASICs that can complement the functionality of a computer's processing unit by performing a limited set of instructions very efficiently.

In this chapter, we have argued that measurements provide a quantitative representation of the physical world and that measuring instruments influence the mathematical structures that we use to design our models. Data analysts use this information to design models that can explain (or at least exploit) the statistics of the data set. Some parametric models can be automatically tuned: every machine learning system defines a space of inference algorithms (the *programs*) and a learning algorithm to select the best program out of this space. Depending on the size of the available data set and on the space cost of the elementary instance (e.g., a vector or an image),

these algorithms can be analysed to determine whether the learning system is a viable solution for the problem under analysis. For example, given a data set of size N, learning algorithms with linear time complexity $O(N)$ are more suitable than learning algorithms with quadratic time complexity $O(N^2)$. Certain applications have additional constraints to be satisfied. For example, the detection of pedestrians and cyclists to avoid road accidents must be performed quickly and predictably (latency constraint); if this detection has to be performed on a battery-powered device, we should be able to execute the program on specialised efficient computers (energy constraint). These constraints tend to favour homogeneous (or at least predictable) computational patterns, avoiding programs which use data-dependent branching instructions.

The metrics to be considered must be selected on a case-by-case base, depending on the point of the spectrum on which the analyst works. Business analytics applications should focus more on modelling and algorithmic properties. We will give examples of such models in Chapter 2 and Chapter 4. Hardware metrics become more important when deploying learning systems on edge computers which are embedded into the environment, and which can, therefore, rely on fewer resources than servers or workstations. We will analyse this problem more in detail in Chapter 5.

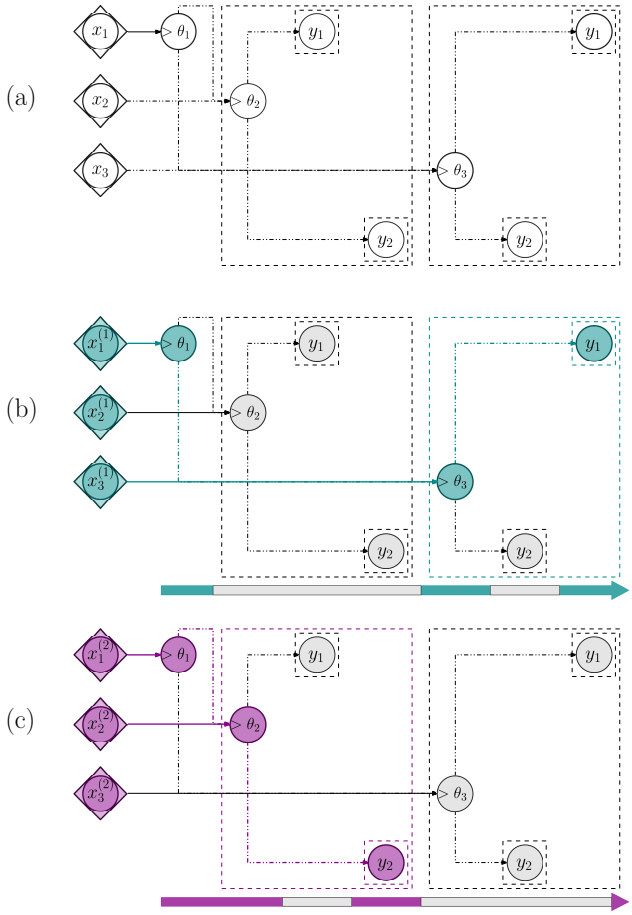


Figure 1.9: An example program that has data-dependent branching instructions. (a) The program computing the function

$$f : \mathbb{R}^3 \to \{y_1, y_2\}$$
$$\mathbf{x} \mapsto \begin{cases} y_1\,, & \text{if } (x_1 > \theta_1 \wedge x_2 > \theta_2) \vee (x_1 \leq \theta_1 \wedge x_3 > \theta_3) \\ y_2\,, & \text{if } (x_1 > \theta_1 \wedge x_2 \leq \theta_2) \vee (x_1 \leq \theta_1 \wedge x_3 \leq \theta_3)\,. \end{cases}$$

(b) An example thread taking the second branch and (c) another example thread taking the first branch. On a SIMD architecture, the two instances cannot be executed efficiently on two different datapaths, since while one datapath is idling the other is executing and viceversa. When the branching structure grows, it becomes also more unlikely that all the instructions can be cached simultaneously: this is harmful since it increases the average number of memory transactions involving lower levels of memory hierarchies, which are more expendive than memory transactions with local cache memories.

Chapter 2

Learning on bags

Real-world entities are more complex than simple functional relationships. For example, molecules often have multiple configurations called *isomers*. A molecule usually vibrates amongst its isomers in a probabilistic way, which is influenced by factors such as the temperature or the internal energy of the isomer. Another example is that of industrial plants which run several production lines. The performance of a single plant can change depending on personnel skill, maintenance policies, energy supply and other factors.

In this chapter we will propose a formalisation of *learning on bags*, a machine learning paradigm where the data representing real-world entities is more complex than a simple numerical vector space. We will also present the *fingerprint* method, a classification algorithm specifically designed to learn on bags, and analyse its algorithmic costs.

2.1 Multi-sets, bags and sequences

Let Z denote some **root set**. For instance, $\mathbf{z} \in Z$ can represent a vector of measurements describing the configuration of a molecule, or it can represent a vector of measurements collected by a suite of sensors installed on an industrial machine. A **multi-set** [9] is a collection

$$\mathfrak{B} = \{\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots, \mathbf{z}^{(N)}\} \tag{2.1}$$

where elements $\mathbf{z}^{(i)} \neq \mathbf{z}^{(j)}$ can be instances of the same element $\mathbf{z} \in Z$. When the order of the measurements is not important, the multi-set is called a **bag**. This distinguishes bags from **sequences**, where the order of the measurements is important. Therefore, a bag $\mathfrak{B}$ is completely characterised

by its **cardinality function**

$$\begin{aligned} \nu_{\mathfrak{B}} \,:\, Z &\to \mathbb{N}_0 \\ \mathbf{z} &\to \#\{\mathbf{z}^{(i)} \in \mathfrak{B} \,|\, \mathbf{z}^{(i)} = \mathbf{z}\}\,, \end{aligned} \tag{2.2}$$

which counts the repetitions of each element $\mathbf{z} \in Z$ into the bag. Consequently, we can define the equivalence relationship on the set of multi-sets (2.1) of size N:

$$\mathfrak{B}^{(1)} \sim \mathfrak{B}^{(2)} \iff \nu_{\mathfrak{B}^{(1)}} = \nu_{\mathfrak{B}^{(2)}}\,. \tag{2.3}$$

We can perform operations with bags. Let $\mathfrak{B}^{(1)}, \mathfrak{B}^{(2)}$ denote two given bags. The **union** of $\mathfrak{B}^{(1)}$ and $\mathfrak{B}^{(2)}$ is any bag $\mathfrak{B}$ such that

$$\nu_{\mathfrak{B}} = \max\{\nu_{\mathfrak{B}^{(1)}}, \nu_{\mathfrak{B}^{(2)}}\}\,.$$

The **sum** of $\mathfrak{B}^{(1)}$ and $\mathfrak{B}^{(2)}$ is any bag $\mathfrak{B}$ with cardinality function

$$\nu_{\mathfrak{B}} = \nu_{\mathfrak{B}^{(1)}} + \nu_{\mathfrak{B}^{(2)}}\,.$$

The **intersection** of $\mathfrak{B}^{(1)}$ and $\mathfrak{B}^{(2)}$ is any bag $\mathfrak{B}$ such that

$$\nu_{\mathfrak{B}} = \min\{\nu_{\mathfrak{B}^{(1)}}, \nu_{\mathfrak{B}^{(2)}}\}\,;$$

two bags are said to be **disjoint** if their intersection satisfies

$$\nu_{\mathfrak{B}} \equiv 0\,. \tag{2.4}$$

When learning on bags, it is common that different entities are represented by bags of different size N: for example, different molecules can have a different number of isomers, and different industrial plants can run a different number of production lines. The definition of machine learning system we gave in Chapter 1 used an input set X containing homogeneous instances. How is it possible to homogenise the interpretation of bags having different cardinalities? Let $\sqcup$ denote an abstract **null element**, and extend the root set Z to include it. We can extend every bag (2.1) to an infinite sequence

$$\mathfrak{B} = \{\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots, \mathbf{z}^{(N)}, \sqcup, \sqcup, \ldots\}\,. \tag{2.5}$$

With this representation, each bag (2.5) is a point in the space

$$Z^* \coloneqq (Z \cup \{\sqcup\})^*$$

of infinite sequences of elements taken from $Z \cup \{\sqcup\}$. Therefore, a machine learning system operating on bags can be modelled as a learning system operating on infinite sequences. The concept of *infinite* holds just for modelling

purposes, and is unacceptable from both an algorithmic and a computer architectures perspective. Therefore, the usual solution is to associate to each bag $\mathfrak{B}$ a probability distribution $\mu_{\mathcal{Z}}(\mathfrak{B})$: the learning problem is then instantiated between the space $X = M_Z$ of probability distributions over Z and a given output space Y. We think to an entity $\mathbf{x} \in X$ as some probability distribution $\mathbf{x} = \mu_{\mathcal{Z}}$ instead that as a simple point or vector. The measurements contained in the bag that represents the entity are interpreted as samples from this distribution.

ML systems designed to solve learning on bags are of two kinds:

- measurement-level systems;
- entity-level systems.

Let

$$\mathfrak{D} = \{(\mathfrak{B}^{(1)}, y^{(1)}), (\mathfrak{B}^{(2)}, y^{(2)}), \ldots, (\mathfrak{B}^{(K)}, y^{(K)})\} \tag{2.6}$$

denote the labelled data set for a supervised learning problem on bags. **Measurement-level systems** work as follows. First, the labelled bags data set (2.6) is turned into an ordinary labelled data set (1.44):

$$\mathfrak{D} = \{(\mathbf{z}^{(1,1)}, y^{(1,1)}), \ldots, (\mathbf{z}^{(K,N_K)}, y^{(K,N_K)})\}\,, \tag{2.7}$$

where $\mathbf{z}^{(k,i_k)} \in \mathfrak{B}^{(k)}$ and $y^{(k,i_k)} = y^{(k)}, i_k = 1, 2, \ldots, N_k$; i.e., the label of the k-th bag is attached to each of its points. Second, an machine learning system designed to work on vectors is trained using this data set. Third, at inference time each measurement $\mathbf{z}^{(i)}$ in a given bag $\mathfrak{B}$ of size N is analysed independently by the system's program, which returns a response $y^{(i)} = f(\mathbf{z}^{(i)})$ for each measurement:

$$\{y^{(1)}, y^{(2)}, \ldots, y^{(N)}\}\,.$$

The final response $\tilde{y} = f(\mathfrak{B})$ is determined by some operation on this **bag of responses**. For example, consider a classification problem, where Y is a finite set. The usual procedure is using the mode:

$$f(\mathfrak{B}) = \arg\max_{\tilde{y} \in Y} \#\{y^{(i)} \,|\, y^{(i)} = \tilde{y}\}\,. \tag{2.8}$$

Measurement-level systems work on the implicit assumption that entities $\mathfrak{B}$ must have corresponding distributions $\mu_{\mathcal{Z}}(\mathfrak{B})$ which are (almost) exact replicas of some *prototypical distributions* $\mu_{\mathcal{Z}}(y)$, $y \in Y$. If the assumption is valid, individual measurements $\mathbf{z} \in \mathfrak{B}$ will be distributed according to the prototypical distribution $\mu_{\mathcal{Z}}(\tilde{y})$ (here, $\tilde{y}$ is the class to which the entity $\mathfrak{B}$

belongs). **Entity-level systems** try instead to discover the relationships between the distributions associated with the bags $\mathfrak{B}^{(k)}$ in the data set and the response variable $y^{(k)}$. At inference time, the distribution $\mu_{\mathcal{Z}}(\mathfrak{B})$ associated with the given test bag $\mathfrak{B}$ is analysed to infer the response $f(\mathfrak{B})$. With respect to measurement-level systems, entity-level systems have pros and cons. On the positive side, entity-level systems are more flexible models, since the distributions $\mu_{\mathcal{Z}}(\mathfrak{B})$ are not assumed to be replicas of prototypical distributions; on the negative side, these systems require more modelling time to characterise the space of distributions. For example, **multiple instance learning** (MIL) [22, 23] classifies a bag $\mathfrak{B}$ depending on whether or not the support of its corresponding distribution $\mu_{\mathcal{Z}}(\mathfrak{B})$ has non-empty intersection with a *key subset* $\hat{Z} \subset Z$; this is accomplished by checking whether a *key instance* $\hat{\mathbf{z}} \in \mathfrak{B}$ exists such that $\hat{\mathbf{z}} \in \hat{Z}$. Often, in real-world scenarios, the measurements in the bags can be interpreted as samples from the Euclidean space $Z = \mathbb{R}^n$, n being some positive integer. In this cases, the bags (2.1) are also called **point clouds**, since they can be visualised as a set of points *sprayed* in $\mathbb{R}^n$. This assumption lies at the core of many classical techniques of multivariate statistics [24]: discriminant analysis (both LDA and QDA) [25] and principal component analysis (PCA) [26] are just some examples.

2.2 A heterogeneous data set

Tetra Pak is the world leader manufacturer of food processing and packaging equipment. Tetra Pak is a *business-to-business* (B2B) company, which means it provides equipment and services to other companies: Tetra Pak's customers are food packaging plants which try to satisfy the always-changing demands of consumers. This makes Tetra Pak's customers a moving target, and it is particularly important for Tetra Pak to anticipate their needs in order to hold its leadership as an equipment provider. Food packaging plants install production lines composed of multiple pieces of equipment. On each piece of Tetra Pak equipment it is installed a system of sensors belonging to the so-called *packaging line monitoring system* (PLMS): a suite of sensors measuring heat, pressures, shafts speeds, machine stops indicators and other information. The problem with food packaging plants is that they greatly vary in size (plants can run from a few lines to dozens of them) and in composition (Tetra Pak is not the only equipment supplier, though it is the largest). In addition, each line can pack food using a different packaging format, described by package volume, package shape and packaging material type. Moreover, geographical information influences the seasonality of the packed products but also, for example, the energy supplies and

the maintenance costs of the plants. This additional information describes a heterogeneous population of industrial plants. For strategic reasons, every year Tetra Pak uses packaging material sales data to classify its customers in *commercial segments*. These commercial segments are designed model the business targets of Tetra Pak's customers, but the packaging material sales data is often not sufficient to provide an accurate description of the customers (for example, when they are buying packaging material also from different suppliers). Would it have been possible to understand the business targets of Tetra Pak's customers by monitoring only the PLMS measurements marked with additional categorical information?

Recent years have seen an increasing interest in analysing data sampled from multi-modal domains. By **multi-modal**, we mean that the data points $\mathbf{z}$ are sampled from the cartesian product $Q \times X$ of a categorical space Q and a numerical space X. Categorical spaces are discrete spaces for which there is no natural embedding in a numerical set. For example, consider food packages. Packages are usually produced using a finite number of formats, characterised by volumes, shapes and packaging material. The set of volumes of the different formats are numbers which can be naturally mapped to a numerical set. Instead, the set of available shapes and that of available packaging material types have no natural ordering relationships that allow mapping them to numerical sets. The elements $\mathbf{q} \in Q$ of a categorical space are called its **levels**. We will suppose that the numerical space X is the Euclidean space $\mathbb{R}^n$.

Consider a set of **entities** about which we can measure tuples

$$\mathbf{z} \coloneqq (\mathbf{q}, \mathbf{x}) \,, \tag{2.9}$$

where $\mathbf{q}$ is the categorical part of the measurement and $\mathbf{x}$ is its numerical part. We will refer to the components of $\mathbf{z}$ using the dotted notation $\mathbf{z}.\mathbf{q}$ and $\mathbf{z}.\mathbf{x}$. The set of measurements collected about an entity is a bag

$$\mathfrak{B}^{(k)} = \{\mathbf{z}^{(k,1)}, \mathbf{z}^{(k,2)}, \ldots, \mathbf{z}^{(k,N_k)}\} \,,$$

which is called the **entity bag**. We define the **point-cloud projection** of a given bag $\mathfrak{B}$ to be the point cloud

$$\mathcal{P} \coloneqq \{\mathbf{z}.\mathbf{x} \,|\, \mathbf{z} \in \mathfrak{B}\} \,.$$

Our goal is to assign each entity to one out of a finite number of classes

$$Y = \{y_1, y_2, \ldots, y_{N_Y}\} \,; \tag{2.10}$$

i.e., we want to solve a classification task on bags. An important note: the figures in this chapter have been generated *ad-hoc* to illustrate the main ideas

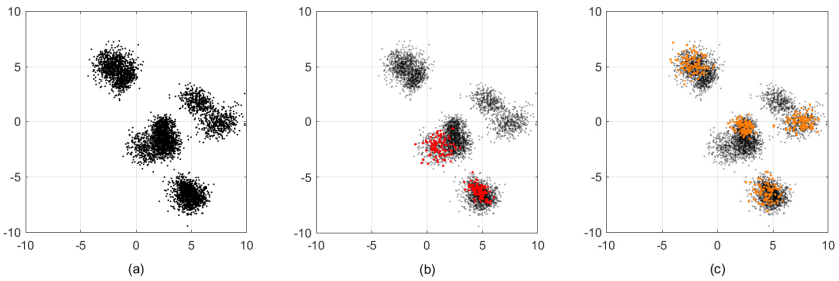


Figure 2.1: A visualisation of the learning problem on bags: (a) the data point cloud in $X = \mathbb{R}^2$ depicting the numerical measurements contained in the available labelled bags, (b) an example entity bag (red point cloud) and (c) another example entity bag (orange point cloud).

since we do not have the permission to report the results on the original data.

Let (2.6) be the available data set of labelled bags, and (2.7) the corresponding labelled measurements data set. For each class $y_j \in Y$, we can create the corresponding unlabelled **class bag**

$$\mathfrak{B}^{(j)} = \{\mathbf{z}^{(k,i_k)} \,|\, y^{(k,i_k)} = y_j\}\,, \tag{2.11}$$

and project it to the **class point cloud**

$$\mathcal{P}^{(j)} = \{\mathbf{z}.\mathbf{x} \,|\, \mathbf{z} \in \mathfrak{B}^{(j)}\}\,. \tag{2.12}$$

To this point cloud we can associate a distribution

$$\mu_{\mathcal{X}}(\mathcal{P}^{(j)})\,, \tag{2.13}$$

which is an n-variate distribution since $X = \mathbb{R}^n$. We isolated four such point clouds from the data set we analysed for Tetra Pak. Applying linear discriminant analysis (LDA) to them revealed us that they had a specific structure: the levels $\mathbf{q} \in Q$ of the categorical variable defined clearly distinguishable sub-populations

$$\mathfrak{B}^{(j,\tilde{\mathbf{q}})} = \{\mathbf{z} \in \mathfrak{B}^{(j)} \,|\, \mathbf{z}.\mathbf{q} = \tilde{\mathbf{q}}\}\,,$$

in the sense that the corresponding point clouds

$$\mathcal{P}^{(j,\tilde{\mathbf{q}})} \tag{2.14}$$

defined distributions $\mu_{\mathcal{X}}(\mathcal{P}^{(j,\tilde{\mathbf{q}})})$ which were n-variate normal distributions with different means and covariance matrices. We called the point clouds (2.14) **homogeneous groups**. The family of point clouds

$$\mathcal{F}^{(j)} = \{\mathcal{P}^{(j,\mathbf{q})}\}_{\mathbf{q}\in Q} \tag{2.15}$$

is called the **fingerprint** of the class y_j. Denoting by $p(x|j)$ and by $p(x|j,\mathbf{q})$ the densities of the measures $\mu_{\mathcal{X}}(\mathcal{P}^{(j)})$ and $\mu_{\mathcal{X}}(\mathcal{P}^{(j,\mathbf{q})})$ respectively, this decomposition process is analogous to interpreting (2.13) as a *mixture* distribution. In fact, the density of a mixture distribution can be expressed as

$$p(x|j) = \sum_{\mathbf{q}\in Q} p(x|j,\mathbf{q})p(\mathbf{q})\,,$$

where the categorical variable $\mathbf{q}$ indexes different components of the mixture; $p(\mathbf{q})$ is a PMF (1.21) on the categorical space Q.

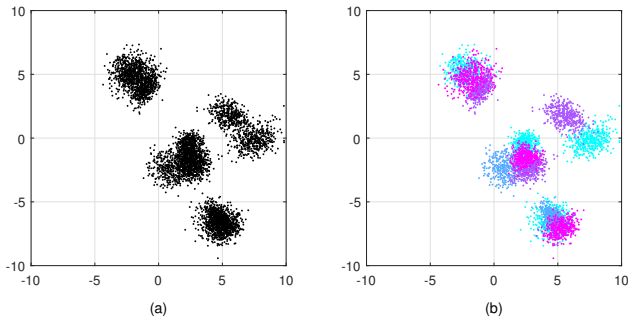


Figure 2.2: The problem of overlapping distributions: (a) the same global point cloud as that in Figure 2.1a is a sample from (b) a collection of $N_Y = 4$ probability distributions whose supports are highly overlapped.

Consider a test entity described by a bag $\mathfrak{B}^{(\tilde{k})}$. To this bag we can attach a probability distribution $\mu_{\mathcal{X}}(\mathcal{P}^{(\tilde{k})})$ derived from its point cloud $\mathcal{P}^{(\tilde{k})}$. We can apply to the bag the same decomposition we applied to the class bags (2.11), obtaining the fingerprint of the entity:

$$\mathcal{F}^{(\tilde{k})} = \{\mathcal{P}^{(\tilde{k},\mathbf{q})}\}_{\mathbf{q}\in Q}\,. \tag{2.16}$$

Algorithm 1 shows how to compute the fingerprint of a given bag. The fingerprint method rephrases the problem of classifying the bag $\mathfrak{B}^{(\tilde{k})}$ as a comparison between the family of distributions

$$\{\mu_{\mathcal{X}}(\mathcal{P}^{(\tilde{k},\mathbf{q})})\}_{\mathbf{q}\in Q}\,, \tag{2.17}$$

associated with the fingerprint of the entity $\mathcal{F}^{(\tilde{k})}$, and the N_Y families of distributions

$$\{\mu_{\mathcal{X}}(\mathcal{P}^{(j,\mathbf{q})})\}_{\mathbf{q}\in Q}\,,\; j=1,2,\ldots,N_Y\,, \tag{2.18}$$

associated with the fingerprints $\mathcal{F}^{(j)}$ of the classes. This comparison is performed on each matching pair of distributions. We say that an entity distribution $\mu_{\mathcal{X}}(\mathcal{P}^{(\tilde{k},\mathbf{q}(\tilde{k}))})$ and a class distribution $\mu_{\mathcal{X}}(\mathcal{P}^{(j,\mathbf{q}(j))})$ are **matching** if they correspond to the same homogeneous group $\mathbf{q}(\tilde{k})=\mathbf{q}(j)$. The instance bag $\mathfrak{B}^{(\tilde{k})}$ is assigned to the class $\tilde{y}$ for which (2.17) and (2.18) result most similar under the specified comparison operation. In the next section, we will describe a specific choice to compare these families of distributions.

Algorithm 1 Split a bag $\mathfrak{B}$ of tuples (list of structures) into its fingerprint $\mathcal{F}$ (dictionary of lists).

Function: `compute_fingerprint`
Input: $\mathfrak{B}$
Output: $\mathcal{F}$

1: $\mathcal{F} \leftarrow dict()$
2: **for** $\mathbf{z} \in \mathfrak{B}$ **do**
3: $\quad \mathbf{q} \leftarrow \mathbf{z}.\mathbf{q}$
4: $\quad$ **if** $\mathbf{q} \notin \mathcal{F}.keys()$ **then**
5: $\quad\quad \mathcal{F}[\mathbf{q}] \leftarrow list()$
6: $\quad$ **end if**
7: $\quad \mathcal{F}[\mathbf{q}].append(\mathbf{z}.\mathbf{x})$
8: **end for**
9: **return** $\mathcal{F}$

2.3 The *fingerprint* algorithm

Our analysis of the point clouds in the specific Tetra Pak data set revealed them to be approximately normally distributed. Therefore, it was sufficient to define a program space capable of characterising the first and second momenta of these distributions. We opted for extracting the mean (Algorithm 2, line 5) and the covariance matrix (Algorithm 2, line 6) of each class point cloud $\mathcal{P}^{(j,\mathbf{q})}$ together with some additional statistics: the average Mahalanobis distance [27] of the points in the point cloud from the mean (Algorithm 2, line 7), the standard deviation of this quantity (Algorithm 2, line 8) and the first n_{PCA} principal components (Algorithm 2, line 9).

Algorithm 2 Given the bag $\mathfrak{B}^{(j)}$ (list of structures) representing a class $y_j \in Y$ (enumerated), compute the statistics $\mathcal{T}[j]$ (dictionary of structures) of its fingerprint.

Input: $\mathfrak{B}^{(j)}, n_{PCA}$
Output: $\mathcal{T}[j]$

1: $\mathcal{F} \leftarrow$ `compute_fingerprint`$(\mathfrak{B}^{(j)})$
2: $\mathcal{T}[j] \leftarrow dict()$
3: **for** $\mathbf{q} \in \mathcal{F}.keys()$ **do**
4: $\quad \mathcal{P} \leftarrow \mathcal{F}[\mathbf{q}]$
5: $\quad \mathcal{T}[j][\mathbf{q}].\mathbf{m} \leftarrow mean(\mathcal{P})$
6: $\quad \mathcal{T}[j][\mathbf{q}].\mathbf{S} \leftarrow cov(\mathcal{P})$
7: $\quad \mathcal{T}[j][\mathbf{q}].m \leftarrow mean(\{\sqrt{(\mathbf{x}-\mathbf{m})\mathbf{S}^{-1}(\mathbf{x}-\mathbf{m})} \,|\, \mathbf{x} \in \mathcal{P}\})$
8: $\quad \mathcal{T}[j][\mathbf{q}].s \leftarrow stdv(\{\sqrt{(\mathbf{x}-\mathbf{m})\mathbf{S}^{-1}(\mathbf{x}-\mathbf{m})} \,|\, \mathbf{x} \in \mathcal{P}\})$
9: $\quad \mathcal{T}[j][\mathbf{q}].N \leftarrow PCA(\mathbf{S}, n_{PCA})$
10: **end for**
11: **return** $\mathcal{T}[j]$

To classify a test entity bag $\mathfrak{B}^{(\tilde{k})}$, the *fingerprint* method compares matching point clouds $\mathcal{P}^{(\tilde{k},\mathbf{q})} \in \mathcal{F}^{(\tilde{k})}$ and $\mathcal{P}^{(j,\mathbf{q})} \in \mathcal{F}^{(j)}$ to compute two quantities (Algorithm 3, lines 3-21):

(i) the average Mahalanobis distance between the points of the entity point cloud and the mean of the class point cloud, computed using the sample covariance matrix associated with the class point cloud as the metric matrix;

(ii) the average of the alignments (i.e., absolute cosines) of the angles between the first n_{PCA} principal components of the two point clouds.

The entity bag $\mathfrak{B}^{(\tilde{k})}$ is then assigned to the class $\tilde{y}$ for which the distances between the centers of the homogeneous groups are minimised and for which the alignments of the principal components of the homogeneous groups are maximised (Algorithm 3, lines 22-33).

Let $\#Q$ denote the number of levels of the categorical variable, which is of course the number of possible homogeneous groups composing every fingerprint. The time cost of the learning Algorithm 2 is $O(nN^2) + O(n^3)$, where N is the size of the data set and n the dimension of the Euclidean space X; the cubic term in n originates in the extraction of the principal components, which requires matrix inversion. Interestingly, we observe that the computations of the statistics associated with the homogeneous groups can be run

in parallel, amortising the term $O(nN^2)$ to an average of $O(nN^2/\#Q)$ (supposing each homogeneous group contains approximately the same number of points). The space cost of the learning algorithm is $O(\#Qn^2)$, since for every homogeneous group it must store the respective covariance matrix.

Learning on bags is a great example of the complexities inherent to real-world data analysis problems. The design is often time-consuming since it requires comprehensive analysis of the data. Moreover, a number of properties of the data set filter into the model in the form of assumptions (in our case, the availability of categorical information and the approximate normality of the corresponding point clouds). With respect to more *general purpose* machine learning systems, this characteristic limits the number of scenarios to which these models can be applied: if these assumptions are violated, the performance of the system will usually degrade.

From an algorithmic perspective, these systems often have quadratic complexities since they extract *global* properties of the data (they compute quantities over all the pairs of points in the experience data set). This characteristic makes them in general more accurate, but their programs are also more cumbersome to train and store.

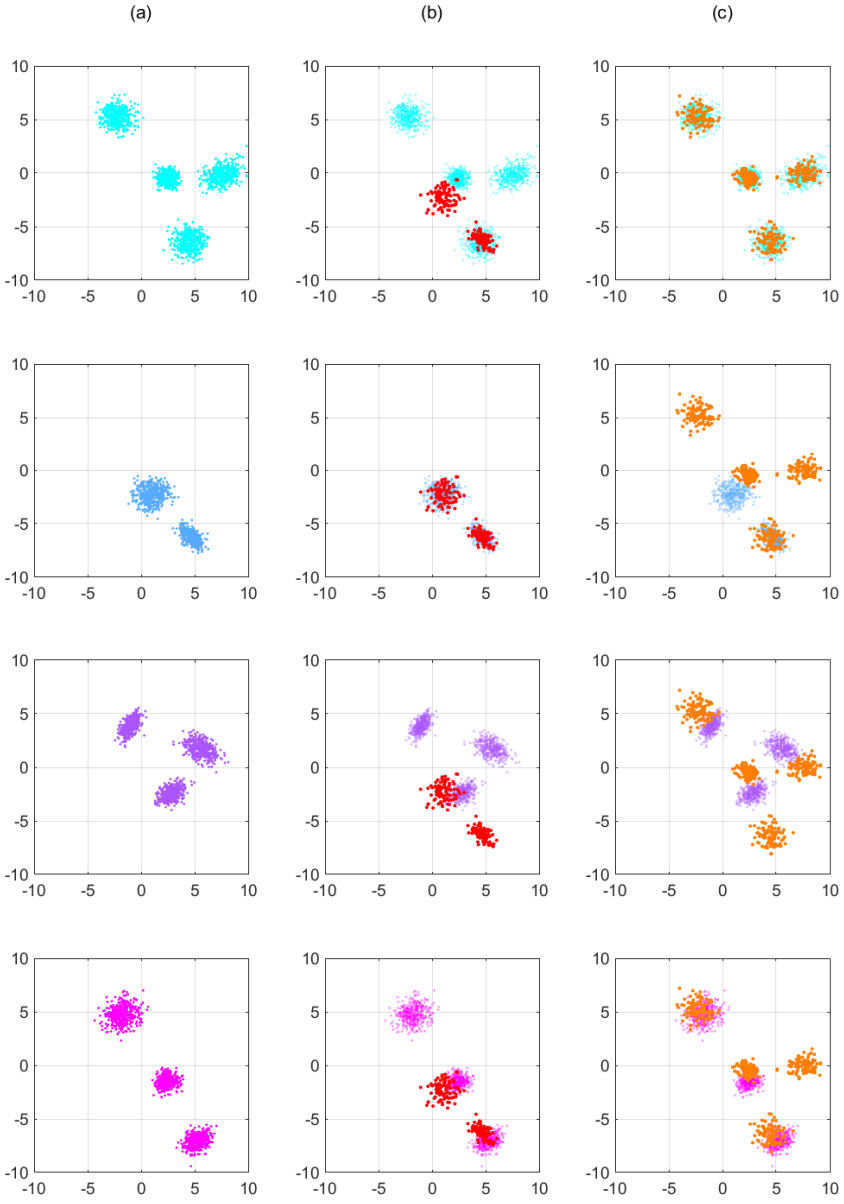


Figure 2.3: The $N_Y = 4$ figures in column (a) separately depict the point clouds $\mathcal{P}^{(j)}$ which appeared mixed in Figure 2.2b: we observe that normal sub-populations emerge. It is therefore easier to compare the entity point cloud of Figure 2.1b and assign it to the class y_2, as shown by the figures in column (b). Figures in column (c) show that the entity point cloud of Figure 2.1c likely belongs to the class y_1.

Algorithm 3 Given the trained statistics $\mathcal{T}$ (dictionary of dictionary of structures) for all the classes $y \in Y$ (enumerated), assign an instance bag $\mathfrak{B}^{(\tilde{k})}$ to a class.

Input: $\mathcal{T}, n_{PCA}, \theta, \mathfrak{B}^{(\tilde{k})}$
Output: $\tilde{y}$

$\tilde{\mathcal{F}} \leftarrow$ `compute_fingerprint`$(\mathfrak{B}^{(\tilde{k})})$
$momenta \leftarrow dict()$
for $y_j \in Y$ **do** ▷ compare matching groups
  $first \leftarrow list()$
  $second \leftarrow list()$
  for $\mathbf{q} \in \tilde{F}.keys()$ **do**
    $\tilde{\mathcal{P}} \leftarrow \tilde{F}[\mathbf{q}]$
    $\mathbf{m} \leftarrow \mathcal{T}[j][\mathbf{q}].\mathbf{m}$
    $\mathbf{S} \leftarrow \mathcal{T}[j][\mathbf{q}].\mathbf{S}$
    $m \leftarrow \mathcal{T}[j][\mathbf{q}].m$
    $s \leftarrow \mathcal{T}[j][\mathbf{q}].s$
    $N \leftarrow \mathcal{T}[j][\mathbf{q}].N$
    $\tilde{m} \leftarrow mean(\{\sqrt{(\mathbf{x}-\mathbf{m})\mathbf{S}^{-1}(\mathbf{x}-\mathbf{m})} \,|\, \mathbf{x} \in \tilde{\mathcal{P}}\})$
    $first.append((\tilde{m}-m)/s)$
    $\tilde{\mathbf{S}} \leftarrow cov(\tilde{\mathcal{P}})$
    $\tilde{N} \leftarrow PCA(\tilde{\mathbf{S}}, n_{PCA})$
    $second.append(mean(\{|\langle \tilde{\mathbf{n}}_i, \mathbf{n}_i \rangle| \,|\, i = 1, 2, \ldots, n_{PCA}\}))$
  end for
  $momenta[j].first \leftarrow mean(first)$
  $momenta[j].second \leftarrow mean(second)$
end for
$y_{j_1} \leftarrow \arg\min_{y_j \in Y} \{momenta[j].first\}$ ▷ classification
$y_{j_2} \leftarrow \arg\min_{y_j \in Y \setminus \{y_{j_1}\}} \{momenta[j].first\}$
if $momenta[j_2].first \geq (1-\theta) momenta[j_1].first$ **then**
  if $momenta[y_{j_1}].second \geq momenta[y_{j_1}].second$ **then**
    $\tilde{y} \leftarrow y_{j_2}$
  else
    $\tilde{y} \leftarrow y_{j_1}$
  end if
else
  $\tilde{y} \leftarrow y_{j_1}$
end if
return $\tilde{y}$

Chapter 3

Network-based learning

Humans are fascinated by their own ability to extract meaning from the surrounding world. Due to its role as the organ responsible of this intriguing capability, the brain is a source of inspiration for many machine learning systems. Amongst these systems, *artificial neural networks* are appealing for many reasons. First, they are powerful, non-linear function approximators. Second, their computational structure is parallelisable and data-independent, properties which make their programs hardware-friendly.

In this chapter we will review the history of artificial neural networks, from *linear threshold units* to the introduction of the *backpropagation* algorithm. We will recall important theoretical results about artificial neural networks, analyse the backpropagation algorithm in detail and describe techniques developed to improve the quality of the learnt programs.

3.1 A specific program space

Let $L \geq 2$ be an integer. For each $\ell \in \{0, 1, 2, \dots, L\}$ we define a positive integer $n_\ell \in \mathbb{N}$. Let $\hat{n}_{\ell'} \coloneqq \sum_{\ell=1}^{\ell'} n_\ell$ and $\hat{n} = \hat{n}_L$. Consider a set of units

$$\mathcal{N} \coloneqq \{N_i = 1, 2, \dots, \hat{n}\}\,. \tag{3.1}$$

Each unit N_i is called a **neuron** and we associate to it a state space X_i; we make a qualitative distinction between the elements $x \in X_i$ in *active states* and *inactive states*. We partition (3.1) into $L + 1$ subsets

$$\mathcal{N}^\ell \coloneqq \{N_{i_\ell}^\ell\,, i_\ell = 1, 2, \dots, n_\ell\} \tag{3.2}$$

that we call **layers**. We define this partition in a conventional way, so that mapping a unit $N_i \in \mathcal{N}$ to the unit $N_{i_\ell}^\ell \in \mathcal{N}^\ell$ can be done using the following

equations:

$$\begin{aligned} \ell(i) &= \underset{\ell \in \{1,2,\dots,L\}}{\arg\max} \{\hat{n}_\ell \,|\, i > \hat{n}_\ell\}\,, \\ i_\ell &= i - \ell(i)\,; \end{aligned}$$

conversely, we can map a unit $N_{i_\ell}^\ell \in \mathcal{N}^\ell$ to $N_i \in \mathcal{N}$ using the equation

$$i = \hat{n}_{\ell-1} + i_\ell\,.$$

We say that *layer* $\mathcal{N}^{\ell_2}$ *follows layer* $\mathcal{N}^{\ell_1}$ if $\ell_2 > \ell_1$. We say that *layers* $\mathcal{N}^{\ell_1}, \mathcal{N}^{\ell_2}$ *are adjacent* if $|\ell_2 - \ell_1| = 1$. To the layers (3.2) we can naturally associate the product state spaces

$$X^\ell \coloneqq X_1^\ell \times X_2^\ell \times \cdots \times X_{n_\ell}^\ell\,, \ell = 0, 1, \dots, L\,, \tag{3.3}$$

which we call **representations spaces**; points $\mathbf{x}^\ell \in X^\ell$ are called **representations**. **Artificial Neural Networks** (ANNs) give (3.1) the interpretation of vertices of a directed graph

$$(\mathcal{N}, \mathcal{E})\,, \tag{3.4}$$

where $\mathcal{E}$ can be described using a connectivity matrix

$$\begin{aligned} \mathbf{A} &= (a_{ij})_{i,j=1,2,\dots,\hat{n}} \\ &= \begin{cases} 0, & \text{if } (i,j) \notin \mathcal{E} \\ 1, & \text{if } (i,j) \in \mathcal{E}\,. \end{cases} \end{aligned} \tag{3.5}$$

Given two layers $\mathcal{N}^{\ell_1}, \mathcal{N}^{\ell_2}$, we denote by $\mathbf{A}^{(\ell_1,\ell_2)}$ the block of $\mathbf{A}$ describing the connections from units $N_{i_{\ell_1}}^{\ell_1} \in \mathcal{N}^{\ell_1}$ to units $N_{i_{\ell_2}}^{\ell_2} \in \mathcal{N}^{\ell_2}$. We say that a network (3.4) is a **feedforward neural network** if its adjacency matrix satisfies the following property:

$$\mathbf{A}^{\ell_1,\ell_2} = \mathbf{0}\,, \forall \ell_1 \geq \ell_2\,.$$

In the following we will discuss only feedforward neural networks. The units are abstractions for the *somas* and *axons* of biological neurons. The links are abstractions for their *synapsis* and *dendrites*. Given an edge $(i,j) \in \mathcal{E}$, the unit $N_i \in \mathcal{N}$ is called the *pre-synaptic* neuron, whereas $N_j \in \mathcal{N}$ is called the *post-synaptic* neuron. To model biological neurons, links can be *excitatory* if activity in the pre-synaptic neuron is likely to generate activity in the post-synaptic neuron; *inhibitory* if activity in the pre-synaptic neuron is likely to

suppress activity in the post-synaptic neuron; *absent* if activity in the pre-synaptic neuron has no effect on the activity of the post-synaptic neuron. The rules that govern the interaction of the neurons are known as *laws of nervous excitation* [28].

To use networks as programs, we must attach to the units in (3.1) some **computational rules**; i.e., some operations that implement a specific model of nervous excitation. To this end, we consider the specific case where $X_{i_\ell}^\ell \subseteq \mathbb{R}$, and consequently $X^\ell \subseteq \mathbb{R}^{n_\ell}$ for every $\ell \in \{0, 1, \dots, L\}$. Fix $\ell' \in \{0, 1, \dots, L\}$. We denote by

$$\hat{\mathbf{x}}^{\ell'} = (\mathbf{x}^0, \mathbf{x}^1, \dots, \mathbf{x}^{\ell'})$$

the joint representations of the first ℓ' layers, and by

$$\widehat{X}^{\ell'} \coloneqq X^0 \times X^1 \times \cdots \times X^{\ell'}$$

the corresponding joint representations space. First, we attach to each edge $(i, j) \in \mathcal{E}$ a **weight** $w_{ij} \in \mathbb{R}$; we collect these weights into the matrix $\mathbf{W} \in W \subseteq \mathcal{M}^{\hat{n}\times\hat{n}}(\mathbb{R})$, that must satisfy

$$a_{ij} = 0 \implies w_{ij} = 0\,;$$

for each pair of layers $\mathcal{N}^{\ell_1}, \mathcal{N}^{\ell_2}$, we define their **connection matrix** to be matrix

$$\mathbf{W}^{\ell_1,\ell_2} \coloneqq (w_{ij})_{\substack{i=\hat{n}_{\ell_1-1}+1,\dots,\hat{n}_{\ell_1} \\ j=\hat{n}_{\ell_2}+1,\dots,\hat{n}_{\ell_2}}}\,, \tag{3.6}$$

taken from the space $W^{\ell_1,\ell_2} \subseteq \mathcal{M}^{n_{\ell_1}\times n_{\ell_2}}(\mathbb{R})$. Again, fix $\ell' \in \{0, 1, \dots, L\}$. We define the weight matrix of layer $\mathcal{N}^{\ell'}$ as the concatenation

$$\widehat{\mathbf{W}}^{\ell'} \coloneqq \begin{pmatrix} \mathbf{W}^{0,\ell'} \\ \mathbf{W}^{1,\ell'} \\ \cdots \\ \mathbf{W}^{\ell'-1,\ell'} \end{pmatrix}\,, \tag{3.7}$$

which lives in the space $\widehat{W}^{\ell'} \subseteq \mathcal{M}^{\hat{n}_{\ell'-1}\times n_{\ell'}}(\mathbb{R})$. The column $\hat{\mathbf{w}}_{i_{\ell'}}^{\ell'}$ is called the **filter** of the neuron $N_{i_{\ell'}}^{\ell'} \in \mathcal{N}^{\ell'}$. We then attach to each neuron $N_{i_\ell}^\ell \in \mathcal{N}^\ell$ a **bias** $b_{i_\ell}^\ell \in B_{i_\ell}^\ell \subseteq \mathbb{R}$; for each layer $\mathcal{N}^\ell$, we group the biases of its neurons in the vector

$$\mathbf{b}^\ell \tag{3.8}$$

taken from a space $B^\ell \subseteq \mathbb{R}^{n_\ell}$. Finally, to each neuron $N_{i_\ell}^\ell \in \mathcal{N}^\ell$ we attach a function

$$\sigma_{i_\ell}^\ell : \mathbb{R} \to \mathbb{R} \tag{3.9}$$

called the **activation function** or **transfer function**. The computational rule attached to neuron $N_{i_{\ell'}}^{\ell'}$ is the following:

$$x_{i_{\ell'}}^{\ell'} = \sigma_{i_{\ell'}}^{\ell'} \left(\hat{\mathbf{x}}^{\ell'-1} \hat{\mathbf{w}}_{i_{\ell'}}^{\ell'} + b_{i_{\ell'}}^{\ell'} \right) , \tag{3.10}$$

where

$$\begin{aligned} s_{i_{\tilde{l}}}^{\ell'} &\coloneqq \hat{\mathbf{x}}^{\ell'-1} \hat{\mathbf{w}}_{i_{\tilde{l}}}^{\ell'} \\ &= \sum_{i=1}^{\hat{n}_{\ell'-1}} \hat{x}_i^{\ell'-1} \hat{w}_{i i_{\ell'}}^{\ell'} \end{aligned}$$

is called the **score** of the neuron and

$$\tilde{s}_{i_{\tilde{l}}}^{\ell'} = s_{i_{\tilde{l}}}^{\ell'} + b_{i_{\tilde{l}}}^{\ell'}$$

is its **adjusted score**. To express the computation of a layer in a more compact form, we will write

$$\begin{aligned} \mathbf{x}^{\ell'} &= \sigma^{\ell'} \left(\hat{\mathbf{x}}^{\ell'-1} \widehat{\mathbf{W}}^{\ell'} + \mathbf{b}^{\ell'} \right) \\ &= \sigma^{\ell'} \left(\sum_{\ell=0}^{\ell'-1} \mathbf{x}^{\ell} \mathbf{W}^{\ell,\ell'} + \mathbf{b}^{\ell'} \right) , \end{aligned} \tag{3.11}$$

where we use the vector notation

$$\sigma \left(s^{\ell'} + \mathbf{b}^{\ell'} \right) = \left(\sigma_1^{\ell'}(s_1^{\ell'} + b_1^{\ell'}), \ldots, \sigma_{n_{\ell'}}^{\ell'}(s_{n_{\ell'}}^{\ell'} + b_{n_{\ell'}}^{\ell'}) \right) .$$

We observe that (3.11) is a map

$$\begin{aligned} \varphi^{\ell'} \; : \; M^{\ell'} \times \widehat{X}^{\ell'-1} &\to X^{\ell'} \\ (\mathbf{m}^{\ell'}, \hat{\mathbf{x}}^{\ell'-1}) &\mapsto \mathbf{x}^{\ell'} , \end{aligned} \tag{3.12}$$

called the **layer map** of layer $\mathcal{N}^{\ell'}$, where $\mathbf{m}^{\ell'} = (\widehat{\mathbf{W}}^{\ell'}, \mathbf{b}^{\ell'})$ is the **parameters** tuple associated with the ℓ'-th layer and $M^{\ell'} \coloneqq \widehat{W}^{\ell'} \times B^{\ell'}$ is its **parameters space**. The collection of layer maps

$$\Phi \coloneqq \{ \varphi^{\ell} , \ell = 1, 2, \ldots, L \} \tag{3.13}$$

is called the **program space of the ANN**. When we say that a network (3.13) is a L**-layers ANN** we are counting the number of layer maps (3.12) and not the number of layers (3.2). The set $\mathcal{N}^0$ is called the **input layer**, $\mathcal{N}^L$

is called the **output layer** and $\mathcal{N}^\ell, \ell \in \{1, 2, \ldots, L-1\}$ are called **hidden layers**. The representations spaces X^0, X^L and $X^\ell, \ell \in \{1, 2, \ldots, L-1\}$ are called the **input space**, **output space** and **hidden spaces** respectively.

An important subclass of ANNs is that where connections between non-adjacent layers (called **skip-connections**) are not allowed. ANNs that satisfy this property are called **simple feedforward neural networks**. In simple feedforward neural networks, the matrices (3.7) are composed of blocks satisfying

$$\mathbf{W}^{\ell,\ell'} = \mathbf{0}\,, \forall \ell < \ell' - 1\,.$$

We can simplify (3.11) by substituting $\widehat{\mathbf{W}}^{\ell'} = \mathbf{W}^{\ell,\ell'}$, obtaining layer functions

$$\mathbf{x}^{\ell'} = \sigma^{\ell'}\left(\mathbf{x}^{\ell'-1}\mathbf{W}^{\ell,\ell'} + \mathbf{b}^{\ell'}\right)\,.$$

In such cases, we can rewrite the layer maps (3.12) as parametric families of functions

$$\{\varphi^\ell_{\mathbf{m}^\ell} \,:\, X^{\ell-1} \to X^\ell\}_{\mathbf{m}^\ell \in M^\ell}\,,\, l = 1, 2, \ldots, L\,,$$

whose elements can be directly composed:

$$\Phi \coloneqq \varphi^L_{\mathbf{m}^L} \circ \cdots \circ \varphi^1_{\mathbf{m}^1}\,. \tag{3.14}$$

These composable structures make simple feedforward neural networks easier to analyse than general ANNs: theoretical results on ANNs are usually derived on the restriction to this sub-class of models.

3.2 From LTUs to ANNs

In 1943, Warren McCulloch and Walter Pitts analysed ANNs (3.13) with Boolean representations spaces

$$X^\ell = \{0,1\}^{n_\ell}\,.$$

They used the Heaviside function

$$H(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases} \tag{3.15}$$

as the activation function $\sigma^\ell_{i_\ell}$ for each neuron $N^\ell_{i_\ell}$. Note that the codomain of the Heaviside function consists of the numerical representations 0/1 (inactive/active) for the false/true Boolean values. The weights w_{ij} could take values in the set

$$\{-\infty\} \cup \{n \in \mathbb{Z} \,|\, n \leq 1\}\,.$$

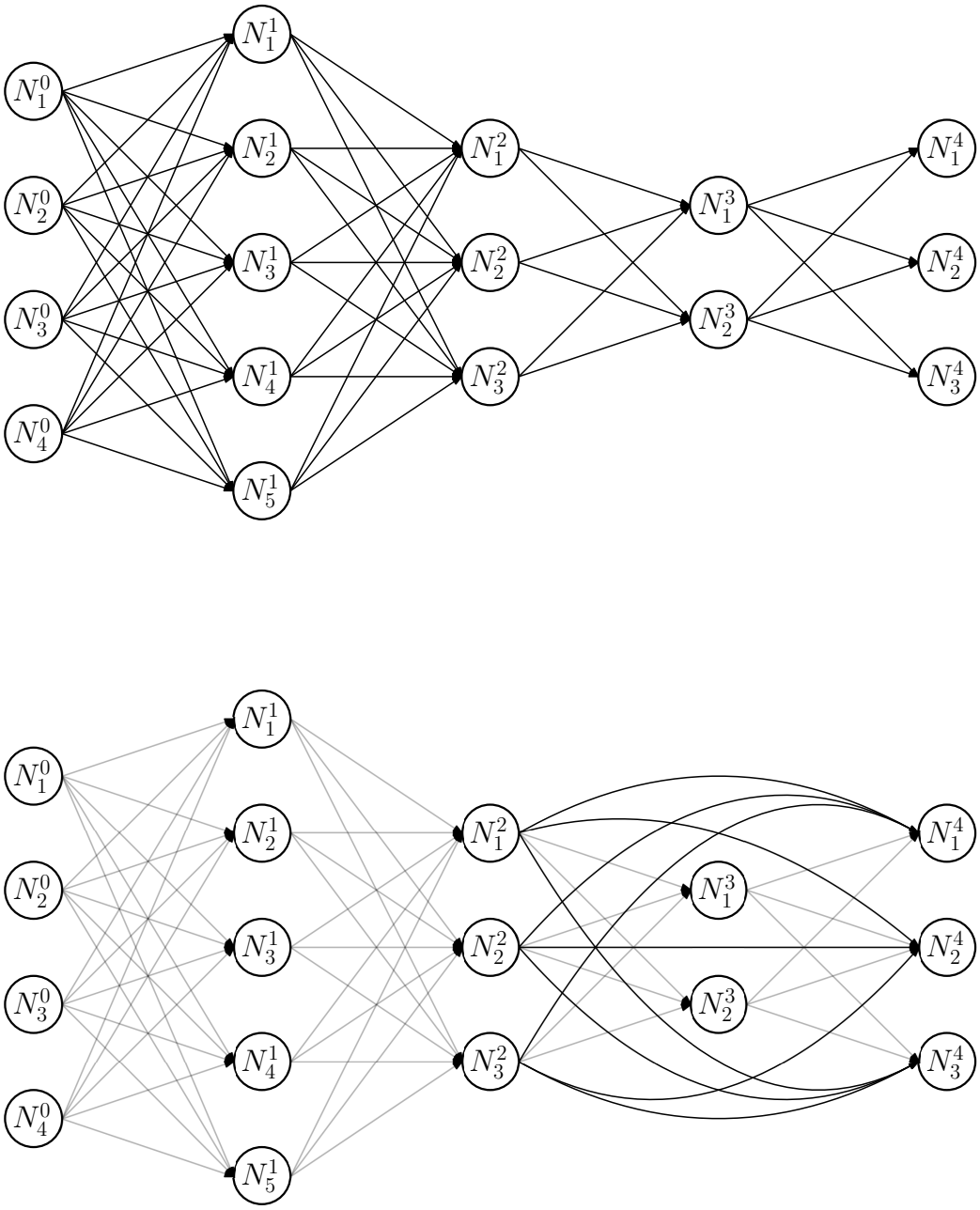


Figure 3.1: Examples of feedforward neural networks: (a) a 4-layers simple feedforward neural network and (b) the same network, where connections between neurons of $\mathcal{N}^2$ and $\mathcal{N}^4$ have been added.

Excitatory connections were modelled by $w_{ij} = 1$, absent connections by $w_{ij} = 0$ and inhibitory connections by either $w_{ij} = n < 0$ (*relative inhibitory*

connections) or $w_{ij} = -\infty$ (*absolute inhibitory* connections). The computational rules used to compute $\hat{\mathbf{x}}^{\ell'-1}\widehat{\mathbf{W}}^{\ell'}$ were taken from the usual integer arithmetic extended to include the following rules:

$$\begin{aligned} 0 \cdot (-\infty) &= 0\,, \\ 1 \cdot (-\infty) &= -\infty\,, \\ n + (-\infty) &= -\infty\,, \forall\, n \in \mathbb{Z}\,. \end{aligned}$$

These rules were intended to model a particular property of biological networks, where certain pairs (N_i, N_j) of neurons are such that the activity of the pre-synaptic neuron N_i is able to completely suppress the activity of the post-synaptic neuron N_j. They combined these rules to model networks of so-called **Linear Threshold Units** (LTUs), which computed

$$\begin{aligned} x_{i_\ell}^{\ell}(t+1) &= H(\hat{\mathbf{x}}^{\ell-1}(t)\widehat{\mathbf{w}}_{i_\ell}^{\ell} + b_{i_\ell}^{\ell}) \\ &= \begin{cases} 0, & \text{if } \hat{\mathbf{x}}^{\ell-1}(t)\widehat{\mathbf{w}}_{i_\ell}^{\ell} < -b_{i_\ell}^{\ell} \\ 1, & \text{if } \hat{\mathbf{x}}^{\ell-1}(t)\widehat{\mathbf{w}}_{i_\ell}^{\ell} \geq -b_{i_\ell}^{\ell}\,, \end{cases} \end{aligned} \tag{3.16}$$

where the biases $b_{i_\ell}^{\ell}$ played the role of *thresholds*. We observe that the model (3.16) included a time dimension, implying that neurons $N_{i_\ell}^{\ell}$ updated their state $x_{i_\ell}^{\ell}(t)$ in a synchronous manner governed by some *system clock*. This dynamics was introduced to model a physical system subject to communication and computation delays. McCulloch and Pitts showed that networks of components (3.16) could be assembled to represent arbitrary logical predicates in the Boolean vector variable $\mathbf{x}^0$. The main limitation of the McCulloch-Pitts model was the lack of the notion of *learnable* parameters. Their analysis was theoretical, and they proposed no practical algorithms to find a network implementing a target predicate.

In 1957, Frank Rosenblatt described a system whose elementary components were capable of modifying their state to improve responses to presented stimuli [29, 30]. This system, named **perceptron**, was a simple 2-layers feed-forward network:

- $\mathcal{N}^0$: *sensory units* or *S-units*;
- $\mathcal{N}^1$: *association units* or *A-units*;
- $\mathcal{N}^2$: *response units* or *R-units*.

In its simplest form, the R-layer consisted of only two units. All the A-units and R-units were LTUs (3.16) implementing the following computations:

$$\begin{aligned} \mathbf{x}^1 &= H(\mathbf{x}^0\mathbf{W}^1 + \mathbf{b}^1)\,, \\ \mathbf{x}^2 &= H(\mathbf{x}^1\mathbf{W}^2 + \mathbf{b}^2)\,. \end{aligned}$$

Here, $\mathbf{W}^1 \in \mathcal{M}^{n_0 \times n_1}(\{-1, 0, 1\})$ was a matrix with ternary components modelling inhibitory, absent and excitatory connections. The matrix was obtained as

$$\mathbf{W}^2 = \left(\mathbf{w}^2, \mathbf{w}^2\right) \odot \mathbf{A}^{1,2} ;$$

this specific form was due to the fact that the *values* in $\mathbf{w}^2$ where not specific to the links between A-units and R-units, but they were implemented as electric charges accumulated on the capacitors of integrator circuits attached to the A-units; therefore, the value associated with the A-unit $N^1_{i_1}$ was a shared weight amongst all the physical links outgoing from it. The vector $\mathbf{w}^2$ represented parameters that could change according to a *reinforcement mechanism*; i.e., a set of rules to modify the system parameters. Let $\mathbf{x}^0 \in X^0$ be a pattern that can be represented by the S-units in $\mathcal{N}^0$ (in the original perceptron, these units were implemented as photocells of a raster screen which could take on binary values). Let $[t_0, t_2)$ denote a given time interval of length $\Delta t = t_2 - t_0$, and let $t_1 \in (t_0, t_2)$ so that $\eta = t_2 - t_1 > 0$. Each pattern $\mathbf{x}^0$ was presented to the perceptron for a total time Δt. During the first part of the interval, $[t_0, t_1)$, the perceptron determined the index $i_2 \in \{1, 2\}$ of the R-unit with greatest score

$$s^2_{i_2} = \mathbf{x}^1(\mathbf{w}^2 \odot \mathbf{a}^{1,2}_{i_2}) + b^2_{i_2} ;$$

during the second part of the interval, $[t_1, t_2)$, the perceptron forced the state of the *loser unit* to zero (even if its score surpassed the threshold), and updated the values of the A-units linked to the *winner unit* according to the following set of rules:

$$\begin{aligned} \Delta\mathbf{w}^2 &= \int_{t_1}^{t_2} \mathbf{x}^1 \odot A^2_{i_2} dt && (3.17)\\ &= \eta \mathbf{x}^1 \odot \mathbf{a}^{1,2}_{i_2} , && (3.18)\\ \mathbf{w}^2(t + \Delta t) &= \mathbf{w}^2(t) + \Delta\mathbf{w}^2 . \end{aligned}$$

Since $\eta \coloneqq t_2 - t_1$ was kept fixed for each pattern, the updates (3.17) were discrete: as $\mathbf{W}^1$, also $\mathbf{W}^2$ took values in some discrete set. The initialisation strategy for the ternary matrix $\mathbf{W}^1$ had been carefully designed, but its components were kept fixed over the course of the learning process. Similarly, the biases $\mathbf{b}^1, \mathbf{b}^2$ of the perceptron were not learnable parameters. Note that the learning process was such to amplify a binary discrimination inherent to the system. In fact, due to the random nature of the connections encoded by $\mathbf{W}^1$ and $\mathbf{A}^{1,2}$, the response units tended to respond differently to different representations $\mathbf{x}^1$ from the very beginning. Moreover, the reinforcement (3.20) required no actions from an external *teacher*, making the perceptron

an example of *unsupervised learning* or *self-organisation*. In particular, this learning rule is an example of *Hebbian learning*:

> 'When an axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.' [31]

In the early 60's, Rosenblatt had developed a more mature theory of perceptrons, and he described a version of the system capable of supervised learning [32, 33]. He used the term *perceptron* to refer to a vast class of lerning systems, including amongst others the unsupervised learning system presented above and what is currently known as perceptron, which he called **simple perceptron** or **error-corrected α-perceptron**. This specific perceptron was designed to solve binary classification, and consisted of a single R-unit connected to all the A-units. This unit was a LTU computing

$$\begin{aligned} x^2 &= H_0^{\{-1,+1\}}(\mathbf{x}^1\mathbf{w}^2 + b^2) \\ &= H_{-b^2}^{\{-1,+1\}}(\mathbf{x}^1\mathbf{w}^2)\,, \end{aligned}$$

where

$$H_\theta^{\{q_0,q_1\}}(x) = \begin{cases} q_0, & \text{if } x < \theta \\ q_1, & \text{if } x \geq \theta \end{cases} \tag{3.19}$$

denotes the **generalised Heaviside function**. This activation function was chosen to return responses in the binary set $Y = \{-1, +1\}$. The system required a human operator (or an equivalent *teacher*) to compare the output x^2 of the R-unit with a desired response $y \in \{-1, +1\}$. The update was then computed according to the following rule:

$$\begin{aligned} \Delta\mathbf{w}^2 &= \begin{cases} 0, & \text{if } x^2 y \geq 1 \\ \eta\mathbf{x}^1, & \text{if } x^2 y < 1 \text{ and } y = 1 \\ -\eta\mathbf{x}^1, & \text{if } x^2 y < 1 \text{ and } y = -1 \end{cases} \\ &= -\eta\mathbf{x}^1(x^2 - y)\,. \end{aligned} \tag{3.20}$$

The number $\eta > 0$ was chosen at design time, and determined the size of the update (i.e., what we could now call a *learning rate*). Note that the update to the values $\mathbf{w}^2$ of the A-units happened only if the response of the R-unit was incorrect, and no correction was applied in case of a correct response; hence the name *error-correcting reinforcement*. Note also that the updates, and therefore the components of $\mathbf{w}^2$, could take on negative values. We remark that Rosenblatt considered what we now call a perceptron just a

very particular case of his more comprehensive theory. Convergence proofs concerning this *perceptron algorithm* in the case of linearly separable sets were immediately developed [34].

In 1960, almost contemporarily to the introduction of the simple perceptron, Bernard Widrow and Ted Hoff developed a system called **ADAptive LInear NEuron** (ADALINE) [35]. The goal of this system was to discover functional relationships between spaces

$$X^0 = \{-1, +1\}^{n_0}\,,$$
$$X^1 = \{-1, +1\}\,,$$

interpreting the learning problem as the search for a switching circuit with n_0 binary inputs and a single binary output. The idea was using the statistics contained in available input/output pairs (a labelled data set (1.44)) to derive the truth table of a circuit which could not be analytically designed. Their network included an input layer $\mathcal{N}^0$ and an output layer $\mathcal{N}^1$ consisting of a single unit (the ADALINE neuron) computing

$$x^1 = H_0^{\{-1,+1\}}(\mathbf{x}^0\mathbf{w}^1 + b^1)\,. \tag{3.21}$$

Similar to the supervised perceptron, the goal of ADALINE's learning algorithm was to perform binary classification in the set $Y = \{-1, +1\}$. Given the electrical engineering context, the output was to be determined by a thresholding operation over an electric potential: this operation was called *quantization*, and the generalised Heaviside was referred to as the *quantizer*. The main difference with respect to the perceptron was that the Heaviside activation function was not implemented as a physical device during the learning phase. Instead, it was computed by a human operator who checked the output of a voltmeter. The learning rule of ADALINE was developed as follows. Given an input pattern $\mathbf{x}^0$, the adjusted score

$$\tilde{s}^1 = \mathbf{x}^0\mathbf{w}^1 + b^1$$

physically corresponded to a voltage, yielding an output $H_0^{\{-1,+1\}}(\tilde{s}^1)$. A human operator (the *boss*, i.e., the *teacher*) would have known that setting $\tilde{s}^1$ close to a desired voltage y could yield the correct system output $H_0^{\{-1,+1\}}(y)$. The problem was therefore transformed into a regression of the correct electric potential value. This problem could be solved modifying both the weights $\mathbf{w}^1$ and the bias b^1 to minimise the distance $|\tilde{s}^1 - y|$, where $y \in \mathbb{R}$ was the reference voltage that the input $\mathbf{x}^0$ should have generated. Widrow and Hoff proposed to minimise this distance by using gradient descent over the function

$$d(\tilde{s}^1, y) \coloneqq \frac{1}{2}(\tilde{s}^1 - y)^2\,. \tag{3.22}$$

The updates were computed as

$$\Delta \mathbf{w}^1 = -\eta \nabla_{\mathbf{w}^1} d(\tilde{s}^1, y)\,,$$
$$\Delta b^1 = -\eta \frac{\partial}{\partial b^1} d(\tilde{s}^1, y)\,,$$

The number $\eta > 0$ was chosen so to obtain $d(\tilde{s}^1, y) = 0$ after the update: since the parameters were physically implemented as potentiometers, this implied that they could take on values along a continuous spectrum. Note that in ADALINE also the bias b^1 was a learnable parameter. Widrow and Hoff motivated the replacement of a binary classification problem with a regression problem on the basis of the fact that solving the regression problem induces convergence also for the classification problem [36].

From a modelling perspective, both the perceptron and ADALINE were limited to learn only linearly separable bisections of the input space X^0 (a bisection is a partition of X^0 in two disjoint sets). The distinction between the two methods lies in that Rosenblatt's perceptron performed classification upon representations $\mathbf{x}^1$ of the inputs $\mathbf{x}^0$ instead of performing it on the raw inputs themselves. Both the R-units and the A-units shared a LTU structure (3.16), with the difference that the parameters $\mathbf{W}^1$ of the links between S-units and A-units could not be learnt. The A-units played the role of what we would now call *feature extractors*; i.e., functions capable of creating abstract representations of their inputs. In the early 60's, Augusto Gamba and his collaborators published a series of papers describing a system similar to the perceptron which they named **Programmatore/Analizzatore Probabilistico Automatico** (PAPA, Automatic Programmer/Probabilistic Analyzer) [37, 38]. They explicitly proposed to stack multiple layers of LTUs,

$$\begin{aligned} \mathbf{x}^1 &= H(\mathbf{x}^0\mathbf{W}^1 + \mathbf{b}^1)\,, \\ \mathbf{x}^2 &= H(\mathbf{x}^1\mathbf{W}^2 + \mathbf{b}^2)\,, \\ &\dots \\ \mathbf{x}^L &= H(\mathbf{x}^{L-1}\mathbf{W}^L - \mathbf{b}^L)\,, \end{aligned} \tag{3.23}$$

to allow the system to determine relevant features autonomously:

> 'In self-learning one leaves PAPA to find an error-free classification, that corresponds obviously to a higher intelligence term. The drawback is that the classification that is eventually reached is generally an irrelevant one from the point of view of a human observer.' [39]

These multi-layer perceptrons were soon recognised to have a richer program space than that of standard perceptrons (for instance, they can learn the

non-linearly separable XNOR logic function), but it was also recognised that they would have behaved like *black-boxes*; moreover, no learning algorithm was proposed to train such learning systems.

The choice of Heaviside functions (3.15) and (3.19) as activation functions created a natural link between network-based learning and Boolean logic, allowing an interpretation of ANNs in terms of mathematical logic. Remarkably, we note that the parameters used by these first ANNs models often took values in discrete spaces. We could say that these models were *quantized neural networks*. In 1969, Marvin Minsky and Seymour Papert published a controversial book on perceptrons [40], which investigated Rosenblatt's simple perceptrons more thoroughly. They also analysed multi-layer perceptrons as the *Gamba machines* (3.23), recognising that such systems could compute complex predicates, but they observed that some concepts (such as the connectedness of a plain figure) could not be computed just by stacking layers of LTUs. This critique has been often misunderstood as the impossibility, for multi-layer perceptrons, to learn complex functions, but it is likely that the real concern of Minsky and Papert was the more practical limitation implied by the exponential explosion in the number of units required to learn specific concepts. The lack of practical algorithms to train multi-layer perceptrons and the realisation that systems parametrised by a very high number of parameters were likely to have multiple local minima (hampering the possibility to reach the global minimum) [41] reflected a growing awareness of the scientific community about the complexities of the learning problem associated with ANNs.

This *crysis* induced a shift in the applications of ANNs from symbolic logic to non-linear regression. In 1974, Paul Werbos replaced quantized functions (3.15) and (3.19) with differentiable activation functions [42]. Important examples of differentiable activation functions are the sigmoid

$$\begin{aligned} sigmoid \,:\, \mathbb{R} &\to (0,1) \\ x &\mapsto \frac{e^x}{e^x+1}\,, \end{aligned} \tag{3.24}$$

and the hyperbolic tangent

$$\begin{aligned} tanh \,:\, \mathbb{R} &\to (-1,1) \\ x &\mapsto \frac{e^x-1}{e^x+1}\,. \end{aligned} \tag{3.25}$$

This replacement allowed deriving gradient-based learning rules (like ADALINE's delta rule (3.23)) even for multi-layer ANNs. ANNs were since interpreted as parametric differentiable approximators of non-linear functions.

The idea of quantized activations disappeared from mainstream research, replaced by that of differentiable activation functions. In 1986, David Rumelhart, Geoffrey Hinton and Ronald Williams popularised backpropagation by showing that ANNs trained using this algorithm encoded useful representations of their inputs [43]. Backpropagation has since become the workhorse of ANNs research.

3.3 Properties of ANNs

Automating a given task T is a difficult problem. The *divide et impera* (divide and conquer) principle suggests to decompose it into a set of simpler sub-tasks $\mathrm{T} = \{\tau^{(1)}, \tau^{(2)}, \ldots, \tau^{(N)}\}$. Then we must model proper input and output domains X, Y for each task $\tau \in \mathrm{T}$ and define a function

$$\begin{aligned} g \,:\, & X \to Y \\ & x \mapsto g(x)\,. \end{aligned} \tag{3.26}$$

If the available knowledge about g is sufficient, it is possible to derive an algorithm α_g which implements g. In this case, we will say that *an explicit algorithm is available for* g. For example, if X denotes the space of all finite sequences of integers and Y denotes the space of all finite ordered sequences of integers, the function g that maps every sequence $x \in X$ into its sorted version $g(x) \in Y$ is called the *sorting function*. Many algorithms α_g are available that implement such a function [17].

Other times, the knowledge of g is insufficient to derive an explicit algorithm α_g. If samples of its input/output pairs are available (for example, in the form of a labelled data set (1.44)), it is reasonable to use a machine learning system whose program space contains an approximation of g. If we have some information about the properties that g should satisfy, then it is important to know which are the limitations of this program space. Scientific experience suggests that many real-world phenomena can be described by functions that satisfy certain regularity conditions. For example, $g \in C^0(X)$ (i.e., g is continuous over its domain). Other applications might enforce differentiability requirements like $g \in C^1(X)$ or $g \in C^2(X)$ (e.g., when g models the movements of a robot). From a modelling perspective, ANNs are appealing since they can approximate large function spaces. To give a more precise explanation, we need an additional definition. Let $X^0 \subset \mathbb{R}^{n_0}$ and $X^2 = \mathbb{R}$. Let $n_1 \in \mathbb{N}$ be a positive integer and define an simple feedforward neural network that computes the following functoin:

$$\Phi(n_1, \hat{\mathbf{m}}, \mathbf{x}^0) = \mathbf{w}^2 \sigma(\mathbf{x}^0 \mathbf{W}^1 + \mathbf{b}^1)\,, \tag{3.27}$$

where $\mathbf{W}^1 \in \mathcal{M}^{n_0 \times n_1}(\mathbb{R}), \mathbf{b}^1 \in \mathcal{M}^{1 \times n_1}(\mathbb{R}), \mathbf{w}^2 = \mathcal{M}^{n_1 \times 1}(\mathbb{R}), b^2 = 0$ and $\hat{\mathbf{m}} = (\mathbf{m}^1, \mathbf{m}^2) = ((\mathbf{W}^1, \mathbf{b}^1), (\mathbf{w}^2, 0))$ is the parameters tuple. These networks have an input layer $\mathcal{N}^0$ with n_0 units, a hidden layer $\mathcal{N}^1$ with n_1 units whose activation function is σ, and an output layer $\mathcal{N}^2$ consisting of a single unit whose activation function is the identity function

$$\begin{aligned} id \,:\, \mathbb{R} &\to \mathbb{R} \\ x &\mapsto x \,, \end{aligned} \tag{3.28}$$

also called the **linear activation function**. The restrictions of (3.4) to pairs of adjacent layers form complete bipartite graphs; we say that the layers of these pairs are **fully-connected**. The program space of this ANN is

$$M(n_1) \coloneqq \{\Phi(\hat{\mathbf{m}}, \cdot) \,:\, X^0 \to \mathbb{R}\} \,.$$

We define

$$M \coloneqq \bigcup_{n_1 \in \mathbb{N}} M(n_1) \tag{3.29}$$

to be the **global program space** of all 2-layers simple feedforward neural networks with fully-connected layers. In 1989, George Cybenko derived an important approximation result about this global program space, when it uses the sigmoid (3.24) (or an equivalent *sigmoidal* activation function for the neurons of the hidden layer [44].

Theorem 1 (**Universal Approximation Theorem** (UAT))**.** *Let $X^0 = [0,1]^{n_0} \subset \mathbb{R}^{n_0}$ be the n_0-dimensional hypercube in $\mathbb{R}^{n_0}$ and denote by $G = C^0(X^0)$ the space of real-valued continuous functions over X^0. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous function satisfying*

$$\sigma(x) \to \begin{cases} 0, & \text{as } x \to -\infty \\ 1, & \text{as } x \to +\infty \,, \end{cases} \tag{3.30}$$

*called a **sigmoidal** function. For any given $g \in G$ and any given $\epsilon > 0$, there exists an integer $n_1 \in \mathbb{N}$ and a network (3.27) such that $\|g - \Phi(n_1, \hat{\mathbf{m}},)\|_\infty < \epsilon$, where $\|g\|_\infty \coloneqq \sup_{\mathbf{x}^0 \in X^0}\{g(\mathbf{x}^0)\}$.*

More general results were derived by Kurt Hornik in 1991 [45].

Theorem 2. *Let $X^0 \subset \mathbb{R}^{n_0}$ be a compact subset of $\mathbb{R}^{n_0}$ and denote by $G = C^0(X^0)$ the space of real-valued continuous functions over X^0. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, bounded and non-constant function. For any given $g \in G$ and any given $\epsilon > 0$, there exists an integer $n_1 \in \mathbb{N}$ and a network (3.27) such that $\|g - \Phi(n_1, \hat{\mathbf{m}})\|_\infty < \epsilon$, where $\|g\|_\infty \coloneqq \sup_{\mathbf{x}^0 \in X^0}\{g(\mathbf{x}^0)\}$.*

Cybenko had required the activation functions to satisfy the sigmoidal property (3.30), whereas Hornik showed that this constraint was not necessary to approximate continuous functions.

Theorem 3. *Let $X^0 \subset \mathbb{R}^{n_0}$ be a compact subset of $\mathbb{R}^{n_0}$. Let μ be a finite measure over $\mathbb{R}^{n_0}$ (e.g., a probability measure). Denote by $G = L^p(\mu)$ the space of real-valued measurable functions over X^0 for which*

$$\|g\|_p \coloneqq \left[\int_{\mathbb{R}^{n_0}} |g(\mathbf{x}^0)|^p \mu(d\mathbf{x}^0)\right]^{\frac{1}{p}} \tag{3.31}$$

is finite. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a bounded and non-constant function. For any given $g \in G$ and any given $\epsilon > 0$, there exists an integer $n_1 \in \mathbb{N}$ and a network (3.27) *such that $\|g - \Phi(n_1, \hat{\mathbf{m}})\|_p < \epsilon$.*

In particular, the closeness between a target measurable function and a simple fully-connected ANN is guaranteed in measure, even when the activation function is discontinuous (and therefore not differentiable). As a corollary, this result holds in the case where σ is the Heaviside (3.15) or the generalised Heaviside (3.19). Nevertheless, the parameters $\hat{\mathbf{m}}$ are still required to take on continuous values. Note that these theorems hold for a very simple subset of the space of all possible ANNs, which is therefore at least as rich. Let M denote the program space of a given machine learnign system. Let G denote the functions space in which the target map (3.26) is supposed to live. Approximation theorems are results about the density of Π in G, with respect to some given metric on G. From an application perspective, the limitation of this results is that the classes M are usually arbitrarily large, since no constraint is enforced on the size program space. This is an unrealistic assumptions for real-world systems, since physical computers have limited memory. For this reason, a small subset $M' \subset M$ of the global program space is usually defined. The problem of selecting a good M' is known as **topology selection**. In Section 4.2 we will briefly summarise how topology selection has evolved for the sub-class of ANNs knowns as *convolutional neural networks*.

Suppose the program space (3.13) of an ANN has been chosen, and denote it by M. According to the definition of machine learning system we gave in Section 1.3, we need to define a learning algorithm α to select the best program out of M. Let $\mathfrak{D}$ be a given labelled data set (1.44). To simplify the discussion we make the following assumptions:

(i) the ANN is a simple feedforward neural network;

(ii) the activation functions $\sigma_{i_\ell}^{\ell}$ are all equal except fot those of the last layer ($\sigma_{i_L}^{L}$), which are allowed to be the identity function (3.28);

(iii) the first representations space of the networks, X^0, and the data set input space X coincide;

(iv) the last representations space of the network, X^L, and the data set labels space Y coincide.

The parameters space $W^\ell \times B^\ell$ associated with the ℓ-th layer is therefore $\mathbb{R}^{(n_{\ell-1}+1)\times n_\ell}$; consequently, the overall parameters space is $\mathbb{R}^{\sum_{\ell=1}^{L}((n_{\ell-1}+1)\times n_\ell)}$. Denote by μ the probability measure according to which $\mathfrak{D}$ has been sampled. A natural way to evaluate the quality of the programs of an ANN is to compute the **loss functional** [46]:

$$\mathcal{L}(\Phi(\hat{\mathbf{m}})) \coloneqq \iint_{X\times Y} d(\Phi(\hat{\mathbf{m}},\mathbf{x}),\mathbf{y})\mu(d\mathbf{x},d\mathbf{y})\,. \tag{3.32}$$

Observe that, when the topology of Φ is fixed, the loss functional is actually a function of the parameters $\hat{\mathbf{m}}$. Here,

$$\begin{aligned} d\,: X^L \times Y &\mapsto [0,+\infty) \\ (\mathbf{x}^L,\mathbf{y}) &\to d(\mathbf{x}^L,\mathbf{y}) \end{aligned} \tag{3.33}$$

is a **loss function** which is non-negative, satisfies the *no-error, zero-loss* property

$$\mathbf{x}^L = \mathbf{y} \implies d(\mathbf{x}^L,\mathbf{y}) = 0\,, \tag{3.34}$$

and is usually differentiable and convex [47]. When μ admits a disintegration of function form $\{\mu_{\mathcal{X}}, \{\delta_{g(\mathbf{x})}\}_{\mathbf{x}\in X}\}$, the loss functional takes the simpler form

$$\mathcal{L}(\Phi(\hat{\mathbf{m}})) \coloneqq \int_X d(\Phi(\hat{\mathbf{m}},\mathbf{x}),g(\mathbf{x}))\mu_{\mathcal{X}}(d\mathbf{x})\,. \tag{3.35}$$

The ideal solution would occur when $\Phi(\hat{\mathbf{m}},\mathbf{x}) = g(\mathbf{x})$, $\forall \mathbf{x} \in X$. This is equivalent to requiring that g belongs to the program space M. Since we usually do not know the space G in which g lives, there is no way to know whether this is a reasonable expectation. Therefore, the guiding principle for a supervised learning algorithm is to find either a global or a local minimum of (3.32). More in general, there is no way to know μ; as we explained is Section 1.2, the only access we have to μ is through data sets (1.41) and (1.44) and their associated empirical measures (1.43). When we plug the empirical measure in (3.32) in place of the real measure μ, the loss functional is turned into a finite sum

$$\mathcal{L}(\Phi(\hat{\mathbf{m}})) = \frac{1}{N}\sum_{i=1}^{N} d(\Phi(\hat{\mathbf{m}},\mathbf{x}^{(i)}),\mathbf{y}^{(i)})\,, \tag{3.36}$$

called the **empirical loss**. Consider an ANN (3.14) composed of layer maps (3.12) whose activation functions σ are differentiable (e.g., (3.24), (3.25) or (3.28)). Since each layer is differentiable, $\Phi(\hat{\mathbf{m}}, \cdot)$ is differentiable with respect to its parameters $\hat{\mathbf{m}}$. Together with the differentiability of the loss function (3.33), this implies that the empirical loss is a differentiable function of $\hat{\mathbf{m}}$. It is possible to compute a gradient-based **error signal**

$$\epsilon_{\hat{\mathbf{m}}} \coloneqq \nabla_{\hat{\mathbf{m}}} \mathcal{L}(\Phi(\hat{\mathbf{m}})) \tag{3.37}$$

directed towards the ANN's parameter. This error signal can then be taken in input by some **gradient descent** (GD) algorithm [48] designed to minimise (3.36) as a function of $\hat{\mathbf{m}}$. The learning problem is therefore interpreted as a numerical optimisation problem.

The computation of the error signal (3.37) is not a trivial task. The typical approaches to compute the gradient of (i.e., to *differentiate*) a function Φ are:

(i) **numerical differentiation**: it is built on top of the definition of *directional derivative*,

$$\frac{d}{d\mathbf{u}}\Phi(\hat{\mathbf{m}}) \coloneqq \lim_{h \to 0} \frac{\Phi(\hat{\mathbf{m}} + h\mathbf{u}) - \Phi(\hat{\mathbf{m}})}{h},$$

and should be used when no analytic expression for the derivative is available; since exhaustive search over all the possible values of $\mathbf{u}$ and h is unfeasible, we must sample arbitrary directions $\mathbf{u}$ and arbitrary arc lengths h to obtain an estimate of the real gradient; it is likely to be imprecise when the argument $\hat{\mathbf{m}}$ lives in a high-dimensional space;

(ii) **symbolic differentiation**: the input function $\Phi(\hat{\mathbf{m}})$ is described as a sequence of operations on simpler functions (e.g., sums, products, compositions), and this expression is analysed to derive a program that implements the analytic gradient of $\Phi(\hat{\mathbf{m}})$; this resulting program must then be evaluated;

(iii) **automatic differentiation**: the input function $\Phi(\hat{\mathbf{m}})$ is described as a graph of function blocks, and the derivative is computed applying the chain rule only; each function block must be programmed to compute its own Jacobians, one for each of its inputs, so that computing the derivative amounts to a composition of Jacobians; with respect to symbolic differentiation, this approach has the advantage of being local: the blocks can be executed without waiting that a global formula for the gradient is available.

We will now analyse automatic differentiation more in-depth. The forward operation of each layer is a composition of a linear map

$$\mathbf{s}^\ell = \mathbf{x}^{\ell-1}\mathbf{W}^\ell\,, \tag{3.38}$$

a translation

$$\tilde{\mathbf{s}}^\ell = \mathbf{s}^\ell + \mathbf{b}^\ell \tag{3.39}$$

and an element-wise non-linear distortion

$$\mathbf{x}^\ell = \sigma(\tilde{\mathbf{s}}^\ell)\,. \tag{3.40}$$

The first function has two inputs ($\mathbf{x}^{\ell-1}$ and $\mathbf{W}^\ell$) and the following Jacobians

$$\begin{aligned} J_{\mathbf{x}^{\ell-1}}(\mathbf{s}^\ell) &= \left(\frac{\partial s_k^\ell}{\partial x_i^{\ell-1}}\right) \\ &= \left(w_{ki}^\ell\right) \\ &= \mathbf{W}^{\ell T} \\ J_{\mathbf{W}^\ell}(\mathbf{s}^\ell) &= \left(\frac{\partial s_k^\ell}{\partial w_{ij}^\ell}\right) \\ &= \left(\tau_{kij}\right) \\ &= \left(\delta_k^j x_i\right)\,; \end{aligned} \tag{3.41}$$

the second function has two inputs ($\mathbf{s}^\ell$ and $\mathbf{b}^\ell$) and the following Jacobians

$$\begin{aligned} J_{\mathbf{s}^\ell}(\tilde{\mathbf{s}}^\ell) &= \left(\frac{\partial \tilde{s}_k^\ell}{s_i^\ell}\right) \\ &= \left(\delta_k^i\right) \\ &= I_{n_\ell} \\ J_{\mathbf{b}^\ell}(\tilde{\mathbf{s}}^\ell) &= \left(\frac{\partial \tilde{s}_k^\ell}{b_i^\ell}\right) \\ &= \left(\delta_k^i\right) \\ &= I_{n_\ell}\,; \end{aligned} \tag{3.42}$$

the third function has one input ($\tilde{\mathbf{s}}^\ell$) and the following Jacobian

$$J_{\tilde{\mathbf{s}}^\ell}(\mathbf{x}^\ell) = \left(\frac{\partial x_k^\ell}{\partial \tilde{s}_i^\ell}\right) \tag{3.43}$$

$$= \left(\delta_k^i \sigma'(\tilde{s}_i^\ell)\right)\,. \tag{3.44}$$

Let d denote a loss function, and denote by

$$\begin{aligned} J_{\mathbf{x}^L}(d) &\coloneqq \nabla_{\mathbf{x}^L}(d) \\ &= \left(\tfrac{\partial d}{\partial x_1^L}, \tfrac{\partial d}{\partial x_2^L}, \cdots, \tfrac{\partial d}{\partial x_{n_L}^L} \right) \end{aligned}$$

its one-dimensional Jacobian (i.e., its gradient). Suppose we want to compute the error signals directed towards the weights $\mathbf{W}^1$. This can be done performing tensor contraction operations between successive Jacobians:

$$\begin{aligned} J_{\mathbf{W}^1}(\tilde{\mathbf{s}}^1) &= J_{\mathbf{s}^1}(\tilde{s}^1) J_{\mathbf{W}^1}(\mathbf{s}^1) \\ J_{\mathbf{W}^1}(\mathbf{x}^1) &= J_{\tilde{\mathbf{s}}^1}(\mathbf{x}^1) J_{\mathbf{W}^1}(\tilde{\mathbf{s}}^1) \\ J_{\mathbf{W}^1}(\mathbf{s}^2) &= J_{\mathbf{x}^1}(\mathbf{s}^2) J_{\mathbf{W}^1}(\mathbf{x}^1) \\ J_{\mathbf{W}^1}(\tilde{\mathbf{s}}^2) &= J_{\mathbf{s}^2}(\tilde{\mathbf{s}}^2) J_{\mathbf{W}^1}(\mathbf{s}^2) \\ J_{\mathbf{W}^1}(\mathbf{x}^2) &= J_{\tilde{\mathbf{s}}^2}(\mathbf{x}^2) J_{\mathbf{W}^1}(\tilde{\mathbf{s}}^2) \\ J_{\mathbf{W}^1}(d) &= J_{\mathbf{x}^2}(d) J_{\mathbf{W}^1}(\mathbf{x}^2) \,. \end{aligned} \tag{3.45}$$

Suppose now we want to compute the error signals directed towards the bias vector $\mathbf{b}^1$. Again, this can be done performing tensor contraction operations between successive Jacobians:

$$\begin{aligned} J_{\mathbf{b}^1}(\mathbf{x}^1) &= J_{\tilde{\mathbf{s}}^1}(\mathbf{x}^1) J_{\mathbf{b}^1}(\tilde{\mathbf{s}}^1) \\ J_{\mathbf{b}^1}(\mathbf{s}^2) &= J_{\mathbf{x}^1}(\mathbf{s}^2) J_{\mathbf{b}^1}(\mathbf{x}^1) \\ J_{\mathbf{b}^1}(\tilde{\mathbf{s}}^2) &= J_{\mathbf{s}^2}(\tilde{\mathbf{s}}^2) J_{\mathbf{b}^1}(\mathbf{s}^2) \\ J_{\mathbf{b}^1}(\mathbf{x}^2) &= J_{\tilde{\mathbf{s}}^2}(\mathbf{x}^2) J_{\mathbf{b}^1}(\tilde{\mathbf{s}}^2) \\ J_{\mathbf{b}^1}(d) &= J_{\mathbf{x}^2}(d) J_{\mathbf{b}^1}(\mathbf{x}^2) \,. \end{aligned} \tag{3.46}$$

If we now expand (3.45) and (3.46),

$$\begin{aligned} J_{\mathbf{W}^1}(d) &= J_{\mathbf{x}^2}(d) J_{\tilde{\mathbf{s}}^2}(\mathbf{x}^2) J_{\mathbf{s}^2}(\tilde{\mathbf{s}}^2) J_{\mathbf{x}^1}(\mathbf{s}^2) J_{\tilde{\mathbf{s}}^1}(\mathbf{x}^1) J_{\mathbf{s}^1}(\tilde{s}^1) J_{\mathbf{W}^1}(\mathbf{s}^1) \,, \\ J_{\mathbf{b}^1}(d) &= J_{\mathbf{x}^2}(d) J_{\tilde{\mathbf{s}}^2}(\mathbf{x}^2) J_{\mathbf{s}^2}(\tilde{\mathbf{s}}^2) J_{\mathbf{x}^1}(\mathbf{s}^2) J_{\tilde{\mathbf{s}}^1}(\mathbf{x}^1) J_{\mathbf{b}^1}(\tilde{\mathbf{s}}^1) \,, \end{aligned}$$

we observe that much of the computation is shared amongst the two evaluations. Since the tensor contraction operation is both left-associative and right-associative, the output of the right-to-left operations chain is the same as the output of the left-to-right operations chain. By computing $J_{\mathbf{W}^1}(d)$ and $J_{\mathbf{b}^1}(d)$ from right-to-left, in what is known as **forward-mode automatic differentiation**, we need to compute the intermediate results multiple times, due to the difference between the *radices* $J_{\mathbf{W}^1}(\tilde{\mathbf{s}}^1)$ and $J_{\mathbf{b}^1}(\tilde{\mathbf{s}}^1)$. We can perform a more efficient evaluation of derivatives by computing tensor contractions left-to-right to reuse intermediate results, in what is known

as **reverse-mode automatic differentiation**. In the above example, this would result in the following sequence of shared computations:

$$\begin{aligned} J_{\tilde{\mathbf{s}}^2}(d) &= J_{\mathbf{x}^2}(d) J_{\tilde{\mathbf{s}}^2}(\mathbf{x}^2) \\ J_{\mathbf{s}^2}(d) &= J_{\tilde{\mathbf{s}}^2}(d) J_{\mathbf{s}^2}(\tilde{\mathbf{s}}^2) \\ J_{\mathbf{x}^1}(d) &= J_{\mathbf{s}^2}(d) J_{\mathbf{x}^1}(\mathbf{s}^2) \\ J_{\tilde{\mathbf{s}}^1}(d) &= J_{\mathbf{x}^1}(d) J_{\tilde{\mathbf{s}}^1}(\mathbf{x}^1) \,. \end{aligned} \tag{3.47}$$

This algorithm for computing gradients is also called **backpropagation** [43, 49]. For additional details about the implementation of backpropagation see, for example, [50]. Once we have computed the gradient (3.37), a learning rule must be defined. Typical learning rules are variant of the steepest gradient descent algorithm that try avoid getting trapped in the many local minima that characterize an ANN's parameters space [51]. The most popular learning rules are **GD with momentum** [52], **Root Mean Square propagation (RMSprop)** [53] and **ADAptive Moment estimation (ADAM)** [54]. These algorithms modify the gradient computed using the backpropagation algorithm to incorporate information about previous iterations, returning an **update direction**

$$\mathbf{u} = \gamma(\epsilon_{\hat{\mathbf{m}}}) \,. \tag{3.48}$$

We derive a learning rule introducing a positive *step-length* parameter

$$\eta > 0 \,, \tag{3.49}$$

which is known in machine learning literature as the **learning rate**. The update to the parameters is computed as

$$\Delta\hat{\mathbf{m}} = -\eta \mathbf{u} \,. \tag{3.50}$$

From a computational perspective, the operations performed by the neurons of an ANN are extremely parallelisable and data-independent:

(i) the score computation $s_{i_\ell}^\ell = \sum_{i_{\ell-1}=1}^{n_\ell - 1} r_{i_{\ell-1} i_\ell}^\ell$, where $r_{i_{\ell-1} i_\ell}^\ell = x_{i_{\ell-1}}^{\ell-1} w_{i_{\ell-1} i_\ell}^\ell$, is the composition of a *map pattern* operation (as the one depicted in Figure 1.6) and a *reduce pattern* operation (as the one depicted in Figure 1.7); in particular, the time cost of this operation can be reduced from $2n_{\ell-1} - 1$ to $1 + \lceil \log_2(n^{\ell-1}) \rceil$, where $\lceil \cdot \rceil$ is the integer rounding up operation and the base of the logarithm is the arity of the addition operation;

(ii) the adjusted score and non-linear distortion operation are sequential, as depicted by a single thread in Figure 1.6;

(iii) theoretically, every neuron $N_{i_\ell}^\ell \in \mathcal{N}^\ell$ could perform these operations in parallel, reducing the time complexity of computing a layer map from $n_\ell(2n_{\ell-1}+1)$ to $1+\lceil \log_2(n^{\ell-1})\rceil+2$, which is independent of the number n_ℓ of units in the layer.

A second level of algorithmic parallelisation can be exposed if we consider *computational graphs*. At the time of writing, the most popular software frameworks to train and study ANNs are TensorFlow [55] and PyTorch [56]. Both frameworks Python [57] front-ends and C/C++ [21] back-ends designed according to the **dataflow** programming model [58, 59]. The dataflow programming model interprets programs as directed **computational graphs**

$$(V, E)\,, \tag{3.51}$$

where the vertices $v \in V$ represent operations and edges $(i, j) \in E$ represent operands communications (i.e., dependencies amongst operations). We remark that the computational graph (3.51) is conceptually different from the network graph (3.4); this distinction is important, since the related ambiguity is a cause of common mistakes. A computational graph (3.51) is called a **differentiable graph** if all the operations $v \in V$ are differentiable with respect to their operands. For a given differentiable graph (3.51), we can define a second computational graph

$$(\dot{V}, \dot{E} \coloneqq \{(j, i) \,|\, (i, j) \in E\})\,, \tag{3.52}$$

where the dependencies between operations are reversed. In this case, we will refer to (3.51) as the **forward computational graph** and to (3.52) as the **backward computational graph**. Examples of differentiable operations are tensor contractions (3.38), tensor sums (3.39) and differentiable activation functions (3.40). The forward and backward computational graphs associated with the networks of Figure 3.1 are depicted in Figure 3.2, whereas details are shown in Figure 3.3 and Figure 3.4.

3.4 Regularisation algorithms

At the time of writing, it has been accepted that finding the global optimum of the empirical loss (3.36) is both impractical and possibly not optimum with respect to the real measure μ according to which the data is distributed. Therefore, learning algorithms should find *good local minima*. A conventional measure of the quality of a local minimum is its **generalisation capability**. By this term, we mean the quality of its performance on **out-of-sample** points

$$\mathbf{x} \in X \,|\, \nexists (\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \in \mathfrak{D}\,, \mathbf{x} = \mathbf{x}^{(i)}\,.$$

A *good solution* $\Phi(\hat{\mathbf{m}}^*)$ is not necessarily one that minimises the loss functional

$$\mathcal{L}(\Phi(\hat{\mathbf{m}}^*)) \coloneqq \iint_{X\times Y\setminus\{(\mathbf{x}^{(i)},\mathbf{y}^{(i)})\in\mathfrak{D})\}} d(\Phi(\hat{\mathbf{m}}^*,\mathbf{x}),\mathbf{y})\mu(d\mathbf{x},d\mathbf{y})$$

outside of the data set. The ultimate performance metric depends on the application, and usually it can not be measured using a loss function. Nevertheless, the choice of the loss function influences the shape of the optimisation landscape [47]. It is a common practice to express (3.33) as the sum of different *penalty terms*

$$d = d_1 + d_2 + \cdots + d_{N_D}$$

intended to communicate to the ANN the different constraints of the problem that it has been designed to solve. The problem of choosing a suitable loss function is therefore as important as the problem of choosing the correct topology. In Section 4.3 we will describe an example reasoning behind these choices. But these problems are determined on a case-by-case manner, since they depend on the specific problem and data set under investigation. We are therefore interested in definitions of good minima that involve only the concept of loss function.

The minimisation of the empirical loss (3.36) is a complex problem. To better understand why, we need to look more carefully at the loss functional (3.32). Suppose that the measures μ over $\mathcal{X}\mathcal{Y}$ is a simple Dirac's delta concentrated at the point $(\mathbf{x},\mathbf{y})$. We define the **loss surface** associated with the network $\Phi(\hat{\mathbf{m}})$ and with point $(\mathbf{x},\mathbf{y})$ as the graph of the function

$$\begin{aligned}\mathcal{L}_{\Phi,(\mathbf{x},\mathbf{y})}(\hat{\mathbf{m}}) \;:\; &\widehat{M} \to [0,+\infty)\\ &\hat{\mathbf{m}} \mapsto d(\Phi(\hat{\mathbf{m}},\mathbf{x}),\mathbf{y})\,,\end{aligned}$$

and we denote it by the symbol $\mathcal{L}_{(\mathbf{x},\mathbf{y})}$. The loss function d is usually convex with respect to its arguments, but the topology of the network Φ makes the loss surface non-convex with respect to $\hat{\mathbf{m}}$, yielding multiple local minima. Therefore, a gradient descent algorithm applied to $\hat{\mathbf{m}}$ would converge towards the nearest local minimum. The loss functional (3.32) can be seen as the average over the ensemble

$$\left(\{\mathcal{L}_{(\mathbf{x},\mathbf{y})}\}_{(\mathbf{x},\mathbf{y})\in X\times Y}\,,\mu\right)\;; \tag{3.53}$$

we call this ensemble the **optimisation landscape**. The empirical loss (3.36) is the value taken by this surface at the point $\hat{\mathbf{m}}$, when μ is the empirical measure associated with the data set $\mathfrak{D}$. This ensemble average

is itself a loss surface. Computing the gradient (3.37) of the empirical loss amounts to averaging the gradients over all the functions $\mathcal{L}_{\Phi,(\mathbf{x},\mathbf{y})}, (\mathbf{x},\mathbf{y}) \in \mathfrak{D}$:

$$\nabla_{\hat{\mathbf{m}}}\mathcal{L}(\Phi(\hat{\mathbf{m}})) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\hat{\mathbf{m}}} d(\Phi(\hat{\mathbf{m}}, \mathbf{x}^{(i)}), \mathbf{y}^{(i)}) \,. \tag{3.54}$$

As for the single-point loss surface $\mathcal{L}_{(\mathbf{x},\mathbf{y})}$, a gradient descent algorithm would converge towards the closest local minimum; therefore, the quality of the end-point depends on the initialisation. The scope of research in gradient descent algorithms for training ANNs [52, 53, 54] is trying to reduce the impact of the initialisation introducing momentum terms, whose goal is keeping track of the *principal modes* of the loss surface to escape the attractors associated with bad initial conditions. Recent research [60] has observed that *flat* and *sharp* minima of the loss surface are correlated with good and poor generalisation capabilities of ANNs, respectively. By flat and sharp, we mean points $\hat{\mathbf{m}} \in \widehat{M}$ which have neighbouroods on which the value of the loss surface is respectively slowly or rapidly changing.

A first strategy to improve generalisation is **mini-batch training**. Let $\mathfrak{D}$ be a given data set (1.44). On-line learning assumes that samples become available one at a time [61]. This suggest to replace the average gradient (3.37) with an approximation

$$\nabla_{\hat{\mathbf{m}}}\mathcal{L}(\Phi(\hat{\mathbf{m}})) \approx \nabla_{\hat{\mathbf{m}}} d(\Phi(\hat{\mathbf{m}}, \mathbf{x}^{(i)}), \mathbf{y}^{(i)}) \,, \tag{3.55}$$

which is of course stochastic since it depends on the sample $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ drawn from the data set $\mathfrak{D}$. This variant of the GD algorithm is therefore called **Stochastic Gradient Descent** (SGD). It can be interpreted as picking just one loss surface from the ensemble (3.53) and performing gradient descent with respect to it. A trade-off between normal GD (3.37) and SGD (3.55) is the so-called **mini-batch SGD**. Denote by N the data set size and let K be a positive integer such that $N = KB$. Denote by

$$\left\{\mathfrak{D}^{(1)}, \mathfrak{D}^{(2)}, \ldots, \mathfrak{D}^{(K)}\right\} \tag{3.56}$$

an exhaustive partition of $\mathfrak{D}$ in mini-batches $\mathfrak{D}^{(k)}, k = 1, 2, \ldots, K$, each of which contains $N_k = B$ data points. To each mini-batch, we can associate an approximated gradient

$$\nabla_{\hat{\mathbf{m}}}\mathcal{L}(\Phi) \approx \frac{1}{B}\sum_{i=1}^{B} \nabla_{\hat{\mathbf{m}}} d(\Phi(\hat{\mathbf{m}}, \mathbf{x}^{(k,i)}), \mathbf{y}^{(k,i)}) \,, \tag{3.57}$$

called the **mini-batch gradient**. This is a compromise between normal GD and SGD, realised to avoid being trapped in the attractor associated with

the initial condition but also to be computationally efficient. In fact, the gradients $\nabla_{\hat{\mathbf{m}}} d(\Phi(\hat{\mathbf{m}}, \mathbf{x}^{(k,i)}), \mathbf{y}^{(k,i)})$ associated with each sample can be computed in parallel on specialised hardware [62]. In ANNs literature, the term SGD is often used to refer to mini-batch SGD. Training ANNs using gradient descent has been interpreted as a dynamical system where the equations of motions are generated by the gradient operator [63, 64]. Making this dynamics stochastic (for example, using SGD or mini-batch SGD) would increase the probability of reaching flat minima of the loss surface. The theory of stochastic differential equations has also been applied trying to improve the convergence of mini-batch SGD [65].

We have observed above that the structure of Φ breaks the convexity of the loss functional with respect to $\hat{\mathbf{m}}$. *Normalisation techniques* have the effect of smoothing the non-convex loss functional by decreasing the Lipschitz constant associated with $\Phi(\hat{\mathbf{m}})$. In 2010, Xavier Glorot and Yoshua Bengio investigated initialisation strategies, and discovered that starting from points $\hat{\mathbf{m}}(0)$ around which Φ is sufficiently Lipschitz increases the probability of convergence towards a good local minimum [66]. More recently, a renormalisation algorithm called **equi-normalisation** has been proposed to preserve this Lipschitnzess property at every iteration, not only at initialisation time [67]. A popular technique, proposed in 2015 by Sergey Ioffe and Christian Szegedy, is **batch normalisation** (BN, also known as batch-norm) [68]. BN has been proposed to counter the *covariate shift* effect [69]. Consider a network (3.13) and a layer (3.12). Let $\mathbf{W}^\ell(t)$ and $\mathbf{b}^\ell(t)$ denote the value of the parameters after the t-it iteration of the gradient descent learning algorithm. For the first layer, the probability distribution according to which

$$\mathbf{x}^1 = \sigma(\mathbf{x}^0 \mathbf{W}^\ell(t) + \mathbf{b}^\ell(t))$$

is distributed are likely different from the one according to which

$$\mathbf{x}^1 = \sigma(\mathbf{x}^0 \mathbf{W}^\ell(t+1) + \mathbf{b}^\ell(t+1))$$

is distributed. The difference between these two distributions is called covariate shift. To reduce this effect, batch normalisation proposes to apply a mini-batch-dependent affine transformation of the mini-batches of scores $\mathbf{s}^\ell, \ell = 1, 2, \ldots, L$:

$$\tilde{\mathbf{s}}^\ell = (\mathbf{s}^\ell - \boldsymbol{\mu})\boldsymbol{\Sigma}^{-1}\boldsymbol{\Gamma} + \boldsymbol{\beta}^\ell \tag{3.58}$$

$$= \mathbf{s}^\ell(\boldsymbol{\Sigma}^{-1}\boldsymbol{\Gamma}^\ell) + (-\boldsymbol{\mu}\boldsymbol{\Sigma}^{-1}\boldsymbol{\Gamma}^\ell + \boldsymbol{\beta}^\ell)\,, \tag{3.59}$$

where $\boldsymbol{\mu}$ is the mean of the score vectors in the k-th batch $\{\mathbf{s}^{\ell,(k,i)}\}_{i=1,2,\ldots,B}$, $\boldsymbol{\Sigma}$ is the diagonal matrix of the variances of their components, $\boldsymbol{\Gamma}^\ell$ is a diagonal

scaling matrix with learnable coefficients and $\boldsymbol{\beta}^\ell$ is a learnable translation vector. The intended purpose of the normalisation operation using μ and Σ was homogenising the statistics of every batch, consequently stabilising the learning of the parameters $\boldsymbol{\Gamma}^\ell$ and $\boldsymbol{\beta}^\ell$. Recent research [70] has experimentally verified that this effect is marginal at the very best, but it has also proved that BN reduces the Lipschitz constant associated with the network Φ, therefore smoothing the optimisation landscape with respect to the parameters $\hat{\mathbf{m}}$.

The regularisation effect of **parameters noise** has also been investigated. At every iteration of the learning algorithm, **dropout** [71] removes a random subset of units from the network graph (3.4). This amounts to set to zero all the weights of links $(i, j) \in \mathcal{E}$ where i is one of the silenced units. When noise ν is added to the parameters $\hat{\mathbf{m}}$, the effect is that of applying a convolution operation over the optimisation landscape:

$$\bar{\mathcal{L}}(\Phi(\hat{\mathbf{m}})) \coloneqq \int_N \iint_{X\times Y} d(\Phi(\hat{\mathbf{m}} + \nu, \mathbf{x}), \mathbf{y})\mu(d\mathbf{x}d\mathbf{y})\mu(d\nu)\,.$$

It is easier to see the smoothing effect if we consider just a single loss surface:

$$\bar{\mathcal{L}}_{\Phi,(\mathbf{x},\mathbf{y})}(\hat{\mathbf{m}}) \coloneqq \int_N d(\Phi(\mathbf{m} + \nu, \mathbf{x}), \mathbf{y})\mu(d\nu)\,.$$

The loss surface $d(\Phi(\hat{\mathbf{m}}, \mathbf{x}), \mathbf{y})$ is replaced by an average of its translations weighted by the probabilities of the translations. This average has been proven to be more Lipschitz than the original loss surface for the case where Φ is a single layer map [72].

So far, artificial neural networks have accompanied all the history of artificial intelligence and machine learning. Original ANNs used non-differentiable activation functions and parameters taken from discrete, or even finite sets, constrained inherited by the physical systems on which they were implemented. Finding suitable learning algorithms for these machine learning systems proved to be a difficult task. The introduction of differentiable activation functions has opened the way to gradient-based learning algorithms, which have allowed training effective systems. From a theoretical perspective, ANNs are general function approximators, as exemplified by Theorem 1. But real-world applications have physical constraints which make selecting a suitable network topology a crucial but complex task. The choice of a good loss function is also a crucial task. In Chapter 4 we will present two scenarios to exemplify these choices.

The real motivation behind the renewed interest in ANNs of the last decade has been the introduction of highly parallel hardware like GPGPUs. We have analysed two levels at which ANNs are parallelisable. The neuron level, characterised by data-independent operations, makes ANNs an ideal fit for SIMD hardware. The computational graph level, which exposes the parallelism of different operations (as depicted in Figure 3.2b1, and Figure 3.2b2), allows executing these blocks concurrently even on different computers at the same time (*distributed computing*).

Current economical trends are demanding to process information directly into the environment where it is extracted. These *edge* computers have more limited processing and memory characteristics with respect to servers and workstations. These constraints have motivated a *return to the origins* of ANNs, investigating the possibility of training models with quantized weights and quantized activation functions. We will investigate this problem in depth in Chapter 5.

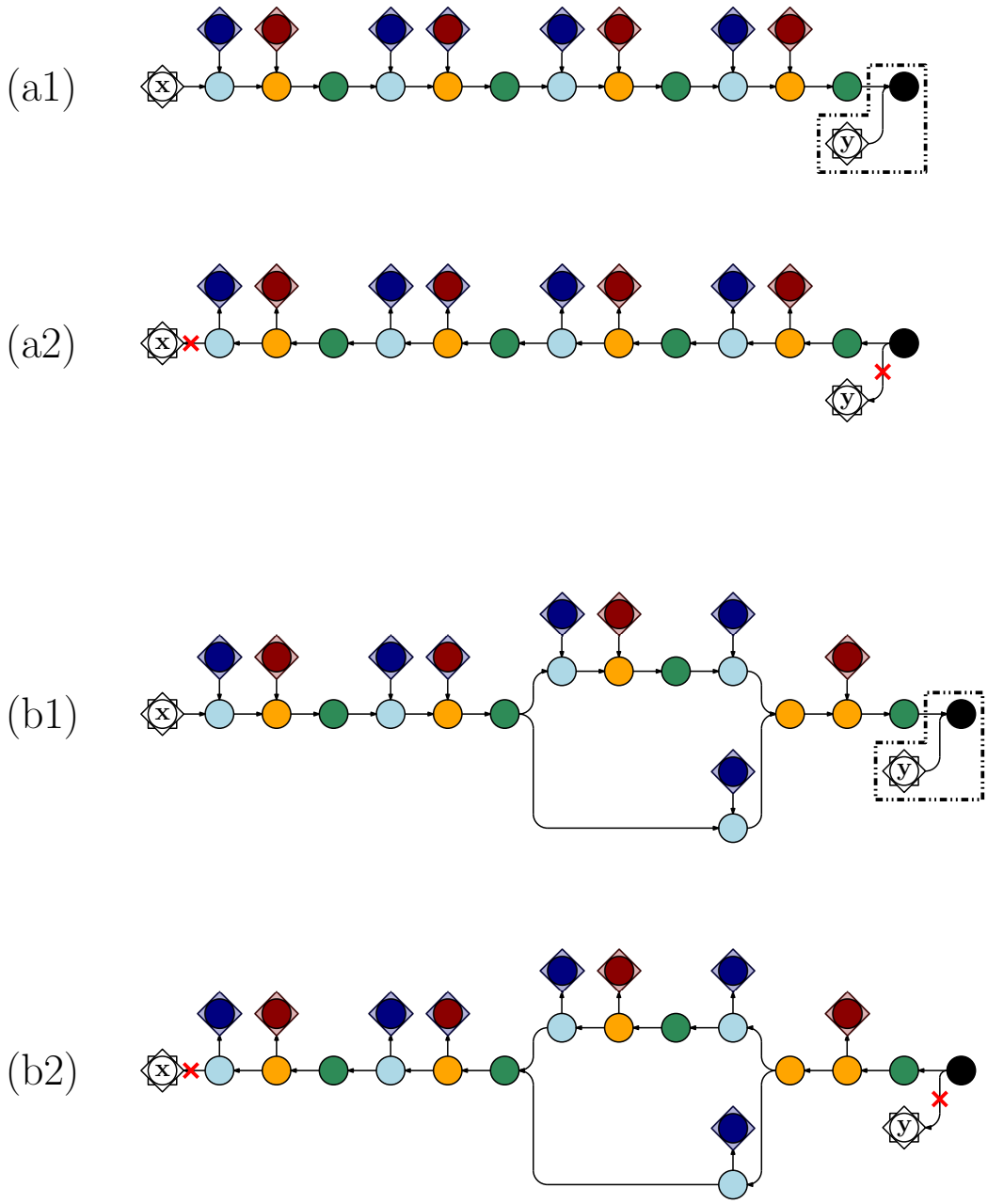


Figure 3.2: Forward (a1) and backward (a2) computational graphs associated with the network of Figure 3.1a; forward (b1) and backward (b2) computational graphs associated with the network of Figure 3.1b. In each graph, nodes represent weights matrices (3.7) (dark blue), bias vectors (3.8) (dark red), linear maps (3.38) (light blue), translations (3.38) (orange nodes), activation functions (3.40) (green nodes) and the loss function (3.33) (black node); of course, the data $(\mathbf{x}, \mathbf{y}) \in \mathfrak{D}$ (white nodes) can not be updated, and the error signals directed towards them are discarded. When the networks are deployed to perform inference, the loss function node and the label node are not needed, and are therefore discarded.

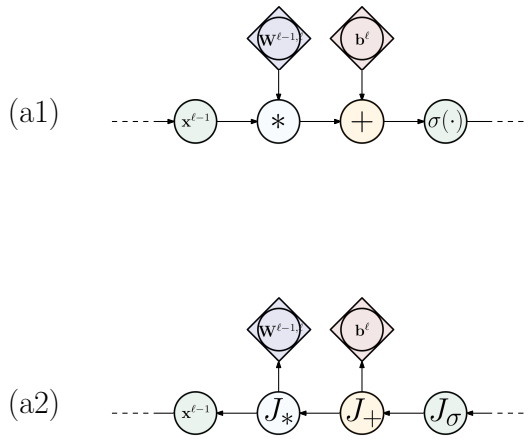


Figure 3.3: Details of (a1) Figure 3.2a1 and (a2) Figure 3.2a2.

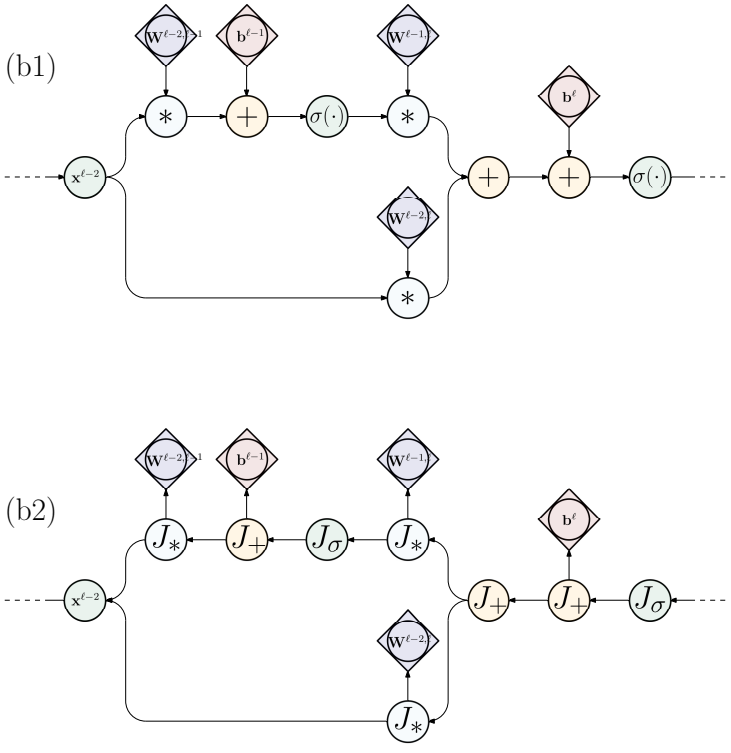


Figure 3.4: Details of (b1) Figure 3.2b1 and (b2) Figure 3.2b2.

Chapter 4

Advanced topics in ANNs

Artificial neural networks are powerful non-linear function approximators. Computer vision is the scientific discipline that studies how to process visual information using computers. Computer vision is amongst those disciplines which have received the most benefit from applying artificial neural networks, motivating the research in a whole sub-class of systems known as *convolutional neural networks*. One of the most useful aspects of artificial neural networks is their *composability* structure: powerful programs can be learnt both *end-to-end* or by stacking different sub-programs that have been trained independently to detect specific features in the available data.

In this chapter we will describe popular computer vision tasks and convolutional neural networks. We will also present a model we developed to analyse time-series using feedforward neural networks, where we extensively leveraged the composability of artificial neural networks.

4.1 Computer vision applications

Computer vision applications extract some abstract information $\mathbf{y}$ from a given digital source $\mathbf{x}$ of visual information. For example, attaching a semantic concept (described by a word in natural language) to a given image. If we denote by X the set of images and by Y the set of labels, a computer vision algorithm should realise a map

$$g : X \to Y \,. \tag{4.1}$$

We will now introduce the common terminology used in computer vision. Let H and W denote positive integers, which we call the **height** and **width** respectively. To encode an image in digital format, one can imagine to lay a plain grid composed of H-by-W squares of unit side on top of a continuous

electromagnetic field. Each square in the grid, identified by a pair of coordinates $(i, j), i \in \{0, 1, \ldots, H-1\}, j \in \{0, 1, \ldots, W-1\}$, is called a **pixel**. The **radiant flux** flowing perpendicularly to the rectangle $[0, W] \times [0, H]$ can be described by a time-dependent function

$$\begin{aligned} f \,:\, \mathbb{R} \times [0, W] \times [0, H] &\to \mathbb{R} \\ (t, x, y) &\mapsto f(t, x, y)\,. \end{aligned} \tag{4.2}$$

The radiant flux through each pixel's surface is processed by a photodetector, a sensor that converts it into a digital representation. This digital representation is characterised by a given **colour model** [73]. Loosely speaking, a colour model with **channels** $k \in \{1, 2, \ldots, C\}$ can be described by a collection of integral functionals of the radiant flux

$$p(k, i, j) \coloneqq \frac{1}{t_{end} - t_{start}} \int_{t_{start}}^{t_{end}} \iint_{W_i \times H_j} f(x, y, t) dx dy dt\,,\ k = 1, 2, \ldots, C\,, \tag{4.3}$$

where $W_i = [i, i+1]$ and $H_j = [j, j+1]$. The most typical colour model of the vision data sets used in machine learning is the RGB model: it consists of $C = 3$ channels that can take on $2^8 = 256$ possible values (thus requiring one byte each to be stored). Using the RGB model, each image $\mathbf{x} \in X$ is encoded as an *array of structures* (AoS) where each of the $H \times W$ structures consists of $C = 3$ bytes. The images in data sets acquired using the RGB colour model are therefore points in the space

$$X = \{0, 1, \ldots, 255\}^{C \times H \times W}\,. \tag{4.4}$$

Let

$$V = \{v_1, v_2, \ldots, v_{N_V}\} \tag{4.5}$$

denote a finite vocabulary of abstract semantic concepts, such as *person*, *tree* or *sky*. When $Y = V$ in (4.1), the vision task is a *classification at image-level*, also called **image classification**. When $Y = \mathcal{N}^{H \times W}$, (4.1) should attach semantic content to each pixel, and the task is a *classification at pixel-level*, also called **image segmentation**.

Another typical vision task is **object localisation**. Localisation requires to predict **bounding boxes**, rectangular containers that can be drawn on the image to enclose areas where objects appear. Bounding boxes are completely determined by pairs of points, the lower left corner and the upper right corner of the rectangle. Let

$$B \coloneqq \{b = \{(x_1, y_1), (x_2, y_2)\} \,|\, x_1 < x_2 \in [0, W], y_1 < y_2 \in [0, H]\} \tag{4.6}$$

denote the space of all possible bounding boxes that can be drawn on an image of sizes H, W. Note that a bounding box $b \in B$ requires a more fine grained labelling than a semantic class $v \in V$, since its components take values in continuous sets. Object localisation is in fact a **regression** task. Since multiple objects can appear concurrently in a single image, we can attach to the image multiple bounding boxes. Therefore, a suitable output space for the localisation task is the space $Y = B^*$ of all possible (finite or infinite) sequences of boxes $b \in B$. Due to the non-uniform length of the sequences in Y, the outputs of localisation algorithms are usually instances of some list data structure. **Object detection** combines classification and localisation into a single task. Given a set of bounding boxes (4.6) and a set of classes (4.5), an **annotation** for the object detection task is a pair

$$(b, v) \in B \times V \,. \tag{4.7}$$

Therefore, the output space for an object detection task is the space $Y = (B \times V)^*$ of all possible (finite or infinite) sequences of annotations (4.7).

The most popular data sets in image classification are the ten-classes **modified National Institute of Standards and Technology** (MNIST) database [49], the ten-classes **Canadian Institute for Advanced Research** (CIFAR-10) data set [74] and the subset of the **ImageNet** database [75] used in the image classification track of the **ImageNet large scale visual recognition challenge** (ILSVRC) [76]. The most popular data set for object detection is Microsoft's **common objects in context** (COCO) [77]. Due to the difficulty of deriving explicit algorithms α_g for tasks (4.1), computer vision has become a typical benchmark for artificial neural networks.

4.2 Convolutional neural networks

Let us add some terminology to that defined in Section 3.1. Consider a feedforward network (3.1) with layers $\mathcal{N}^0, \mathcal{N}^1, \ldots, \mathcal{N}^L$. Let C_ℓ, H_ℓ, W_ℓ be positive integers such that the number of units in the ℓ-th layer can be expressed as $n_\ell = C_\ell H_\ell W_\ell$. We organise the units in the layer in a three-dimensional array structure whose units are indexed by the triple

$$(k_\ell, i_\ell, j_\ell)\,, k_\ell \in \{0, 1, \ldots, C_\ell - 1\}, i_\ell \in \{0, 1, \ldots, H_\ell - 1\}, j_\ell \in \{0, 1, \ldots, W_\ell - 1\}\,. \tag{4.8}$$

We call C_ℓ, H_ℓ and W_ℓ the **depth**, **height** and **width** of the layer respectively. In analogy with the mathematical object, we call these structured layers **tensors**. See Figure 4.1a for an example. The triples (4.8) can be mapped to unique indices in $\mathcal{N}^\ell$ by the simple equation

$$u_\ell = k_\ell H_\ell W_\ell + i_\ell W_\ell + j_\ell + 1\,, \tag{4.9}$$

where the index u_ℓ takes value in the range $\{1, 2, \dots n_\ell\}$. To convert the neuron linear index u_ℓ into neuron tensor coordinates (k_ℓ, i_ℓ, j_ℓ) we use the equations

$$\begin{aligned} k(u_\ell) &= (u_\ell - 1)/(H_\ell W_\ell)\,, \\ i(u_\ell) &= ((u_\ell - 1) \bmod H_\ell W_\ell)/W_\ell\,, \\ j(u_\ell) &= (u_\ell - 1) \bmod W_\ell\,, \end{aligned} \tag{4.10}$$

where / denotes integer division and mod denotes the remainder of integer division. We will therefore refer to a unit with both the notations $N^\ell_{u_\ell}$ and $N^\ell_{(k,i,j)}$. We partition $\mathcal{N}^\ell$ in C_ℓ **planes**

$$\mathcal{P}^\ell_{k_\ell} \coloneqq \{N^\ell_{u_\ell} \,|\, k(u_\ell) = k_\ell\}\,, k_\ell = 0, 1, \dots, C_\ell - 1\,, \tag{4.11}$$

each of which contains exactly $H_\ell W_\ell$ units. In Chapter 3 we considered fully-connected networks, where each unit in $\mathcal{N}^\ell$ is connected to every unit in $\mathcal{N}^{\ell+1}$. Mathematically speaking, these are networks (3.4) where restricting $\mathcal{N}$ to pairs of adjacent layers $\mathcal{N}^\ell, \mathcal{N}^{\ell+1}$ yields a complete bipartite graph. Consequently, the adjacency matrices $\mathbf{A}^{\ell,\ell+1}$ of these sub-graphs are **dense**. In the following we will discuss a strategy to make this connectivity **sparse**.

Let F^H_ℓ, F^W_ℓ be positive integers such that F^H_ℓ is relatively small with respect to H_ℓ and F^W_ℓ is relatively small with respect to W_ℓ. The set

$$\mathcal{C}^\ell_{(0,0)} \coloneqq \{N^\ell_{(k,i,j)} \in \mathcal{N}^\ell \,|\, i \in \{0, 1, \dots, F^H_\ell - 1\}, j \in \{0, 1, \dots, F^W_\ell - 1\}\} \tag{4.12}$$

is called a *column* of neurons. Note that neurons have neighbouring spatial coordinates (i.e., a column has **local** spatial support). Let now S^H_ℓ, S^W_ℓ and $H_{\ell+1}, W_{\ell+1}$ be positive integers satisfying

$$\begin{aligned} H_\ell &= F^H_\ell + S^H_\ell(H_\ell + 1)\,, \\ W_\ell &= F^W_\ell + S^W_\ell(W_\ell + 1)\,. \end{aligned} \tag{4.13}$$

We can identify other columns

$$\begin{aligned} \mathcal{C}^\ell_{(i_{\ell+1}, j_{\ell+1})} \coloneqq \{&N^\ell_{(k,i,j)} \in \mathcal{N}^\ell \,| \\ &i \in \{S^H_\ell i_{\ell+1}, \dots, S^H_\ell i_{\ell+1} + F^H_\ell - 1\}, \\ &j \in \{S^W_\ell j_{\ell+1}, \dots, S^W_\ell j_{\ell+1} + F^W_\ell - 1\}\}\,, \end{aligned} \tag{4.14}$$

where $i_{\ell+1} \in \{0, 1, \dots, H_{\ell+1} - 1\}$ and $j_{\ell+1} \in \{0, 1, \dots, W_{\ell+1} - 1\}$. They can be visualised by shifting an imaginary box enclosing (4.12) by steps of length S^H_ℓ and S^W_ℓ respectively. The pair (S^H_ℓ, S^W_ℓ) is called the **striding** of the support

and its components are called **strides**. See Figure 4.1b and Figure 4.1c for examples of columns. Now, consider a tensor $\mathcal{N}^{\ell+1}$ consisting of a single plane $\mathcal{N}_0^{\ell+1}$ with $H_{\ell+1}W_{\ell+1}$ units. Each unit $N^{\ell+1}_{(0,i_{\ell+1},j_{\ell+1})}$ is connected to all the neurons in a column (4.14), but to none of the neurons of $\mathcal{N}^{\ell}$ which are not in the column. We can of course stack multiple planes $\mathcal{N}^{\ell+1}_{k_{\ell+1}}$ of neurons in the $(\ell+1)$-th layer, where neurons $N^{\ell+1}_{(k_1,i,j)}, N^{\ell+1}_{(k_2,i,j)}$ from different planes but with the same spatial coordinates share the same support (4.14). This structure reduces the connectivity from

$$n_{\ell+1}n_\ell = C_{\ell+1}H_{\ell+1}W_{\ell+1}C_\ell H_\ell W_\ell \tag{4.15}$$

connections to

$$C_{\ell+1}H_{\ell+1}W_{\ell+1}C_\ell H_\ell W_\ell \frac{F_\ell^H F_\ell^W}{H_\ell W_\ell} = C_{\ell+1}H_{\ell+1}W_{\ell+1}C_\ell F_\ell^H F_\ell^W \,. \tag{4.16}$$

We now want to define an additional constraint: **parameters sharing**. Let $N^{\ell+1}_{(k_{\ell+1},i_1,j_1)}$ and $N^{\ell+1}_{(k_{\ell+1},i_2,j_2)}$ be units belonging to the $k_{\ell+1}$-th plane of layer $\mathcal{N}^{\ell+1}$, and let $\mathcal{C}^\ell_{(i_1,j_1)}$ and $\mathcal{C}^\ell_{(i_2,j_2)}$ be their respective supports. We say that units

$$\begin{aligned} N^\ell_{u_1} &\in \mathcal{C}^\ell_{(i_1,j_1)}\,, \\ N^\ell_{u_2} &\in \mathcal{C}^\ell_{(i_2,j_2)}\,, \end{aligned}$$

occupy the same relative positions in the supports of $N^{\ell+1}_{(k_{\ell+1},i_1,j_1)}$ and $N^{\ell+1}_{(k_{\ell+1},i_2,j_2)}$ if the following properties are satisfied:

$$\begin{aligned} k(u_2) &= k(u_1)\,, \\ i(u_2) - i(u_1) &= S_\ell^H(i_2 - i_1)\,, \\ j(u_2) - j(u_1) &= S_\ell^W(j_2 - j_1)\,; \end{aligned} \tag{4.17}$$

in this case we will write

$$N^\ell_{u_1} \sim N^\ell_{u_2}\,. \tag{4.18}$$

We say that the k-th plane of the layer $\mathcal{N}^{\ell+1}$ satisfies the **weights-sharing property** if

$$\begin{aligned} &\forall\, N^{\ell+1}_{(k_{\ell+1},i_1,j_1)}, N^{\ell+1}_{(k_{\ell+1},i_2,j_2)} \in \mathcal{N}^{\ell+1}_{k_{\ell+1}}\,, \\ &\quad w(N^\ell_{u_1}, N^{\ell+1}_{(k_{\ell+1}),i_1,j_1)}) = w(N^\ell_{u_2}, N^{\ell+1}_{(k_{\ell+1}),i_2,j_2)})\,, \\ &\quad \forall\, N^\ell_{u_1} \sim N^\ell_{u_2}\,. \end{aligned} \tag{4.19}$$

When the weights-sharing property is satisfied, the number of connections is still (4.16), but the number of parameters is reduced:

$$C_{\ell+1}H_{\ell+1}W_{\ell+1}C_\ell F_\ell^H F_\ell^W \frac{1}{H_{\ell+1}W_{\ell+1}} = C_{\ell+1}C_\ell F_\ell^H F_\ell^W \,. \tag{4.20}$$

Remember that each unit $N^{\ell+1}_{(k,i,j)}$ also has its own bias parameter $b^{\ell+1}_{(k,i,j)}$. We say that the k-th plane of layer $\mathcal{N}^{\ell+1}$ satisfies the **bias-sharing property** if

$$\begin{aligned} \forall\, & N^{\ell+1}_{(k_{\ell+1},i_1,j_1)}, N^{\ell+1}_{(k_{\ell+1},i_2,j_2)} \in \mathcal{N}^{\ell+1}_{k_{\ell+1}}\,, \\ & b(N^{\ell+1}_{(k,i_1,j_1)}) = b(N^{\ell+1}_{(k,i_2,j_2)}) \\ & \forall i_1, i_2 \in \{0,1,\dots,H_{\ell+1}-1\}, \\ & j_1, j_2 \in \{0,1,\dots,W_{\ell+1}-1\}\,. \end{aligned} \tag{4.21}$$

We say that the k-th plane of layer $\mathcal{N}^{\ell+1}$ satisfies the **parameters-sharing** property when both the weights-sharing (4.19) and bias-sharing (4.21) properties are satisfied.

To get a better understanding of the operation performed by the layer maps (3.12) implemented by these *locally connected* networks, consider adjacent tensors $\mathcal{N}^\ell, \mathcal{N}^{\ell+1}$ such that $C_\ell = C_{\ell+1} = 1$ (i.e., they consist of just one plane each). In this case, the plane can be considered a discretisation of a two-dimensional domain $K \subset \mathbb{R}^2$, and a representations $\mathbf{x}^\ell$ is a sample from a function defined on K. Columns (4.14) are reduced to sub-planes. When the weights-sharing property is satisfied, the linear part of the layer map performs the following operation:

$$\sum_{N^\ell_u \in \mathcal{C}^\ell_{(i,j)}} x^\ell_u w(N^\ell_u, N^{\ell+1}_{(0,i,j)})\,. \tag{4.22}$$

This is the discrete counterpart of the continuous-valued integral

$$\mathbf{s}^\ell(y) = \int_K \mathbf{x}^\ell(x)\mathbf{w}^\ell(y-x)dx\,, \tag{4.23}$$

which is a convolution of the map $\mathbf{x}^\ell$ with a kernel $\mathbf{w}$ of compact support. The locality of the connections of $N^{\ell+1}_v$ with neurons of the tensor $\mathcal{N}^\ell$ implies that the weights of the links with neurons which are outside of the column enclosing the support of the kernel are set to zero. This analogy with the convolution operation is the reason why neural networks satisfying the feedforward property, the locality property and the weights-sharing property are called **convolutional neural networks** (CNNs). The neuron

map $\varphi^{\ell+1}_{k_{\ell+1}}$ computed by each neuron of the $k_{\ell+1}$-th plane of tensor $\mathcal{N}^{\ell+1}$ is called a **feature map**. The filter $\mathbf{w}^{\ell+1}_{k_{\ell+1}}$ associated with the feature map is called the **kernel** of the feature map. The representations $\mathbf{x}^{\ell+1}_{k_{\ell+1}}$ living in the codomain of the $k_{\ell+1}$-th feature map are called **features**, and they quantify the presence/absence of a specific feature in the different columns (4.14).

Different planes $\mathcal{N}^{\ell+1}_{k_{\ell+1}}$ of a tensor should encode different features. To enforce this diversification, it is possible to constrain the supports (4.14) even more. For example, when the height and width of the columns are constrained to $F^H_\ell = F^W_\ell = 1$, the respective feature map is called a **pointwise convolution**. Another common practice is associating to each plane $\mathcal{N}^{\ell+1}_{k_{\ell+1}}$ a **group** of planes belonging to the previous layer,

$$G^\ell_{k_{\ell+1}} \subseteq \{0, 1, \ldots, C_\ell - 1\}\,, \tag{4.24}$$

so that the support of neuron $N^{\ell+1}_{(k_{\ell+1},i,j)}$ is reduced from $\mathcal{C}^{\ell+1}_{(i,j)}$ to

$$\mathcal{C}^{\ell+1}_{(i,j)} \bigcap \left(\bigcup_{k \in G^\ell_{k_{\ell+1}}} \mathcal{N}^\ell_k \right)\,. \tag{4.25}$$

When $C_{\ell+1} = C_\ell = C$ and $G^\ell_k = \{k\}, k = 0, 1, \ldots, C-1$, each feature map is a **spatial convolution** (i.e., it does not mix the information contained in different planes of $\mathcal{N}^\ell$).

A last notion we must mention is **padding**. When either F^H_ℓ or F^W_ℓ are greater than one, it is not possible to place columns (4.14) in a number of positions $H_{\ell+1}W_{\ell+1} = H_\ell W_\ell$. This therefore reduces the spatial dimensions of following tensors. To prevent the collapse of the spatial dimensions of these tensors, it is desirable to preserve the sizes $H_{\ell+1} = H_\ell$ and $W_{\ell+1} = W_\ell$. To this end, it is usual to **pad** the input tensor $\mathcal{N}^\ell$ with an apron of units, so that the spatial sizes of the extended tensor are

$$\begin{aligned} H'_\ell &= P^H_\ell + H_\ell + P^H_\ell\,, \\ W'_\ell &= P^W_\ell + W_\ell + P^W_\ell\,, \end{aligned} \tag{4.26}$$

where P^H_ℓ, P^W_ℓ are positive integers. The tuple (P^H_ℓ, P^W_ℓ) is called **padding**. The units in the apron are just virtual: they are never implemented, since they are supposed to have no connections with units from previous layers and their states are always zero.

The visual cortex is the part of the human brain that processes visual information. Visual information is acquired by specialised cells of the eye which are sensitive to the radiations belonging to the part of the electromagnetic

spectrum which we call *visible light*. The radiant flux (4.2) exhibits spatial correlation [78]. Between the 60's and the 70's, David Hubel and Torsten Wiesel conducted a series of studies in the neurophysiology of mammals' visual cortex, discovering that their brains extract visual information in a hierarchical fashion to exploit this spatial correlation [79, 80, 81, 82]. They suggested that the neurons of the visual cortex are organised into multiple layers, where each neuron is connected only to a small (local) subset of neighbouring cells belonging to the previous layer. This model brought along the concepts of **locality** and **hierarchy** of information processing: global information could be extracted by stacking components with local connectivity. The concept of local connectivity (i.e., a few lines afferent to a single computational unit) was already used by Rosenblatt in his original perceptron model [29, 30]. Nevertheless, learning visual patterns remained a hard task when these patterns where translated or rotated in the receptive field of the learning system: perceptrons where not able to recognize isometries. This was the so-called problem of *position invariance* of the stimuli. In 1980, Kunihiko Fukushima made an important step towards solving it by designing an ANN called the **neocognitron** [83, 84]. The neocognitron included a hierarchy of layers whose neurons were connected to columns (4.14) of neurons of the previous layer. Moreover, neurons were arranged in planes and forced to satisfy the weights-sharing property (4.19). In 1989, Yann LeCun and his collaborators pioneered the application of the backpropagation algorithm to train ANNs for visual information processing, marking the birth of CNNs [85]. Their model, named **LeNet**, satisfied the local connectivity and weights-sharing properties (but not the bias-sharing property). Following their introduction, CNNs have been applied to vision tasks associated with document recognition [86]. Remarkably, the values of the weights learnt by a CNNs trained with the backpropagation algorithm remind of the excitatory and inhibitory patterns of the synapsis between the neurons in the visual cortex identified by Hubel and Wiesel [87]. In recent years, CNNs have been the focus of a vast amount of research. Why did this happen? As we saw in Section 1.4, the physical limitation called the *power wall* has motivated a transition from single-core computer architectures to parallel multi- and many-core architectures. In particular, vendors of graphics processing units (GPUs) have spent considerable resources in expanding their instruction set architectures (ISAs) to make them general purpose programmable devices (GPGPUs) that can run also programs not specific to computer graphics. We pointed out in Section 3.3 that the computational properties of ANNs make their programs suitable for GPGPUs. GPGPUs were one of the reasons behind the success of **AlexNet**, a CNN topology designed to solve the image classification task on the ILSVRC2012 data set [88]. Also, AlexNet

replace classical activation functions (3.24) and (3.25) with the **rectified linear unit** (ReLU) function

$$\begin{aligned} ReLU &: \mathbb{R} \to [0, +\infty) \\ x &\to \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0\,, \end{cases} \end{aligned} \tag{4.27}$$

which is still (almost everywhere) differentiable but can be computed more efficiently on digital computers. To reduce the number of model parameters, Oxford's visual geometry group developed the **VGG** network family [89]. The innovation of VGG was replacing convolutional layers using supports with spatial dimensions $F_\ell^H, F_\ell^W > 3$ with stacks of convolutional layers using only supports with spatial dimensions $F_\ell^H = F_\ell^W = 3$; the *receptive field* of the hierarchy was made larger by adding layers (i.e., by *going deeper*) instead of using larger supports. To improve the quality of the learnt features, the **GoogLeNet** network family used **inception** modules. An inception module is a convolutional layer structured into multiple branches constrained to model feature maps of different nature; these smaller sub-networks could still be executed in parallel [90, 91]. The idea behind inception modules was extended in two ways. First, **residual networks** (ResNets) reduced the number of branches to two: an identity (or linear) branch, and an additive residual that should model variations over this *baseline* [92]. ResNets were the first CNNs explicitly modelled to learn *high-level* representations combining different *low-level* representations of the same input. Second, **extreme inception** (Xception) [93] introduced the idea of modelling residual branches using stacks of spatial convolutions and pointwise convolutions. These advancements have produced CNNs which are more compact in terms of the number of parameters. These models, such as the **MobileNet** network families [94, 95], can consequently be deployed also on resource-constrained devices.

4.3 A model of subjective driving perceptions

Maserati is a world excellence in the automotive sector. As a **business-to-consumer** (B2C) company, its product development processes are strongly oriented to improving the user experience. The problem with user experience is that it is often a subjective perception. Automobiles handling performances are an example. In human-vehicle systems, the human driver represents a very complex control in a feedback-loop. During the interaction between the vehicle and the external environment, the mechanical characteristics of the vehicle concur to determine the forces that act on it and its

occupants. The driver, perceiving these forces and integrating them with sensory information about the environment (visual, auditory), performs actions on the steering wheel, the brake or the throttle pedals, and the gear lever to control the vehicle. This control causes the vehicle to change its state and its interaction with the environment, starting the loop over. Quantifying the perceived quality of an automobile's handling properties is a difficult task. Therefore, automotive companies like Maserati hire professional automotive test drivers to evaluate these performances, in a *design-test-analysis* cycle that works as follows. Before the first iteration of this cycle, the engineers and the drivers agree on a set of properties (e.g., understeering, oversteering, stability during overtakings) that should be assessed, and define a set of maneuvers (e.g., tight curves at high speed, overtakings) that the drivers can use to evaluate the properties. Then the cycle begins. At design time, the engineers define some *synthesis parameters* which describe a prototype automobile; from them, a physical prototype can be dimensioned and built, or a virtual model can be computed and loaded onto a driving simulator. Then, the drivers perform *driving experiences* on a test circuit (physical or simulated) where they can execute the agreed maneuvers using the prototype; during this test phase, data is collected from a suite of sensors installed on the automobile. Finally, the drivers evaluate the handling properties according to their driving experiences; the engineers analyse these results together with the objective data collected during the experience and try to understand the reasons behind the drivers' evaluations. When the analysis is completed, the engineers can define a new set of synthesis parameters and start a new iteration of the experiment. The design-test-analysis cycle continues until satisfying performances are attained. This product development process is expensive and time consuming. Is it possible to predict the evaluations of the drivers just by looking at the objective data collected during their driving experiences? Is it possible to create a model that explains the reasons behind these predictions? More generally, can we reuse such a model to assess also performances other than handling? A positive answer to these questions would make the analysis phase more accurate, reducing the number of iterations of the experiment and, in the end, saving both money and time.

Our data set consisted of labelled **sequences**

$$\mathfrak{D} = \{(\mathfrak{Z}^{(1)}, \mathbf{y}^{(1)}), (\mathfrak{Z}^{(2)}, \mathbf{y}^{(2)}), \ldots, (\mathfrak{Z}^{(N)}, \mathbf{y}^{(N)})\} \tag{4.28}$$

called **records**. Each record

$$\mathfrak{Z}^{(n)} = \{\mathbf{z}^{(n,1)}, \mathbf{z}^{(n,2)}, \ldots, \mathbf{z}^{(n,T_n)}\}, n = 1, 2, \ldots N, \tag{4.29}$$

contains the measurements collected during a driving experience. Each vector $\mathbf{z}$ consists of n_Z components called **channels**, which correspond to ac-

celerometers, dynamometers, speedometers and actuators installed on the automobile. Each sequence (4.29) represents the history of the physical state of an automobile and its driver's actions, recorded during a lap performed on a given test circuit. Since the test circuit had fixed length, but the speed at which it was run across could vary, the records had different sizes. For a reason that we will clarify later, we split each vector

$$\mathbf{z}^{(n,t)} = (\mathbf{x}^{(n,t)}, \mathbf{c}^{(n,t)})$$

to separate sensors data (the state) from actuators data (the control), obtaining two sequences

$$\mathfrak{X}^{(n)} = \{\mathbf{x}^{(n,1)}, \mathbf{x}^{(n,2)}, \ldots, \mathbf{x}^{(n,T_n)}\}, \tag{4.30}$$

$$\mathfrak{C}^{(t)} = \{\mathbf{c}^{(n,1)}, \mathbf{c}^{(n,1)}, \ldots, \mathbf{c}^{(n,T_n)}\}. \tag{4.31}$$

The labels $\mathbf{y}^{(n)}$, called **scores**, measured the properties on which engineers and drivers had agreed at the beginning of the design-test-analysis cycle. Each component $y_i^{(n)}$ takes values in the finite set

$$Y = \{1.0, 1.5, \ldots, 9.5, 10.0\}, \tag{4.32}$$

which is called the **scores set**. To simplify our discussion, we will consider labels $\mathbf{y}^{(n)}$ with just one component and therefore replace this symbol with $y^{(n)}$.

In this context, modelling the human driver amounts to building a machine learning system capable of classifying sequences (4.29) and which is able to identify which parts of the sequence are *important* for its classification. Since multi-dimensional time series (4.30) are implemented using array data structures

$$\mathtt{x} = \mathtt{x}[\mathtt{t}, \mathtt{i}], \mathtt{t} = 1, 2, \ldots, \mathtt{T_n}, \mathtt{i} = 1, 2, \ldots, \mathtt{n_X},$$

we modelled this second task as the search for probability mass function (1.21) over the space

$$\{1, 2, \ldots, T_n\} \times \{1, 2, \ldots, n_X\}. \tag{4.33}$$

To simplify the design of the system, we took advantage of the fact that the test circuit had fixed length parametrised by a curvilinear abscissa. We use the curvilinear abscissa to determine L equally spaced **waypoints** on the circuit, and transform sequences (4.29) into sequences

$$\mathfrak{Z}^{(n)} = \{\mathbf{z}^{(n,1)}, \mathbf{z}^{(n,2)}, \ldots, \mathbf{z}^{(n,L)}\}, \tag{4.34}$$

where $\mathbf{z}^{(n,l)}$ is obtained by linear interpolation between the vectors $\mathbf{z}^{(n,t(l))}$ and $\mathbf{z}^{(n,t(l)+1)}$ ($t(l)$ indexes the latest vector of the original sequence whose curvilinear abscissa does not exceed that of the l-th waypoint). We homogenised (4.31) and (4.30) analogously. Consequently, instead of outputting a PMF over (4.33), our system is designed to output a PMF over the space

$$\{1, 2, \ldots, L\} \times \{1, 2, \ldots, n_X\}\,. \tag{4.35}$$

As we pointed out in Section 3.4, the design of successful machine learning systems takes considerable modelling effort to incorporate domain knowledge into them. To exemplify this process, we will therefore explain the rationale behind all our assumptions.

Maserati's engineers pointed out that the channels contain redundant information. The physical nature of accelerations and forces induced us to interpret the records (4.30) as trajectories living on a manifold embedded in some high-dimensional space $\mathbb{R}^{n_X}$. We chose to use a manifold learning technique to eliminate redundancies by reducing the dimensionality of these trajectories. Autoencoders [96] are feedforward networks designed to project points from a high-dimensional Euclidean space R^{n_X} onto a low-dimensional space $\mathbb{R}^{n_E}$. Autoencoders achieve this goal by using two networks

$$\begin{aligned}\varphi^{(e)} &: \mathbb{R}^{n_X} \to \mathbb{R}^{n_E}\,,\\ \varphi^{(d)} &: \mathbb{R}^{n_E} \to \mathbb{R}^{n_X}\,,\end{aligned}$$

called the **encoder** and **decoder**, that are stacked to perform compression and decompression of input vectors. The autoencoder is trained by minimising the loss function called **mean square error** (MSE) loss:

$$d(\varphi^{(d)}(\varphi^{(e)}(\mathbf{x}^0)), \mathbf{x}^0) \coloneqq \|\varphi^{(d)}(\varphi^{(e)}(\mathbf{x}^0)) - \mathbf{x}^0\|^2\,.$$

The effect of the backpropagation algorithm is therefore to make $\varphi^{(d)} \circ \varphi^{(e)}$ as similar to the identity function as possible. Since the dimensions satisfy $n_E \ll n_X$, it is impossible for the encoder to learn the identity map. Denoising autoencoders [97] are an improvement over classical autoencoders. At training time, the encoder of a denoising autoencoder is fed a noisy version of the input point and the loss is therefore

$$d(\varphi^{(d)}(\varphi^{(e)}(\mathbf{x}^0 + \boldsymbol{\nu})), \mathbf{x}^0) \coloneqq \|\varphi^{(d)}(\varphi^{(e)}(\mathbf{x}^0 + \boldsymbol{\nu})) - \mathbf{x}^0\|^2\,. \tag{4.36}$$

The regularisation effect of this probabilistic alteration is similar to that of parameters additive noise described in Section 3.4. For our model we trained a 4-layers denoising autoencoder

$$\varphi_1^4 \circ \varphi_1^3 \circ \varphi_1^2 \circ \varphi_1^1 \,:\, \mathbb{R}^{n_X} \to \mathbb{R}^{n_X}$$

where the first and third layers contain ten units each; to set the dimension n_E of the space of encoded features (i.e., the number of units in the second layer), we applied another technique from the field of manifold learning [98] that provided us an estimate of $n_E = 4$. All the layers except the last one are activated by the hyperbolic tangent function (3.25). We used the sum of all the multi-sets (4.31) as a data set to train the denoising autoencoder; we applied the Adam optimisation algorithm [54] with a learning rate of 0.001 and a batch size of 25 for 500 epochs. During training, we applied a zero-mean Gaussian noise $\boldsymbol{\nu}$ with a standard deviation of 0.01 to all the components of the input vectors. The encoder

$$\varphi_1 = \varphi_1^2 \circ \varphi_1^1 : \mathbb{R}^{n_X} \to \mathbb{R}^{n_E}$$

can be applied pointwise to a trajectory (4.30). This pointwise transformation of the trajectory is called the **trajectory condenser**:

$$\begin{aligned} \Phi_1 : \quad & X^0 \mapsto X^1 \\ & \{\mathbf{x}^{0,(l)}\}_{\ell=1,2,\ldots,L} \mapsto \{\mathbf{x}^{1,(l)}\} = \{\varphi_1(\mathbf{x}^{0(1)}), \varphi_1(\mathbf{x}^{0(2)}), \ldots, \varphi_1(\mathbf{x}^{0(L)})\}\,, \end{aligned} \tag{4.37}$$

where $X^0 \subset (\mathbb{R}^{n_X})^L$ and $X^1 \subset (\mathbb{R}^{n_E})^L$. The sequence

$$\{\mathbf{x}^{1(l)}\}_{l=1,2,\ldots,L} \tag{4.38}$$

is called the **condensed trajectory**.

Discussions with the drivers suggested that their final evaluation summarises a sequence of perceptions collected during the tests, plus some global contextual information (e.g., the time required to complete a lap on the test circuit). We thus modelled the evaluation task (i.e., the assignment of a score to a vehicle) as a *supertask* accomplished using information collected while performing simpler *subtasks*, hierarchically. The natural subtasks we selected are the controls (4.31) actuated by the driver on the vehicle: we hypothesised that the information used by the driver to control the vehicle also informs his final evaluation. This hierarchical composition of information, from subtasks to supertasks, has already been applied successfully to document classification using hierarchical attention mechanisms [99]. Attention mechanisms were developed to analyse natural language data [100]. Let $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(L)}\}$ denote a sequence. The extraction of a single vector from the sequence is known as a *query* in databases terminology. If the elements of the sequence belong to a space where scalar multiplication and addition are defined (i.e., to a vector space), then querying the element $\mathbf{x}^{\tilde{l}}$ can be interpreted as computing the sum

$$\sum_{l=1}^{L} \delta_l^{\tilde{l}} \mathbf{x}^{(l)}\,,$$

where $\delta_l^{\tilde{l}}$ denotes Kronecker's delta. The idea of attention mechanisms is to perform a *soft query* by using a probabilistic average

$$\sum_{l=1}^{L} \theta^{(l)} \mathbf{x}^{(l)} \, , \sum_{l=1}^{L} \theta^{(l)} = 1 \, . \tag{4.39}$$

Suppose that in the space from which the sequence has been sampled it is defined an inner product. A **query vector w** can then be used in conjunction with a softmax function

$$\begin{aligned} softmax \, : \, \mathbb{R}^L \to \{\mathbf{x} \in \mathbb{R}^L, | \, \sum_{l=1}^{L} x_l = 1\} \\ \mathbf{x} \mapsto \left(\frac{e^{x_1}}{\sum_{l=1}^{L} e^{x_l}}, \frac{e^{x_2}}{\sum_{l=1}^{L} e^{x_l}}, \dots, \frac{e^{x_L}}{\sum_{l=1}^{L} e^{x_l}} \right) \, , \end{aligned} \tag{4.40}$$

to define the coefficients:

$$\theta_{\mathbf{w}}^{(\tilde{l})} = \frac{e^{\langle \mathbf{x}^{(\tilde{l})}, \mathbf{w} \rangle}}{\sum_{l=1}^{L} e^{\langle \mathbf{x}^{(l)}, \mathbf{w} \rangle}} \, . \tag{4.41}$$

Due to the differentiable structure of both (4.40) and (4.41), the query vector can be learnt using the backpropagation algorithm. The advantage of attention mechanisms is that they natively compute a probability mass function over the input sequence, which will be helpful towards the goals of our model. The operations performed by attention mechanisms are hard to contextualise in the framework described in Section 3.1: the analogy with biological neural networks is lost. Nevertheless, these operations still satisfy the desirable computational properties described in Section 3.3, and attention mechanisms are recognised as a valid ANN operation.

To decompose our problem, we first defined two positive integers L_1, L_2 such that $L = L_1 L_2$. We call L_1 the **window size**, and we call a **window** each sequence

$$\mathfrak{W}^{(l)} \coloneqq \{\mathbf{x}^{1,(l+1)}, \mathbf{x}^{1,(l+2)}, \dots, \mathbf{x}^{1,(l+L_1)}\} \tag{4.42}$$

consisting of L_1 consecutive points taken from any condensed trajectory. To each window, we can associate the controls $\mathbf{c}^{(l+L_1+1)} \in \mathfrak{C}$ actuated by the driver at the immediately successive waypoint. The data points $(\mathfrak{W}, \mathbf{c}^{(l+L_1+1)})$ built up a labelled data set that we used to train a 2-layer feedforward neural network

$$\varphi_2^2 \circ \varphi_2^1 \, : \, (\mathbb{R}^{n_E})^{L_1} \to [-1, +1]^2, \tag{4.43}$$

to mimic the decision process the drivers apply to decide the controls (throttle/brake pedals, steering wheel left/right). This **controls predictor** network is composed of an attention mechanism

$$\begin{aligned} \varphi_2^1 \,:\, (\mathbb{R}^{n_E})^{L_1} &\to \mathbb{R}^{n_e} \\ \mathfrak{W}^{(\tilde{l})} &\mapsto \mathbf{x}^{2,(\tilde{l})} = \sum_{l=\tilde{l}+1}^{\tilde{l}+L_1} \theta^{(l)}_{\mathbf{w}_2^1} \mathbf{x}^{1,(l)} \end{aligned} \tag{4.44}$$

parametrised by $\mathbf{w}_2^1 \in \mathbb{R}^{n_E}$ and by a layer map

$$\begin{aligned} \varphi_2^2 \,:\, \mathbb{R}^{n_E} &\to [-1,+1]^2 \\ \mathbf{x}^{2,(\tilde{l})} &\mapsto tanh(\mathbf{x}^{2,(\tilde{l})}\mathbf{W}_2^2 + \mathbf{b}_2^2) \end{aligned}$$

activated by a hyperbolic tangent function trained to predict the control $\mathbf{c}^{(l+L_1+1)}$ actuated by the driver. We minimised the MSE loss applying the Adam optimisation algorithm [54] with a learning rate of 0.001 and a batch size of 10 for 20 epochs. We empirically found an optimal window size of $L_1 = 10$. Remember that we chose L_1, L_2 so that $L = L_1 L_2$. This choice was made to partition the condensed trajectory (4.38) with L_2 non-overlapping windows of size L_1 each. The **trajectory analyser** is realised applying the first layer of the controls predictor (4.44) to every window that partitions the compressed trajectory:

$$\begin{aligned} \Phi_2 \,:\, & X^1 \mapsto X^2 \\ & \{\mathbf{x}^{1,(l)}\}_{l=1,2,\dots L} \mapsto \{\varphi_2^1(\mathfrak{W}^{(1)}), \varphi_2^1(\mathfrak{W}^{(L_1+1)}), \dots, \varphi_2^1(\mathfrak{W}^{((L_2-1)L_1+1)})\}. \end{aligned} \tag{4.45}$$

The sequence

$$\{\mathbf{x}^{2,(m)}\}_{m=1,2,\dots,L_2} \tag{4.46}$$

is called the **compressed trajectory**. Observe that the application of φ_2^1 to the windows (4.42) returns L_2 probability distributions, one over each window:

$$\{p^m_{\varphi_2^1}(l)\,, l = L_1(m-1)+1, \dots, L_1(m-1)+L_1\}\,, m = 1, 2, \dots, L_2\,. \tag{4.47}$$

Since we are interested in the features on which the controls decisions are taken, not in the controls themselves, we discard the regression layer φ_2^2 after training.

Finally, we defined the **trajectory classifier**

$$\begin{aligned} \varphi_3 \,:\, (\mathbb{R}^{n_E})^{L_2} &\to \mathbb{R}^{n_Y} \\ \{\mathbf{x}^{2,(m)}\}_{m=1,2,\dots,L_2} &\mapsto \mathbf{x}^3\,. \end{aligned} \tag{4.48}$$

Similarly to (4.43), we implemented it using a 2-layer feedforward neural network:

$$\varphi_3^2 \circ \varphi_3^1 \,:\, (\mathbb{R}^{n_E})^L \to \mathbb{R}^{n_Y}.$$

The map

$$\begin{aligned} \varphi_3^1 \,:\, (\mathbb{R}^{n_E})^{L_2} &\to \mathbb{R}^{n_E} \\ \{\mathbf{x}^{2,(m)}\}_{m=1,2,\dots,L_2} &\mapsto \mathbf{x}^3 = \sum_{m=1}^{L_2} \theta^{(m)}_{\mathbf{w}_3^1} \mathbf{x}^{2,(m)} \end{aligned} \tag{4.49}$$

is an attention mechanism parametrised by $\mathbf{w}_3^1$. The second map is instead a layer map using the linear activation function (3.28):

$$\begin{aligned} \varphi_3^2 \,:\, \mathbb{R}^{n_E} &\to \mathbb{R}^{n_Y} \\ \mathbf{x}^3 &\mapsto \tilde{\mathbf{x}} = \mathbf{w}_3^2 \mathbf{x}^3 + \mathbf{b}_3^2 \,; \end{aligned}$$

To transform $\tilde{\mathbf{x}}$ in a probability distribution over Y, we applied the softmax function (4.40) so that the classification problem is turned into a regression over the n_Y-dimensional simplex (this is a common practice when solving classification problems using ANNs). We trained the trajectory classifier (4.48) using the **cross entropy** (CE) loss upon the softmax function and applying the Adam optimisation algorithm with a learning rate of 0.001 and a batch size of 1 for 350 epochs. Observe that the application of φ_3^1 returns a probability distribution over the compressed trajectory:

$$p_{\phi_3^1}(m),\ m = 1, 2, \dots, L_2. \tag{4.50}$$

We can define a PMF over the product space (4.33) as the product

$$p(l, i) = p_L(l) * p_l(i)\,, l = 1, 2, \dots, L\,, i = 1, 2, \dots, n_X\,,$$

where

$$p_L \,:\, \{1, 2, \dots, L\} \to [0, 1]$$

is a fixed **trajectory PMF** and p_l belongs to the collection of **channels PMFs**

$$\{p_l \,:\, \{1, 2, \dots, n_X\} \to [0, 1]\}_{l=1,2,\dots,L}\,.$$

The trajectory PMF can be obtained combining the PMFs (4.47) and (4.50) associated with the attention mechanisms. To derive a channels PMF p_l for each waypoint, we used the concept of **saliency maps** [101]. Saliency maps were developed to better understand the workings of convolutional neural networks by visualising the importance (i.e., the saliency) of their

hidden representations with respect to some *saliency target* d. The technique amounts to computing the gradient of a specified loss function with respect to the representations of interest $\mathbf{x}^\ell$,

$$\mathbf{g} = \nabla_{\mathbf{x}^\ell} d\,,$$

and turn this gradient into a histogram using its $L1$-norm:

$$p(\tilde{i}) = \frac{|g_{\tilde{i}}|}{\sum_{i=1}^{n_\ell} |g_i)|}\,, \tilde{i} = 1, 2, \ldots, n_\ell\,.$$

In our scenario, we considered the encoder of the trajectory condenser (4.37) as the target network and defined the saliency target

$$d(\hat{\mathbf{x}}^1, \varphi^1(\mathbf{x}^0)) \coloneqq \|\hat{\mathbf{x}}^1 - \varphi^1(\mathbf{x}^0)\|^2\,. \tag{4.51}$$

For each input $\mathbf{x}^{0,(l)}$ we defined $\hat{\mathbf{x}}^1 = \mathbf{x}^{1,(l+1)}$ as its target vector so that the computed saliency gradient is

$$\mathbf{g}^{(l)} = \nabla_{\mathbf{x}^0} d(\mathbf{x}^{1,(l+1)}, \mathbf{x}^{1,(l)} = \varphi^1(\mathbf{x}^{0,(l)}))\,.$$

We selected this expression since it indicates the channels whose changes are the most important during the movement of the automobile from the l-th waypoint to the $l+1$-th waypoint. An example heatmap is depicted in Figure 4.2.

To validate the system and check whether it was actually detecting useful features, we proceeded as follows. First, for each score type (i.e., for each property assessed by the drivers) we trained a different trajectory classifier (4.48) using a 10-fold cross validation [102]. Let us denote by $N_k \equiv S$ the size of each fold and by $\{\mathfrak{X}^{(k,1)}, \mathfrak{X}^{(k,2)}, \ldots, \mathfrak{X}^{(k,S)}\}$ the set of records in the validation set of the k-th fold. For each record in this set, denote by $p^{(k,i_k)}(l, i)$ the corresponding PMF computed by the model. We define the **channels saliency** to be the PMF

$$\begin{aligned} p\,:\,\{1, 2, \ldots, n_X\} &\to [0, 1] \\ i &\mapsto \frac{1}{K}\sum_{k=1}^{K}\frac{1}{S}\sum_{i_k=1}^{S}\sum_{l=1}^{L}\pi^{(k,i_k)}(l, i)\,. \end{aligned} \tag{4.52}$$

The system assigned importance to different channels in a similar way as professional test drivers do, providing evidence that it is not only able to predict the evaluation of an automobile, but can focus the attention of engineers on the relevant properties of the problem. The results of this experiment are reported in Figure 4.3.

Artificial neural networks are not a *magic bullet*: they require careful design and much experimentation in order to be effective at the target task. The choice of an appropriate loss functions allows extracting features which, by leveraging the modular structure of an ANN, can be transferred to different tasks. Therefore, neural networks are ideal to build modular machine learning systems, which can save practitioners valuable modelling and training time.

Convolutional neural networks, a broad sub-class of artificial neural networks designed to mimic the functioning of mammals' visual cortex, are succesful examples of hierarchical (hence, modular) learning systems. For example, consider a convolutional neural network trained on an image classification task. Suppose we now want to solve an object detection task: we can detach the last layers of the network we have trained, transfer the remaining layers to the new object detection system, add a stack of new layers designed to solve the object detection task and define a new loss function to communicate to our system the new learning target. This adjustment process is known as *transfer learning*, and the application of a learning algorithm (e.g., backpropagation with gradient descent) to the new system is known as a *fine-tuning* of the older system on the new task.

In Section 4.3 we described a model designed to find the objective reasons behind subjective evaluations of automobiles. An autoencoder is explicitly designed to be the composition of two parts (the encoder and the decoder) which can be used independently: once such an ANN has been trained, the two functions can be decoupled. The trajectory analyser of the system was trained to predict the driver's responses, but we designed it so that we could reuse only the features extracted by the attention mechanism. The hierarchical structure we enforced on the system allowed transferring the extracted features to different product development processes.

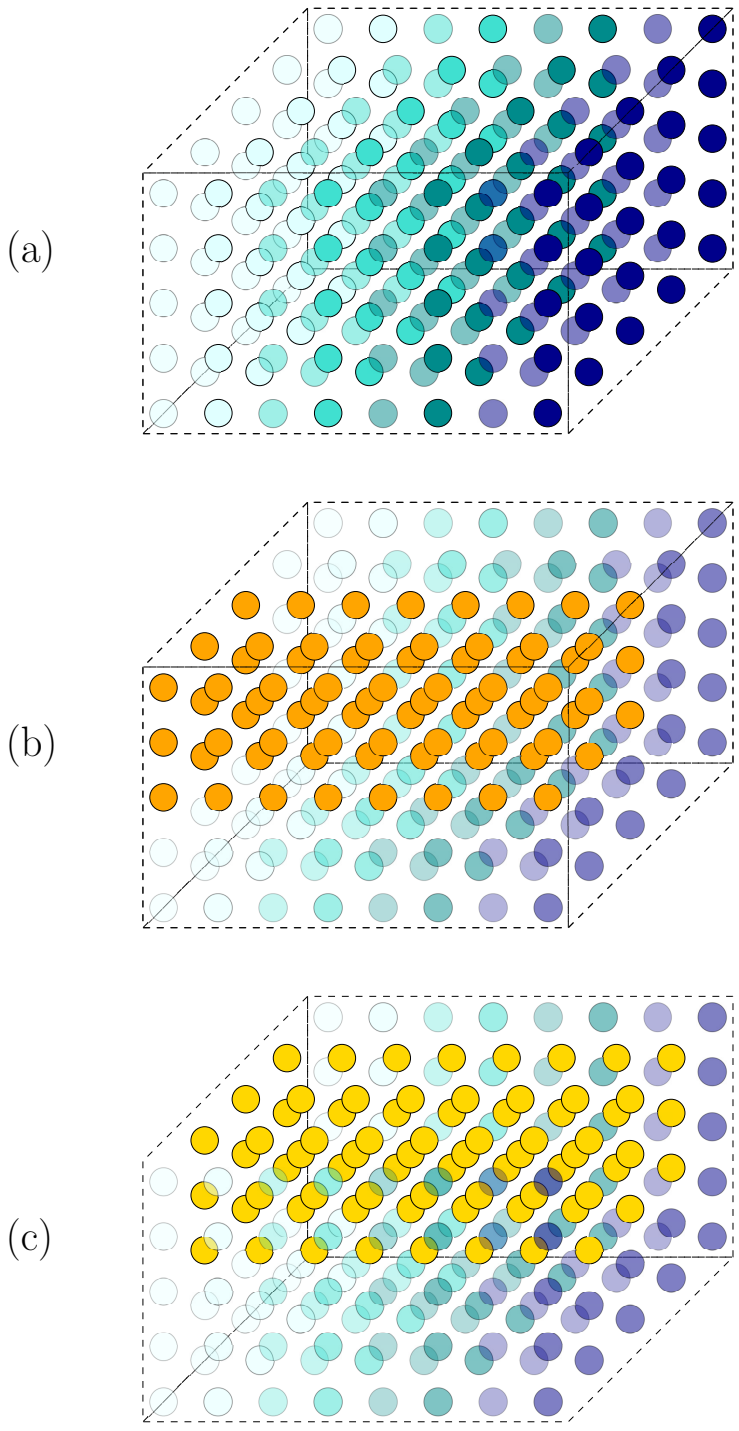


Figure 4.1: An example layer of a convolutional neural network with depth $C = 8$, height $H = 5$ and width $W = 5$: (a) planes of neurons are marked by different shades, (b) an example column with spatial dimensions $F^H = F^W = 3$ and (c) another column, obtained by shifting the previous one by one step along the width dimension. Note that the two columns contain common neurons.

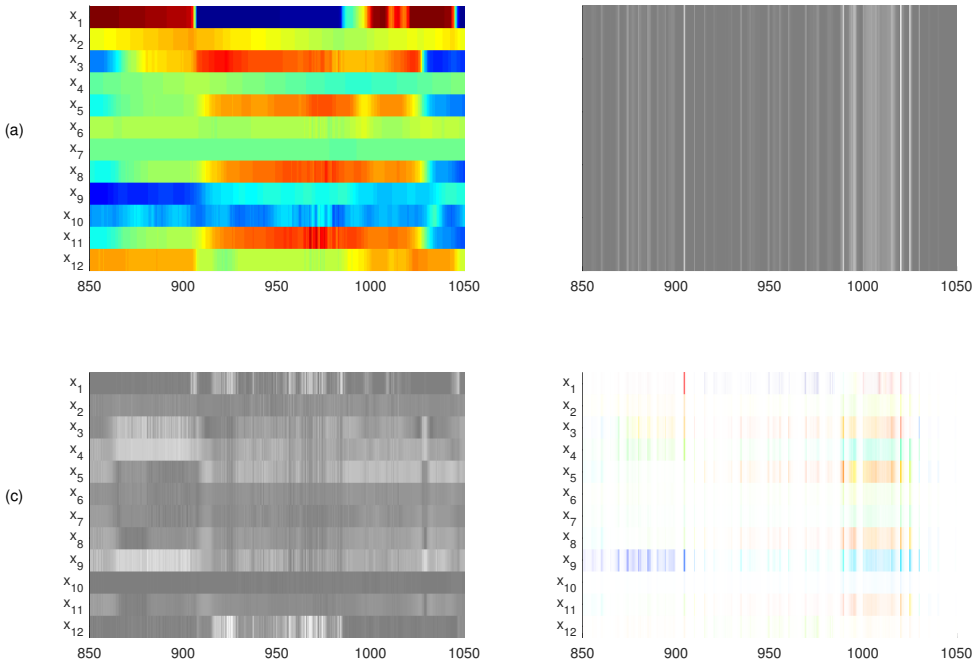


Figure 4.2: An example of the action of our model. (a) The original data is composed by measurements from 12 different channels sampled on 400 waypoints (different colour levels represent different values of the measurements); (b) the probability distribution assigned by the attention mechanisms to the analysed trajectory (whiter values represent higher probabilities) and (c) the probabilities assigned by the saliency maps to the measurement vectors at each waypoint are combined to generate (d) a heatmap which allows identifying the portions of the trajectory which influenced the system's predictions.

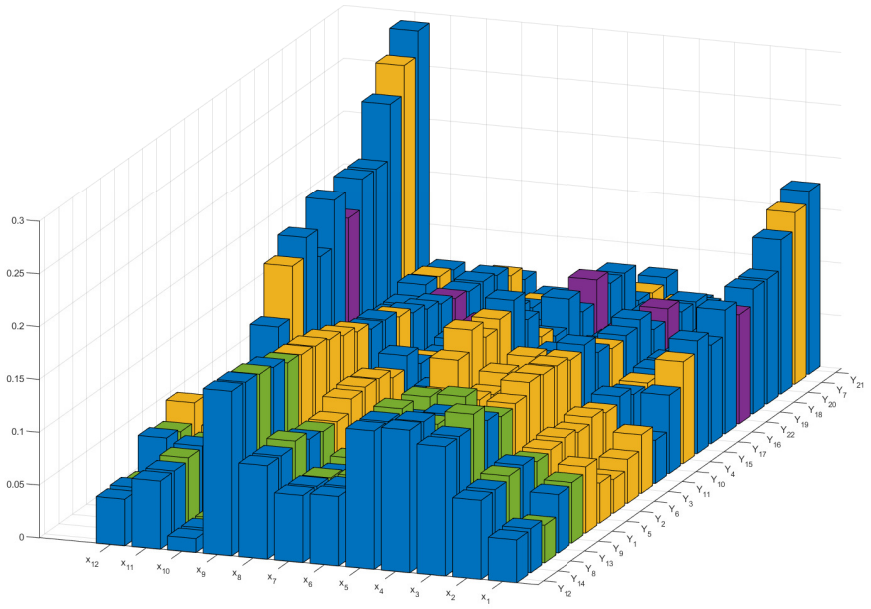


Figure 4.3: Channels importance for different score types. The model assigns more probability mass to channels related to longitudal acceleration (X_1 and X_{12}) when evaluating properties concerning the steering wheel (for example, score class Y_{21}); it assigns instead more probability mass to channels related to vehicle alignment (X_5 and X_9) when evaluating the performances on turns (for example, score class Y_{12}).

Chapter 5

Quantized neural networks

At the time of writing, electronic devices have reached characteristics of miniaturisation and production cost-efficiency that allow incorporating them into mobile phones, household appliances and industrial machinery, adding functionalities with respect to older systems. The amount of available data makes it possible to improve these functionalities even further by exploiting the statistical patterns inherent to the environments where the systems are deployed. For example, mobile phones incorporate automatic speech recognition software that allows users to issue vocal commands without using the keyboard/screen interface; light sensors capable of automatically detecting the presence of people in a room allow switching lamps on and off more efficiently; sensors installed on industrial machinery allow monitoring its efficiency and programming its maintenance more effectively. Machine learning systems are becoming critical components to solve these statistical tasks. The design of the computers embedded in these specific systems usually trades processing speed, physical memory, general purpose programmability and energy budget against cost-effectiveness and the advantage of being physically closer to the data sources (*edge computing*). These small computers are therefore resource-constrained with respect to computer clusters and desktop computers. The use of machine learning systems on resource-constrained computers is a challenging taks.

In this chapter we will introduce *quantized neural networks*, artificial neural networks whose weights and representations are constrained to take values in finite spaces. After reviewing the related literature, we will analyze their approximation properties. Since the transition from continuous to finite spaces wipes away the differentiability property on which the backpropagation algorithm is based, we will use some probabilistic arguments to show how *approximate differentiability* can be retrieved. We will then describe the *additive noise annealing* algorithm that we developed on top of this idea. Fi-

nally, we will present the *QuantLab* framework we programmed to benchmark the algorithm on image classification tasks and report our results.

5.1 Related research

In Section 3.3 we showed that ANNs are general function approximators and that they have remarkable computational properties. These characteristics have made **deep learning** (DL) [103, 104, 105] the backbone of present-day artificial intelligence. For example, specific ANN topologies have been developed to perform object detection [106, 107], autonomous navigation [108, 109] or decision making in both complete and incomplete information games [110, 111]. Designed to optimise statistical fit metrics, these models often require billions of parameters and multiply-accumulate (MAC) operations to perform inference on a single data point. These properties translate into prohibitive latency and energy requirements for resource-constrained computers. Real-time applications must satisfy specific latency constraints [112]. Energy-aware applications must satisfy limited peak-power constraints or average energy consumption constraints [113]. Some applications are both latency-constrained and energy-constrained [114]. Therefore, research on constrained machine learning systems has become a very active field.

In particular, constrained deep learning is concerned with the development of systems whose programs do not exceed given latency constraints and which have a limited memory footprint, a property that depends on both the number of operands and the precision of their hardware representations. In Section 4.2 we briefly discussed some of the techniques introduced by the research on convolutional neural network topologies: these techniques have allowed designing models with less parameters and operations. Hardware-related optimisations have recently attracted much interest [115, 116, 117], evolving into the field of **quantized neural networks** (QNNs). QNNs use low-bitwidth operands and corresponding hardware instructions to reduce the models' memory footprints, their execution latency and their energy consumption.

The numerical variables that represent the operands of an ANN can be partitioned in parameters and representations. Parameters are further partitioned in weights and biases. We observed in Section 3.2 that the weights of ANNs have taken on values in continuous spaces since the introduction of the delta rule [35]. The transition from quantized activation functions to differentiable activation functions happend later [42], and opened the way to the backpropagation algorithm [43]. The commonly used activation functions sigmoid (3.24), hyperbolic tangent (3.25) and ReLU (4.27) have continuous

real-valued codomains, which by definition yield continuous representations. We call **quantization set** or *codebook* any finite non-empty set Q of real numbers called **quantization levels**. A *weights-Q-quantized* ANN is an ANN whose weights take values in Q. An *activations-Q-quantized* ANN is an ANN whose activation functions have Q as their codomain. With these simple definitions, the problem of ANNs quantization can be defined as the investigation of neural networks models that are weights-Q-quantized, activations-Q-quantized, or both.

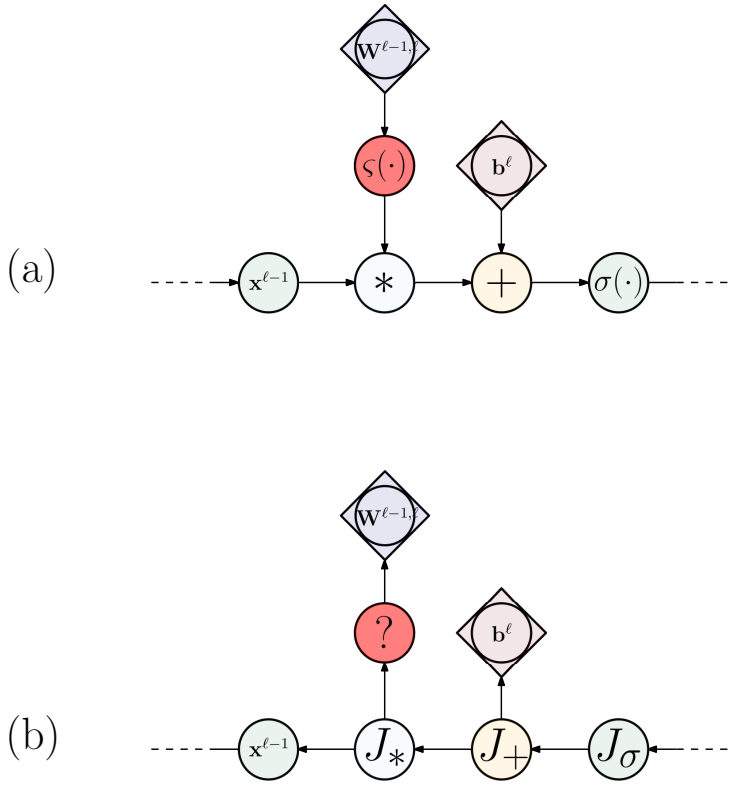


Figure 5.1: Consider the compuational graphs of Figure 3.2a1 and Figure 3.2a2, and their details in Figure 3.3. Training the corresponding neural network using the binary connect algorithm requires adding non-differentiable functions in (a) the forward graph (red node) between each weights matrix (blue node) and the corresponding linear map (light blue node). This yields (b) a backward graph which has undefined operations. Still, since the weights are leaves of the graph of Figure 3.2a2, the gradients can still be exactly computed for all the layers and the biases.

Examples of gradient-based learning algorithms for weights-Q-quantized ANNs are **binary connect** (BC) [118], **incremental network quantiza-**

tion (INQ) [119] and the **alternating direction method of multipliers for neural networks** (ADMM-NN) [120]. BC is given a network topology initialised with continuous **shadow weights**. Before inference is performed, the **sign function**

$$\begin{aligned} \zeta \,:\, \mathbb{R} &\to \{-1,+1\} \\ w &\mapsto H_0^{\{-1,+1\}}(w)\,, \end{aligned} \tag{5.1}$$

(which can be thought of as a specific case of the generalised Heaviside (3.19)) is applied to these shadow weights and returns binary values. This amounts to inserting a node between each weight tensor and the corresponding linear map in the computational graph associated with the network, as in Figure 5.1. Note that such a node is not differentiable. Data is then propagated forward using the outputs of these quantization operations as weights, the loss is evaluated, and error signals are computed using the backpropagation algorithm. Since the quantization operations are not differentiable, it is in principle impossible to define the gradient of the loss functional with respect to the shadow weights. To circumvent this issue, the error signals directed to the quantization operations are copied and applied to the shadow weights directly, passing through the sign functions unaltered. This rule is called the **straight-through estimator** (STE), since the error signals which are obtained by *passing directly through* the quantization operations are intended to be estimates (in the statistical sense) of the correct error signals. Let

$$\begin{aligned} htanh \,:\, \mathbb{R} &\to [-1,1] \\ x &\mapsto \begin{cases} -1, & \text{if } x \in (-\infty,-1] \\ 0, & \text{if } x \in (-1,1] \\ 1, & \text{if } x \in (1,+\infty) \end{cases} \end{aligned} \tag{5.2}$$

denote the piece-wise linear function called **hard hyperbolic tangent**. STE replaces the exact distributional derivative $D\zeta = 2\delta_0$ of the sign function with the derivative of the hard hyperbolic tangent:

$$D\zeta(x) \approx \frac{d}{dx} htanh(x) \begin{cases} 0, & \text{if } x \notin [-1,1] \\ 1, & \text{if } x \in (-1,1)\,. \end{cases} \tag{5.3}$$

INQ is given a network topology initialised with full-precision weights. The algorithm defines a quantization set Q, a set $\{1,2,\ldots,T\}$ of quantization time steps and partitions the weights set $\widehat{\mathbf{W}}$ into a corresponding number of subsets $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots, \mathbf{W}^{(T)}$. INQ then starts training the full-precision model. When a quantization time step $t \in \{1,2,\ldots,T\}$ is reached, the

corresponding weights subset $\mathbf{W}^{(t)}$ is *frozen* (i.e., its elements are projected onto the nearest quantization levels in Q and never updated again). When the last quantization time step $t = T$ is reached, all the weights of the network take values in Q and the resulting program is thus weight-Q-quantized. ADMM-NN splits the learning algorithm in two stages. We call the network to be quantized the *target network*. The target network is initialised with quantized weights from the quantization set Q. Then, the first stage of ADMM-NN performs ordinary gradient descent but including an *elastic constraint* that tries to keep the continuous-valued solution close to the starting (quantized) values. Then, the second stage projects the solution found onto the quantization set.

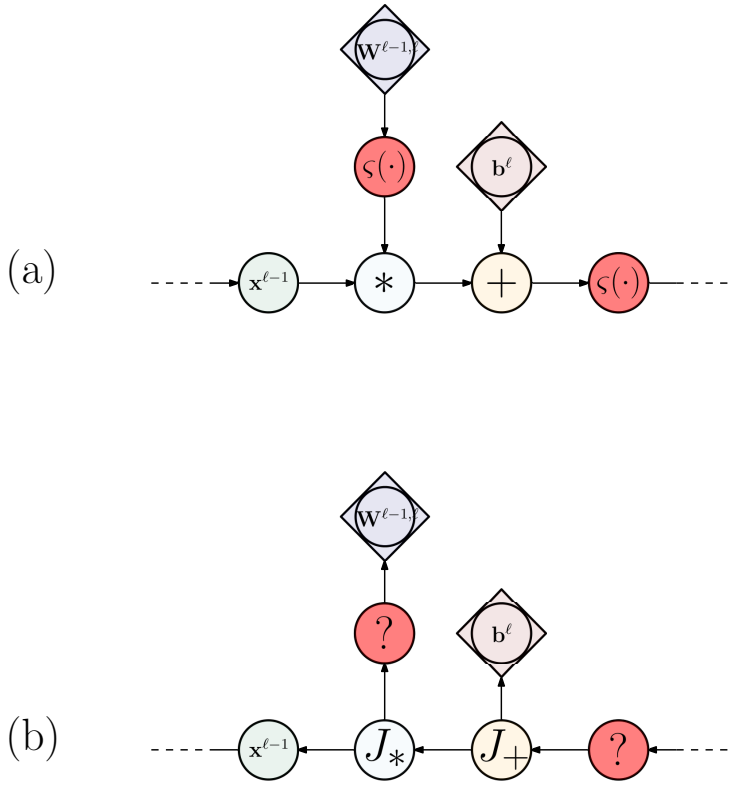


Figure 5.2: Detail of the computational graphs of a QNN which is both weight-Q-quantized and activation-Q-quantized. With respect to the graphs in Figure 5.1, the problem of non-differentiability is worsened when also (a) the activation functions (red node) are not differentiable, since (b) these nodes are dependencies for all the operations of preceeding layers.

As an extension of BC, gradient-based training of activations-Q-quantized DNNs was first investigated by Hubara *et al.* [121, 122]. They used the sign

function also as the activation function in all the models they tested, making the respective computational graphs non-differentiable even in the activation function operations. Example computational graphs of such networks are depicted in Figure 5.2. Nevertheless, STE provided sufficiently good error signals to achieve effective results. Building on STE, many algorithms have been developed. **XNOR-Networks** [123] tried to preserve inner products between full-precision weights and representations by replacing both with optimal binary counterparts. More formally, consider a representations vector $\mathbf{x}^{\ell-1}$ and a weight column vector $\mathbf{w}^{\ell}_{i_\ell}$ taken from the weight matrix $\mathbf{W}^{\ell}$. XNOR-Networks look for parameters $c_{\mathbf{x}^{\ell-1}}, c_{\mathbf{w}^{\ell}_{i_\ell}}$ such that the approximation

$$\mathbf{x}^{\ell-1}\mathbf{w}^{\ell}_{i_\ell} \approx c_{\mathbf{x}^{\ell-1}}\zeta(\mathbf{x}^{\ell-1})c_{\mathbf{w}^{\ell}_{i_\ell}}\zeta(\mathbf{w}^{\ell}_{i_\ell})\,, \tag{5.4}$$

where ζ is the sign function (5.1) applied component-wise to the vectors, is optimal. The optimality is intended in the L^2-norm sense:

$$\min_{c_{\mathbf{x}^{\ell-1}}, c_{\mathbf{w}^{\ell}_{i_\ell}}} \|\mathbf{x}^{\ell-1}\mathbf{w}^{\ell}_{i_\ell} - c_{\mathbf{x}^{\ell-1}}\zeta(\mathbf{x}^{\ell-1})c_{\mathbf{w}^{\ell}_{i_\ell}}\zeta(\mathbf{w}^{\ell}_{i_\ell})\|_2\,. \tag{5.5}$$

The minimiser is given by

$$\begin{aligned} c_{\mathbf{x}^{\ell-1}} &= \|\mathbf{x}^{\ell-1}\|_1\,, \\ c_{\mathbf{w}^{\ell}_{i_\ell}} &= \|\mathbf{w}^{\ell}_{i_\ell}\|_1\,, \end{aligned}$$

where $\|\cdot\|_1$ denotes the usual L^1 norm on $\mathbb{R}^{n_{\ell-1}}$. Using the bi-linearity of the dot product we can rewrite the second term in (5.4) as

$$c_{\mathbf{x}^{\ell-1}}\zeta(\mathbf{x}^{\ell-1})c_{\mathbf{w}^{\ell}_{i_\ell}}\zeta(\mathbf{w}^{\ell}_{i_\ell}) = c_{\mathbf{x}^{\ell-1}}c_{\mathbf{w}^{\ell}_{i_\ell}}\zeta(\mathbf{x}^{\ell-1})\zeta(\mathbf{w}^{\ell}_{i_\ell})\,. \tag{5.6}$$

This is a trivial property from a modelling perspective, but its impact on the hardware efficiency is noticeable. In fact, a dot product between two floating point vectors with $n_{\ell-1}$ components requires $n_{\ell-1}$ floating point multiplications and $n_{\ell-1}-1$ floating point additions. Instead, (5.6) requires $2(n_{\ell-1}-1)$ floating point additions to compute the coefficient $c_{\mathbf{x}^{\ell-1}}, c_{\mathbf{w}^{\ell}_{i_\ell}}$ and two floating point multiplications, executing a dot product between binary vectors using the efficient `xnor` and `popcount` hardware operations. Therefore, networks trained with the XNOR-Networks algorithm do not use quantized operands (since neither $c_{\mathbf{x}^{\ell-1}}\zeta(\mathbf{x}^{\ell-1})$ nor $c_{\mathbf{w}^{\ell}_{i_\ell}}\zeta(\mathbf{w}^{\ell}_{i_\ell})$ are), but constrain them to assume a specific form so that the resulting hardware operations are efficient. Error signals are backpropagated through the sign functions ζ using the STE. **Accurate Binary Convolutional Networks** (ABC-Networks) [124] tried to

decompose filters and representations onto basis of binary vectors:

$$\tilde{\mathbf{x}}^{\ell-1} \approx \sum_{j=1}^{J} c^{j}_{\mathbf{x}^{\ell-1}} H^{\{-1,+1\}}_{\theta^{j}_{\mathbf{x}^{\ell-1}}}(\mathbf{x}^{\ell-1})\,, \tag{5.7}$$

$$\tilde{\mathbf{W}}^{\ell} \approx \sum_{k=1}^{K} c^{k}_{\mathbf{W}^{\ell}} H^{\{-1,+1\}}_{\theta^{k}_{\mathbf{W}^{\ell}}}(\mathbf{W}^{\ell})\,. \tag{5.8}$$

The thresholds $\theta^{j}_{\mathbf{x}^{\ell-1}}, \theta^{k}_{\mathbf{W}^{\ell}}$ and the parameters $c^{j}_{\mathbf{x}^{\ell-1}}$ are learnt using back-propagation and STE. The parameters $c^{k}_{\mathbf{W}^{\ell}}$ are optimised by solving

$$\min_{c^{k}_{\mathbf{W}^{\ell}}} \|\mathbf{W}^{\ell} - \sum_{k=1}^{K} c^{k}_{W^{\ell}} H^{\{-1,+1\}}_{\theta^{k}_{\mathbf{W}^{\ell}}}(\mathbf{W}^{\ell})\|_2 \tag{5.9}$$

every time that the functions $H^{\{-1,+1\}}_{\theta^{k}_{\mathbf{W}^{\ell}}}$ are updated. Without loss of generality, we can suppose that the thresholds $\theta^{j}_{\mathbf{x}^{\ell-1}}$ (equivalently, $\theta^{k}_{\mathbf{W}^{\ell}}$) are sorted in increasing order. Therefore, the effect of the approximations (5.7) and (5.8) is to pass each element in the tensors $\mathbf{x}^{\ell-1}$ and $\mathbf{W}^{\ell}$ through multi-step functions

$$\zeta(x) = -\sum_{j=1}^{J} c^{j}_{\mathbf{x}^{\ell-1}} + \sum_{j \,|\, x \geq \theta^{j}_{\mathbf{x}^{\ell-1}}} 2c^{j}_{\mathbf{x}^{\ell-1}}\,,$$

$$\zeta(w) = -\sum_{k=1}^{K} c^{k}_{\mathbf{W}^{\ell}} + \sum_{k \,|\, x \geq \theta^{k}_{\mathbf{W}^{\ell}}} 2c^{k}_{\mathbf{W}^{\ell}}\,,$$

respectively. Analogously to XNOR-Networks, ABC-Networks use the bi-linearity of the dot product to replace full-precision dot products with sums of JK binary dot products:

$$\begin{aligned}
\mathbf{x}^{\ell-1}\mathbf{w}^{\ell}_{i_\ell} &= \left(\sum_{j=1}^{J} c^{j}_{\mathbf{x}^{\ell-1}} H^{\{-1,+1\}}_{\theta^{j}_{\mathbf{x}^{\ell-1}}}(\mathbf{x}^{\ell-1})\right)\left(\sum_{k=1}^{K} c^{k}_{\mathbf{W}^{\ell}} H^{\{-1,+1\}}_{\theta^{k}_{\mathbf{W}^{\ell}}}(\mathbf{w}^{\ell}_{i_\ell})\right) \\
&= \sum_{j=1}^{J}\sum_{k=1}^{K} c^{j}_{\mathbf{x}^{\ell-1}} c^{k}_{\mathbf{W}^{\ell}} \left(H^{\{-1,+1\}}_{\theta^{j}_{\mathbf{x}^{\ell-1}}}(\mathbf{x}^{\ell-1}) H^{\{-1,+1\}}_{\theta^{k}_{\mathbf{W}^{\ell}}}(\mathbf{w}^{\ell}_{i_\ell})\right)\,.
\end{aligned}$$

We observe here that the error signal is passed through the functions $H^{\{-1,+1\}}_{\theta^{j}_{\mathbf{x}^{\ell-1}}}(\mathbf{x}^{\ell-1})$

using the derivative of the piece-wise linear function

$$f_{\theta^j_{\mathbf{x}^{\ell-1}}} : \mathbb{R} \to [-1,1]$$
$$x \mapsto \begin{cases} -1, & \text{if } x \in (-\infty, \theta^j_{\mathbf{x}^{\ell-1}} - 0.5) \\ 2(x - \theta^j_{\mathbf{x}^{\ell-1}}), & \text{if } x \in [\theta^j_{\mathbf{x}^{\ell-1}} - 0.5, \theta^j_{\mathbf{x}^{\ell-1}} + 0.5) \\ 1, & \text{if } x \in [\theta^j_{\mathbf{x}^{\ell-1}} + 0.5, +\infty)\,, \end{cases}$$

which is different from the hard hyperbolic tangent (5.2). Since ABC-Networks achieves good results even using a different estimator than the STE, this suggests that the specific form (5.3) is not essential to the success of the algorithm. **Binary-real networks** (Bi-real networks) [125] proposed to preserve information by computing

$$\mathbf{x}^\ell = \zeta(\mathbf{x}^{\ell-2}\mathbf{W}^{\ell-1} + \mathbf{x}^{\ell-1}\mathbf{W}^\ell)$$

at every layer instead of $\mathbf{x}^\ell = \zeta(\mathbf{x}^{\ell-1}\mathbf{W}^\ell)$. Moreover, similarly to ABC-Networks they proposed to use a different estimator for the gradients of non differentiable functions, further validating the hypothesis that the specific form of the STE is not essential. Taking the concept of XNOR-Networks and ABC-Networks even further, **group-Networks** [126, 127] partitioned a network (3.13) into *groups* of consecutive layers, and modelled each group using binary weights and binary activation functions. Differently from XNOR-Networks and ABC-Networks, Group-Networks expand each *group* of layers into an ensemble of sub-networks whose operations are actually binary, with no continuous parameters involved except for biases. **Parametrised clipping activation/statistics-aware weight binning** (PACT/SAWB) [128] uses different quantization strategies for weights and activations. During the learning algorithm, before evaluating the forward computational graph, the statistics of the shadow weights are used to compute the quantization levels (i.e., the *bins*) dynamically; after the evaluation of the backward computational graph, the error signal is used to update the shadow weights directly. The activation functions are instead parametric and learnt for each layer. They take the form

$$\zeta^\ell(x) = \begin{cases} 0, & \text{if } x \in (-\infty, 0 - \Delta_b(a^\ell)/2] \\ \left\lfloor \frac{x}{\Delta_b(a^\ell)} \right\rceil \Delta_b(a^\ell), & \text{if } x \in (0 - \Delta_b(a^\ell)/2, a^\ell + \Delta_b(a^\ell)/2] \\ a^\ell, & \text{if } x \in [a^\ell + \Delta_b(a^\ell)/2, +\infty)\,, \end{cases} \tag{5.10}$$

where $\lfloor \cdot \rceil$ is the rounding operation, b is the fixed number of bits used to encode the number of quantization levels (which are therefore 2^b), a^ℓ is the

learnable parameter and

$$\Delta_b(a^\ell) \coloneqq \alpha^\ell/(2^b - 1)$$

is the size of each quantization interval of the function's domain. The advantage of using equally spaced quantization levels is that the corresponding dot products can be implemented using fixed-point arithmetic as opposed to floating-point arithmetic [129]. Fixed-point instructions are essentially integer instructions, and can be implemented more efficiently than floating-point instructions on specialised hardware accelerators.

Recently, partially gradient-free learning algorithms for QNNs have questioned the necessity for continuous shadow weights. The **method of successive approximations** (MSA) [130] trained weights-Q-quantized DNNs leveraging Pontryagin's maximum principle [131]. The **gated XNOR networks** (GXNOR-Nets) algorithm [132] was developed to train systems which were both weights-Q-quantized and activations-Q-quantized. It computes gradients through ternary activation functions applying yet another variant of STE, but uses this information to implement weights updates as probabilistic state transitions in a discrete weights space. The main obstacle to the adoption of gradient-free learning algorithms for QNNs is that these are usually not efficiently implementable in the most popular ANNs frameworks such as TensorFlow and PyTorch. For this reason, we want to develop a full gradient-based learning algorithm to train QNNs; therefore, we will not delve further into the details of MSA and GXNOR-Nets.

The transition from continuous to finite spaces wipes out the differentiability property on which backpropagation is based. In this context, the learning problem is converted from a numerical optimisation problem into a discrete optimisation one, where the state space is finite but very large. Gradient-free optimisation algorithms such as integer programming or Monte Carlo methods are usually unfeasible due to the combinational number of configurations. Gradient-based optimisation has been brought back into the picture by the STE and its variants. Nonetheless, since it is formally incorrect to compute gradients of discontinuous (and thus non-differentiable) functions, the reason for this success is still unclear. In Section 5.3 we propose a probabilistic model based on two arguments. First, the stochastic nature of the quantization process [118, 121, 122]. Second, the possibility of retrieving quantized functions as the limit of some annealing processes on classes of differentiable, parametric functions [133]. We observe that, at the same time as we developed this idea, similar concepts have appeared in the literature. Some of them use only the annealing argument [134, 135]. A second approach noted that additive noise on the parameters makes the

cumulative distribution function (CDF) differentiable, and uses backpropagation to update the probability mass of the quantiles corresponding to the quantization values [136].

5.2 Approximation properties of QNNs

Let K denote a positive integer and

$$Q = \{q_0 < q_1 < \cdots < q_K\} \subset \mathbb{R} \tag{5.11}$$

denote a finite set of real numbers called the **quantization set**. We call the elements $q_k \in Q$ **quantization values**. We say that a matrix $\mathbf{w} \in \mathbb{R}^{m\times n}$ is Q-quantized if $w_{ij} \in Q$ for all i, j. Similarly, we say that a function $\sigma : \mathbb{R} \to \mathbb{R}$ is Q-quantized if its codomain $\sigma(\mathbb{R})$ is Q. We will more generally call **quantization function** any function that is Q-quantized for some Q. We say that a layer (3.12) is **weight-Q-quantized** if W^ℓ is Q-quantized. We say that the layer is **activation-Q-quantized** if σ^ℓ is Q-quantized. Then, the layer is said **Q-quantized** if W^ℓ and σ^ℓ are Q-quantized. Note that $\mathbf{b}^\ell$ is not required to be Q-quantized. We say that a network (3.13) is a **quantized neural network** if $L \geq 2$ and its layers φ^ℓ are Q-quantized for every $\ell = 1, \ldots, L-1$. Note that the last layer φ^L is not required to be Q-quantized. In particular, when $Q = \{-1, +1\}$ we say that Φ is a **binary neural network** (BNN). Similarly when $Q = \{-1, 0, +1\}$ we say that Φ is a **ternary neural network** (TNN).

Existing experimental research on QNNs has consistently shown accuracy gaps with respect to full-precision networks. We remarked in Section 3.3 that the classical approximation results for ANNs [44, 45] assume continuous-valued parameters. A natural doubt arises about whether QNNs can approximate the same function classes as more flexible DNNs.

Lemma 1. *Let $n_0 > 0$ be a given integer and let $I_1, I_2, \ldots, I_{n_0}$ be bounded intervals in $\mathbb{R}$. Let $P = I_1 \times I_2 \times \cdots \times I_{n_0} \subset \mathbb{R}^{n_0}$ be the hyperparallelepiped obtained as the cartesian product of these intervals. There exists a ternary FNN that represents the characteristic function $\chi_P(\mathbf{x}^0)$.*

Proof. P can be interpreted as the intersection of n_0-dimensional *hyperstripes*:

$$P = \cap_{i=1}^{n_0} \{\mathbf{x}^0 \,|\, \pi_i(\mathbf{x}^0) = x_i^0 \in I_i\}\,,$$

where $\pi_i : \mathbb{R}^{n_0} \to \mathbb{R}$ is the projection associated with the i-th component. In turn, these hyperstripes can be represented as intersections of half-spaces.

Depending on whether the extremes do or do not belong to the intervals, there are four possible cases:

$$\{\mathbf{x}^0 \,|\, x_i^0 \in I_i\} = \begin{cases} \{\mathbf{x}^0 \,|\, x_i^0 \geq p_i\} \cap \{\mathbf{x}^0 \,|\, x_i^0 \leq q_i\}, & \text{if } I_i = [p_i, q_i] \\ \{\mathbf{x}^0 \,|\, x_i^0 \geq p_i\} \cap \{\mathbf{x}^0 \,|\, x_i^0 < q_i\}, & \text{if } I_i = [p_i, q_i) \\ \{\mathbf{x}^0 \,|\, x_i^0 > p_i\} \cap \{\mathbf{x}^0 \,|\, x_i^0 \leq q_i\}, & \text{if } I_i = (p_i, q_i] \\ \{\mathbf{x}^0 \,|\, x_i^0 > p_i\} \cap \{\mathbf{x}^0 \,|\, x_i^0 < q_i\}, & \text{if } I_i = (p_i, q_i) \end{cases} \quad i = 1, 2, \ldots, n_0\,. \tag{5.12}$$

Each of the forms (5.12) can be rewritten in term of two equations:

$$\{\mathbf{x}^0 \,|\, x_i^0 \in I_i\} = \left\{ \begin{array}{l|l} \mathbf{x}^0 & \left\{ \begin{array}{l} \sigma^+(1 \cdot x_i^0 + (-p_i)) = 1 \\ \sigma^+(-1 \cdot x_i^0 + q_i) = 1 \end{array} \right. \quad \text{if } I_i = [p_i, q_i] \\ \mathbf{x}^0 & \left\{ \begin{array}{l} \sigma^+(1 \cdot x_i^0 + (-p_i)) = 1 \\ \sigma^-(-1 \cdot x_i^0 + q_i) = 1 \end{array} \right. \quad \text{if } I_i = [p_i, q_i) \\ \mathbf{x}^0 & \left\{ \begin{array}{l} \sigma^-(1 \cdot x_i^0 + (-p_i)) = 1 \\ \sigma^+(-1 \cdot x_i^0 + q_i) = 1 \end{array} \right. \quad \text{if } I_i = (p_i, q_i] \\ \mathbf{x}^0 & \left\{ \begin{array}{l} \sigma^-(1 \cdot x_i^0 + (-p_i)) = 1 \\ \sigma^-(-1 \cdot x_i^0 + q_i) = 1 \end{array} \right. \quad \text{if } I_i = (p_i, q_i) \end{array} \right\} \quad i = 1, 2, \ldots, n_0, \tag{5.13}$$

where

$$\sigma^+(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases} \quad \text{and} \quad \sigma^-(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ 1, & \text{if } x > 0\,. \end{cases}$$

These functions are quantization functions. Let us define the network

$$\chi_{\mathbf{w},\mathbf{b}}(\mathbf{x}^0) \coloneqq \sigma^{(2)}\left(\sigma^{(1)}(\mathbf{w}^1 \cdot \mathbf{x}^0 + \mathbf{b}^1) \cdot \mathbf{w}^2 + b^2\right)\,,$$

where $\mathbf{w}^1 \in \{-1, 0, 1\}^{2n_0 \times n_0}, \mathbf{b}^1 \in \mathbb{R}^{2n_0}, \mathbf{w}^2 = \mathbf{1}_{2n_0}, b^2 = -2n_0, \sigma^{(2)} = \sigma^+$ and

$$\sigma^{(1)}(\mathbf{w}^1 \cdot \mathbf{x}^0 + \mathbf{b}^1) = \begin{pmatrix} \sigma_1^{(1)}\left(\sum_{j=1}^{n_0} w_{1j}^1 x_j^0 + b_1^1\right) \\ \sigma_2^{(1)}\left(\sum_{j=1}^{n_0} w_{2j}^1 x_j^0 + b_2^1\right) \\ \cdots \\ \sigma_{2n_0}^{(1)}\left(\sum_{j=1}^{n_0} w_{2n_0 j}^1 x_j^0 + b_{2n_0}^1\right) \end{pmatrix}^T,$$

where

$$\sigma_i^{(1)} = \begin{cases} \sigma^+, & \text{if } p_i \in I_i \\ \sigma^-, & \text{if } p_i \notin I_i \end{cases} \quad \text{and} \quad \sigma_{i+n_0}^{(1)} = \begin{cases} \sigma^+, & \text{if } q_i \in I_i \\ \sigma^-, & \text{if } q_i \notin I_i\,, \end{cases}$$

for $i = 1, \dots, n_0$. In particular, $\mathbf{w}^1, \mathbf{b}^1$ are defined by

$$w^1_{ij} = \begin{cases} \delta_{ij}, & \text{if } 1 \le i \le n_0 \\ -\delta_{(i-n_0)j}, & \text{if } n_0 + 1 \le i \le 2n_0 \end{cases} \quad \text{and} \quad b^1_i = \begin{cases} p_i, & \text{if } 1 \le i \le n_0 \\ q_{i-n_0}, & \text{if } n_0 + 1 \le i \le 2n_0 \,, \end{cases}$$

for $i = 1, \dots, 2n_0$ and $j = 1, \dots, n_0$, where δ_{ij} denotes the Kronecker delta. Let us observe that $\chi_{\mathbf{w},\mathbf{b}}(\mathbf{x}^0) = 1$ if and only if the argument of the outer activation $\sigma^{(2)}$ is non-negative, that is,

$$\sigma^{(1)}(\mathbf{w}^1 \cdot \mathbf{x}^0 + \mathbf{b}^1) \cdot \mathbf{w}^2 \ge -b^2 \,,$$

which is satisfied if and only if one has equality in the inequality above, which means

$$\sigma^{(1)}(\mathbf{w}^1 \cdot \mathbf{x}^0 + \mathbf{b}^1) = \mathbf{w}^2 \,.$$

The latter vectorial equation corresponds to satisfying all the systems

$$\begin{cases} \sigma^{(1)}_i \left(\sum_{j=1}^{n_0} w^1_{ij} x^0_j + b^1_i \right) = 1 \\ \sigma^{(1)}_{i+n_0} \left(\sum_{j=1}^{n_0} w^1_{(i+n_0)j} x^0_j + b^1_{i+n_0} \right) = 1 \,, \end{cases} \qquad \forall\, i = 1, \dots, n_0 \,,$$

which are equivalent, by the definitions of w^1_{ij}, b^1_i and $\sigma^{(1)}_i$, to the corresponding systems introduced in (5.13). This amounts to requiring that $\mathbf{x}^0 \in \{\mathbf{x}^0 \,|\, x^0_i \in I_i\}$, $\forall\, i = 1, \dots, n_0$, and thus that $\mathbf{x}^0 \in P$. The idea is to use the neurons in the first layer, with parameters $(\mathbf{w}^1_i, b^1_i)$, to test whether $\mathbf{x}^0$ belongs to the corresponding half-space. The single neuron in the second layer computes the intersection between all the half-spaces enclosing P. Hence $\chi_{\mathbf{w},\mathbf{b}}(\mathbf{x}^0) = \chi_P(\mathbf{x}^0)$. □

Theorem 4 (Uniform approximation by QNNs)**.** *Let $X^0 = [0, S]^{n_0} \subset \mathbb{R}^{n_0}$ and denote by $\mathrm{Lip}_\lambda(X^0)$ the class of bounded functions $f : X^0 \to \mathbb{R}$ with Lipschitz constant $\le \lambda$. For every $\epsilon > 0$ and $f \in \mathrm{Lip}_\lambda(X^0)$, there exists a network*

$$\Phi_{\hat{\mathbf{m}}} = \varphi_{\mathbf{m}^3} \circ \varphi_{\mathbf{m}^2} \circ \varphi_{\mathbf{m}^1} : X^0 \to \mathbb{R} \,,$$

such that $\sup_{\mathbf{x} \in X^0} |\Phi_{\hat{\mathbf{m}}}(\mathbf{x}) - f(\mathbf{x})| \le \epsilon$. The layer maps $\varphi_{\mathbf{m}^1}$ and $\varphi_{\mathbf{m}^2}$ take the form (3.12), with quantized weights $\mathbf{w}^1 \in \{-1, 0, +1\}^{2n_0 \times n_0}, \mathbf{w}^2 \in \{0, 1\}^{N \times n_0}$ (where $N = N(\epsilon)$) and $\{0, 1\}$-quantized activation functions. The layer $\varphi_{\mathbf{m}^3}$ is an affine map parametrised by continuous-valued parameters. The number of neurons required by $\Phi_{\hat{\mathbf{m}}}$ to reach the given degree of approximation ϵ is limited by $(2n_0 + 2)\lceil \frac{2\lambda\sqrt{n_0}S}{\epsilon} \rceil^{n_0}$.

Proof. First, we explicitly construct a ternary FNN that can represent a function f which is constant on hyper-parallelepipeds. Let N be a positive integer. Let $\{P_1, \dots, P_N\}$ be a family of closed hyper-parallelepipeds such that $X^0 = \bigcup_{s=1}^{N} P_s$ and $P_{s_1} \cap P_{s_2} = \emptyset$ (i.e., $\{P_s\}_{s=1,\dots N}$ is a partition of X^0). Set now $n_1 = 2n_0N$ and define $\mathbf{w}^1 = (\mathbf{w}_1^1, \dots, \mathbf{w}_N^1), \mathbf{b} = (\mathbf{b}_1^1, \dots, \mathbf{b}_N^1)$, where $\mathbf{w}_s^1 \in \{-1, 0, 1\}^{2n_0 \times n_0}$ and $\mathbf{b}_s^1 \in \mathbb{R}^{2n_0}$ for $s = 1, \dots, N$. Define

$$\begin{aligned} \varphi_{\mathbf{m}^1} \,:\, \mathbb{R}^{n_0} &\to \{0,1\}^{n^1} \\ \mathbf{x}^0 &\mapsto \left(\chi_{\mathbf{w}_1^1, \mathbf{b}_1^1}(\mathbf{x}^0), \dots, \chi_{\mathbf{w}_N^1, \mathbf{b}_N^1}(\mathbf{x}^0)\right) , \end{aligned}$$

where $\chi_{\mathbf{w}_s^1, \mathbf{b}_s^1}$ is the ternary network representation of the half-spaces enclosing χ_{P_s} (i.e., the first layer described in Lemma 1 but applied to every P_s in parallel). Now define

$$\begin{aligned} \varphi_{\mathbf{m}^2} \,:\, \{0,1\}^{n_1} &\to \{0,1\}^N \\ \mathbf{x}^1 &\mapsto \sigma(\mathbf{x}^1 \cdot \mathbf{w}^2 + b^2) , \end{aligned}$$

where $\mathbf{w}^2 \in \{0,1\}^{N \times n_1}$ and $\mathbf{b}^2 \in \mathbb{R}^N$ are such that

$$w_{sj}^2 = \begin{cases} 1, & \text{if } 2n_0(s-1) + 1 \le j \le 2n_0 s \\ 0, & \text{otherwise} \end{cases}$$

and

$$b_s^2 = -2n_0$$

for all $s = 1, \dots N$. The map $\varphi_{m^2} \circ \varphi_{m^1}$ thus measures the membership of a point $\mathbf{x}^0$ to all the hyper-parallelepipeds P_s. Since $\{P_s\}_{s=1,\dots N}$ partitions X^0, just one neuron of the second layer can be active at a time. Finally, for a given $\mathbf{w}^3 \in \mathbb{R}^N$ we define the affine map

$$\begin{aligned} \varphi_{\mathbf{m}^3} \,:\, \{0,1\}^N &\to \mathbb{R} \\ \mathbf{x}^2 &\mapsto \mathbf{x}^2 \cdot \mathbf{w}^3 . \end{aligned}$$

Finally, we set

$$\Phi_{\hat{\mathbf{m}}} = \varphi_{\mathbf{m}^3} \circ \varphi_{\mathbf{m}^2} \circ \varphi_{\mathbf{m}^1} . \tag{5.14}$$

We will now prove the uniform approximation properties of these ternary FNNs. Let f be an arbitrary, fixed function in $\mathrm{Lip}_\lambda(X^0)$. Let n be an integer that satisfies

$$n \ge \frac{2\lambda\sqrt{n_0}S}{\epsilon} ,$$

then choose $N = n^{n_0}$. Consider the family of hypercubes P_s with side length $\delta = S/n$, forming a partition $\{P_s\}_{s=1,\dots,N}$ of X^0. For every $s = 1, \dots, N$ we

can identify the hypercube P_s by the index tuple $(i^{s_0}, \dots, i^{s_{n_0-1}})$ whose n_0 components are the unique integers $i^{s_k} \in \{0, \dots, n_0 - 1\}$ such that

$$s - 1 = i^{s_0} + i^{s_1} n + i^{s_2} n^2 + \cdots + i^{s_{n_0-1}} n^{n_0-1} .$$

Then the hypercube P_s is given by

$$P_s = I_{s_0} \times \cdots \times I_{s_{n_0-1}}$$

where

$$I_{s_k} = \begin{cases} [i^{s_k}\delta, (i^{s_k}+1)\delta), & \text{if } 0 \leq s_k < n_0 - 1 \\ [i^{s_k}\delta, (i^{s_k}+1)\delta], & \text{if } s_k = n_0 - 1 . \end{cases}$$

Define w_s^3 as the integral average of f on P_s for each $s = 1, \dots, N$, so that $\mathbf{w}^3 = (w_1^3, \dots, w_N^3)$. Define $\Phi_{\hat{\mathbf{m}}}$ as above, but applying Lemma 1 with $P = P_s$ for $s = 1, \dots, N$. We are now left with showing that

$$|\Phi_{\hat{\mathbf{m}}}(\mathbf{x}^0) - f(\mathbf{x}^0)| \leq \epsilon \ \forall \mathbf{x}^0 \in X^0 . \tag{5.15}$$

Let $\mathbf{x}^0 \in X^0$ and $s \in \{1, \dots, N\}$ be such that $\mathbf{x}^0 \in P_s$. Then $\Phi_{\hat{\mathbf{m}}}(\mathbf{x}^0) = \varphi_{\mathbf{m}^3}(\varphi_{\hat{\mathbf{m}}^2}(\mathbf{x}^0)) = w_s^3 = f(\mathbf{x}_s^0)$ for some $\mathbf{x}_s^0 \in P_s$, hence by the Lipschitz property of f we obtain

$$|\Phi_{\hat{\mathbf{m}}}(\mathbf{x}^0) - f(\mathbf{x}^0)| = |f(\mathbf{x}_s^0) - f(\mathbf{x}^0)| \leq \lambda |\mathbf{x}_s^0 - \mathbf{x}^0| \leq L \operatorname{diam}(P_s) = \lambda \sqrt{n_0} S/n \leq \epsilon/2 < \epsilon . \tag{5.16}$$

Since (5.16) holds for every $\mathbf{x}^0 \in X^0$, we obtain (5.15), as wanted. □

Remarkably, since $\cup_\lambda \operatorname{Lip}_\lambda(X^0)$ is dense in $C^0(X^0)$, QNNs show approximation capabilities equivalent to those of continuous-valued networks. *The theorem implies that the accuracy gaps observed experimentally between QNNs and continuous-values DNNs are not intrinsic to quantized networks.* Notice that the last layer must be an affine map parametrised by continuous variables. This might seem as an odd constraint for a QNN (after all, we would prefer fully quantized models), but it is consistent with the models described in the related literature. For example, the last layers of the models described in [122] are compositions of batch normalisation transformations (3.58) with linear maps parametrised by quantized weights; since a batch normalisation corresponds to the composition of a translation with an axis-parallel scaling, the resulting composed maps are actually affine.

5.3 Regularising quantization functions

We now introduce a useful class of quantization functions. A **generalised Heaviside function** is the map:

$$H_\theta^{\{q_0,q_1\}}(x) = \begin{cases} q_0, & \text{if } x < \theta \\ q_1, & \text{if } x \geq \theta\,. \end{cases}$$

The standard Heaviside function is recovered when $q_0 = 0, q_1 = 1$ and $\theta = 0$; for simplicity, we will refer to it with the symbol $H(x)$. We observe that a generalised Heaviside can be expressed in terms of the canonical Heaviside by $H_\theta^{\{q_0,q_1\}}(x) = q_0 + \Delta q_1 H(x-\theta)$, where $\Delta q_1 = q_1 - q_0$. Let $\Theta = \{\theta_1 < \theta_2 < \cdots < \theta_K\} \subset \mathbb{R}$ be a finite ordered set of real thresholds. Let $Q = \{q_0 < q_1 < \cdots < q_K\} \subset \mathbb{R}$ be a quantization set. The **K-step function** with thresholds Θ and quantization set Q is the non-decreasing map $\sigma : \mathbb{R} \to Q$ defined as

$$\sigma(x) = q_0 + \sum_{k=1}^{K} \Delta q_k H(x - \theta_k)\,, \tag{5.17}$$

where $\Delta q_k = q_k - q_{k-1}$ are the *jumps* between consecutive quantization levels. The generic term **multi-step function** will denote a function of the form (5.17). We can use multi-step functions to describe QNNs. Suppose $\mathbf{w}^\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ is a real-valued matrix. We notice that (3.12) can be equivalently rewritten by introducing the evaluation of the identity function on top of the weights space:

$$\varphi_{\mathbf{m}^\ell}(\mathbf{x}^{\ell-1}) = \sigma^\ell\left(id(\mathbf{w}^\ell)\cdot\mathbf{x}^{\ell-1} + \mathbf{b}^\ell\right)\,.$$

If we now replace id with a weight quantization function ζ^ℓ of the form (5.17), and consider an activation function σ^ℓ also of the same form, the map

$$\varphi_{\mathbf{m}^\ell}(\mathbf{x}^{\ell-1}) = \sigma^\ell\left(\zeta^\ell(\mathbf{w}^\ell)\cdot\mathbf{x}^{\ell-1} + \mathbf{b}^\ell\right) \tag{5.18}$$

is clearly Q-quantized since both $\zeta^\ell(\mathbf{w}^\ell)$ and σ^ℓ are Q-quantized. Although the idea of a weight quantization function seems unusual when considering the common definition of parameters in neural networks, yet this model is consistent with many of the QNNs training algorithms based on STE [122, 123, 124, 126]. Replacing layer maps of the form (3.12) with layers of the form (5.18) in the map (3.14) yields a QNN.

The **supervised learning** problem [46] to be solved is the minimisation of the **loss functional**

$$\mathcal{L}_{g,\gamma}(\Phi_{\hat{\mathbf{m}}}) = \int_{X^0\times X^L} d(\Phi_{\hat{\mathbf{m}}}(\mathbf{x}^0), g(\mathbf{x}^0))\, d\gamma(\mathbf{x}^0)\,, \tag{5.19}$$

where $(\mathbf{x}^0, \mathbf{y} = g(\mathbf{x}^0))$ is a sampled observation that associates an input instance $\mathbf{x}^0$ with its label $\mathbf{y}$ (obtained through an unknown *oracle* function $g : X^0 \to X^L$), d is a differentiable non-negative function called the **loss function**, and γ is an unknown probability measure on X^0. In practical applications, g is known only on a finite data set $\{\mathbf{x}_t^0\}_{t=1,\dots,T}$, and the measure γ is approximated by the empirical measure $\tilde{\gamma}(\mathbf{x}) = \sum_{t=1}^{T} \delta_{\mathbf{x}_t^0}(\mathbf{x})/T$. By **training** of a FNN we mean any algorithm that aims at minimising $\mathcal{L}_{g,\gamma}(\Phi_{\hat{\mathbf{m}}})$ as a function of $\hat{\mathbf{m}} \in \widehat{M}$. For example, when $\Phi_{\hat{\mathbf{m}}}$ is differentiable with respect to its parameters $\hat{\mathbf{m}}$, the gradient $\nabla_{\hat{\mathbf{m}}}\mathcal{L}_{g,\gamma}(\Phi_{\hat{\mathbf{m}}})$ can be computed applying the chain rule (due to the compositional structure of $\Phi_{\hat{\mathbf{m}}}$) as in the backpropagation algorithm [43], and $\hat{\mathbf{m}}$ can be updated via gradient descent optimisation. However, if we plug an ANN (3.14) composed of non-differentiable building blocks (5.18) into (5.19), the chain rule can no longer be applied. A multi-step function (5.17) is not differentiable at the thresholds, since its (distributional) derivative is a weighted sum of Dirac's deltas centered at the thresholds. Interestingly, when noise satisfying mild regularity properties is added to their argument and the expectation operator is applied, the resulting function turns out to be Lipschitz or even differentiable in classical sense. Let us introduce the essential definitions and notation needed to understand the statement of Theorem 5 and its proof. For further details, see for instance [137]. Given a function $f : \mathbb{R} \to \mathbb{R}$, which is measurable with respect to the Borel σ-algebra, we say that $f \in L^p(\mathbb{R})$ for $1 \leq p < \infty$ if its p-norm $\|f\|_p \coloneqq \left(\int_{\mathbb{R}} |f(x)|^p dx\right)^{1/p}$ is finite. We say that $f \in L^\infty(\mathbb{R})$ if its ∞-norm $\|f\|_\infty \coloneqq \inf\left\{t \geq 0 \,\middle|\, \int_{\{x \,|\, |f(x)|>t\}} dx = 0\right\}$ is finite. Given $f, g \in L^1(\mathbb{R})$, we call the function

$$\begin{aligned} f * g \,:\, \mathbb{R} &\to \mathbb{R} \\ x &\mapsto (f*g)(x) \coloneqq \int_{\mathbb{R}} f(y)g(x-y)dy \end{aligned}$$

the *convolution* between f and g. Given $f \in L^1(\mathbb{R})$, we call *distributional derivative* of f the (linear, continuous) functional Df defined by

$$Df(\phi) \coloneqq -\int_{\mathbb{R}} f(x)\frac{d\phi}{dx}(x)dx$$

on any C^∞-smooth test function ϕ that vanishes outside some compact interval of $\mathbb{R}$. When Df can be represented by a function, that is, there exists $g \in L^1(\mathbb{R})$ such that we have the integration by parts formula

$$Df(\phi) = \int_{\mathbb{R}} g(x)\phi(x)dx\,,$$

we say that f belongs to the *Sobolev space* $W^{1,1}(\mathbb{R})$, and that $Df = g$ is the *weak derivative* of f. Similarly, but more generally, when Df is represented via integration by parts by a signed Borel-regular measure Df on $\mathbb{R}$, whose total variation $|Df|(\mathbb{R})$ (roughly speaking, a generalisation of the L^1-norm of the derivative of f) is finite, then we say that f is a function of *bounded variation*, i.e., that it belongs to the space $BV(\mathbb{R})$. For instance, the characteristic function $\chi_{[a,b]}$ of a bounded interval $[a, b] \subset \mathbb{R}$ is a BV function, and its distributional derivative $D\chi_{[a,b]}$ is given by the signed measure $\delta_a - \delta_b$, where δ_x denotes the Dirac's delta measure centered at x. In order to provide an interpretation of STE, and in accordance with the choices of the noise in our experiments, we assume that the probability measure μ that describes the noise distribution has a PDF which is either a Sobolev or, more generally, a BV function on $\mathbb{R}$. Hence, the generalised derivative $D\mu$ will be either an L^1 function or a signed measure with finite total variation. With a slight abuse of notation, we will use the symbol μ to identify both the probability measure and its density. Given a random variable f, its *expectation* with respect to the probability measure μ is the integral

$$\mathbb{E}_\mu[f] \coloneqq \int_{\mathbb{R}} f(x)\mu(x)dx\,.$$

Lemma 2. *Let $\sigma : \mathbb{R} \to \mathbb{R}$ and $\mu : \mathbb{R} \to \mathbb{R}$ be given functions. The following facts hold:*

(i) if $\sigma \in L^\infty(\mathbb{R})$ and $\mu \in BV(\mathbb{R})$ then $\sigma * \mu \in W^{1,\infty}(\mathbb{R})$;

(ii) if $\sigma \in L^\infty(\mathbb{R})$ and $\mu \in W^{1,1}(\mathbb{R})$ then the weak derivative of $\sigma * \mu$ satisfies

$$D(\sigma * \mu)(x) = \sigma * D\mu(x)$$

for almost all $x \in \mathbb{R}$;

(iii) if $\sigma \in BV(\mathbb{R})$ and $\mu \in W^{1,1}(\mathbb{R})$ then $\sigma * \mu \in C^1(\mathbb{R})$, its derivative is uniformly continuous and one has

$$\frac{d(\sigma * \mu)}{dx}(x) = \sigma * D\mu(x)$$

for all $x \in \mathbb{R}$.

Proof. We first show (i). It is immediate to check that

$$\|\sigma * \mu\|_\infty \leq \|\sigma\|_\infty \|\mu\|_1 < +\infty\,. \tag{5.20}$$

We choose $x_1 < x_2 \in \mathbb{R}$ and, thanks to Fubini's theorem and change of variable, we obtain the following estimate:

$$\begin{aligned}
|\sigma * \mu(x_1) - \sigma * \mu(x_2)| &= \left| \int_{y\in\mathbb{R}} \big(\mu(x_1 - y) - \mu(x_2 - y)\big)\sigma(y)dy \right| \\
&\leq \int_{y\in\mathbb{R}} |D\mu|\big([x_1 - y, x_2 - y]\big)|\sigma(y)|\, dy \\
&\leq \|\sigma\|_\infty \int_{y\in\mathbb{R}} \int_{z\in\mathbb{R}} \chi_{[x_1,x_2]}(z + y)d|D\mu|(z)dy \\
&= \|\sigma\|_\infty \int_{z\in\mathbb{R}} \int_{y\in\mathbb{R}} \chi_{[x_1,x_2]}(z + y)dyd|D\mu|(z) \\
&\leq \|\sigma\|_\infty |D\mu|(\mathbb{R})\, |x_1 - x_2|\,.
\end{aligned}$$

This shows that $\sigma*\mu$ is a Lipschitz function (with Lipschitz constant bounded above by $\|\sigma\|_\infty |D\mu|(\mathbb{R})$). Therefore the proof of (i) follows from the Sobolev characterisation of Lipschitz functions combined with (5.20). Let us prove (ii) by showing that $\sigma * \mu$ is weakly differentiable, thus providing a point-wise almost everywhere representation of its weak derivative. Let $\phi \in C_c^\infty(\mathbb{R})$ be a given test function. By using Fubini's Theorem, the definition of weak derivative, and the change of variable in the integration, we obtain

$$\begin{aligned}
\int_{\mathbb{R}} \sigma * \mu(x) \frac{d\phi}{dx}(x)dx &= \int_{x\in\mathbb{R}} \int_{y\in\mathbb{R}} \sigma(y)\mu(x - y)dy \frac{d\phi}{dx}(x)dx \\
&= \int_{y\in\mathbb{R}} \int_{x\in\mathbb{R}} \mu(x - y) \frac{d\phi}{dx}(x)dx\sigma(y)dy \\
&= -\int_{x\in\mathbb{R}} \int_{y\in\mathbb{R}} \sigma(y)D\mu(x - y)dy\phi(x)dx \\
&= -\int_{x\in\mathbb{R}} \sigma * D\mu(x)\phi(x)dx\,.
\end{aligned}$$

This shows (ii). We finally prove (iii). By (i) and (ii) we already know that $\sigma * \mu \in W^{1,\infty}(\mathbb{R})$ and its weak derivative satisfies $D(\sigma * \mu)(x) = \sigma * D\mu(x)$ for almost all $x \in \mathbb{R}$. The conclusion is achieved as soon as we show that

$\sigma * D\mu(x)$ is a continuous function. We have for $x_1 < x_2 \in \mathbb{R}$

$$\begin{aligned}
|\sigma * D\mu(x_1) - \sigma * D\mu(x_2)| &= \left| \int_{y\in\mathbb{R}} D\mu(x_1 - y)\sigma(y)dy - \int_{y\in\mathbb{R}} D\mu(x_2 - y)\sigma(y)dy \right| \\
&= \left| \int_{z\in\mathbb{R}} D\mu(z)\big(\sigma(x_1 - z) - \sigma(x_2 - z)\big)dz \right| \\
&\leq \int_{z\in\mathbb{R}} \int_{t\in\mathbb{R}} \chi_{[x_1,x_2]}(t + z)d|D\sigma|(t)|D\mu(z)|dz \\
&= \int_{t\in\mathbb{R}} \int_{z\in\mathbb{R}} \chi_{[x_1,x_2]}(t + z)|D\mu(z)|dzd|D\sigma|(t) \\
&= \int_{t\in\mathbb{R}} \int_{z\in\mathbb{R}} \chi_{[x_1,x_2]}(t + z)|D\mu(z)|dzd|D\sigma|(t)\,.
\end{aligned}$$

Denote by ν the non-negative, finite Borel measure defined by $d\nu = |D\mu(z)|dz$. Since ν is absolutely continuous with respect to the Lebesgue measure, for all $\epsilon > 0$ there exists $\delta > 0$ such that, if $|x_1 - x_2| < \delta$ then $\nu([x_1, x_2]) < \epsilon$. Therefore we get

$$|\sigma * D\mu(x_1) - \sigma * D\mu(x_2)| \leq \int_{t\in\mathbb{R}} \nu([x_1 - t, x_2 - t])d|D\sigma|(t) \leq \epsilon|D\sigma|(\mathbb{R})$$

as soon as $|x_1 - x_2| \leq \delta$, which proves the uniform continuity of $\sigma * D\mu$ and concludes the proof. □

Theorem 5 (Noise regularisation effect on multi-step functions)**.** *Let $\sigma : \mathbb{R} \to Q$ be a multi-step function. Let ν be a zero-mean random variable whose distribution has density $\mu(-\nu)$. Define the random function $\sigma_\nu(x) \coloneqq \sigma(x+\nu)$ for $x \in \mathbb{R}$. Then:*

(i) $\mathbb{E}_\mu[\sigma_\nu] = \sigma * \mu$*;*

(ii) $\mu \in W^{1,1}(\mathbb{R})$ *implies that* $\mathbb{E}_\mu[\sigma_\nu]$ *is differentiable, its derivative is bounded, continuous, and satisfies* $\frac{d}{dx}\mathbb{E}_\mu[\sigma_\nu] = \sigma * D\mu$*;*

(iii) $\mu \in BV(\mathbb{R})$ *implies that* $D\mathbb{E}_\mu[\sigma_\nu] = \sigma * D\mu$ *almost everywhere and it is bounded by* $\lambda = \|\sigma\|_\infty |D\mu|(\mathbb{R})$*.*

Proof. The first claim (i) simply follows from the change of variables $z = x + \nu$:

$$\mathbb{E}_\mu[\sigma_\nu](x) = \int_{\mathbb{R}} \sigma(x + \nu)\mu(-\nu)d\nu = \int_{\mathbb{R}} \sigma(z)\mu(x - z)dz = (\sigma * \mu)(x),\ \forall\, x \in \mathbb{R}\,.$$

The proofs of (ii) and (iii) follow from the application of Lemma 2 noticing that, by definition of multi-step function, $\sigma \in L^\infty(\mathbb{R})$ (in particular, $\|\sigma\|_\infty = \max_{q\in Q}\{|q|\}$). □

In other words, the expectation acts as a convolution on the non-differentiable function σ (with the noise playing the role of the kernel), and the regularity of μ is transferred to $\mathbb{E}_\mu[\sigma_\nu]$. To simplify technical computations, we used $\mu(-\nu)$ instead of $\mu(\nu)$, but we notice that for symmetric distributions (such as the uniform or the Gaussian) $\mu(-\nu) = \mu(\nu)$.

5.4 The *additive noise annealing* algorithm

Modern deep learning frameworks such as TensorFlow [55] or PyTorch [56] have been designed to optimised the execution of gradient-based learning algorithms. This optimisation and the vast community support make it unappealing for users to transition towards gradient-free learning algorithms. The computational graph associated with a QNN is not differentiable, and therefore it does not support gradient computation. In the following we will show how to use the results of the previous Section 5.3 to turn the functions that compose a QNN into differentiable computational nodes.

Let $\varphi_{\mathbf{m}^\ell}$ be a quantized layer (5.18) with parameters $\mathbf{w}^\ell$ and $\mathbf{b}^\ell$. Let $\mathbf{n}_{\mathbf{w}^\ell}$ and $\mathbf{n}_{\mathbf{b}^\ell}$ be random variables distributed according to zero-mean measures which have probability densities $\mu_{\mathbf{w}^\ell}$ and $\mu_{\mathbf{b}^\ell}$. The random variables $\boldsymbol{\omega}^\ell \coloneqq \mathbf{w}^\ell + \mathbf{n}_{\mathbf{w}^\ell}, \boldsymbol{\beta}^\ell \coloneqq \mathbf{b}^\ell + \mathbf{n}_{\mathbf{b}^\ell}$ are distributed according to the measures with densities $\mu_{\boldsymbol{\omega}^\ell}(\boldsymbol{\omega}) = \mu_{\mathbf{w}^\ell}(\boldsymbol{\omega} - \mathbf{w}^\ell)$ and $\mu_{\boldsymbol{\beta}^\ell}(\boldsymbol{\beta}) = \mu_{\mathbf{b}^\ell}(\boldsymbol{\beta} - \mathbf{b}^\ell)$, and have means $\mathbf{w}^\ell$ and $\mathbf{b}^\ell$. If we define $\boldsymbol{\xi}^\ell \coloneqq (\boldsymbol{\omega}^\ell, \boldsymbol{\beta}^\ell)$, the quantized layers (5.18) become random functions:

$$\varphi_{\boldsymbol{\xi}^\ell}(\mathbf{x}^{\ell-1}) \coloneqq \sigma^\ell \left(\zeta^\ell(\boldsymbol{\omega}^\ell) \cdot \mathbf{x}^{\ell-1} + \boldsymbol{\beta}^\ell\right) . \tag{5.21}$$

The L-layer network

$$\Phi_{\hat{\boldsymbol{\xi}}} = \varphi_{\xi^L} \circ \varphi_{\xi^{L-1}} \circ \cdots \circ \varphi_{\xi^1}$$

is a random function that we call a **stochastic configuration** of a QNN. We define the random variable $\hat{\boldsymbol{\xi}} \coloneqq (\boldsymbol{\xi}^1, \ldots, \boldsymbol{\xi}^L) \in \widehat{M}$, distributed according to the product measure with density $\hat{\mu} = \mu^1 \times \cdots \times \mu^L$. The set $\{\Phi_{\hat{\boldsymbol{\xi}}}\}_{\hat{\boldsymbol{\xi}} \in \widehat{M}}$ of all possible stochastic configurations of the network and the measure $\hat{\mu}$ over $\widehat{M}$ define the **ensemble**

$$(\{\Phi_{\hat{\boldsymbol{\xi}}}\}_{\hat{\boldsymbol{\xi}} \in \widehat{M}}, \hat{\mu}) . \tag{5.22}$$

Rewriting (5.21) as

$$\sigma^\ell \left(\zeta^\ell(\boldsymbol{\omega}^\ell) \cdot \mathbf{x}^{\ell-1} + \boldsymbol{\beta}^\ell\right) = \sigma^\ell_{\mathbf{n}_{\mathbf{b}^\ell}} \left(\zeta^\ell_{\mathbf{n}_{\mathbf{w}^\ell}}(\mathbf{w}^\ell) \cdot \mathbf{x}^{\ell-1} + \mathbf{b}^\ell\right) .$$

and applying Theorem 5 we get the differentiable function

$$\tilde{\varphi}_{\mathbf{m}^\ell}(\mathbf{x}^{\ell-1}) \coloneqq \tilde{\sigma}^\ell \left(\tilde{\zeta}^\ell(\mathbf{w}^\ell) \cdot \mathbf{x}^{\ell-1} + \mathbf{b}^\ell\right) , \tag{5.23}$$

where $\tilde{\sigma}^\ell = \mathbb{E}_{\mu_{\mathbf{b}^\ell}}[\sigma^\ell_{\mathbf{n}_{\mathbf{b}^\ell}}]$ and $\tilde{\zeta}^\ell = \mathbb{E}_{\mu_{\mathbf{w}^\ell}}[\zeta^\ell_{\mathbf{n}_{\mathbf{w}^\ell}}]$. We call (5.23) the **mean field layer** associated with layer (5.18) and noise densities $\mu_{\mathbf{w}^\ell}, \mu_{\mathbf{b}^\ell}$. Note that the mean field layer can be thought of as an approximation of the expectation

$$\tilde{\varphi}_{\mathbf{m}^\ell}(\mathbf{x}^{\ell-1}) \approx \mathbb{E}_{\mu^\ell}[\varphi_{\boldsymbol{\xi}^\ell}](\mathbf{x}^{\ell-1})\,,$$

where $\mu^\ell \coloneqq \mu_{\mathbf{w}^\ell} \times \mu_{\mathbf{b}^\ell}$ denotes the density associated with the natural product measure on M^ℓ.

Assuming that each layer map $\varphi_{\boldsymbol{\xi}^\ell} : X^{\ell-1} \to X^\ell$ is uniformly continuous with respect to its parameters $\boldsymbol{\xi}^\ell$, the next theorem states that the composition of expectations $\mathbb{E}_{\mu^L}[\varphi_{\boldsymbol{\xi}^L}] \circ \cdots \circ \mathbb{E}_{\mu^1}[\varphi_{\boldsymbol{\xi}^1}]$ point-wise converges to the (deterministic) feedforward neural network $\Phi_{\hat{\mathbf{m}}}$ as soon as $\hat{\mu} = \mu^1 \times \cdots \times \mu^L$ converges to $\delta_{\hat{\mathbf{m}}} = \delta_{\mathbf{m}^1} \times \cdots \times \delta_{\mathbf{m}^L}$ (i.e., to the product of Dirac's deltas concentrated at the parameters means). It is worth recalling the notion of weak-$*$ convergence for a (probability) measure: we say that a sequence μ_t of Borel measures restricted to a compact subset $M \subset \mathbb{R}^n$ converges weakly-$*$ to a Borel measure μ on M if, for every continuous function $\phi : M \to \mathbb{R}$, one has $\mathbb{E}_{\mu_t}[\phi] \to \mathbb{E}_\mu[\phi]$ as $t \to \infty$.

Theorem 6 (Continuity of composed expectations)**.** *Let M^ℓ and $X^{\ell-1}$ be compact subsets of some Euclidean spaces. Assume that, for all $\ell = 1, \dots, L$, the map $\varphi_{\boldsymbol{\xi}^\ell}(\mathbf{x}^{\ell-1}) = \varphi^\ell(\boldsymbol{\xi}^\ell, \mathbf{x}^{\ell-1})$ is continuous in both variables $\boldsymbol{\xi}^\ell$ and $\mathbf{x}^{\ell-1}$. Let $\{\mu^\ell_t\}_{t\in\mathbb{N}}$ be a sequence of probability measures on M^ℓ converging to the Dirac's delta $\delta_{\mathbf{m}^\ell}$ for suitable $\mathbf{m}^\ell \in M^\ell$ and for all $\ell = 1, \dots, L$. Then $\lim_{t\to\infty} \mathbb{E}_{\mu^L_t}[\varphi_{\boldsymbol{\xi}^L}] \circ \cdots \circ \mathbb{E}_{\mu^1_t}[\varphi_{\boldsymbol{\xi}^1}](\mathbf{x}) = \Phi_{\hat{\mathbf{m}}}(\mathbf{x})$, $\forall\, \mathbf{x} \in X^0$.*

We recall the notation (3.14) for a parametric composition of maps, as Theorem 6 applies to this situation as well. We recall in particular the map $\Psi_{\hat{\mathbf{m}}^\ell} : X^0 \to X^\ell$ defined as $\Psi_{\hat{\mathbf{m}}^\ell} = \psi_{\mathbf{m}^\ell} \circ \cdots \circ \psi_{\mathbf{m}^1}$, as it will play a role in the induction argument below.

Proof. First of all we observe that the continuity assumption on ψ^ℓ implies the existence of a modulus of continuity η (i.e., $\eta : [0, +\infty) \to [0, +\infty)$ is continuous, strictly increasing, and satisfies $\eta(0) = 0$) such that

$$\sup_{\boldsymbol{\xi}^\ell \in M^\ell} |\psi_{\boldsymbol{\xi}^\ell}(\mathbf{x}) - \psi_{\boldsymbol{\xi}^\ell}(\mathbf{y})| \le \eta(|\mathbf{x} - \mathbf{y}|),\ \forall\, \mathbf{x}, \mathbf{y} \in X^{\ell-1},\ \forall\, \ell = 1, \dots, L\,. \quad (5.24)$$

Denote by $\mathbb{E}_{\ell,t}$ the expectation operator associated with the measure μ^ℓ_t. We proceed by induction on $\ell = 1, \dots, L$. The basis of the induction consists in showing that

$$\lim_{t\to\infty} \mathbb{E}_{1,t}[\psi_{\boldsymbol{\xi}^1}](\mathbf{x}) = \psi_{\mathbf{m}^1}(\mathbf{x}),\ \forall\, \mathbf{x} \in X^0\,. \quad (5.25)$$

We observe that

$$J_{1,t}(\mathbf{x}) \coloneqq \mathbb{E}_{1,t}[\psi_{\boldsymbol{\xi}^1}](\mathbf{x}) = \int_{M^1} \psi(\boldsymbol{\xi}^1, \mathbf{x}) d\mu_t^1(\xi^1)\,,$$

hence (5.25) directly follows from the definition of convergence of μ_t^1 to $\delta_{\mathbf{m}^1}$. In the next, inductive step we shall apply the uniform continuity assumption (5.24) in an essential way. Let us set

$$J_{\ell,t}(\mathbf{x}) \coloneqq \mathbb{E}_{\ell,t}[\psi_{\boldsymbol{\xi}^\ell}] \circ \cdots \circ \mathbb{E}_{1,t}[\psi_{\boldsymbol{\xi}^1}](\mathbf{x})$$

and assume by induction that

$$\lim_{t\to\infty} J_{\ell,t}(\mathbf{x}) = \Psi_{\hat{\mathbf{m}}^\ell}(\mathbf{x}),\ \forall\, \mathbf{x} \in X^0\,. \tag{5.26}$$

We then have to prove that, for some $1 \leq \ell < L$ and for all $\mathbf{x} \in X^0$,

$$\lim_{t\to\infty} \mathbb{E}_{\ell+1,t}[\psi_{\boldsymbol{\xi}^{\ell+1}}] \circ J_{\ell,t}(\mathbf{x}) = \psi_{\mathbf{m}^{\ell+1}} \circ \Psi_{\hat{\mathbf{m}}^\ell}(\mathbf{x}) = \Psi_{\hat{\mathbf{m}}^{\ell+1}}(\mathbf{x})\,. \tag{5.27}$$

First, we rewrite

$$\begin{aligned}
&\mathbb{E}_{\ell+1,t}[\psi_{\boldsymbol{\xi}^{\ell+1}}] \circ J_{\ell,t}(\mathbf{x}) - \psi_{\mathbf{m}^{\ell+1}} \circ \Psi_{\hat{\mathbf{m}}^\ell}(\mathbf{x}) = \\
&\quad = \int_{M^{\ell+1}} \Big(\psi_{\boldsymbol{\xi}^{\ell+1}}(J_{\ell,t}(\mathbf{x})) - \psi_{\mathbf{m}^{\ell+1}}(\Psi_{\hat{\mathbf{m}}^\ell}(\mathbf{x}))\Big) d\mu_t^{\ell+1}(\boldsymbol{\xi}^{\ell+1}) \\
&\quad = \int_{M^{\ell+1}} \Big(\psi_{\boldsymbol{\xi}^{\ell+1}}(J_{\ell,t}(\mathbf{x})) - \psi_{\boldsymbol{\xi}^{\ell+1}}(\Psi_{\hat{\mathbf{m}}^\ell}(\mathbf{x}))\Big) d\mu_t^{\ell+1}(\boldsymbol{\xi}^{\ell+1}) \\
&\qquad + \int_{M^{\ell+1}} \Big(\psi_{\boldsymbol{\xi}^{\ell+1}}(\Psi_{\hat{\mathbf{m}}^\ell}(\mathbf{x})) - \psi_{\mathbf{m}^{\ell+1}}(\Psi_{\hat{\mathbf{m}}^\ell}(\mathbf{x}))\Big) d\mu_t^{\ell+1}(\boldsymbol{\xi}^{\ell+1}) \\
&\quad = A + B\,,
\end{aligned} \tag{5.28}$$

where the first equality follows from $\mu_t^{\ell+1}$ being a probability measure. The first addend in equation (5.28) satisfies

$$|A| \leq \eta(|J_{\ell,t}(\mathbf{x}) - \Psi_{\hat{\mathbf{m}}^\ell}(\mathbf{x})|)\,,$$

since $\mu_t^{\ell+1}$. By the inductive hypothesis (5.26), this term goes to zero when $t \to \infty$. The term B in equation (5.28) goes to zero thanks to the weak-$*$ convergence hypothesis on $\mu_t^{\ell+1}$. This proves (5.27) and completes the proof of the theorem. □

Even though the σ^ℓ and ζ^ℓ used in Section 5.2 to model QNNs are not continuous, Theorem 6 motivates ANA. None of the networks corresponding to the individual configurations in (5.22) is differentiable, but each of them is

quantized. The approximate ensemble average (the *mean field network*) built using blocks (5.23) is instead differentiable as long as the conditions provided by Theorem 5 are satisfied, but it is usually not quantized. Therefore, in order to retrieve a QNN, the probability measure of the ensemble must collapse onto a single element of the configuration space. At every training iteration (Algorithm 4, lines 1-20), ANA sets probability measures $\mu_{\boldsymbol{\omega}^\ell}, \mu_{\boldsymbol{\beta}^\ell}$ for each parameter $\boldsymbol{\omega}^\ell$ and $\boldsymbol{\beta}^\ell$ (lines 2-5). ANA then replaces the non-differentiable functions with regularised counterparts (lines 6-9), so that gradients can be computed (lines 10-17). We remark that the measures $\mu_{\boldsymbol{\omega}^\ell}, \mu_{\boldsymbol{\beta}^\ell}$ depend on time, and should be annealed to Dirac's deltas as $t \to T$. We discuss the implemented annealing strategies in Section 5.5.

Algorithm 4 Synchronous additive noise annealing

Input: $\Phi_{\hat{\mathbf{m}}_0}, \lambda_0, T, \{(\mathbf{x}_t^0, \mathbf{y}_t)\}_{t=1,\dots,T}$
Output: $\Phi_{\hat{\mathbf{m}}_T}$

1: **for** $t \leftarrow 1, T$ **do**
2: **for** $\ell \leftarrow 1, L$ **do** ▷ additive noise
3: $\mu_{\boldsymbol{\omega}^\ell} \leftarrow set\_noise(t, \mathbf{w}^\ell)$
4: $\mu_{\boldsymbol{\beta}^\ell} \leftarrow set\_noise(t, \mathbf{b}^\ell)$
5: **end for**
6: **for** $\ell \leftarrow 1, L$ **do** ▷ inference
7: $\tilde{\mathbf{w}}^\ell \leftarrow \mathbb{E}_{\mu_{\boldsymbol{\omega}^\ell}}[\zeta^\ell(\boldsymbol{\omega}^\ell)]$
8: $\mathbf{x}_t^\ell \leftarrow \mathbb{E}_{\mu_{\boldsymbol{\beta}^\ell}}[\sigma^\ell(\tilde{\mathbf{w}}^\ell \cdot \mathbf{x}_t^{\ell-1} + \boldsymbol{\beta}^\ell)]$
9: **end for**
10: $g_{\mathbf{x}^L} \leftarrow \nabla_{\mathbf{x}^L} d_L(\mathbf{x}_t^L, \mathbf{y}_t)$ ▷ backpropagation
11: **for** $\ell \leftarrow L, 1$ **do**
12: $g_{\mathbf{s}^\ell} \leftarrow g_{\mathbf{x}^\ell} \cdot \nabla_{\mathbf{s}^\ell} \mathbb{E}_{\mu_{\boldsymbol{\beta}^\ell}}[\sigma^\ell(\mathbf{s}^\ell + \boldsymbol{\beta}^\ell)]$
13: $g_{\mathbf{b}^\ell} \leftarrow g_{\mathbf{s}^\ell}$
14: $g_{\tilde{\mathbf{w}}^\ell} \leftarrow g_{\mathbf{s}^\ell} \cdot \mathbf{x}_t^{\ell-1}$
15: $g_{\mathbf{w}^\ell} \leftarrow g_{\tilde{\mathbf{w}}^\ell} \cdot \nabla_{\mathbf{w}^\ell} \mathbb{E}_{\mu_{\boldsymbol{\omega}^\ell}}[\zeta^\ell(\boldsymbol{\omega}^\ell)]$
16: $g_{\mathbf{x}^{\ell-1}} \leftarrow g_{\mathbf{s}^\ell} \cdot \tilde{\mathbf{w}}^\ell$
17: **end for**
18: $\hat{\mathbf{m}}_t \leftarrow optim(\lambda_{t-1}, \hat{\mathbf{m}}_{t-1}, g_{\hat{\mathbf{m}}})$ ▷ gradient descent
19: $\lambda_t \leftarrow lr\_sched(t, \lambda_{t-1}, \hat{\mathbf{m}}_t)$
20: **end for**
21: **return** $\Phi_{\hat{\mathbf{m}}_T}$

Using Theorem 5, we can interpret STE [138] as a particular instance that applies noise to the argument of the sign function according to two different densities μ^f, μ^b during the forward and backward passes.

Corollary 1. *Let $\sigma(x) = H_0^{\{-1,+1\}}(x)$ denote the univariate sign function. Define forward and backward probability measures with generalised densities $\mu^f = \delta_0$ and $\mu^b = \mathcal{U}[-1,+1]$, and denote with ν^f, ν^b the random variables distributed accordingly. By Theorem 5 we have*

$$\mathbb{E}_{\mu^f}[\sigma(x+\nu^f)] = \sigma(x) \quad \textit{and} \quad \frac{d}{dx}\mathbb{E}_{\mu^b}[\sigma(x+\nu^b)] = \begin{cases} 0, & \textit{if } x \notin [-1,1] \\ 1, & \textit{if } x \in [-1,1]\,. \end{cases}$$

This observation led us to devise a generalised version of Algorithm 4 that allows defining different measures $\mu^f_{\boldsymbol{\omega}^\ell} \neq \mu^b_{\boldsymbol{\omega}^\ell}$ and $\mu^f_{\boldsymbol{\beta}^\ell} \neq \mu^b_{\boldsymbol{\beta}^\ell}$ in lines 2-5. This idea amounts to replacing the non-differentiable nodes in the computational graph with differentiable approximations that can change between the inference and training passes. Notice that, to get a QNN at inference time, it is sufficient that only one measure (the one used during the forward pass) converges to a Dirac's delta. Using differentiable functions during the backward pass allows the gradients to flow (and for training to continue) in the lower layers also when $t \to T$. To distinguish this variant of ANA from the previous one, we will refer to the former with the term **synchronous ANA**.

5.5 QuantLab

The activity of collecting measurements about a physical phenomenon where certain variables (called *explanatory variables*) can be controlled is called **experimentation**. The activity of collecting performance measurements about an engineered artifact where certain parameters of the artifact can be arbitrarily set is called **benchmarking**. Benchmarking is often confused with experimentation. Both the activities are empirical in nature: scientists and engineers collect measurements to quantify desired relationships. Experimentation studies the relationships between the explanatory variables and the *response variables* of some physical phenomenon to provide insights into its physics. Benchmarking studies the relationships between the parameters of a given artifact and some performance metrics to provide operational guidelines to future users of the artifact. Usually, benchmarking involves a series of tasks (and associated metrics) to evaluate the performance of the artifact; each task is called a **benchmark**. In the machine learning literature, and specifically in deep learning literature, the term *experiments* or *experimental results* are improperly used to refer to benchmarking.

QuantLab is an add-on for PyTorch [56], designed to enable the benchmarking of the additive noise annealing algorithm over multiple ANN topologies and multiple data sets. It is a Python package containing configuration

Algorithm 5 Additive noise annealing

Input: $\Phi_{\hat{\mathbf{m}}_0}, \lambda_0, T, \{(\mathbf{x}_t^0, \mathbf{y}_t)\}_{t=1,\dots,T}$
Output: $\Phi_{\hat{\mathbf{m}}_T}$

for $t \leftarrow 1, T$ **do**
  for $\ell \leftarrow 1, L$ **do** ▷ additive noise
    $\mu^f_{\boldsymbol{\omega}^\ell}, \mu^b_{\boldsymbol{\omega}^\ell} \leftarrow set\_noise(t, \mathbf{w}^\ell)$
    $\mu^f_{\boldsymbol{\beta}^\ell}, \mu^b_{\boldsymbol{\beta}^\ell} \leftarrow set\_noise(t, \mathbf{b}^\ell)$
  end for
  for $\ell \leftarrow 1, L$ **do** ▷ inference
    $\tilde{\mathbf{w}}^\ell \leftarrow \mathbb{E}_{\mu^f_{\boldsymbol{\omega}^\ell}}[\zeta^\ell(\boldsymbol{\omega}^\ell)]$
    $\mathbf{x}_t^\ell \leftarrow \mathbb{E}_{\mu^f_{\boldsymbol{\beta}^\ell}}[\sigma^\ell(\tilde{\mathbf{w}}^\ell \cdot \mathbf{x}_t^{\ell-1} + \boldsymbol{\beta}^\ell)]$
  end for
  $g_{\mathbf{x}^L} \leftarrow \nabla_{\mathbf{x}^L} d_L(\mathbf{x}_t^L, \mathbf{y}_t)$ ▷ backpropagation
  for $\ell \leftarrow L, 1$ **do**
    $g_{\mathbf{s}^\ell} \leftarrow g_{\mathbf{x}^\ell} \cdot \nabla_{\mathbf{s}^\ell} \mathbb{E}_{\mu^b_{\boldsymbol{\beta}^\ell}}[\sigma^\ell(\mathbf{s}^\ell + \boldsymbol{\beta}^\ell)]$
    $g_{\mathbf{b}^\ell} \leftarrow g_{\mathbf{s}^\ell}$
    $g_{\tilde{\mathbf{w}}^\ell} \leftarrow g_{\mathbf{s}^\ell} \cdot \mathbf{x}_t^{\ell-1}$
    $g_{\mathbf{w}^\ell} \leftarrow g_{\tilde{\mathbf{w}}^\ell} \cdot \nabla_{\mathbf{w}^\ell} \mathbb{E}_{\mu^b_{\boldsymbol{\omega}^\ell}}[\zeta^\ell(\boldsymbol{\omega}^\ell)]$
    $g_{\mathbf{x}^{\ell-1}} \leftarrow g_{\mathbf{s}^\ell} \cdot \tilde{\mathbf{w}}^\ell$
  end for
  $\hat{\mathbf{m}}_t \leftarrow optim(\lambda_{t-1}, \hat{\mathbf{m}}_{t-1}, g_{\hat{\mathbf{m}}})$ ▷ gradient descent
  $\lambda_t \leftarrow lr\_sched(t, \lambda_{t-1}, \hat{\mathbf{m}}_t)$
end for
return $\Phi_{\hat{\mathbf{m}}_T}$

files to optimise its performances on the underlying computing infrastructure. According to the definition of machine learning system given in Section 1.3, we designed QuantLab to include

- a sub-package `indiv` that implements a library of quantized layer maps which can be used to build different program spaces;
- a sub-package `treat` that implements the operations of ANA, our learning algorithm for QNNs;
- multiple problem sub-packages implementing different machine learning systems on different data sets; we implemented packages `MNIST`, `CIFAR-10` and `ImageNet` for the corresponding image classification tasks.

Theorem 4 requires that weights take values in a superset of $\{-1, 0, +1\}$ and that representations take values in a superset of $\{0, 1\}$. BNNs do not

satisfy these conditions. The only research work satisfying the conditions prescribed by the theorem while using low-bitwdith operands is GXNOR-Nets [132]. As we observed in Section 5.1, GXNOR-Nets uses a gradient-free update rule for the weights, whereas ANA is a fully gradient-based training algorithm. For these two reasons, we decided to experiment with TNNs instead of BNNs, but also compared with BNNs trained using gradient-based algorithms. We set all the weights quantization functions and the activation functions of our systems to be the multi-step function

$$\varsigma(x) = \begin{cases} -1, & \text{if } x \in (-\infty, -0.5) \\ 0, & \text{if } x \in [-0.5, 0.5) \\ 1, & \text{if } x \in [0.5, +\infty)\,. \end{cases}$$

The shadow weights were initialised uniformly around the thresholds. In Section 5.4, we remarked that the measures $\mu^f_{\boldsymbol{\omega}^\ell}$ and $\mu^f_{\boldsymbol{\beta}^\ell}$ need to converge (in the weak-$*$ sense) to Dirac's deltas in order to yield a QNN at inference time. Note that a uniform distribution $\mathcal{U}[a, b]$ over a non-empty real interval $[a, b]$ can also be described as $\mathcal{U}[\frac{a+b}{2} - \sqrt{3}\varsigma, \frac{a+b}{2} + \sqrt{3}\varsigma]$, where ς is its standard deviation. Modelling $\varsigma = \varsigma(t)$ as a time-dependent quantity, we see that the zero-mean distribution

$$\mathcal{U}[-\sqrt{3}\varsigma(t), \sqrt{3}\varsigma(t)] \tag{5.29}$$

converges to δ_0 if $\varsigma(t) \to 0$ as $t \to 0$. We thus modelled $\mu^f_{\boldsymbol{\omega}^\ell}, \mu^b_{\boldsymbol{\omega}^\ell}, \mu^f_{\boldsymbol{\beta}^\ell}$ and $\mu^b_{\boldsymbol{\beta}^\ell}$ as products of independent uniform distributions of the form (5.29). In other words, we added a zero-mean uniform noise of given standard deviation to each parameter. We defined these random variables to be independent but not necessarily identically distributed (i.e, different parameters could be added noises whose measures had different standard deviations). To anneal the measures to Dirac's deltas, we used their component's standard deviations as time-dependent hyperparameters. We ran all the experiments for 1000 epochs on machines equipped with an Intel Xeon E5-2640v4 CPU, four Nvidia GTX1080 Ti GPUs, and 128 GB of memory.

CIFAR-10 The CIFAR-10 data set [74] consists of 32×32 pixels RGB images grouped into ten classes. It comprises $50k$ training points and $10k$ test points. For our experiments, we split the $50k$ images training set in a $45k$ images actual training set and a $5k$ validation set [102]. We performed data augmentation by resizing, random cropping and random flipping the images. The preprocessing consisted of a normalization of the three channels described by means $\mu = (0.4914, 0.4822, 0.4465)$ and standard deviations $\sigma = (0.2470, 0.2430, 0.2610)$. For the sake of comparison with related work, we

used the same VGG-like architecture used by [121, 122] and [132], consisting of six convolutional layers followed by three fully connected layers. The exact topology is described in Table 5.5. We used the per-class **hinge loss** also

Layer	Input shape	Type	C_ℓ/n_ℓ	P	F	S	Output shape
φ^1	$3 \times 32 \times 32$	Conv2d	128	1	3	1	$128 \times 32 \times 32$
		BatchNorm2d	-				
		QuantAct	-				
φ^2	$128 \times 32 \times 32$	Conv2d	128	1	3	1	$128 \times 16 \times 16$
		MaxPool2d	-	0	2	2	
		BatchNorm2d	-				
		QuantAct	-				
φ^3	$128 \times 16 \times 16$	Conv2d	256	1	3	1	$256 \times 16 \times 16$
		BatchNorm2d	-				
		QuantAct	-				
φ^4	$256 \times 16 \times 16$	Conv2d	256	1	3	1	$256 \times 8 \times 8$
		MaxPool2d	-	0	2	2	
		BatchNorm2d	-				
		QuantAct	-				
φ^5	$256 \times 8 \times 8$	Conv2d	512	1	3	1	$512 \times 8 \times 8$
		BatchNorm2d	-				
		QuantAct	-				
φ^6	$512 \times 8 \times 8$	Conv2d	512	1	3	1	$512 \times 4 \times 4$
		MaxPool2d	-	0	2	2	
		BatchNorm2d	-				
		QuantAct	-				
φ^7	8192	FC	1024	-			1024
		BatchNorm1d	-				
		QuantAct	-				
φ^8	1024	FC	1024	-			1024
		BatchNorm1d	-				
		QuantAct	-				
φ^9	1024	FC	10	-			10
		BatchNorm1d	-				

Table 5.1: The VGG-like network used for CIFAR-10 experiments. For each map, C_ℓ/n_ℓ indicate the number of neuron planes in a convolutional layer or the number of neurons in a normal layer, P the padding, F the spatial dimensions of the neuron columns and S the stride.

used by [121, 122] in combination with the Adam optimisation algorithm [54].

The learning rate was initialized to 0.001 and decreased to 0.0001 at epoch 700. We set a batch size of 256. In the first experiment we used syncrhonous ANA (Algorithm 4). We heuristically opted for hierarchical annealing of the noise, starting from layer $\varphi_{\mathbf{m}^1}$ to layer $\varphi_{\mathbf{m}^L}$. This strategy was meant to allow the representations of lower layers to stabilise before transitioning to differentiable approximations with too high Lipschitz constants in the upper layers thus avoiding the *exploding gradients* problem (this potentially disruptive effect can derived as a corollary from Theorem 5). The standard deviations regulating the noises of the layers $\varphi_{\boldsymbol{\xi}^\ell}, \ell = 1, \dots, L$ were initialized to $\varsigma^f_{\boldsymbol{\omega}^\ell}(0) = \varsigma^b_{\boldsymbol{\omega}^\ell}(0) = \varsigma^f_{\boldsymbol{\beta}^\ell}(0) = \varsigma^b_{\boldsymbol{\beta}^\ell}(0) = \varsigma_\ell(0) = \sqrt{3}/6$ (in order to describe the uniform distribution $\mathcal{U}[-1, +1]$) and annealed following the linear decay

$$\varsigma_\ell(t) = \left(1 - \frac{\min(\max(0, t - 50(\ell - 1)), 50}{50}\right) \varsigma_\ell(0)\,, \tag{5.30}$$

where t denotes the training epoch. The idea was that ς_ℓ should decay from $\sqrt{3}/6$ at epoch $50(\ell - 1)$ to zero at epoch 50ℓ. Since the noise distribution had to be annealed to get a quantized network, the limitation of synchronous ANA was that it prevented gradients from flowing through multi-step functions and adjusting the lower layers' parameters. To circumvent this problem, we experimented with Algorithm 5 using forward measures $\mu^f_{\boldsymbol{\omega}^\ell}, \mu^f_{\boldsymbol{\beta}^\ell}$ whose controlling standard deviations were again initialized to $\varsigma^f_{\boldsymbol{\omega}^\ell}(0) = \varsigma^b_{\boldsymbol{\omega}^\ell}(0) = \varsigma^f_{\boldsymbol{\beta}^\ell}(0) = \varsigma^b_{\boldsymbol{\beta}^\ell}(0) = \sqrt{3}/6$. We annealed the standard deviations of the forward noises trying both the linear annealing (5.30) and the quadratic annealing

$$\varsigma_\ell(t) = \left(1 - \frac{\min(\max(0, t - 50(\ell - 1)), 50}{50}\right)^2 \varsigma_\ell(t)\,.$$

The standard deviations regulating the backward noises were kept constant. Since all the network's parameters could be updated even after the noise had been removed from the forward pass computational graph, both the linear and quadratic settings outperformed synchronous annealing, showing no relevant mutual difference. Validation accuracies for the linear annealing scheme are reported in Table 5.5.

ImageNet The part of the ImageNet database [75] used for the image classification track of the ILSVRC challenge [76] consists of 224×224 pixels RGB images grouped into 1000 classes. It comprises $1.2M$ training points and $50k$ validation points. We performed data augmentation using a pipeline of random resizing and cropping, random flipping, random colours alterations and

random PCA-based lighting changes. The colour alterations consisted of random changes of brightness, contrast and saturation (these transformations are essentially linear interpolations between the original image and transformations of its greyscale version). The lighting changes were performed by randomly rescaling the three RGB channels using coefficients which depend on the eigenvalues obtained applying a per-pixel PCA on the ImageNet dataset. The preprocessing consisted of a normalization of the three channels described by means $\mu = (0.485, 0.456, 0.406)$ and standard deviations $\sigma = (0.229, 0.224, 0.225)$. For the sake of comparison with related research, we experimented on AlexNet [88]. This network topology is described in Table 5.5. We used the cross-entropy loss function in combination with the Adam optimisation algorithm. The learning rate was initialized to 0.001 and decreased to 0.0001 at epoch 700. The batch size was set to 512. The standard deviations regulating the noises of the layers $\varphi_{\boldsymbol{\xi}^\ell}, \ell = 1, \ldots, L$ were initialized to $\varsigma^f_{\boldsymbol{\omega}^\ell}(0) = \varsigma^b_{\boldsymbol{\omega}^\ell}(0) = \varsigma^f_{\boldsymbol{\beta}^\ell}(0) = \varsigma^b_{\boldsymbol{\beta}^\ell}(0) = \varsigma_\ell(0) = \sqrt{3}/6$. We annealed the forward noises' standard deviations using the delayed linear annealing

$$\varsigma_\ell(t) = \left(1 - \frac{\min(\max(0, t - 50(\ell)), 50}{50}\right) \varsigma_\ell(0)\,,$$

where t denotes the training epoch. The idea was that ς_ℓ should have decayed from $\sqrt{3}/6$ at epoch 50ℓ to zero at epoch $50(\ell + 1)$. The standard deviations regulating the backward noises were kept constant at $\sqrt{3}/6$. We also experimented with mixed precision networks. In particular, we quantized the residual branches of a MobileNetV2 [95] while keeping the bottleneck layers at full-precision (see also [125]). We also took care of quantizing the input tensors to all the residual branches to ensure the computationally costly convolution operations are performed with ternary operands. Although this yields a network which is only partially quantized, the operations performed in the residual branches of a MobileNetV2 amount to approximately 70% of the total operations and contain approximately 30% of the total parameters, yielding a significant reduction in both computational effort and memory footprint. Validation accuracies are reported in Table 5.5.

Comparisons between ANA and similar low-bitwidth (BNNs, TNNs) quantization methods are reported in Table 5.5. To perform a more direct comparison to a TNN trained with GXNOR-Net [132], we evaluated our method on CIFAR-10 with a VGG-like network. Here we observed a 1.76% inferior accuracy; however, we could not verify the scalability of the GXNOR-Net algorithm to deeper networks. We remark that deep learning frameworks do not implement optimised libraries to support gradient-free algorithms, and we think an exploration of initialisation strategies for ANA could further improve the resulting accuracy. On AlexNet we achieved a validation accuracy

Layer	Input shape	Type	C_ℓ/n_ℓ	P	F	S	Output shape
φ^1	$3 \times 227 \times 227$	Conv2d	64	2	11	4	$64 \times 27 \times 27$
		MaxPool2d	-	0	3	2	
		BatchNorm2d	-				
		QuantAct	-				
φ^2	$64 \times 27 \times 27$	Conv2d	192	2	5	1	$192 \times 13 \times 13$
		MaxPool2d	-	0	3	2	
		BatchNorm2d	-				
		QuantAct	-				
φ^3	$192 \times 13 \times 13$	Conv2d	384	1	3	1	$384 \times 13 \times 13$
		BatchNorm2d	-				
		QuantAct	-				
φ^4	$384 \times 13 \times 13$	Conv2d	256	1	3	1	$256 \times 13 \times 13$
		BatchNorm2d	-				
		QuantAct	-				
φ^5	$256 \times 13 \times 13$	Conv2d	256	1	3	1	$256 \times 6 \times 6$
		MaxPool2d	-	0	3	2	
		BatchNorm2d	-				
		QuantAct	-				
φ^6	9216	FC	4096	-			4096
		BatchNorm1d	-				
		QuantAct	-				
φ^7	4096	FC	4096	-			4096
		BatchNorm1d	-				
		QuantAct	-				
φ^8	4096	FC	1000	-			1000
		BatchNorm1d	-				

Table 5.2: The AlexNet model used in part of the experiments on ImageNet. For each map, C_ℓ/n_ℓ indicate the number of neuron planes in a convolutional layer or the number of neurons in a normal layer, P the padding, F the spatial dimensions of the neuron columns and S the stride.

of 45.8%. In comparison to the most widely known STE (41.8% accuracy) [121, 122], the network trained with ANA showed a clear reduction of the accuracy gap to the full-precision network (54.7% accuracy) by 31%. Comparing to a BNN trained with XNOR-Net, our trained TNN apparently has a richer configuration space and the achieved accuracy gain is less distinct. However, note that XNOR-Net does not quantize the first and last layers [123]. Moreover, this algorithm require to dynamically compute the $L1$-norm

Problem	Network Topology	Top-1 accuracy	
		Full-precision	Ternary
CIFAR-10	VGG-like	94.40%	90.74%
ImageNet	AlexNet	54.71%	45.80%
	MobileNetV2	71.25%	64.79%[1]

[1] Only the residual branches of MobileNetV2 were quantized (∼70% of MACs and ∼30% of parameters).

Table 5.3: Top-1 validation accuracy of ANA on both CIFAR-10 and ImageNet datasets.

of every representation at each forward pass. To analyze the application of TNNs to a more recent and compute-optimised network, we evaluated the accuracy of MobileNetV2 with quantized residuals (i.e., training a partially quantized network). This simplified 70% of the network's convolution operations and reduced by 30% its memory foortprint, while introducing an accuracy loss of only 6.46% from 71.25% to 64.79%.

Algorithm	Type	CIFAR-10	ImageNet	
		VGG-like	AlexNet	MobileNetV2[2]
STE [121, 122]	BNN	89.85%	41.80%	-
XNOR-Net[1][123]	BNN	-	44.20%	-
GXNOR-Net [132]	TNN	92.50%	-	-
ANA	TNN	90.74%	45.80%	64.79%

[1] The first layer is not quantized; the linear part of the last layer does not use quantized parameters.
[2] Only the residual branches of MobileNetV2 were quantized (∼70% of MACs and ∼30% of parameters).

Table 5.4: Comparison between the Top-1 validation accuracies of ANA and similar methods described in QNN literature.

Artificial neural networks have proven to be general purpose machine learning systems, and have therefore seen widespread adoption throughout the scientific community. We ascribe the major part of the merit of this success to their fitness to parallel computers. Recent years have seen an

increase in research about computer architectures specifically designed for ANNs, like *tensor processing units* (TPUs). But in order to deploy ANNs on resource-constrained devices such as *embedded devices*, we need to simplify their arithmetic and program spaces further. This ultimately translates into dropping the differentiability which lies at the heart of backpropagation [42, 43].

The idea of quantized neural networks is not new: research in ANNs was born around systems whose weights and activation functions were quantized [28, 29, 30, 35, 37, 32, 39, 33, 38]. Nevertheless, we can now address this *return to the origins* with more tools than those which were available fifty years ago. In particular, we have attacked this problem using functional analysis and probability theory. Similarly to quantum mechanics, to recover continuity and differentiability on a space of non-differentiable functions we need to interpret it as an ensemble. In order to train our systems efficiently on existing software frameworks [55, 56] we had to develop a gradient-based rule similar to backpropagation. Therefore, we have traded some formal correctness against efficient training. Preliminary benchmark results of our algorithm are promising: we plan to analyse more in-depth the annealing strategies for the probability measures according to which the additive noise is distributed.

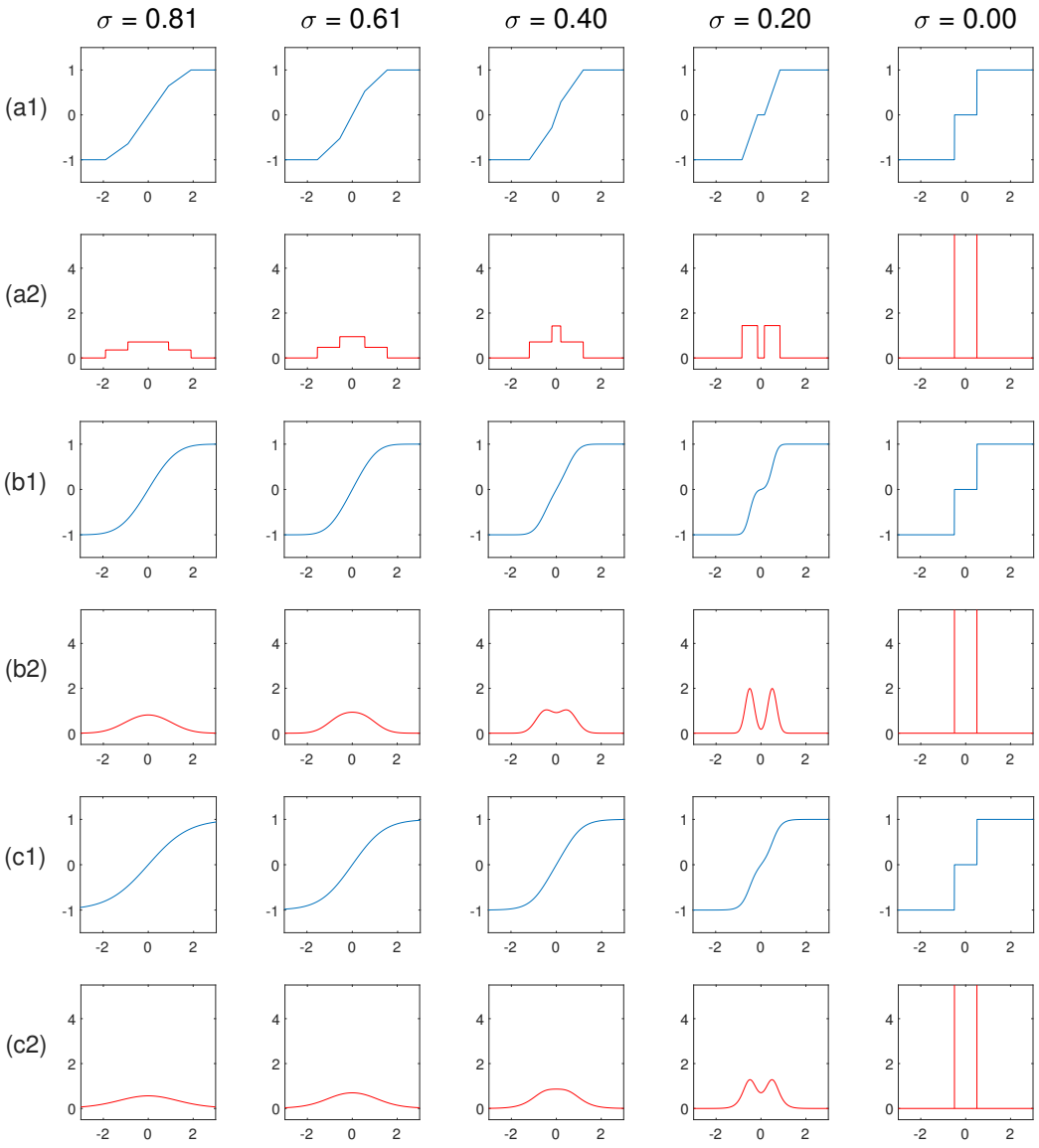


Figure 5.3: The regularisation effect of additive noise on the non-differentiable multi-step function

$$\varsigma(x) = \begin{cases} -1, & \text{if } x \in (-\infty, -0.5) \\ 0, & \text{if } x \in [-0.5, 0.5) \\ 1, & \text{if } x \in [0.5, +\infty)\,. \end{cases}$$

Blue lines represent $(\varsigma * \mu)(x)$, and red lines its derivative $\frac{d(\varsigma*\mu)}{dx}(x)$. Different examples of probability measures μ are depicted for different values of their variances σ: (a1, a2) uniform, (b1, b2) Gaussian and (c1, c2) logistic.

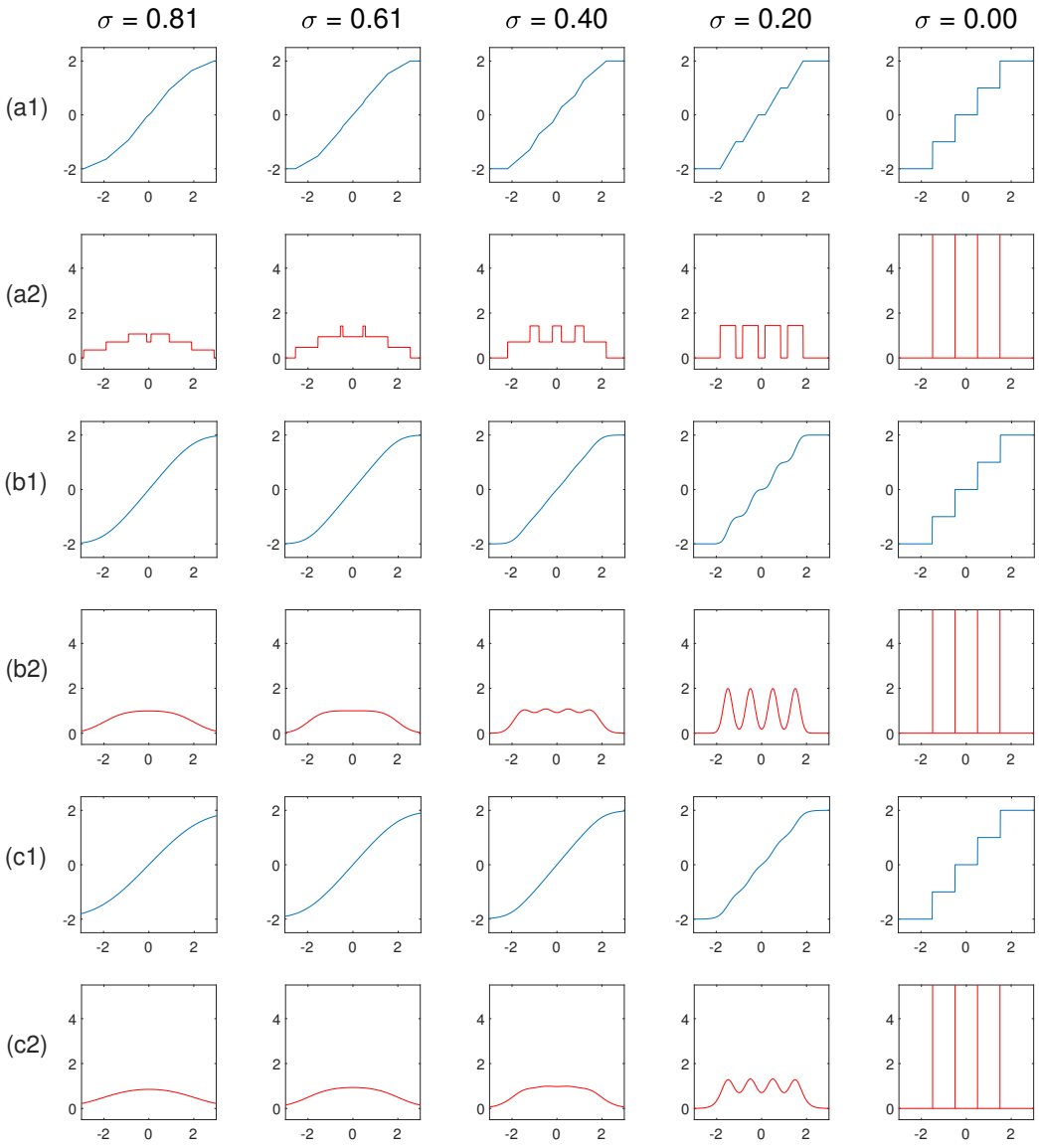


Figure 5.4: The regularisation effect of additive noise on the non-differentiable multi-step function

$$\varsigma(x) = \begin{cases} -2, & \text{if } x \in (-\infty, -1.5) \\ -1, & \text{if } x \in [-1.5, -0.5) \\ 0, & \text{if } x \in [-0.5, 0.5) \\ 1, & \text{if } x \in [0.5, 1.5) \\ 2, & \text{if } x \in [1.5, +\infty)\,. \end{cases}$$

Blue lines represent $(\varsigma * \mu)(x)$, and red lines its derivative $\frac{d(\varsigma*\mu)}{dx}(x)$. Different examples of probability measures μ are depicted for different values of their variances σ: (a1, a2) uniform, (b1, b2) Gaussian and (c1, c2) logistic.

Chapter 6

Conclusions

To get meaningful insights, we need clear questions. Evaluating machine learning systems is a particularly challenging task since it requires answering questions from diverse fields: mathematics, algorithms theory and computer architectures. From a modelling perspective, we need to know which classes of objects (for example, which function spaces) the system can approximate, and how many components it needs to reach a desired degree of approximation (for example, the number of trees in a random forest or the number of neurons and layers in an artificial neural network). From an algorithmic perspective, we need to estimate the complexity of learning and inference algorithms in terms of the size of the individual input sample (usually an array) and of the size of the system's data set. From an engineering perspective, we need to assess the latency and energy consumption of the system when it is run on a given computer architecture.

From a modelling perspective, support vector machines and their kernel extensions [139, 140, 141] can implement every function in a *reproducing kernel Hilbert space* (RKHS) [142], but choosing the kernels and tuning them to acceptable levels of accuracy require time and expertise; moreover, the resulting programs can not be directly reused on related tasks. Decision trees [143, 144, 145] have a branching structure that can be understood by humans (they are so-called *white-box* models) and they can process in a unified way both categorical and numerical data, but they are poor function approximators for all but the simplest functions. Random forests are ensemble machine learning systems based on decision trees [146]. At the time of writing, the theory behind random forests is not fully understood, and just a few efforts have been devoted to their analysis [147, 148]; they seem to be powerful function approximators which do not suffer from overfitting but are also *black-box* models which are not easy to reuse after being trained on a specific task. Artificial neural networks are dense in different spaces of

real functions defined on compact domains [44, 45], and the interest received by these systems in recent years has motivated a detailed analysis of their representational capabilities [149, 150]. With respect to other machine learning systems, artificial neural networks have the advantage of having modular structures: with proper design at modelling time, the representations computed by a hidden layer of a neural network trained on one problem can be reused to solve problems that share similarities with the original one. This property can save practitioners valuable modelling time. We exemplified this point with the analyser of driving perceptions in Section 4.3. All these machine learning systems can directly operate only on spaces of simple entities, usually vector spaces. We have also presented learning systems capable of analysing spaces of probability distributions. As exemplified by the *fingerprint* method in Section 2.2, these systems require complex *ad-hoc* modelling and are often built upon assumptions which limit the number of problems to which they can be applied.

When time constraints are not an issue, it is sufficient to consider statistical metrics. However, in many real-world scenarios, the family of systems that can actually be considered is constrained by the data set size (for learning) or by the structure of the system's programs (for inference). Let N denote the data set size and n denote the size of a generic input instance. The learning algorithm for a support vector machine is a quadratic programming problem [48], and its time cost is $O(nN^2)$. Inference is not as problematic since it requires $O(nN)$ operations. The cost of training a decision tree is $O(nN^2)$, although we must observe that heuristics that take linear time $O(nN)$ have been proposed [151]. The cost of inference is $\Omega(\log(N))$ and $O(N)$, depending on the structure of the grown tree. Random forests just multiply these costs by the number of trees they are composed of and therefore have the same complexity. The most used learning algorithm for artificial neural networks (backpropagation combined with stochastic gradient descent) is an online algorithm with cost $O(nN)$. The cost of inference is $O(n^2)$. The inference algorithms for all these systems (except decision trees) can be highly parallelised. As for the learning algorithms, artificial neural networks seem to be the clear winner, although we must observe that the size n of their inputs is usually greater than that of other algorithms. For example, the inputs of computer vision tasks are often images with thousands of pixels: in these cases, the impact of the input size is not negligible.

Computer architectures metrics are an essential filter, especially for applications with constrained resources, be it latency, energy or both. Support vector machines rely on floating-point arithmetic: this is not a limitation by itself, but it requires ISAs including floating-point instructions. Random forests are useful at many tasks, but they have data-dependent branching

structures which make it hard to use SIMD architectures efficiently. The parallel nature of artificial neural networks and the homogeneity of the computational primitive implemented by each neuron (the dot product) are instead optimal for SIMD computer architectures. Since quantized neural networks are in principle equivalent to classical artificial neural networks (at least on the set of targets represented by continuous functions defined on compact domains), it makes sense to develop specialised hardware accelerators. As we said, learning on bags is a relatively unexplored field and it is consequently hard to make general evaluations about the hardware suitability of the respective systems.

System	Approximation	Reuse	—- Time cost —-		SIMD
			Training	Inference	
SVM	High	✗	$O(nN^2)$	$O(nN)$	✓
DT	Low	✓	$O(nN^2)$	$O(N)$	✗
RF	High	✗	$O(nN^2)$	$O(N)$	✗
ANN	High	✓	$O(nN)$	$O(n^2)$	✓

Table 6.1: Comparison of different supervised machine learning systems. Artificial neural networks (ANNs) have many desirable properties when compared to support vector machines (SVMs), decision trees (DTs) and random forests (RFs). Amongst the systems which have program spaces which are dense in large function spaces, only ANNs can save modelling time. Their learning algorithm has linear time complexity in the data set size, which makes them suitable to solve *big data* problems. Moreover, their computational structure makes them suitable for parallel SIMD computer architectures.

Quantized neural networks can be executed efficiently on specialised hardware accelerators, making their deployment on edge computers affordable. We showed that quantized neural networks can approximate any real continuous function defined on a compact domain with arbitrary accuracy, though the bound on the number of required units is exponential. We have developed *additive noise annealing*, a gradient-based learning algorithm that leverages the role of probability in the regularisation of non-differentiable functions. The interpretation of quantized neural networks, but also of more general artificial neural networks, as ensembles of deterministic programs poses intriguing questions. For example, which classes of objects can such systems approximate? How could we model their evolution? Research on gradient-free probabilistic learning algorithms would be interesting, but could not rely on the support of software libraries and specialised hardware, two facts

that can significantly reduce the pace of innovation in this direction. Are there alternatives to improve the statistical fit of quantized neural networks? The program space of a machine learning system plays a capital role in determining its approximation properties. *Neural architecture search* (NAS) techniques dynamically alter the program space of artificial neural networks. These techniques have been explored using genetic algorithms [152] or reinforcement learning [153], techniques which have slow convergence times and require hardware support which is unaffordable for all but the biggest technology companies. More recently, efficient gradient-based NAS algorithms [154] or randomly structured programs [155] have been proposed. We argue that, in the short-to-medium term, the joint application of gradient-based learning and efficient NAS algorithms will be a more realistic and practical alternative to train quantized neural networks than using probabilistic program spaces and gradient-free learning algorithms.

During this thesis project, we came to realize that learning theory and learning technology are two sides of the same coin. Though not yet established as a discipline by its own, machine learning lives at the intersection of diverse disciplines: it can receive invaluable benefits from collaborations between experts of these disciplines, but it requires them to grow more aware of the relationships between their approach and those of other scientists. We hope that this thesis can serve them as a compass along this path.

Bibliography

[1] P. Halmos, *Naive Set Theory*. Van Nostrand, 1960.

[2] J. L. Bell and M. Machover, *A Course in Mathematical Logic*. North Holland, 1977.

[3] I. N. Herstein, *Topics in Algebra*. John Wiley & Sons, 1975.

[4] P. R. Halmos, *Measure Theory*. Springer, 1974.

[5] G. R. Grimmett and D. R. Stirzaker, *Probability and Random Processes*. Oxford University Press, 2001.

[6] P. A. M. Dirac, *The Principles of Quantum Mechanics*. Oxford University Press, 1948.

[7] C. Dellacherie and P.-A. Meyer, *Probabilities and Potential*. North Holland Publishing Co., 1978.

[8] A. S. Morris, *Measurement and Instrumentation Principles*. Butterworth-Heinemann, 2001.

[9] W. D. Blizard, "The development of multiset theory," *Modern Logic*, vol. 1, pp. 319–352, 1991.

[10] A. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, pp. 433–460, 1950.

[11] J. S. Russell and P. Norvig, *Artificial Intelligence: a Modern Approach*. Prentice Hall, 2010.

[12] S. Amari, *Information Geometry and Its Applications*. Springer, 2016.

[13] M. Campbell, "Knowledge discovery in Deep Blue," *Communications of the ACM*, vol. 42, pp. 65–67, 1999.

[14] M. Campbell, A. J. Hoane, and F. Hsu, "Deep Blue," *Artificial Intelligence*, vol. 134, pp. 57–83, 2002.

[15] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[16] M. E. Glickman, "A comprehensive guide to chess ratings," *American Chess Journal*, vol. 3, pp. 59–102, 1995.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2013.

[18] G. E. Blelloch and B. M. Maggs, "Parallel algorithms," in *Algorithms and Theory of Computation Handbook*, Chapman & Hall, 2010.

[19] C. Slot and P. van Emde Boas, "On tape versus core; an application of space efficient perfect hash functions to the invariance of space," in *Proceedings of the 16th annual ACM Symposium on Theory of Computing*, ACM, 1984.

[20] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*. Morgan Kauffman, 2018.

[21] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice Hall, 1978.

[22] T. G. Dietterich, L. H. Lathrop, and T. Lozano-Pérez, "Solving the multiple instance problem with axis-parallel rectangles," *Artificial Intelligence*, vol. 89, pp. 31–71, 1997.

[23] G. Doran and S. Ray, "A theoretical and empirical analysis of support vector machine methods for multiple-instance classification," *Machine Learning*, vol. 97, pp. 79–102, 2014.

[24] A. C. Rencher, *Methods of Multivariate Analysis*. Wiley, 2003.

[25] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, pp. 179–188, 1936.

[26] K. Pearson, "On lines and planes of closest fit to systems of points in space," *The London, Edinburgh and Dublin Phylosophical Magazine and Journal of Science*, vol. 2, pp. 559–572, 1901.

[27] P. C. Mahalanobis, "On the generalized distance in statistics," *Proceedings of the National Institute of Sciences of India*, vol. 2, pp. 49–55, 1936.

[28] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[29] F. Rosenblatt, "The perceptron: a perceiving and recognizing automaton," tech. rep., Cornell Aeronautical Laboratory, Inc., January 1957.

[30] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, pp. 386–408, 1958.

[31] D. O. Hebb, *The Organization of Behavior: a Neuropsychological Theory*. John Wiley & Sons, Inc., 1949.

[32] F. Rosenblatt, "Principles of neurodynamics: perceptrons and the theory of brain mechanisms," tech. rep., Cornell Aeronautical Laboratory, Inc., March 1961.

[33] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1962.

[34] A. Novikoff, "On convergence proofs for perceptrons," in *Proceedings of the Symposium on the Mathematical Theory of Automata*, Polytechnic Press, 1962.

[35] B. Widrow and M. E. Hoff, "Adaptive switching circuits," tech. rep., Stanford Electronics Laboratories, June 1960.

[36] B. Widrow, "A study of rough amplitude quantization by means of Nyquist sampling theory," *IRE Transactions on Circuit Theory*, vol. 3, pp. 266–276, 1956.

[37] A. Borsellino and A. Gamba, "An outline of a mathematical theory of PAPA," *Il Nuovo Cimento*, vol. 20, pp. 571–581, 1961.

[38] A. Gamba and E. Wanke, "A pattern recognition machine," *Kybernetik*, vol. 4, pp. 69–80, 1968.

[39] A. Gamba, "A multilevel PAPA," *Il Nuovo Cimento*, vol. 26, pp. 176–177, 1962.

[40] M. L. Minsky and S. A. Papert, *Perceptrons*. MIT Press, 1969.

[41] M. L. Minsky and O. G. Selfridge, "Learning in random nets," in *Proceedings of the Fourth London Symposium on Information Theory*, Butterworth, Ltd., 1961.

[42] P. Werbos, *Beyond regression: new tools for prediction and analysis in the behavioral sciences*. PhD thesis, Harvard University, January 1974.

[43] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[44] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, pp. 303–314, 1989.

[45] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, pp. 251–257, 1991.

[46] V. N. Vapnik, *Statistical Learning Theory*. John Wiley & Sons, Ltd, 1998.

[47] L. Rosasco, E. De Vito, A. Caponnetto, M. Piana, and A. Verri, "Are loss functions all the same?," *Neural Computation*, vol. 16, pp. 1063–1076, 2004.

[48] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer, 2006.

[49] Y. LeCun, C. Cortes, and C. J. C. Burges. `http://yann.lecun.com/exdb/mnist/`, 1998.

[50] C. C. Margossian, "A review of automatic differentiation and its efficient implementation," *WIREs Data Mining and Knowledge Discovery*, vol. 9, pp. 1–19, 2019.

[51] R. S. Sutton, "Two problems with backpropagation and other steepest-descent learning procedures for networks," in *Proceedings of the 8th Annual Conference of the Cognitive Science Society*, Lawrence Erlbaum Associates, 1986.

[52] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, pp. 145–151, 1999.

[53] G. E. Hinton. `https://www.cs.toronto.edu/~hinton/coursera_lectures.html`, 2012.

[54] D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization," *CoRR*, 2014.

[55] M. Abadi, P. Barham, J. Chen, J. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, G. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: a system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, USENIX, 2016.

[56] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *Neural Information Processing Systems Workshop 2017*, Neural Information Processing Systems (NIPS), 2017.

[57] G. van Rossum, "Python tutorial," tech. rep., Python Software Foundation, September 2018.

[58] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, Springer, 1974.

[59] W. M. Johnston, J. R. Paul Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, pp. 1–34, 2004.

[60] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: generalization gap and sharp minima," in *Proceedings of Machine Learning Research*, International Conference on Machine Learning (ICML), 2017.

[61] T. M. Heskes and B. Kappen, "On-line learning processes in artificial neural networks," *Mathematical Approaches to Neural Networks*, vol. 51, pp. 199–233, 1993.

[62] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural Networks: Trick of the Trade*, Springer, 1998.

[63] Y. LeCun, "A theoretical framework for back-propagation," in *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann, 1988.

[64] A. Betti and M. Gori, "The principle of least cognitive action," *Theoretical Computer Science*, vol. 633, pp. 83–99, 2016.

[65] N. Ye, Z. Zhu, and R. K. Mantiuk, "Langevin dynamics with continuous tempering for training deep neural networks," in *Advances in Neural Information Processing Systems 30*, Neural Information Processing Systems (NIPS), 2017.

[66] X. Glorot and Y. Bengio, "Understanding the difficulties of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, International Conference on Artificial Intelligence and Statistics (AISTATS), 2010.

[67] P. Stock, B. Graham, R. Gribonval, and H. Jégou, "Equi-normalization of neural netorks," in *International Conference on Learning Representations 2019*, International Conference on Learning Representations (ICLR), 2019.

[68] S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, International Conference on Machine Learning (ICML), 2015.

[69] H. Shimodaira, "Improving predictive inference under covariate shift by weighting the log-likelyhood function," *Journal of Statistical Planning and Inference*, vol. 90, pp. 227–244, 2000.

[70] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?," in *Advances in Neural Information Processing Systems 31*, Neural Information Processing Systems (NIPS), 2018.

[71] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[72] M. Zhou, T. Liu, Y. Li, D. Lin, E. Zhou, and T. Zhao, "Towards understanding the importance of noise in training neural networks," in *Proceedings of Machine Learning Research*, International Conference on Machine Learning (ICML), 2019.

[73] M. D. Fairchild, *Color Appearance Models*. John Wiley & Sons, Ltd, 2005.

[74] A. Krizhevsky, V. Nair, and G. E. Hinton. `https://www.cs.toronto.edu/~kriz/cifar.html`, 2014.

[75] 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 (CVPR)*, IEEE, 2009.

[76] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, pp. 211–252, 2015.

[77] 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 *Computer Vision - ECCV 2014*, Springer, 2014.

[78] J. Huang and D. Mumford, "Statistics of natural images and models," in *Proceedings. IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, IEEE, 1999.

[79] D. H. Hubel and T. N. Wiesel, "Receptive fields of single neurones in the cat's striate cortex," *The Journal of Physiology*, vol. 148, pp. 574–591, 1959.

[80] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of Physiology*, vol. 160, pp. 106–154, 1962.

[81] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *The Journal of Physiology*, vol. 195, pp. 215–243, 1968.

[82] D. H. Hubel and T. N. Wiesel, "Functional architecture of macaque monkey visual cortex," *Proceedings of the Royal Society of London*, vol. 198, pp. 1–59, 1977.

[83] K. Fukushima, "Neocognitron: a self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, pp. 193–202, 1980.

[84] K. Fukushima, "Neocognitron: a hierarchical neural networks capable of visual pattern recognition," *Neural Networks*, vol. 1, pp. 119–130, 1988.

[85] 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," *Neural Computation*, vol. 1, pp. 541–551, 1989.

[86] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 11, pp. 2278–2324, 1998.

[87] E. P. Simoncelli and B. A. Olshausen, "Natural image statistics and neural representation," *Annual Review of Neuroscience*, vol. 24, pp. 1193–1216, 2001.

[88] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, Neural Information Processing Systems (NIPS), 2012.

[89] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations 2015*, International Conference on Learning Representations (ICLR), 2015.

[90] M. Lin, Q. Chen, and S. Yan, "Network in network," in *International Conference on Learning Representations 2014*, International Conference on Learning Representations (ICLR), 2014.

[91] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2015.

[92] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2016.

[93] F. Chollet, "Xception: deep learning with depthwise separable convolutions," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2017.

[94] 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," *CoRR*, 2017.

[95] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: inverted residuals and linear bottlenecks," in *2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2018.

[96] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, pp. 504–507, 2006.

[97] P. Vincent, H. Larochelle, Y. Bengio, and P. A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th International Conference on Machine Learning*, International Conference on Machine Learning (ICML), 2008.

[98] M. Hein and A. J.-Y., "Intrinsic dimensionality estimation of submanifolds in $\mathbb{R}^d$," in *Proceedings of the 22nd International Conference on Machine Learning*, International Conference on Machine Learning (ICML), 2005.

[99] Z. Yang, D. Yang, C. Dyer, X. He, A. J. Smola, and E. H. Hovy, "Hierarchical attention networks for document classification," in *Proceedings of the 15th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2016.

[100] D. Bahdanau, K. H. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, 2014.

[101] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional neural networks: visualizing image classification models and saliency maps," *CoRR*, 2013.

[102] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, ACM, 1995.

[103] G. E. Hinton, "Learning multiple layers of representation," *Trends in Cognitive Sciences*, vol. 11, pp. 428–434, 2007.

[104] Y. Bengio, "Learning deep architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, pp. 1–127, 2009.

[105] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[106] 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, 2016.

[107] J. Redmon and A. Farhadi, "YOLOv3: an incremental improvement," *CoRR*, 2018.

[108] D. Gandhi, L. Pinto, and A. Gupta, "Learning to fly by crashing," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017.

[109] A. Loquercio, A. I. Maqueda, C. R. del Blanco, and D. Scaramuzza, "DroNet: learning to fly by driving," *IEEE Robotics and Automation Letters*, vol. 3, pp. 1088–1095, 2018.

[110] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, T. Guez, A. Hubert, L. Baker, A. Lai, M. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[111] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, 2019.

[112] E. Quiñones, M. Bertogna, E. Hadad, A. J. Ferrer, L. Chiantore, and A. Reboa, "Big data analytics for smart cities: the H2020 CLASS project," in *Proceedings of the 11th ACM International Systems and Storage Conference*, ACM, 2018.

[113] P. Mayer, M. Magno, and L. Benini, "Self-sustaining acoustic sensor with programmable pattern recognition for underwater monitoring," *IEEE Transactions on Instrumentation and Measurement*, vol. x, p. x, 2019.

[114] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini, "A 64mW DNN-based visual navigation engine for autonomous nano-drones," *CoRR*, 2019.

[115] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd annual International Symposium on Computer Architecture*, IEEE, 2016.

[116] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: an architecture for ultralow power binary-weight CNN acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, pp. 48–60, 2018.

[117] L. Cavigelli and L. Benini, "Extended bit-plane compression for convolutional neural network accelerators," in *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, IEEE, 2019.

[118] M. Courbariaux, Y. Bengio, and J.-P. David, "Binary connect: training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems 28*, Neural Information Processing Systems (NIPS), 2015.

[119] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: towards lossless CNNs with low-precision weights," in *International Conference on Learning Representations 2017*, International Conference on Learning Representations (ICLR), 2017.

[120] R. Ao, T. Zhang, S. Ye, J. Li, W. Xu, X. Qian, x. Lin, and Y. Wang, "ADMM-NN: an algorithm-hardware co-design framework of DNNs using alternating direction methods of multipliers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.

[121] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems 29*, Neural Information Processing Systems (NIPS), 2016.

[122] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: training neural networks with low precision weights and activations," *Journal of Machine Learning Research*, vol. 18, pp. 1–30, 2018.

[123] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Computer Vision - ECCV 2016*, Springer, 2016.

[124] X. Lin, C. Zhao, and W. Pan, "Towards accurate binary convolutional neural network," in *Advances in Neural Information Processing Systems 30*, Neural Information Processing Systems (NIPS), 2017.

[125] Z. Liu, B. Wu, W. Luo, X. Yang, W. Liu, and K.-T. Cheng, "Bi-real net: enhancing the performance of 1-bit CNNs with improved representational capability and advanced training algorithm," in *Computer Vision - ECCV 2018*, Springer, 2018.

[126] B. Zhuang, C. Shen, and I. Reid, "Training compact neural networks with binary weights and low precision activations," *CoRR*, 2018.

[127] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid, "Structured binary neural networks for accurate image classification and semantic segmentation," in *2019 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2019.

[128] J. Choi, S. Venkataramani, V. Srinivasan, K. Gopalakrishnan, Z. Wang, and P. Chuang, "Accurate and efficient 2-bit quantized neural networks," in *Proceedings of the 2nd SysML Conference*, Conference on Systems and Machine Learning (SysML), 2019.

[129] B. Jacob, S. Kligys, B. Chen, and M. Zhu, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2018.

[130] Q. Li and S. Hao, "An optimal control approach to deep learning and applications to discrete-weight neural networks," in *Proceedings of Machine Learning Research*, International Conference on Machine Learning (ICML), 2018.

[131] E. J. McShane, "The calculus of variations from the beginning through optimal control theory," *SIAM Journal on Control and Optimization*, vol. 27, pp. 916–939, 1989.

[132] 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," *Neural Networks*, vol. 100, pp. 49–58, 2018.

[133] E. Agustsson, F. Mentzer, M. Tschannen, L. Cavigelli, R. Timofte, L. Benini, and L. Van Gool, "Soft-to-hard vector quantization for

end-to-end learning compressible representations," in *Advances in Neural Information Processing Systems 30*, Neural Information Processing Systems (NIPS), 2017.

[134] J. Yang, X. Shen, J. Xing, X. Tian, H. Li, B. Deng, J. Huang, and X. Hua, "Quantization networks," in *2019 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2019.

[135] W. Severa, C. M. Vineyard, R. Dellana, S. J. Verzi, and J. B. Aimone, "Training deep neural networks for binary communication with the Whetstone method," *Nature Machine Intelligence*, vol. 1, pp. 86–94, 2019.

[136] C. Louizos, M. Reisser, T. Blankevoort, E. Gavves, and M. Welling, "Relaxed quantization for discretized neural networks," in *International Conference on Learning Representations 2019*, International Conference on Learning Representations (ICLR), 2019.

[137] L. Ambrosio, N. Fusco, and D. Pallara, *Functions of bounded variation and free discountinuity problems*. Clarendon Press Oxford, 2000.

[138] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *CoRR*, 2013.

[139] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the 5th Annual Workshop on Computational Learning Theory*, ACM, 1992.

[140] C. Cortes and V. N. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, 1995.

[141] M. A. Aizerman, E. M. Braverman, and L. I. Rozoner, "Theoretical foundations of the potential function method in pattern recognition learning," *Automation and remote control: a publication of the Russian Academy of Sciences*, vol. 25, pp. 821–837, 1964. Translated from the Russian "Autmotika i Telemekhanika", volume 25, pages 921–936.

[142] B. Schölkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2001.

[143] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification And Regression Trees*. Chapman & Hall, 1984.

[144] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[145] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kauffman Publishers Inc., 1993.

[146] L. Breiman, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, 2001.

[147] G. Biau, "Analysis of a random forest model," *Journal of Machine Learning Research*, vol. 13, pp. 1063–1095, 2012.

[148] M. Denil, D. Matheson, and N. De Freitas, "Narrowing the gap: random forests in theory and practice," in *Proceedings of the 31st International Conference on Machine Learning*, International Conference on Machine Learning (ICML), 2014.

[149] H. Bölcskei, P. Grohs, G. Kutyniok, and P. Petersen, "Optimal approximation with sparsely connected deep neural networks," *SIAM Journal on Mathematics of Data Science*, vol. 1, pp. 8–45, 2019.

[150] I. Daubechies, R. DeVore, S. Foucart, B. Hanin, and G. Petrova, "Nonlinear approximation and deep (ReLU) networks," *CoRR*, 2019.

[151] J. Su and H. Zhang, "A fast decision tree learning algorithm," in *Proceedings of the 21st National Conference on Artificial Intelligence*, AAAI Press, 2006.

[152] K. O. Stanley and R. Mikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, pp. 99–127, 2002.

[153] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *International Conference on Learning Representations 2017*, International Conference on Learning Representations (ICLR), 2017.

[154] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, Y. Jia, and K. Keutzer, "FBNet: hardware-aware efficient ConvNet design via differentiable neural architecture search," *CoRR*, 2018.

[155] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," *CoRR*, 2019.