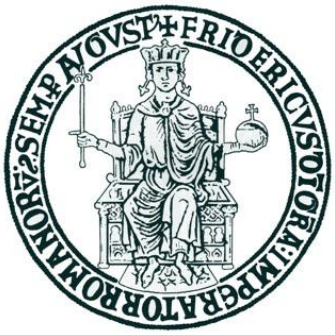
Università Degli Studi Di Napoli
Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Matematica e Applicazioni
"Renato Caccioppoli"

Tesi per il dottorato di ricerca in
Scienze Matematiche e Informatiche
XXXII Ciclo

Quantum Machine Learning: A Comparison Between Quantum and Classical Support Vector Machine

Claudio Pipicelli

Contents

To Alessandro, my young niece
To my family

Acknowledgements

At the end of the run, I would like to express my sincere gratitude to the key persons who supported me throughout the last three years.

Firstly, I wish to strongly thank my scientific advisor, Prof. Giovanni Acampora. His guidance, motivation and incitements have been allowing me to train and increase my research attitudes.

Besides my supervisor, I am grateful to Dr. Autilia Vitiello and Dr. Mariacarla Staffa for being present in many situation I needed.

A special mention goes to my PhD colleagues, among various Federico II's departments and all my friends. The enjoyable moments I spent with them helped me to face this almost four-years run less hardly.

Finally, thanks go to my big and magnificent family and to my beloved nephew for the tremendous support and limitless endurance. I dedicate this thesis to them all.

Summary and foreword

This thesis is mainly focused on the study of Quantum Support Vector Machine (QSVM), a very important member of the recent and innovative Quantum Machine Learning field, and its comparison with conventional Support Vector Machine (SVM) [Cortes and Vapnik, 1995].

Machine Learning [Bishop, 2006], a prominent subset of Artificial Intelligence, perhaps the most developed in the last few decades, is progressively pursuing significant growth prospects and achieving important results in many fields nowadays, such as computer vision, speech recognition, natural language processing, robot control, predictive maintenance, and so on, with a growing spread in other sectors, for example health care, manufacturing, education, financial modeling, policing, and marketing [Jordan and Mitchell, 2015]. Machine learning algorithms can figure out how to perform important tasks by generalizing from exhibited examples. This is conceptually a very attractive alternative to manually design an algorithm to perform the same task. Moreover, in several research and application fields, this approach is often feasible and cost-effective where explicit, conventional programming is not. As more data becomes available, in principle more ambitious problems can be tackled by machine learning techniques. However, every machine learning approach to solve a not trivial problem postulates the access to sufficient amounts of useful data and requires accordingly to solve an optimization problem: in fact, these are determinants which are the most appropriate choices for parameters (and possibly hyperparameters), in order to improve the execution of a specific task using a particular model.

The adoption of learning algorithms characterized by high computational complexity and the use of large amounts of training data (the so colled "big data"), can therefore make the application of effective machine learning really infeasible to solve some classes of useful problems, even on the fastest and most expensive electronic digital computers.

Moreover, because Moore's Law [Moore et al., 1965], the principle that has powered the information technology revolution since the beginning of microelectronics in 1960s, seems that is reaching its end as lithographic processes shrink down [Chien and Karamcheti, 2013; Markov, 2014], approaching the extreme ultraviolet (from deep ultraviolet), so to involve the construction of transistors constituted by an increasingly smaller number of atoms, other models of computation urgently need to be evaluated and implemented.

Inspired by quantum mechanics laws, one really encouraging computing model is quantum computing [Feynman, 1982; Nielsen and Chuang, 2010], which includes both ad hoc solutions (such as D-Wave quantum annealing machines [Adachi and Henderson, 2015; Finnila et al., 1994; Rønnow et al., 2014]) both universal, general purpose solutions, from several multinational information technology company such as I.B.M., Google, Intel, Microsoft as well as promising startups such as Rigetti and Xanadu.
Quantum computers leverage peculiar quantum effects, such as superposition and entanglment [Chuang and Shor, 2018b; Gottesman and Chuang, 1999; Rycerz et al., 2015; Simon, 1997], to evaluate simultaneously more computational paths, but they are subjected to stringent limits, both for the current shortage of quantum bits and gates, both

for the ineluctable presence of various kinds of single and multi-qubit errors, decoherence phenomena and so on [Chuang and Harrow, 2018; Chuang and Shor, 2018c; Nielsen and Chuang, 2010], which, in some ways, make quantum computers more similar to old analogue computers (almost never used in the last few decades) than digital ones, especially with regard to the reproducibility of results and tolerance to errors.

Anyway, quantum computing is quite a promising approach to solve some kind of hard problem, such as high complexity optimization problems, significantly faster and more efficient than classical digital computers.

The first and most famous quantum algorithms [Chuang and Shor, 2018b; Nielsen and Chuang, 2010] exhibit indeed considerable comptutational advantages with rispect to conventional computers: Deutsch-Jozsa algorithm [Deutsch and Jozsa, 1992] allows resolution of a specific class of "on promise" problems with just one function ("oracle") evaluation; Shor algorithm [Shor, 1994] provides an exponential speed-up for prime number factorization (anyway, until now the only real implementations are able to work with very small numbers [Monz et al., 2016]); Grover algorithm [Grover, 1996] allows a quadratic speed-up for searching an unstructured database. Very recently, both the scientific and non-specialist press has emphasized Google's bold statement that it has achieved the so called "quantum supremacy"[Arute et al., 2019]: to complete a specific task, used as benchmark, a million times, Google's Sycamore quantum processor takes about 200 seconds, while Google's forecast indicates that the equivalent task for a state-of-the-art classical supercomputer would take approximately 10,000 years.

In the last few years, Quantum Machine Learning is emerging as the union between quantum computing and information systems and machine learning [Biamonte et al., 2017; Lloyd et al., 2013; Otterbach et al., 2017; Ristè et al., 2017; Ruan et al., 2016; Schuld et al., 2015a; Wilson et al., 2018], including quantum neural networks [Carleo et al., 2018; Chen et al., 2017; Da Silva et al., 2012; da Silva et al., 2016; Fard et al., 2018; Gupta and Zia, 2001; Hu, 2018; Liu et al., 2013; Rebentrost et al., 2017; Schuld et al., 2014, 2015b; Verdon et al., 2017; Zhao et al., 2018] and quantum support vector machines[Ding et al., 2019; Havlíček et al., 2019; Havlivcek et al., 2018; Rebentrost et al., 2014; Schuld and Killoran, 2018; Willsch et al., 2019].

There are multiple reasons to support this union. In fact, in recent years the application of machine learning algorithms in multiple sectors (industrial applications, automation technologies, autonomous driving systems, road surveillance, facial recognition, bank fraud detection and eventually prevention, and so on) requires more and more computing power , also because often these algorithms are applied to the so called "Big Data". As a result, there has been a rapid transition from general purpose CPU code execution to massive GPU adoption, as well as the development of ad hoc solutions, in the form of FPGAs and ASICs carefully designed to support high performance. The preference for these hardware configurations is related to the fact that the main machine learning algorithms are based on the repeated application of relatively simple processing kernels applied to large amounts of data; moreover, these kernels are generally massively based on linear algebra operations, therefore well adaptable to special purpose accelerators consisting of hundreds or thousands of computational cores, albeit simpler than a general purpose CPU, or even networks of appropriately designed calculation elements.

In this context, the idea of having a quantum computer perform at least significant portions of the machine learning algorithms is decidedly plausible. First, the evolution of quantum systems is described by operators in Hilbert spaces, which suggests the possibility of applying specific machine learning strategies by breaking them down into operators that characterize evolution starting from an initial state of the system. In addition, efficient implementations of quantum algorithms based on linear algebra, such as Fourier Transform and Principal component analysis, have already been demonstrated in the sci-

entific literature [Biamonte et al., 2017; Bowden et al., 2002; Liu and Rebentrost, 2018; Lloyd et al., 2014; Nam et al., 2018]. Furthermore, the intrinsically probabilistic nature of the result of the processing of a quantum computer should not affect significantly the goodness of the result, at least in principle, in machine learning applications, because they are systems that don't guarantee absolute certainty, but they provide results that can be used in a statistical sense (provided, of course, that the errors introduced by the quantum computer are under control, to be sufficiently modest). Furthermore, especially in the inferential phase, high precision in the execution of the single calculations is often not required: indeed, the emerging result of a suitably trained system is therefore stable for small (and sometimes even relatively large) errors of the single calculation units. So this aspect also suggests that the error rate found so far on quantum computers could affect machine learning applications less than other fields of application.

Recently, a quantum implementation has been conceived for a machine learning technique widely used in classification problems: the Support Vector Machine (SVM) was born embryonic from the simple idea of identifying the hyperplane that guarantees maximum separation between linearly separable sets [Burges, 1998; Cortes and Vapnik, 1995]; this algorithm is generalized, through the so-called kernel trick, to classes that are not linearly separable, usually immersing the feature space in a space of higher dimensionality, in which it is possible to identify an optimal hyperplane for classification; in addition, even when it is not possible to identify a separation surface between the classes subject to the classification problem, there is a generalization of SVM based on the least squares technique [Suykens et al., 2000; Suykens and Vandewalle, 1999]. Therefore, a quantum version of Least Square Support Vector Machine was developed and it is receiving a lot of attention for the possibility of running also on current quantum systems, characterized by an extremely small number of functioning qubits [Havlivcek et al., 2018; Schuld and Killoran, 2018]. Just as Google's quantum computer has shown excellent performance in an area of little concrete interest, working on an artificial case designed specifically to be difficult for a classic computer, so the first implementation of Quantum Support Vector Machine (QSVM) has shown interesting performance applied to artificial datasets generated to put it under the best conditions in comparison with a classic SVM [Havlíček et al., 2019].

In this paper, I have worked on the application of Quantum Support Vector Machine algorithm, that runs on near term quantum processors from I.B.M., through IBM Quantum Experience cloud service, to a set of supervised machine learning case studies and I compared its performance with classical Support Vector Machine algorithm; net of the enormous hype surrounding the proliferation of quantum technologies in recent years, are we beginning to glimpse an application of real interest in which quantum systems, albeit with limitations, offer concrete improvements already now?

Chapter 1

Introduction

1.1 Classical computing

Since ancient times, humanity has been interested in the development of methods that would allow solving some classes of problems, operating in a uniform, repeatable way. Therefore, throughout the history of Mathematics, techniques and strategies have been developed to obtain exact or at least approximate solutions to problems of the most disparate nature. Classic examples can be considered the iterative method attributed to the Babylonians to extract the square root, the iterative strategy exposed in Euclid's Elements to determine the greatest common divisor between two natural numbers or the Sieve of Eratosthenes, which gives step by step instructions for quickly removing all non-prime numbers from a defined set of numbers (for instance, between 1 and 100) until only prime numbers are left. So there has always been considerable interest in identifying a finite sequence of well-defined, (as much as possible) unambiguous instructions whose purpose is to solve a class of problems or to perform a calculation; informally, such kind of procedures is named "algorithm", a word that has its roots in Latinizing the name of a Persian mathematician, al-Khwarizmi. During first decades of the 1900s, computability theory was born, as part of a profound process of revision of the foundations of mathematics. Indeed, in the 19th century the discovery of consistent alternative to Euclidean geometry (hyperbolic and elliptic geometry) started a crisis of the foundations of mathematics, a crisis progressively aggravated by developments such as the Cantor's paradox ("there is no greatest cardinal number") or the Russell paradox (concerning "the set of all non-self-membered sets"). So, to avoid further confusion and satisfactorily answer paradoxical results, a new and more rigorous foundation for mathematics was necessary. Therefore it was in that period that mathematicians wondered about the possibility of formalizing the concept of algorithm and of determining which functions are in principle computable, starting from certain set of assumptions. For example, what problems can a man solve by using pen and paper? Is there an effective method to check if every well-formed formula of propositional logic is true or false? And for first-order logic? And so on. Driven by this lively and varied research, computability theory quickly began to develop, producing various formal models of effective computation, such as Godel and Herbrand's partial recursive functions, Church's λ-calculus, Turing machines, Kleene's equation calculus.

Under very general hypotheses, all those formal models describe classes of computable functions which are proved to be equivalent; moreover, the Church–Turing thesis (also known as computability thesis) states that any function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing machine, so to exclude the possibility of a more general model of computation.

Although tools and "machines" have been designed or at least conceived since an-

cient times to carry out mathematical operations, for a long time they have been of rather limited use, for example various kind of abacus were useful mainly to carry out additions and subtractions, but eventually also multiplications and divisions, seen respectively as repeated additions and subtractions. The first appearance of abacus was over 4 thousand years ago in Mesopotamia: Sumeric abacus had a table of successive columns which delimited the successive orders of magnitude of their sexagesimal number system. Obviously the abacus cannot be considered a mechanical calculator as it does not have mechanisms: human operator must perform all the operations manually, nothing happens automatically. Blaise Pascal, instead, invented Pascaline, a mechanical instrument with an automatic carry mechanism to add and subtract two numbers directly and to perform multiplication and division through repeated addition or subtraction. During 1800, Charles Babbage created at first a difference engine, i.e. a special-purpose automatic mechanical calculator designed to tabulate logarithms and trigonometric functions by evaluating finite differences to create approximating polynomials. In general, a difference engine can interpolate any sufficiently regular function by using a small set of polynomial coefficients, which were tabulated using the method of divided differences, i.e. a recursive division process. Following the failures with several difference engine prototypes, Babbage then tried to develop the Analytical Engine, a mechanical computing device which aspired to be a general-purpose computer; someway it anticipated the first digital computers of the following century, with a sort of arithmetic logic unit, control flow in the form of conditional branching and loops, and integrated memory. Babbage also established a collaboration with Ada Lovelace, who developed a way to calculate Bernoulli numbers using the Analytical Engine, so she has been described as the first computer programmer (in her honor, in the late 70s of the twentieth century, a programming language bearing her name was developed). While Babbage was never able to complete any of his machines due to inadequate funding and technical troubles, the Analytical Engine was its most successful achievement.

Beside the formulation of a mathematical theory of computation, since around 1930s the first electro-mechanical computers were build and soon after them the first digital electronic computers also, based on thermionic valves (tubes) and using mostly binary arithmetic and Boolean logic. During the World War II, the Allies developed several complex machines to decode German messages; for example, in 1943 they made the Colossus: it consisted of about 2,400 valves and was the first partially programmable electronic computer. It allowed to decode encrypted messages generated even by the famous German machine Lorenz SZ40/42. Konrad Zuse, in 1941 developed Z3, based on the binary system (implementing a Leibniz's idea). Z3 used about 2600 relays, i.e. electromechanical switches. The IBM Automatic Sequence Controlled Calculator (ASCC), best knwon as Harvard Mark I, was put into operation in 1944 and it was a special purpose programmable machine to solve numerically differential equations. ENIAC (Electronic Numerical Integrator and Calculator) is considered the world's first fully operational electronic general-purpose computer: the project was developed by Eckert and Mauchly and it was funded by the U.S. Army and became operational during World War II, but it was not publicly disclosed until 1946 and it remained in operation until 1955. This enormous machine, with 18,000 vacuum tube (it had a 10-bit register), was used supposedly for computing artillery firing tables, but the ENIAC provided even conditional jumps and it was the first truly programmable computer, which clearly distinguished it from earlier special purpose calculators, because it could be used for any type of calculation: in principle, it was equipotent to a universal Turing machine (although in fact subject to the constraints of a finite memory). EDVAC (Electronic Discrete VAriable Computer) and IAS (Institute for Advanced Study) computer were the first computer with an internal memory that could store program instructions .

Although, for some decades, analog computers were used in scientific and industrial applications, because at the time they were typically faster than digital computers, revealing especially well-suited to representing situations described by differential equations [Care, 2010], they started to become obsolete for general purpose application as early as the 1950s, because there were serious concerns about analogue computing, such as its limitations on scale, reliability (noise and temperature sensitivity, for example), and long-running error-free computation (by virtue of errors accumulation processes).

Indeed, during the 1950s to 1970s, digital computers became more economical and precise, offered greater flexibility and largely replaced analog computers. Their success depended even on reliability and reproducible results: the choice to encode all program and data information with sequences of binary digits, only zeros and ones, represented with quite distinct high or low voltage signals, simplifies and strengthens information storage and communication.

A fundamental point, around 1945, was the definition of the so called von Neumann architecture, i.e. a computer architecture which consists of a central processing unit (CPU), that contains (at least) an arithmetic logic unit (which implements the fundamental arithmetic and Boolean logic operations) and a control unit (which manages the sequential program execution and conditional instructions), then central working (usually volatile) memory that stores data and instructions, external persistent mass storage and usually input and output devices (which, respectively, allows users to feed the computer with data and programs or transmit the computation's results and other computer's information to the user). Almost all digital computer made since then are based on von Neumann paradigm or some variant of it; for example, in the Harvard paradigm there are separate memories for the program code and for the data to be processed.

In the second half of 1900, other computational models, more adequate to be concretely implemented with the available technology, such as Unlimited Register Machine, were conceived and it was shown that, under appropriate hypotheses, they were computationally equivalent to Turing machines.

In the meanwhile, the the point-contact transistor was invented in 1947 by William Shockley, John Bardeen and Walter Brattain at Bell Labs, followed by the bipolar junction transistor in 1948. From 1955 onward transistors replaced vacuum tubes in computer designs, giving rise to a new generation of computers. Compared to vacuum tubes, transistors have many advantages: they are smaller, require less power and usually last longer.

The MOSFET (metal-oxide-semiconductor field-effect transistor), also known as the MOS transistor, was later invented by Mohamed Atalla and Dawon Kahng at Bell Labs in 1959, which led to the mass-production of MOS transistors for a wide range of uses. The MOSFET has since become the most widely manufactured device in history. With its high scalability, and much lower power consumption and higher density than bipolar junction transistors, the MOSFET made it possible to build high-density integrated circuits (ICs), allowing the integration of several thousands transistors in a single IC.

The monolithic integrated circuit was first introduced as a concept by British radar engineer Geoffrey Dummer on May 7, 1952. It was later developed by Jack Kilby and Robert Noyce, and successfully demonstrated on September 12, 1958. Then on April 25th, 1961 the first patent was awarded to Robert Noyce for an integrated circuit.

The scientific and commercial success of deterministic general purpose electronic digital computing was driven by the rapid, substantially exponential growth of both computational power and available memory.

Around the mid-60s, it was observed that the average cost per component is nearly inversely proportional to the number of components, for small integrated circuits, but as more components are added, decreased yields of whole IC more than compensate for the increased complexity, tending to raise the average cost per component. Thus, the function

that expresses the dependence of the average price of a circuit component (transistor or logic gate) on the number of components used, for a fixed production process, is a convex function, so there is a minimum cost at any given time in the evolution of the technology.

The so called Moore's law predicted that such optimal circuit size, i.e. the number of transistor for which average unitary price reaches its minimum, doubles every year[Moore et al., 1965]. Afterwards, the prediction was changed so that the optimal number of components in integrated circuits double every eighteen or twenty-four months.

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000."[Moore et al., 1965]

Doubling the transistor budget for integrated circuits has allowed on the one hand the memory capacity to expand rapidly, on the other hand it has made possible to create ever more sophisticated microprocessors. In 1968 Gordon Moore and Bob Noyce founded Intel, short for "integrated electronics" The first commercial single-chip microprocessor, the Intel 4004, was developed by Federico Faggin, using his silicon-gate MOS IC technology, with Intel engineers Marcian Hoff and Stan Mazor, and Busicom engineer Masatoshi Shima. It was a 4-bit central processing unit (CPU) released by Intel Corporation in 1971 on a 10 μn process and it was originally designed and built for Busicom's 141-PF electronic calculator: it was the most advanced integrated circuit (IC) design undertaken up until then[1]. The Intel 4004 had 2250-2300 transistor and it operated at 108 KHz frequency. Only one year later, Intel released the 8008 microprocessor, with 3500 transistor, still on a 10 μn process, with 8 bit registers and doubled clock. With the new 6 μn process, in 1974 Intel, which became an important chip manufacturer, released 8080 8-bit microprocessor, with 6000 transistor and 2 MHz clock rate. His second 8-bit microprocessor became one of the first widespread microprocessors in the world: 8080 became indeed the engine of the Altair 8800, the first prototype "personal" computer and it was the original target CPU for the famous CP/M operating systems. Distributed in mounting kit or already assembled, Altair was an unexpected success, and its manufacturer, MITS, earned its first supply contract together with Microsoft (at the time Micro-Soft), which provided the Altair BASIC programming language.

With the rapidly increasing transistor budget, microprocessors have progressively improved by increasing some crucial parameters or by adopting a plethora of increasingly refined solutions[Hennessy and Patterson, 2011], for example:

- increasing the number (from a few to many tens or a few hundreds), flexibility (ad hoc usage - such as accumulator, index, loop control, etc. - or general purpose) and size of the registers (usually, power of 2, from 4 bits to 8, 16, 32, 64 bits);

- expanding Arithmetic and Logic Unit (ALU), to manipulate longer sequence of bits (for some simple operations, such addition, subtraction, AND, OR and so on, transistor budget depends linearly on word size, i.e. the number of bits processed in one elementary step of computation, but for complex operations, such as multiplication and division, the budget is usually at least quadratic on word size);

- introducing pipelined execution, dividing incoming instructions execution into a series of stages, i.e. sequential steps, to allow increasing CPU throughput, at the cost of worst latency and virtually pipeline bubbles;

[1] (`http://www.vintagecalculators.com/html/busicom_141-pf_and_intel_4004.html`

- superscalar design, using more parallel pipelines, to extract instruction level parallelism, i.e. trying to execute instructions without mutual data dependency at the same time in different (rarely perfectly symmetric) pipelines;
- Out Of Order Execution (OoOE), to allow instructions to begin execution as soon as their operands are ready, more accurately in an order governed by the availability of input data and physical execution units, rather than by their original order in a program;
- Speculative Execution, to reduce the cost of conditional branch instructions using schemes that predict the execution path of a program based on the history of branch executions, even at the risk of having to cancel part of the work done, in case of mis-prediction; rarely eager execution is used, a form of speculative execution where both sides of the conditional branch are executed, than only the results from right one are kept;
- Simultaneous multithreading (SMT), to permits multiple independent software threads of execution to better utilize the execution resources provided;
- adding specialized units, such as:
  - Memory Managment Unit (MMU), to provide the virtualization of memory access;
  - Floating-Point Unit (FPU, aka mathematical co-processor or Numeric Processor Unit), to allow the rapid execution of arithmetic operations and transcendental functions with - "approximate" - real numbers (floating-point is indeed a mechanism to encode a "good enough" subset of real numbers, with a trade-off between range and precision; since at least the last 35 year almost all processors adhere to the IEEE 754 format);
  - special cache memories, to speed up memory access, exploiting the principles of spatial and temporal locality of reference (multilevel cache memory hierarchies have been gradually adopted);
  - Translation Lookaside Buffers (TLB), a sort of small address-translation cache, to accelerate virtual memory to physical memory indexing (even for TLBs, multilevel cache memory hierarchies have been gradually adopted);
  - Branch Prediction Unit (BPU), to predict the status, taker or not taken, of a branch, i.e. to accelerate selective and iterative structures execution;
  - Single Instruction Multiple Data units (SIMD), to speed up array operations, issuing the same instruction simultaneously to several sequential elements in an array (usually to speed up vector processing);
  - Hardware Data Prefetch (to copy speculatively into the fast cache memory the data needed before they are actually used).

1.1.1 A note on choosing x86 CPUs as a reference

In formulating concrete examples and evaluating some trends in the ICT market, I will mainly refer to general purpose processors based on the x86 ISA conceived by Intel and to its 64-bit extension developed by AMD. This choice is motivated by multiple considerations. First of all, it is the most widespread type of processor in the world, if we take into consideration laptops, desktops, workstations, HPC (High Performance Computing) and servers (excluding, that is, controllers and embedded processors, smartphones and IoT devices). Moreover, this is probably the longest-running CPU family in

history, as the progenitor, i8086, was introduced in 1978: this ineluctably also entails deleterious consequences, since a series of compromises that were reasonable four decades ago, when the transistor budget was a few tens of thousands of elements (for example, the scarcity of architectural registers, variable-length instructions op-codes, instructions that combine both access to memory and logical-arithmetic operations), are much less appropriate in the present days, when it is possible to economically make chips with a few billion transistors. Anyway, in the last twenty-five years, almost all x86 CPUs uncouple an in order front-end that decodes x86 instructions and a parallel, out-of-order back-end, introducing many registers for renaming operations, mitigating the defects due to the age of the ISA (indeed, some aspects, such as higher code density, are potentially transformed into advantages, because they improve the cache hit percentage). x86 CPUs first experienced widespread use for the nascent Personal Computer market, dominated by IBM and compatible devices, but starting in the second half of the nineties, the x86 CPUs have progressively assaulted and they gradually replaced other architectures in the upper sectors of the market, leading in fact to the end of the development of some competing processor families, mainly of type RISC (Reduced Instruction Set Computing), or relegating them to increasingly niche markets.

For example, during the 80s, Motorola 68k CISC (Complex Instruction Set Computing) CPUs were excellent products and they were widely used in multiple Amiga and Macintosh computers, but they could not keep up the pace of development of Intel CPUs, so in 1991 Motorola, Apple (its best client) and IBM created an alliance to develop PowerPC (Performance Optimization With Enhanced RISC – Performance Computing), which initially included 32-bit microprocessors, later evolved into 64-bit solutions. In the early 1990s, IBM wanted indeed to free itself from the dependence on the Intel-Microsoft ("Wintel") duopoly for the supply of CPUs and operating systems or, at the very least, to differentiate its product offerings; in fact, in addition to developing the PowerPC project, IBM invested in the OS/2 operating system (as an alternative to Microsoft DOS and Windows for its PC) and in the manufacture of x86 CPUs designed by Cyrix (a fab-less independent developer).

Anyway, after a dozen years of profitable use of PowerPC CPU in Apple systems and after having repeatedly supported the virtues of this architecture (moreover, in 2004 Steve Jobs, at that time CEO and guru of Apple, during a presentation event with strong media resonance, ridiculed the Intel CPUs, declaring that he would never use one of them, in a video that had a viral spread), in 2005 Apple announced it would no longer use PowerPC processors in its Macintosh computers and it decided to adopt only Intel processors, starting in 2006 with Yonah and Merom CPUs, while roughly in the same period IBM sold its Personal Computing Division to the Chinese Lenovo (that acquisition was completed on May 1, 2005). Even the PowerPC adoption as a gaming console CPU has proved ephemeral, involving only one generation of Xbox, the second one (Xbox 360; the first one and all subsequent models are x86 based, instead), only one generation of Playstation (the third, the following are all x86) and three Nintendo's generations (GameCube, Wii and Wii U), before Japanese console makers moved to ARM solutions for the Switch console. So, to the present day, the PowerPC survives only in the embedded sector and in some very high-end IBM POWER platforms.

Alpha was instead a 64-bit RISC CPU family developed by Digital Equipment Corporation (DEC); Alpha CPUs were intended to be 64-bit high-performance design from the inception and they were used in a variety of workstations and servers. The first version, the Alpha 21064 (EV4) was introduced in November 1992 at 750 nm processing node, while the Alpha 21264 (EV6) was a valuable source of inspiration for AMD's K7 project; in fact, the first AMD Athlon in many ways resembled a sort of Alpha EV6 with the addition of a front-end that allowed to decode the x86 instructions into RISC-like Macro-ops

sequences; even the interconnection bus used by K7 and derivatives was exactly the EV6 bus from DEC, obtained under license from AMD. But in 1998 Compaq, a very important Intel customer (for example, in September 1986, Compaq released Deskpro 386, the first 386 PC, beating IBM by 7 months), bought most parts of DEC, including the Alpha architecture, which was phased out in favor of the forthcoming Hewlett-Packard/Intel Merced/Itanium architecture. In 2001 Intel bought all Alpha intellectual property, while later HP purchased Compaq later that same year, so Alpha development was killed, although the sales to the existing customer base continued until 2007.

SPARC (Scalable Processor Architecture) is a RISC CPU family originally developed by Sun Microsystems (later acquired by Oracle) and Fujitsu. The first implementation of the original 32-bit architecture (SPARC V7) was released already in 1987. SPARC V9, released in 1993, introduced a 64-bit architecture and was first released in Sun's UltraSPARC processors in 1995. In 2006, Sun released UltraSPARC T1, based on chip multi-threading (CMT) and massive multi-core: the so-called "Niagara" project pursued Throughput Computing and it provided indeed a "torrent" of hardware threads, with an octo-core single die CPU, where each core had 4 hardware threads, for a total of 32 logical processors, at a time when the first dual core CPUs were beginning to spread into the PC world. The latest commercial high-end SPARC processors are Fujitsu's SPARC64 XII (introduced in 2017 for its SPARC M12 server) and Oracle's SPARC M8 introduced in September 2017 for its high-end servers. Oracle claimed that it terminated SPARC design after the completion of the M8.

In the mid-80s, Hewlett Packard started Spectrum program, a project for a 32 bit RISC architecture, that was sold as Precision Architecture RISC (PA-RISC). Albeit in 1996 PA-RISC 2.0 introduced 64 bits support and it also added fused multiply–add instructions, HP had meanwhile entered into an agreement with Intel for the Merced project. Though HP stopped selling PA-RISC-based HP 9000 systems at the end of 2008, in the last decade of his commercial life its development has been very slow and the priority for HP was the Merced platform.

In 1989, HP researchers started indeed to investigate a new architecture that can allow the processor to execute multiple instructions in each clock cycle, exploiting multiple execution units and issue ports. To simplify control logic and reduce energy consumption (by eliminating the need for complex run-time scheduling circuitry), HP chose a form of very long instruction word (VLIW) architecture, later named Explicitly Parallel Instruction Computing (EPIC). With EPIC, Instruction Level Parallelism is exploited by the compiler, that determines in advance which instructions can be executed at the same time, so the microprocessor's transistor budget is basically invested entirely in execution units, registers and caches, enabling deeper inspection of the code at compile time to identify additional opportunities for parallel execution with the respect to any form of hardware scheduling and out of order window. In 1994 Intel joined HP to develop this EPIC paradigm under the Merced project, which conducted to IA-64 architecture. HP and Intel initiated a large joint development effort with a goal of delivering the first product, Merced, in 1998, with the ambitious goal of quickly conquering important positions in the high-end server and workstation sectors, as well as High Performance Computing, and of reaching high-end PCs within 5 years, and then became a replacement for the original x86 architecture. Indeed, at some point, Intel boldly decided to rename Merced project into P7, to indicate that the future of the x86 microprocessors would consist of a conversion to the new EPIC architecture. HP and Intel were so convincing that Compaq and Silicon Graphics decided to abandon further development of the Alpha and MIPS architectures respectively in favor of migrating to IA-64, but when Itanium was released in June 2001 it was far below expectations: only a few thousand systems using the original Merced Itanium processor were sold, due to relatively poor performance, high cost

and limited software availability. One year later, the Itanium 2 was released and it was marketed for enterprise servers rather than for the whole gamut of high-end computing. Despite huge and protracted economic efforts, Itanium has never been a high-volume product for Intel; the last attempt was to develop a single interconnection bus for both x86 and Itanium (CSI, i.e. Common System Interface, later named QPI, Quick Path Interconnect) multi-socket systems, in the hope of reducing the development costs of the motherboards for Itanium, but in fact this product line has substantially extinct after 2010.

1.1.2 Evaluating CPU performance

In the case of a single central processing unit, i.e. a system with only a single core microprocessor, in a first approximation the performances can be considered corresponding to the product of the operating frequency, or how many execution cycles can be carried out in a second, and the average amount of actual work done in a clock cycle, represented by the IPC (Instructions Per Second):

$$Performance = IPC \times Frequency$$

For many years, operating frequency of a CPU was a fixed value, then at a certain point more and more sophisticated arrangements were developed to vary the frequency according to the expected workload. The first version of Intel SpeedStep Technology acted only on products intended for laptops and it allowed the switching between only two modes: low consumption for battery operation, high frequency if connected to the power supply. Later, more advanced technologies, such as AMD PowerNow!, AMD Cool'n'Quiet, Intel Enhanced SpeedStep adopted increasingly sophisticated dynamic frequency scaling strategies to reduce consumption even on desktop systems and in general to modulate operating frequencies and power voltages dynamically depending on the workload. In the last decade, newer solutions have also been adopted to increase operating frequencies beyond the base value in a controlled, reliable and warranted way.

Anyway, for a long time, the operating frequency, which was precisely a fixed and carefully determined CPU parameter, obtained the greatest emphasis on performance evaluation. This choice also depends on the fact that the IPC does not assume a constant value over time and its average value also cannot in general be accurately estimated, as it depends on a plethora of factors. For example, in the case of a modern superscalar CPU, although in theory it is possible to achieve an IPC of 4 or more instructions per cycle in optimal conditions (such as small memory footprint for high percentage of cache and TLB hits, regular memory access patterns to "cooperate" well with hardware data prefetcher, a small jump density to not put too much effort on the branch prediction unit and to pay small penalities for mispredicted branches, and so on), it is not uncommon to find use cases in which the average IPC does not even reach 0.5.

Once a specific hardware configuration has been fixed, the IPC varies according to the exact instructions used, therefore it is possible to evaluate different average IPC values not just by tackling totally different software but even with the aim of solving a specific problem, for example by changing the solving algorithm, using a different compiler, changing the compiler's options or running the software on a different operating system.

Given how extremely complicated it is to evaluate "absolute" mean IPC for a CPU or platform, it can be emphasized that it is very difficult to even evaluate a sort of "relative" IPC: in general, between two CPUs based on different ISAs, different microarchitectures or even simply different platforms (motherboard, memory subsystem, graphic section, mass storage compartment, etc.), any attempt to compare IPCs should be evaluated with extreme caution. Even with the same ISA and using identical, or very similar, platforms,

the ratio between IPCs of two CPUs with different microarchitecture is generally not constant, furthermore often the order relationship is not invariant either, in the sense that a microarchitecture can offer a higher IPC with certain software and the other, instead, is the best with different workloads.

Moreover, if a specific benchmark has been selected and a very precise hardware and software platform has been set, the following considerations can be found by modifying only the microprocessor:

- if a CPU X is replaced by a CPU Y of the same microarchitecture, with the only difference of an increase in operating frequency in favor of the latter, the average IPC tends to decrease, because of factors such as higher memory access latency (measured in clock cycles) and lower average bandwidth for clock cycle;

- if a CPU X is replaced by a CPU Y, of the same microarchitecture or an extremely similar one (for example, a "die shrink"), in which the only or main difference is the cache memory hierarchy, if the newer product has a larger cache but with a higher (so worse) latency, there may be a slight decrease in the average IPC in the processing of small datasets and an increase with large datasets;

- with identical chips (and same active features), the adoption of a faster connection with the rest of the system can lead to a significant increase in IPC, at least for certain workloads (for example, there were three versions of the Pentium 4 Northwood at 2.8 GHz, denoted by the suffixes A, B and C, respectively with Fronst Side Bus - FSB - at 400, 533 and 800 MHz, each one faster than the previous one).

In order to assess the relative performance of computer systems or CPUs, because is very difficult to predict their performance simply by looking at their specifications (especially since they can be the result of divergent choices and different compromises), special programs were developed that allowed comparison of different architectures: the benchmarks.

Maybe one of the first benchmark was the Whetstone, a synthetic benchmark for evaluating the performance of scientific computers that was first written in Algol 60 in 1972 and primarily measures the floating-point arithmetic performance [Curnow and Wichmann, 1976]. Usually it reports performance profile using the Millions of Whetstone Instructions Per Second (MWIPS) metric. In 1984, choosing the name as a pun on Whetstone, Dhrystone was developed by Reinhold P. Weicker as a synthetic computing benchmark, originally written in Ada, to evaluate system integer performance (eventually with over-representation of string operations). It grew to become representative of general processor performance, expressed as the number of Dhrystones per second (the number of iterations of the main code loop per second).

Standard Performance Evaluation Corporation (SPEC) is a non-profit corporation that aims to "produce, establish, maintain and endorse a standardized set" of performance benchmarks for computers that are reasonably scientific, unbiased, meaningful and relevant. It was founded in 1988 and it released its first set of benchmarks, the SPEC Benchmark Suite for UNIX Systems version 1.0, which consisted of 10 computation-intensive benchmark programs, the same year. This suite consisted of 10 programs which could be run and measured to produce three scores: integer SPECmark (later renamed to SPECint and finally to SPECint89), floating-point SPECmark (later renamed as SPECfp and afterwards SPECfp89), and overall SPECmark. In January 1992 SPEC released updated benchmark suites, SPECint92 and SPECfp92 (dropping a combined index), each with a greater number of programs than its predecessor, and each using larger amounts of code and larger datasets.

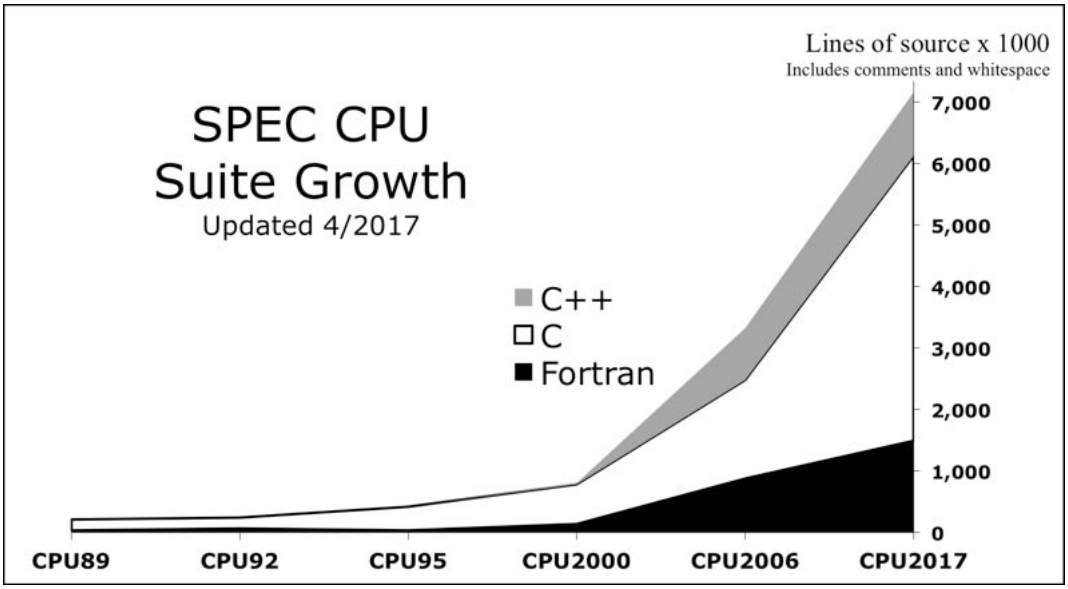


Figure 1.1: SPEC CPU Suite Growth across all of its iterations.

SPEC distributes source code files to users wanting to test their systems. These files are written in a standard programming language, which is then compiled for each particular CPU architecture and operating system. Thus, the performance measured is that of the CPU, RAM, and compiler, and does not test I/O, networking, or graphics. Two metrics are reported for a particular benchmark, "base" and "peak": allowed compiler options account for the difference between the two numbers. In 1992, SPEC92 (subsequently known as SPEC CPU92) debuts with 20 benchmark programs and up to 10.2 billion dynamic instruction count. With the release of this suite, the Baseline rule was introduced; in which vendors are no longer allowed to optimize the compilation of the code without reporting it.

It was followed by CPU95, CPU2000, and CPU2006. The latest standard is SPEC CPU_2017 and consists of SPECspeed and SPECrate: the first data is used for comparing time for a computer to complete single tasks, while the second one measure the throughput or work per unit of time. 1995 SPEC CPU95 goes live, with a SPARCstation 10 model 40 as its reference machine and

2017 SPEC releases SPEC CPU2017, an all-new version of its flagship performance evaluation suite, with 43 individual benchmarks organized into four categories. The SPECspeed 2017 Integer and SPECspeed 2017 Floating Point suites are . The SPECrate 2017 Integer and SPECrate 2017 Floating Point suites

Intel Comparative Microprocessor Performance (iCOMP) was an index published by Intel and it was used to measure the relative performance of its microprocessors, as in Fig. 1.2; it allowed the customers to compare the performance of Intel's various processor among various families and models using a simple, relative measure of microprocessor performance. Introduced in 1992, it compared CPU from 386 SX at 16 MHz processors to the P54C Pentium, choosing a 486 SX at 25 MHz as an arbitrary reference and giving it the conventional score 100. iComp was based on a mix of 16 and 32 bit benchmarks, with emphasis on 16-bit tests, in accordance with the fact that most of the software actually in use was just 16-bit. The overall score was indeed determined on 70% of 16-bit tests and 30% of 32-bit tests: in more detail, 68% of the total score was assigned by virtue of PC Bench 7.01, released by Ziff Davis Labs to evaluate traditional business applications (at 16-bit), 2% for 16-bit Whetsone, 25% for SPECint92 and 5% for SPECfp92 (the last

two were the 32-bit tests). iComp index was useful to show that a processor of a more advanced architecture could outperform a rival (a 486 compared to a 386, a Pentium compared to a 486) even operating at lower frequencies: for example, 486 SX at only 20 MHz could outperform 386 DX at 33 MHz (and almost reach it at just 16 MHz), Pentium at 60 MHz could exceed 486 DX4 at 100 MHz. Converserly, it could show significant differences at the same frequency: at 25 MHz, 386 SX scored 39 points, 386 DX 49, 486 SX 100 and 486 DX 122 points; at 100 MHz, 486 DX4 scored 435 points, while Pentium scored 815.

iCOMP was updated twice; version 2.0 was introduced in 1996, it used only 32-bit benchmarks and it chose the Pentium 120 as baseline, because the Pentium 120 scored 1000 in iCOMP 1.0; it computed the weighted geometric mean of a processor's relative performance on each of the component benchmarks compared to the chosen base processor and multiplied by 100. The weights were 40% for CPUmark32 (to represent traditional business applications, such as Microsoft Office and Lotus Smart Suite), 15% Norton SI-32 (to evaluate high-end applications, such as Adobe Photoshop and Autodesk's Autocad), 20% for SPECint_base95 (general purpose integer computation), 5% SPECfp_base95 (general purpose floating-point calculations), and 20% for Intel Media Benchmark. iComp 2.0 upgraded support from SPEC92 to SPEC95, which included 18 benchmark programs with up to 520.4 billion dynamic instruction count. Moreover, it presented higher focus on FPU performance, because competitors (AMD, Cyrix) FPU performance were inferior to Intel's, and on new MMX instructions, as it allowed the new Pentium MMX and Pentium II to stand out against its predecessors Pentium and Pentium Pro, as well as competing products, which supported MMX instructions with severely reduced speed. According to Intel, at that time there was no industry standard multimedia benchmark which measures video, audio, imaging, and 3D performance, so Intel developed the Media Benchmar specifically to illustrate the advantages of the new MMX instructions, so much so that it was the only program in the iComp 2.0 suite to use these instructions; the overall score of the Media Benchmark was calculated at 40% for Video (video playback with MPEG1 decompression), 30% for 3D Geometry (a mix of Direct3D and OpenGL 3D geometry computations), 25% Audio (decompression and playback of MPEG1 stereo audio clip, but event sample rate conversion, special effects and stereo mixing), 5% Image (applications of digital filters on 24-bit "true coloro" bitmap images). In Tab. 1.1, a comparison among three 200 MHz Intel CPUs is shown: in Media Benchmark, the most significant increase in performance offered by MMX instructions was found in the Image section (+ 370%, P55C vs P54C), to which however only 5% of the sub-score was assigned; in the Audio section there was the second improvement (+ 118%„ P55C vs P54C), while in Video section there was a + 74% improvement; 3D Geometry requires floating-point calculations and it is unaffected by MMX instruction set, while P6 beefier FPU conquered +36.4%. Overall, the P55C outperformed the P54C by 28.2% (identical advantage in SPECint_base95), due to the doubling of size and set associativity of the L1 cache, a new pipeline balance, enhanced branch prediction and deeper write buffers.

The last version of iCOMP was released in 1999; iComp 3.0 abandoned the tradition of using conventional SPEC tests (SPECint and SPECfp), but it used six benchmarks: Wintune 98 Advanced CPU Integer test (20%), CPUmark 99 (20%), than 3D WinBench 99-3D Lighting and Transformation test (20%), MultimediaMark 99 (25%), Jmark 2.0 Processor Test (10%), and WinBench 99-FPU WinMark (5%). These benchmarks were selected to help give a better rating of the Intel Pentium 3 processors that included additional "multimedia" instructions, i.e. ISSE: indeed, at the same 450 MHz frequency, Pentium III Katmai overtook Pentium II Deschutes with 1500 points versus 1240 (+21%), with

Benchmark	P54C	P55C	P6
iComp 2.0	142	182	220
CPUmark32	382	423	553
Norton SI-32	43.8	56.7	90.0
SPECint_base95	5.00	6.41	8.20
SPECfp_base95	2.98	3.90	5.54
Intel Media Benchmark	153.06	253.08	196.29
Video	153.42	267.23	160.97
3D Geometry	155.69	160.19	212.41
Audio	148.50	323.81	239.27
Image Processing	157.77	742.65	222.04

Table 1.1: Comparison of three 350nm Intel CPU at 200MHz: P54C (Pentium), P55C (Pentium MMX), P6 (Pentium Pro).

the same cache memory hierarchy, FSB and any other parameter. Officially, the selection of tests had to reflect the increasing use of 3D, multimedia, and Internet technology and software, so the biggest weight was assigned to the MultimediaMark 99, which measured performance of audio, video, imaging, educational, creativity and numerous Internet applications. Anyway, the aggregate value of Wintune 98 Advanced CPU Integer test and CPUmark 99 was 40%: they were used as indicative of productivity applications (32-bit integer performance). An important addition to this iteration of icomp index was Jmark, a benchmark that measured performance of Java code, which was increasingly becoming a widely accepted technology in the Internet; Java applications are typically compiled to an intermediate representation called Java bytecode, instead of directly to architecture-specific machine code, so this bytecode can run on any Java virtual machine (JVM) regardless of the underlying computer architecture and this aspect is particularly useful for client-server web applications. The Java programming language was indeed spreading rapidly, because it is a general-purpose object-oriented programming language that is intended from its conception to let application developers write once, run anywhere (WORA), because it is architecture-neutral and portable since Java Runtime Environment (JRE) are provided for free and constantly updated on popular platforms. While the use of bytecode makes application porting simple, the overhead of interpreting bytecode into machine language instructions slow down execution speed with to respect to an optimized native executable. So, since the early stage of Java development Just-in-time (JIT) compilers were developed to compile byte-codes to machine code during runtime; JMark showed even the efficiency of the byte-code interpreter of Java enviroment.

Pentium II Deschutes at 350MHz was chosen as reference point (its index was 1000) because it was the entry level (i.e. slower) among FSB100 CPUs supported by Seattle chipset, alias 440BX; iComp 3.0 was computed by calculating the weighted geometric mean of a processor's relative performance on each of the component benchmarks compared to the base CPU. As soon as Intel introduced the Pentium 4, Intel switched to use only frequency for performance classification.

Usually, CPU manufacturers who could not reach the higher frequencies, tried to make up for this lack by introducing some alternative metrics, although aware that it could not receive an ubiquitous diffusion among potential buyers and users.

The PR (Performance Rating, P-Rating or Pentium-rating) system was a figure of merit developed by AMD, Cyrix, IBM Microelectronics and SGS-Thomson as a method of comparing their x86 processors to those of rival Intel. AMD was lagging behind in the development of its fifth generation processor, the K5, so it introduced an enhanced 486 at high frequency a stop-gap product; this 486 processor at 133 MHz was sold as

5x86-PR75, because it was as fast as a Pentium running at 75 MHz, according to Ziff-Davis Winstone 96 for Windows 95 benchmark, which run 13 applications (mostly 16-bit) in four categories: Business Graphics/DTP (Adobe Pagemaker 5.0a, CorelDRAW! 5.02E2, PowerPoint 4.0c), Database (Borland dBASE 5.0, Borland Paradox 5.0, Access 2.0c, Works 3.0b), Spreadsheet (Lotus 1-2-3 Release 5, Excel 5.0c, Works 3.0b, Novell Quattro Pro 6.01), Word Processing (Lotus Ami Pro 3.1, Word 6.0c, Works 3.0b, Novell WordPerfect 6.1). Similarly, Cyrix, which was lagging behind in the development of its M1 project (commercially called 6x86), developed a sort of simplified version with 486 bus, sold under the name 5x86. Cyrix 5x86 at 120 MHz scored 47.1 points in Winstone 96 versus 45.2 points of Pentium 90, so was solded as 5x86-PR90.

While in these first uses in 1995 the PR index served to compare processors that needed higher operating frequencies to the Pentium, starting from 1996, the AMD K5 and Cyrix 6x86 consistently outperformed higher-frequency Pentium processors on Winstone 96, so both manufactures PR-rated the chips one or two Pentium speed grades higher than clock speed. In 1997, after Pentium II introduction, a PR2 rating system was introduced, but used only by Cyrix for its M2 processor (6x86MX, later renamed MII). PR rating has been heavily criticized for using only business application, leaving out completely high-end applications, games, audio and video processing and so on.

AMD revived it in 2001 with the introduction of its Athlon XP line of processors. In reaction to the consumers' misconception, AMD reinstated the PR to compare their Athlon XP microprocessors. AMD made sure to advertise the PR number of its microprocessors rather than their raw clock speeds believing that customers would compare the PR of AMD's processors to the clock speed of Intel's processors. The PR number was originally believed to show the clock speed (in megahertz) of an equivalent Pentium 4 processor, but this was never confirmed by AMD. As part of its marketing, AMD even made sure that motherboard manufacturers conspicuously showed the PR number of the microprocessor in the motherboards' POST and not include the processors' clock speeds anywhere except within the BIOS Between 2001 and 2003, Intel and AMD made few changes to the designs of their processors. Most performance increases were created by raising the processor's clock speed rather than improving the microprocessor's core. Around mid-2004, Intel encountered serious problems in increasing their Pentium 4's clock speed beyond 3.4 GHz because of the enormous amount of heat generated by the already hot Prescott core processor when working at higher clock speeds. Quickly switching transistors lose more leakage power: power leaking became a serious problem at 90 nm. A CPU that needs to run at very high frequency is limited by wire delays. The result is many repeaters and extra pipelines stages just to get the signal across the die. More pipeline stages and more repeaters mean more logic, more power. SOI – Silicon on Insulator - has made process technology even more complex but it improved the insulation of the gate and thus reduces leakage currents; anyway, the most spectacular reduction of leakage came from "high-k" materials, which replaced the previously used silicon dioxide gate dielectric. In response, Intel started exploring ways to improve the performance of its microprocessors in ways other than raising the clock speeds of the processors such as increasing the sizes of the processors' caches, using a P6 microarchitecture descendant in Pentium M CPUs and beyond, and using multiple processing cores in its processors. Because of the philosophy change, Intel now faces the challenge of making consumers compare its processors based on the PR system rather than raw clock speed, ironically a problem which Intel created itself. Some analysts regard the PR scheme (and a raw MHz / GHz rating) as nothing more than a marketing tactic, rather than as a useful measure of CPU performance.

$$Power = kCV^2Af$$

In other words, dissipated power increases quadratically with the CPU's core voltage

2.0

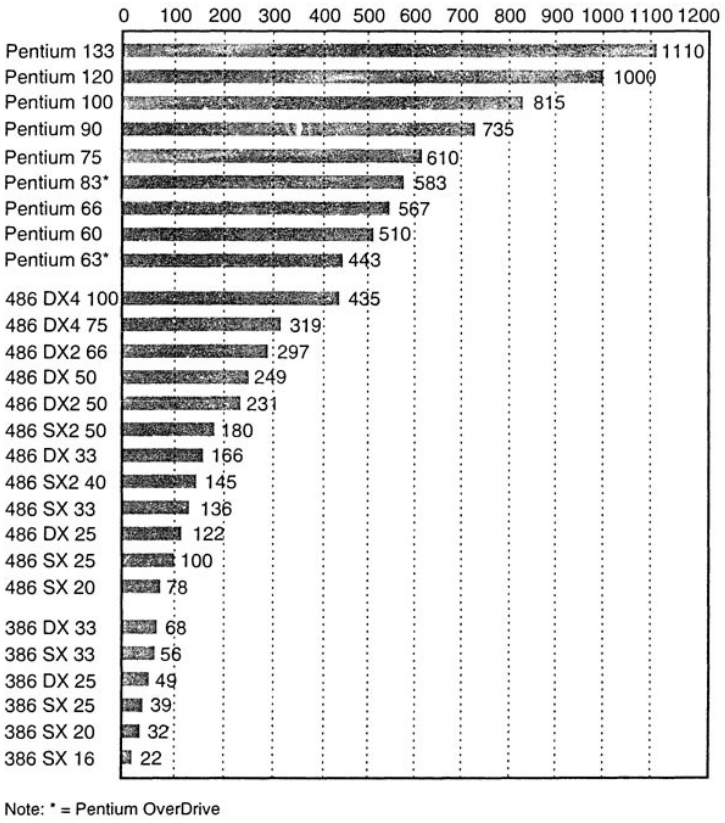


Figure 1.2: iComp 1.0 performance for Intel CPU from 386SX at 16 MHz up to Pentium P54C at 133 MHz.

2.0

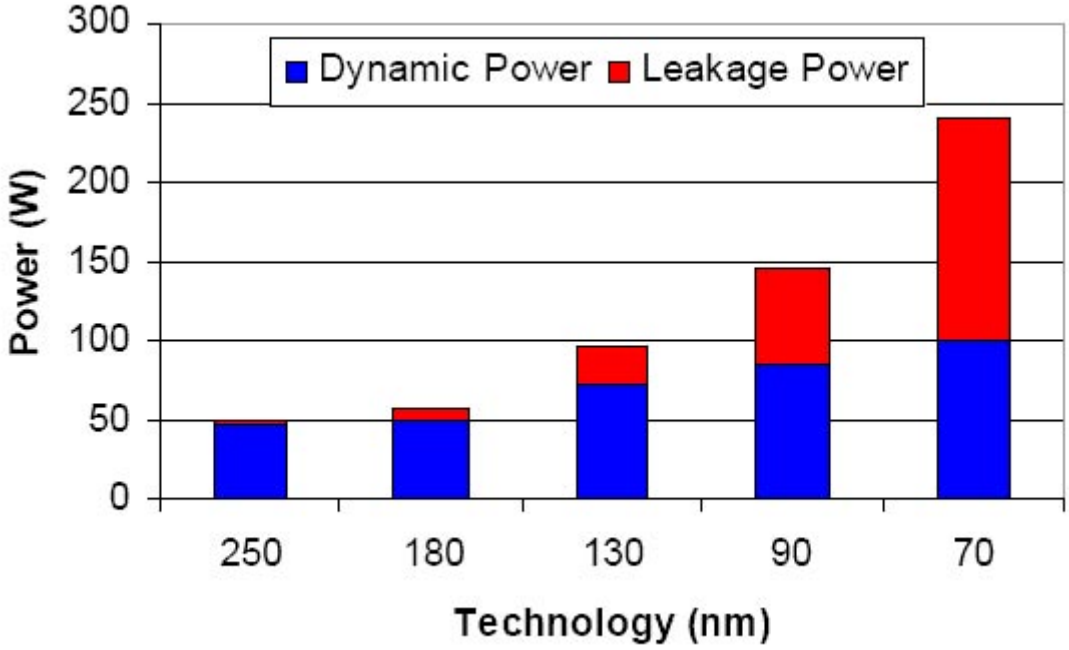


Figure 1.3: Leakage Power Estimated by Intel in 2004 from 250 to 65 nm (70 nm reported).

(V), but it is linear with the effective capacitance (C), activity(A) and frequency(f). Effective capacitance depends essentially on lithographic process, transistor count and circuit layout, while k denotes a proportionality coefficient. Activity is the factor that is influenced by the software used; the more intensive the software, the higher the amount of the time that the transistors are active; a higher IPC can contribute to increase the activity, but a high activity is not necessarily associated with a high IPC: in this regard, the replay mechanism for NetBurst microprocessors proves almost the exact opposite.

Of course, as CPUs extract more ILP and have deeper pipelines, they become more complex and use more transistors. Clock gating logic only activate the clocks selectively in a Functional Unit Block when it needs to work, so it's a power-saving technique implemented extensively.

1.1.3 Megahertz Myth (around 1980-2004)

Moreover, for many decades, the technological development and the progressive miniaturization of transistors has allowed to raise the microprocessors' operating frequencies with relative simplicity, from under a MHz to a few GHz, so the fundamental problem of raw computer power was slowly being overcome with such a rapid growth.

That extremely rapid increase in operating frequencies involved the spread of the so-called "Megahertz myth", the misconception of only using clock rate to evaluate microprocessor and, by extension, the whole digital computer's performance.

This myth arose because of the simplicity inherent in the adoption of a naive metric based on a single parameter, based on the idea that higher frequencies always corresponded to a higher performance, so clock rate was promoted in advertising and by enthusiasts without taking into account other factors.

While frequencies are a valid way of comparing the performance of different microprocessor, all the rest being equal, including details of the computer's configuration, other

2.0

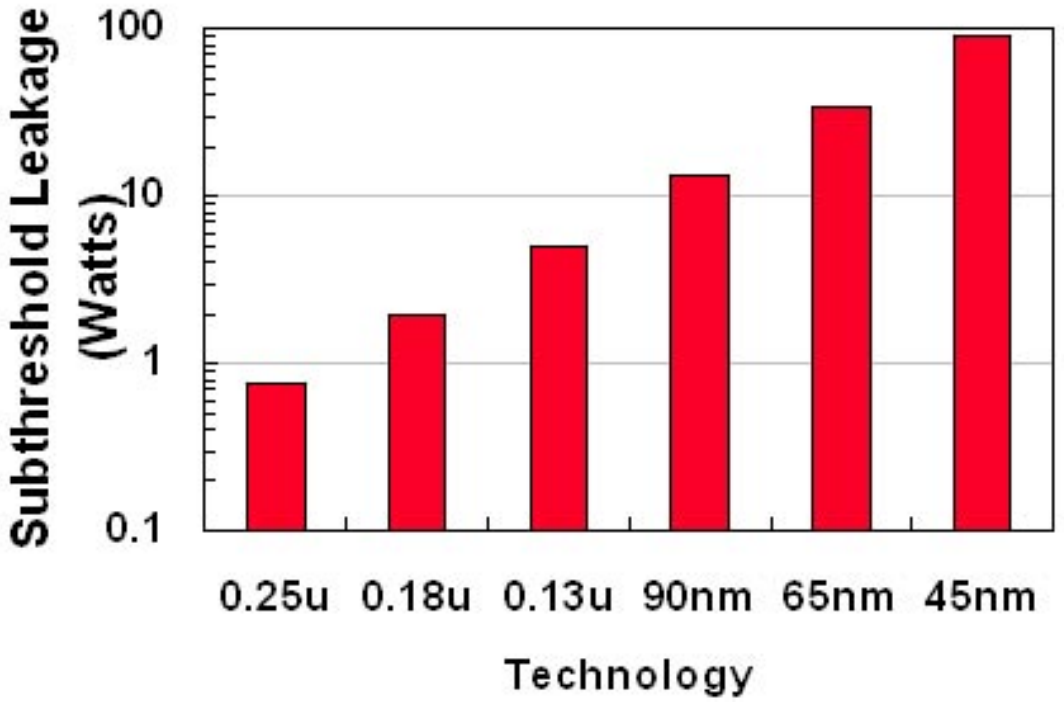


Figure 1.4: Subthreshold Leakage Power Estimated by Intel in 2005 from 250 to 45 nm.

factors such as instruction set, amount of execution units, pipeline depth, cache hierarchy, branch prediction and memory subsystem can greatly affect the performance when considering microprocessors of different "families" and or generations.

Because for some decades clock rates speed up seemed correlated, more or less, to transistor doubling (really with migration to new process node), frequently there has been overlap or confusion between Moore's law and MHz "race", a sort of corollary of MHz myth, in the sense that competition between different microprocessor manufacturers placed the emphasis on this specific feature.

At the start of 2004, Intel decided to move towards a new "MHz less" model name, on the one hand due to the difficulties in increasing frequency encountered by Prescott, on the other to emphasize the virtues of Dothan (second generation Pentium M).

For example, at the beginning of the year 2000 there was a bitter battle between AMD, with the Athlon processor family, and Intel, with the Pentium III, both at 180 nm process node, to reach the psychological barrier of 1 GHz. In the end, AMD announced its Athlon 1GHz on the March, 6th, 2000 and Intel ran for cover two days later. At that specific juncture, however, while AMD was able to supply the market with CPUs at 800, 850, 900 and 950 MHz, in addition to the flagship 1 GHz model, Intel was able to distribute a limited number of CPUs at 1 GHz, while ordinary supplies were limited to 800 MHz.

It is important to note that although two antithetical approaches, braniacs (focused on high-IPC - instruction per cycle) and speed demon (striving for high clock speeds), have long been faced in the development of digital computer processors, at one point the MHz Myth was so widespread as to induce Intel to develop a microarchitecture, called Netburst, which he advanced the achievement of higher frequencies at any cost. Starting from an initial design, progressively better IC processes usually have greatly improved both the IPC and the cycle time of microprocessors, leading some vendors to claim to deliver the best of both worlds, braniacs and speed demon. For example, Intel Pentium

release date	process (nm)	Product	Freq. (MHz)
22/03/1993	800	Pentium P5	67
07/03/1994	600	Pentium P54C	100
27/03/1995	350	Pentium P54CQS	120
01/06/1995	350	Pentium P54C	133
08/01/1997	350	Pentium P55C	200
02/06/1997	350	Pentium P55C	233
01/11/1995	350	Pentium P6	200
07/05/1997	350	Pentium II Klamath	300
26/01/1998	250	Pentium II Deschutes	333
15/04/1998	250	Pentium II Deschutes	400
24/08/1998	250	Pentium II Deschutes	450
26/02/1999	250	Pentium III Katmai	500
17/05/1999	250	Pentium III Katmai	550
02/08/1999	250	Pentium III Katmai	600
25/10/1999	180	Pentium III Copermine	733
20/12/1999	180	Pentium III Copermine	800
08/03/2000	180	Pentium III Copermine	1000
20/11/2000	180	Pentium 4 Willamette	1500
23/04/2001	180	Pentium 4 Willamette	1700
02/07/2001	180	Pentium 4 Willamette	1800
27/08/2001	180	Pentium 4 Willamette	2000
07/01/2002	130	Pentium 4 Northwood	2200
02/04/2002	130	Pentium 4 Northwood	2400
25/08/2002	130	Pentium 4 Northwood	2800
12/11/2002	130	Pentium 4 Northwood	3067
14/04/2003	130	Pentium 4 Northwood	3000
23/06/2003	130	Pentium 4 Northwood	3200
02/02/2004	130	Pentium 4 Northwood	3400
21/06/2004	90	Pentium 4 Prescott	3600
12/11/2004	90	Pentium 4 Prescott	3800

Table 1.2: Intel CPU release dates during the so called MHz Race. Intel was very aggressive in releasing processors operating at higher frequencies as quickly as possible, regardless of other factors, such as increased energy consumption or decreased average IPC

reached both a significantly higher IPC than i80486 both up to double operative frequency. Anyway, the "true" speed demon philosophy dictate that a processor's cycle time should be the minimum required to cycle an ALU and pass the result to the next instruction, at a fixed process node, rather than exploit only IC process gains. In the case of the Pentium 4 Willamette, based precisely on the Netburst microarchitecture, Intel decided not only to substantially double the length of the typical execution pipeline, but also adopted special fast-ALUs capable of operating at a frequency twice that of the rest of the microprocessor, breaking down the execution of the most common (and simple, such as several types of addition and subtraction) 32-bit instructions into a pair of just 16 bits (a further half cycle is necessary to update the status register, for example to take into account any carry or borrow).

In Tab. 1.2 e in Fig.1.5, the concrete implementation of the MHz Myth is shown in

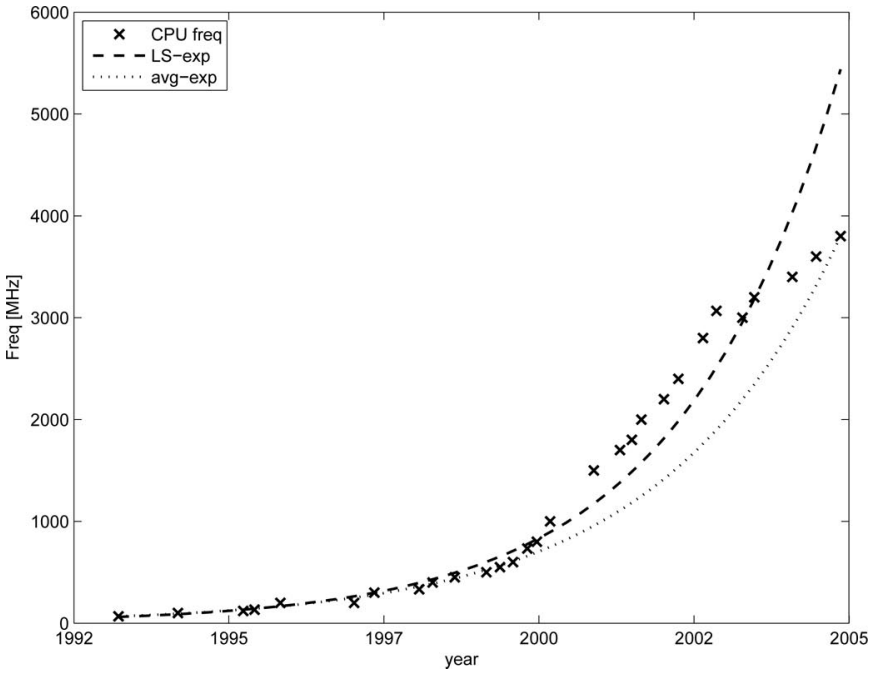


Figure 1.5: MHz Race in action: fastest clock speed (in MHz) for Intel CPU for more than a decade, from 1993 to 2003 (and the subsequent slowdown). The cross data points correspond to the values in Tab. 1.2. The dashed line shows the exponential trend obtained through the least squares method. The dotted line, on the other hand, represents the "average" exponential.

an exemplary way: there is an exponential growth in operating frequency over a period of a dozen years. The dashed line shows the exponential trend obtained through the least squares method: for 2005 it contemplates exceeding the 5 GHz threshold, in line with the expectations that Intel and the market had for the Prescott and Tejas CPUs. It is characterized by a time constant $\tau = 656.66 days = 1.8 years$. The dotted line, on the other hand, represents the "average" exponential: it is simply the exponential curve that passes through the first and last of the data points taken into consideration, that is, showing the expected rhythm for a constant cadence of the frequency increases. It is characterized by a time constant $\tau = 730.04 days = 2.0 years$. Therefore, there is a doubling of operating frequency with a substantially constant rate, about every two years, over a time span of a dozen years. In Fig.1.5, it can be found that both exponential curves underestimate the rapidity of growth in operating frequency between the end of 2000 and the beginning of 2003: this is due to the introduction of the Netburst microarchitecture, with its hyperpipeline. By adhering to the "speed daemon" design paradigm, the aim was to increase frequency at any cost, even at the expense of IPC and energy consumption. "While architectural enhancements are important, Intel intends to continue its lead in raw speed [i.e. frequency]. Otellini demonstrated a new high-frequency mark for processors, running a Pentium 4 processor at 4.7 GHz" (Intel press release after IDF Fall 2002[2]) (Otellini was executive vice president and general manager of the Intel Architecture Group and in 2002 he was elected to the board of directors and became president and Chief Operating Officer at the company). Later, Prescott's data point are instead under the least squares exponential curve, because the introduction of a new lithographic process (from 130 to 90 nm) and the 55% increase in the hyperpipeline (from 20 to 31 stages) did not produce the desired improvements in terms of frequency. Although Intel had placed

[2] `https://www.anandtech.com/show/1611`

much emphasis in 2001 in highlighting that it took less than 18 months, after the 1 GHz milestone, to reach the 2 GHz mark, in spite of the fact during the IDF 2002 it made very bold statements, it became overly complicated to reach operating frequencies. "What you have seen is a public demonstration of 4 GHz silicon straight off our manufacturing line. We have positive indications to be able to take Netburst to the 10 GHz space" (IDF Spring 2002[3]). Instead, the ambitious Tejas project was officially canceled on May, 7th 2004, the planned 4GHz Prescott was erased also and it took about 10 years for an Intel CPU to reach the 4 GHz milestone. To pursue Moore's law and to increase overall computing power, Intel started the multi-core era: by cramming two complete computational cores into a single package, then an increasing number of cores, to improve performance in multi-user or multi-tasking environments or with multi-thread applications.

In Tab. 1.3 e in Fig.1.6, the frequency slowdown during recent multicore era is showed. Despite the awareness of a certain amount of arbitrariness, it was considered reasonable to select, at each major release of new CPUs destined to the PC and HEDT (High-End Desktop) market, the CPU with the maximum operating frequency (observing the maximum frequency in turbo mode for a single active core, possibly for the best core in the case of CPUs that support Turbo 3.0 mode). The first dual core CPUs were based on the Netburst microarchitecture and could operate at frequencies close to 4 GHz, but with the transition to the Merom / Conroe microarchitecture the operating frequencies were drastically reduced, so in Fig.1.6 I prepared two least squares regression lines: the dotted line was computed using all data points in Tab. 1.3, while dashed line was calculated excluding Netburst processors (Pentium 4, Pentium D and Extreme Edition). It seems evident from the graph that the growth of the maximum frequency in the last fifteen years follows definitely a linear trend, with a relatively modest slope. In fact, evaluating the overall CPU portfolio, there is an increase from 3.8 GHz (Pentium 4 Prescott, 12/11/2004) to 5.0 GHz (Core i7 8086K Coffee Lake, 08/06/2018), i.e. by 31.6 % in fourteen years, in stark contrast to exponential growth with a doubling of frequency every 18-24 months. Even excluding the parenthesis of products based on the NetBurst microarchitecture, over a dozen years there has been a 70.5% increase in frequency since the the Core 2 Duo Conroe X6800 release at 2933 MHz.

1.1.4 Multi core era (around 2005-nowadays)

Hyperthreading [Akkary and Driscoll, 1998] Turbo Mode Intel Dynamic Acceleration (IDA) More cores only helps users workloads that scale across multiple cores, or gives an opportunity for more work at once. There also has to be an interconnect to feed those cores, which scales out the power requirements. Cores doesn't always help everyone, but it can be one of the easier ways to scale out certain types of performance.

1.1.5 Slowdown of Moore's Law

The doubling of components in integrated circuits has already started to falter, thanks both to increasing difficulty in developing new lithographic processes and to the heat that is unavoidably generated when more and more silicon circuitry is jammed into the same small area.

Over the last few years, there have been some crucial events that have shown the difficulty in the progress of production processes:

[3] `https://www.anandtech.com/show/1611`

release date	(nm)	cores/thr.	Product	Freq. (MHz)
21/02/2005	90	1/2	Pentium 4 Prescott 2M EE	3733
18/04/2005	90	2/4	Pentium XE Smithfield	3200
12/06/2005	90	1/2	Pentium 4 Prescott 571	3800
14/11/2005	90	1/2	Pentium 4 Presctott 2M 672	3800
27/12/2005	65	2/4	Pentium XE Presler	3466
05/01/2006	65	1/2	Pentium 4 Cedar Mill 661	3600
22/03/2006	65	2/4	Pentium XE Presler	3733
27/07/2006	65	2/2	Core 2 Duo Conroe X6800	2933
16/07/2007	65	2/2	Core 2 Duo Conroe E6850	3000
07/01/2008	45	2/2	Core 2 Duo Wolfdale E8500	3167
10/08/2008	45	2/2	Core 2 Duo Wolfdale E8600	3333
17/11/2008	45	4/8	Core i7 EE Bloomfield 965	3467
02/06/2009	45	4/8	Core i7 EE Bloomfield 975	3600
08/09/2009	45	4/8	Core i7 Lynnfield 870	3600
07/01/2010	32	2/4	Core i5 Clarkdale 670	3733
18/04/2010	32	2/4	Core i5 Clarkdale 680	3867
30/05/2010	45	4/8	Core i7 Lynnfield 880	3733
09/01/2011	32	4/8	Core i7 Sandy Bridge 2600K	3800
13/02/2011	32	6/12	Core i7 EE Gulftown 990X	3733
23/10/2011	32	4/8	Core i7 Sandy Bridge 2700K	3900
14/11/2011	32	6/12	Core i7 Sandy Bridge-E 3960X	3900
23/04/2012	22	4/8	Core i7 Ivy Bridge 3770K	3900
12/11/2012	32	6/12	Core i7 Sandy Bridge-E 3970X	4000
02/06/2013	22	4/8	Core i7 Haswell 4770K	3900
10/09/2013	22	6/12	Core i7 Ivy Bridge-E 4960X	4000
11/05/2014	22	4/8	Core i7 Haswell Refresh 4790	4000
02/06/2014	22	4/8	Core i7 Devil's Canyon 4790K	4400
05/08/2015	14	4/8	Core i7 Skylake 6700K	4200
03/01/2017	14	4/8	Core i7 Kaby Lake 7700K	4500
26/06/2017	14	10/20	Core i9 Skylake-X 7900X	4500
25/09/2017	14	18/36	Core i9 Skylake-X 7980XE	4400
05/10/2017	14	6/12	Core i7 Coffee Lake 8700K	4700
08/06/2018	14	6/12	Core i7 Coffee Lake 8086K	5000
08/10/2018	14	18/36	Core i9 Skylake-X Refresh 9980XE	4500
08/10/2018	14	8/16	Core i9 Coffee Lake-R 9900K	5000
07/10/2019	14	18/36	Core i9 Cascade-X 10980XE	4800
28/10/2019	14	8/16	Core i9 Coffee Lake-R 9900KS	5000

Table 1.3: Intel CPU: max frequency evolution in the last 15 years, during multi-core age. For each CPU model, max operative frequency is reported (such as 1-core turbo mode, best core for Turbo 3.0)

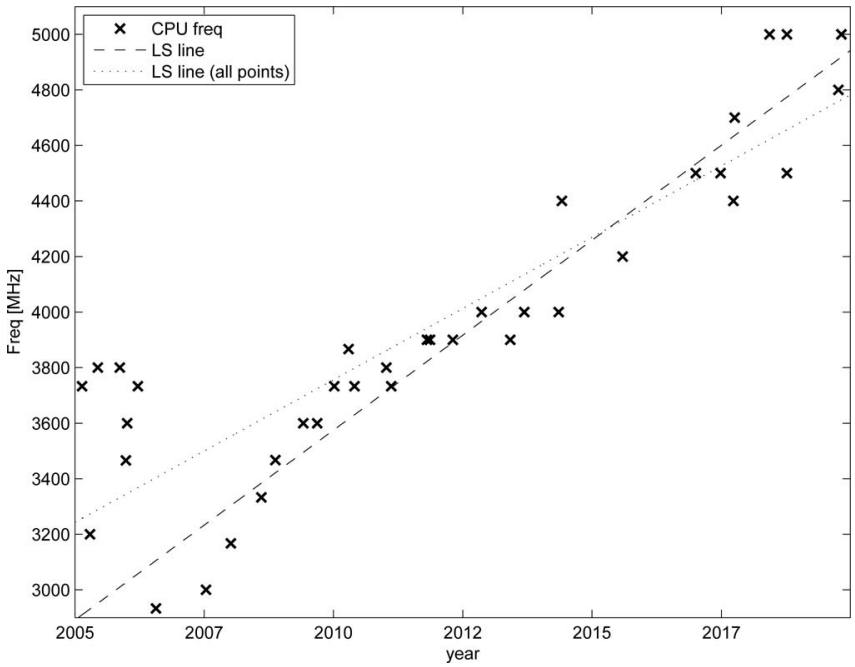


Figure 1.6: Intel CPU maximum frequency evolution in the last 15 years, during multicore age. The cross data points correspond to the values in Tab. 1.3. The dashed line shows the best fit regression line over all CPU since Conroe's introduction. The dotted line, instead, shows the best fit regression line over all data points plotted, including Pentium 4 and Pentum D Netburst processors.

- 2004 Intel Prescott delay at 90 nm, with final frequencies far below promises (up to 3.8 GHz, while the promise before the commercial release of the product was of 5 GHz and above)
- 2007 IBM's discussion of the inevitable compromises for the scaling from 90 nm to 65 nm of Cell processor (jointly developed by Sony, Toshiba, and IBM, an alliance known as "STI"): with that transition, for the first time IBM was forced to choose at most two directions of improvement, including greater density of transistors, higher operating frequency and reduction of energy consumption (from that moment, the producers of integrated circuits have often been forced to release significant variations of a certain process node, privileging different aspects such as high operating frequency or very low power usage)
- 2014 Intel difficulties with manufacturing and shipping 14 nm products (Broadwell delay, niche market release, Haswell refresh introduction at old 22 nm node)
- 2019 Intel prolonged difficulties with manufacturing and shipping 10 nm CPU (Cannonlake, formerly named Skymont, Ice Lake are low volume, niche products, only for low frequency, low core market segment)

The most significant effects of the difficulties encountered in the transition to more advanced lithographic processes were the sharp reduction in the rate of increase in operating frequency and the transition to multicore architectures. For example, in about ten years there was the transition from 66 MHz to 3800 MHz, in the next fifteen years maximum frequency rose up only to 5000 MHz (in peculiar "turbo mode").

At the end of the year 2000, NetBurst based processors somewhere between 8 – 10GHz were considered realistically a concrete objective to be reached in the following five years,

year	producer	product	lithography (μm)	transistor (k)	freq. (MHz)
1971	Intel	4004	10	2.25	0.4-0.7
1972	Intel	8008	10	3.5	0.5-0.8
1974	Intel	8080	6	6.0	2-3
1976	Intel	8085	3	6.5	3-6
1978	Intel	8086	3	29	5-10
1979	Intel	8088	3	29	5-8
1982	Intel	80286	1.5	134	6-12.5
1985	Intel	80386	1.5-1.0	275	12-33
1989	Intel	80486	1.0	1200	25-33
1992	Intel	80486DX2	0.8	1200	50-66
1993	Intel	Pentium P5	0.8	3100	60-66
1994	Intel	80486DX4	0.6	1600	75-100
1994	Intel	Pentium P54C	0.6	3300	75-120

Table 1.4: Intel and compatible CPU until 1994[4]

year	producer	product	lithography (nm)	transistor (M)	freq. (MHz)
1995	Intel	Pentium P54CS	350	3.3	133-200
1995	Intel	Pentium P6	350	5.5	150-200
1997	Intel	Pentium P55C	350	4.5	166-233
1997	Intel	Pentium II Klamath	350	7.5	233-300
1998	Intel	Pentium II Deshutes	250	7.5	333-450
1999	Intel	Pentium III Katmai	250	9.5	450-600
1999	Intel	Pentium III Coppermine	180	28	600-1000
2000	Intel	Pentium 4 Willamette	180	42	1300-2000
2001	Intel	Pentium III Tualatin	130	44	1400
2002	Intel	Pentium 4 Northwod	130	55	2000-3400
2003	Intel	Pentium M Banias	130	77	900-1300
2004	Intel	Pentium 4 Prescott	90	125	2600-3800

Table 1.5

before a more radical architecture's change was needed again. It was even speculated that such very high frequency CPU would be based on a manufacturing process forecasted to debut in 2005 at about 70 nm, for these processors who were expected to run at less than 1 volt, 0.85v being the more accurate estimate (based on what happened in the previous 3 years, when there was a decrease from 3.3 V of P54C and P6, operating at 200MHz on 350 nm node, to 1.5-1.8 V for Tualatin, Thundirbird and Willamette, operating at 1.4-2 GHz on 180 nm process). Although the story has followed a radically different path, it is curious to note that at the end of 2005 Intel had actually introduced processors with a 65 nm lithographic process and that these products could operate under 1 V, although only in low consumption mode (around 1 GHz for mobile Yonah and Merom).

year	producer	product	lithography (nm)	transistor (M)	core count	freq. (MHz)
2006	Intel	Core 2 Duo Conroe	65	291	2	1833-3000
2006	Intel	Core 2 Quad Kentsfield	65	2x291	4	2400-3000
2007	Intel	Core 2 Duo Penryn	45	410	2	2667-3466
2007	Intel	Core 2 Quad Yorkfield	45	2x410	2	2400-3200
2009	Intel	Core i7 Nehalem	45	2x410	2	2666-3466

Table 1.6

2.0

Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

Figure 1.7: DRAM specifications

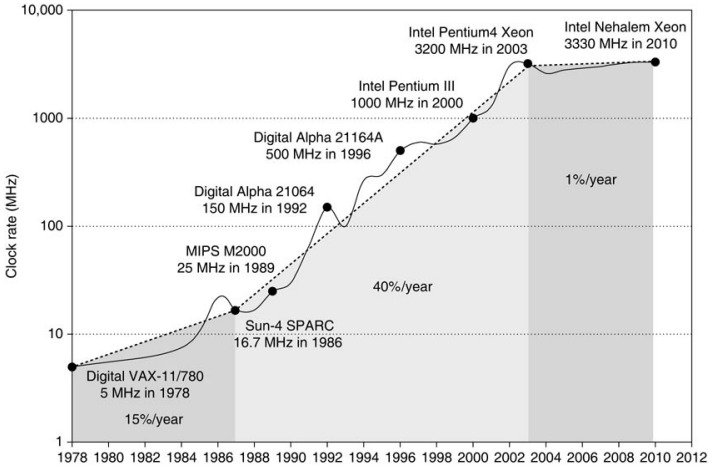


Figure 1.8: CPU clock rates

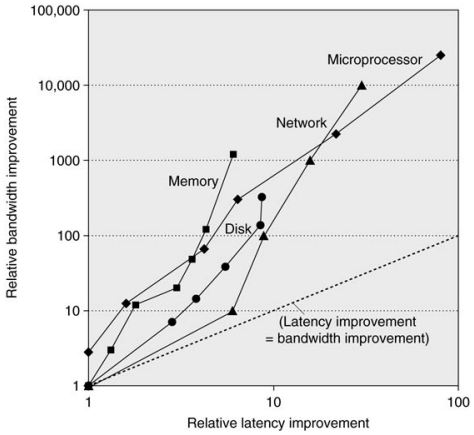


Figure 1.9: bandwidth vs latency

CA:AQA Edition	Year	DRAM growth rate	Characterization of impact on DRAM capacity
1	1990	60%/year	Quadrupling every 3 years
2	1996	60%/year	Quadrupling every 3 years
3	2003	40%–60%/year	Quadrupling every 3 to 4 years
4	2007	40%/year	Doubling every 2 years
5	2011	25%–40%/year	Doubling every 2 to 3 years

Figure 1.10: DRAM rate improvement

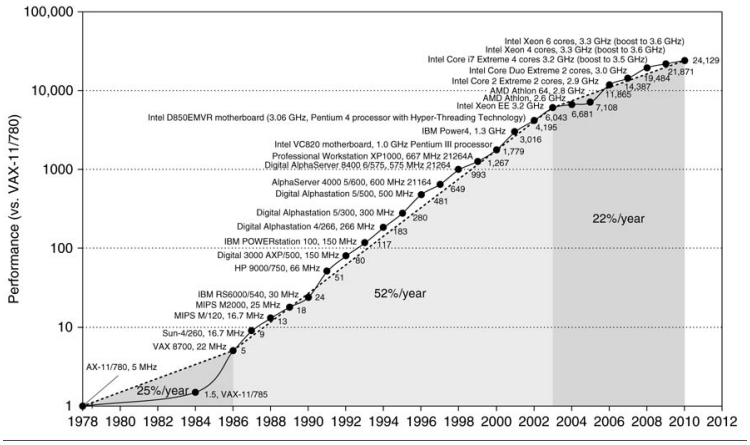


Figure 1.11: SPEC Growth CPU perf

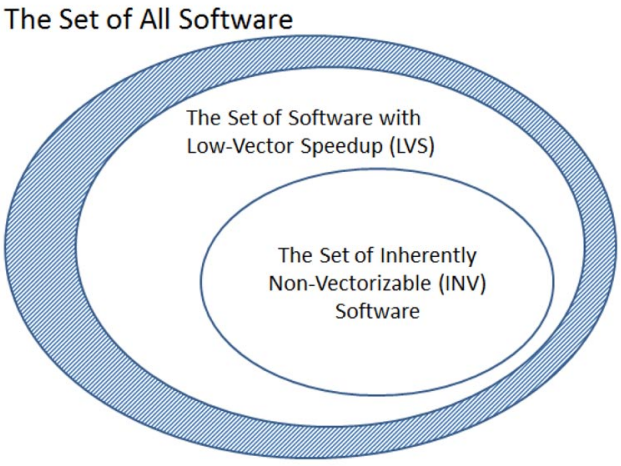


Figure 1.12: Relationships software vectorizable

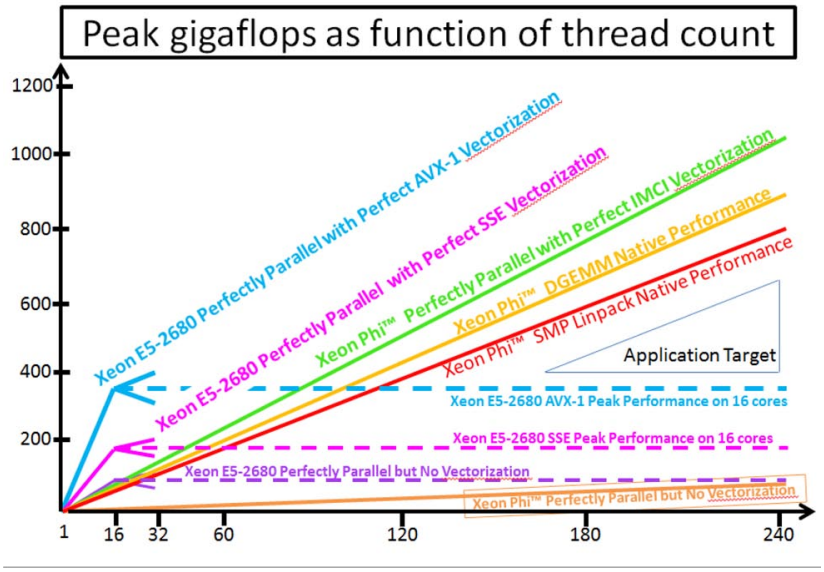
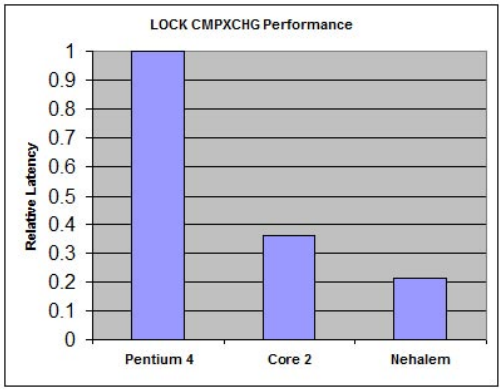
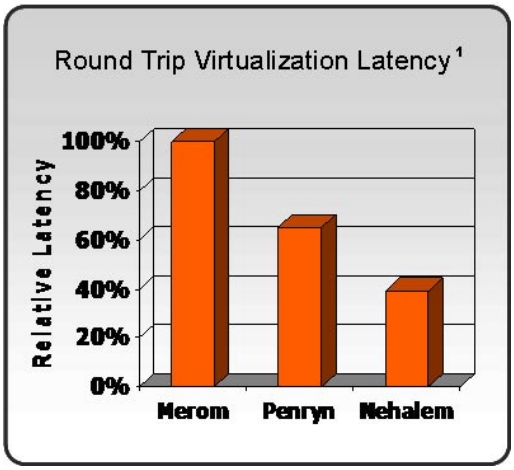


Figure 1.13: vectorization GigaFlops Thread Count



(a) An example of synchronization primitives's speed-up



(b) From Merom (Conroe) to Nehalem there was a 2.5x improvement in about 2 years and half

Figure 1.14: Some examples of ad hoc improvements. When it is difficult to improve performance in a generalized way, efforts are made to identify particular use cases that are subject to significant improvements.

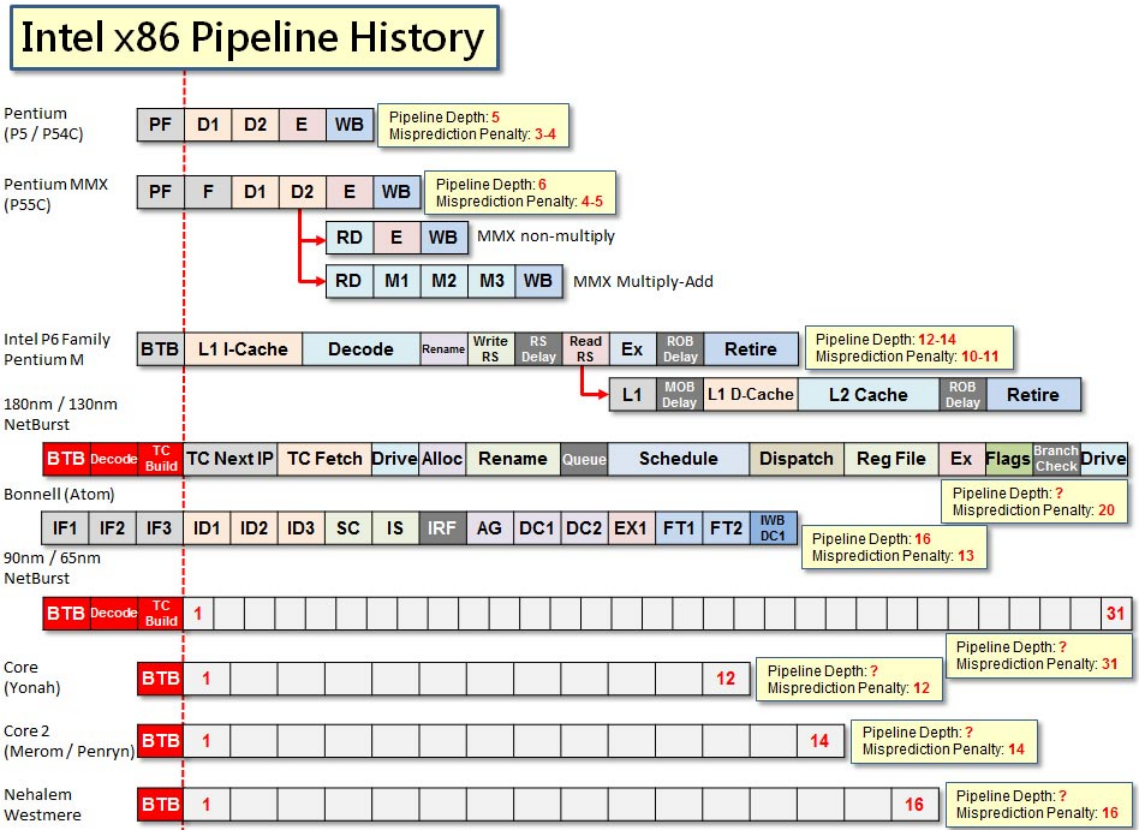


Figure 1.15: Schematic representation of x86 pipelines evolution.

objective	ALU size		ALU size		ALU size	
	16-bit ALU		32-bit ALU		64-bit ALU	
	ADC	ADD	ADC	ADD	ADC	ADD
16-bit	0	1	0	1	0	1
32-bit	1	1	0	1	0	1
64-bit	3	1	1	1	0	1
128-bit	7	1	3	1	1	1
256-bit	15	1	7	1	3	1
512-bit	31	1	15	1	7	1

Table 1.7: ALU operations needed to compute big numbers additions. When you want to work with numbers that require a higher number of bits than that provided by hardware, i.e. registers and ALUs, you must decompose the execution of each abstract arithmetic operation into several elementary instructions. In this case, for each addition, the ratio between the number of bits desired and those available to the hardware gives the number of needed arithmetic operations. Of these operations, one will be a simple ADD, while all the others will be ADC, or additions with carry. This table shows the operations necessary to compute additions between numbers up to 512 bits on 16, 32 or 64 bit ALUs. This table enumerates only arithmetic operations, but it is usually necessary to carry out a series of support instructions such as moves or pushes and pops, to remedy the limitations of registers.

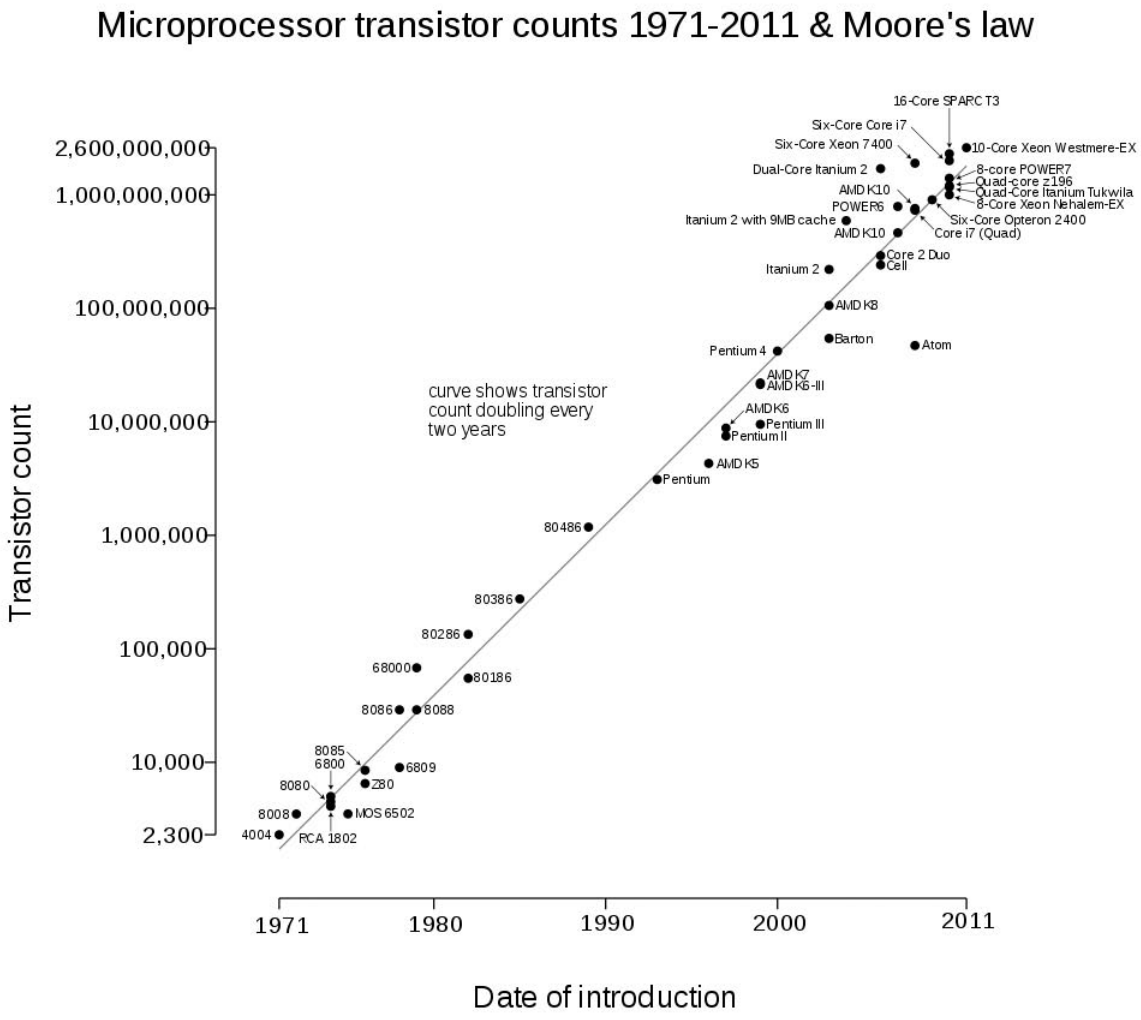


Figure 1.16: Microprocessor's transistor count from 1971 to 2011[5]

objective	ALU size			ALU size		
	32-bit			64-bit		
	MUL	ADC	ADD	MUL	ADC	ADD
32-bit	1	0	0	1	0	0
64-bit	4	5	3	1	0	0
128-bit	16	29	15	4	5	3
256-bit	64	125	63	16	29	15
512-bit	256	509	255	64	125	63

Table 1.8: ALU operations needed to compute big numbers multiplications. When you want to work with numbers that require a higher number of bits than that provided by hardware, i.e. registers and ALUs, you must decompose the execution of each abstract arithmetic operation into several elementary instructions. In this case, for each multiplication, the number of elementary multiplications grows as the square of the ratio between the number of bits desired and those available to the hardware, but a large number of support additions are needed. This table shows the operations necessary to compute multiplications between numbers up to 512 bits on 32 or 64 bit ALUs. This table enumerates only arithmetic operations, but it is usually necessary to carry out a series of support instructions such as moves or pushes and pops, to remedy the limitations of registers.

1.1.6 ALU size effectivness

ADDIZ. MOLTIPL. ADDIZ. MOLTIPL. 64-bit 2 12 1 1 128-bit 4 60 2 12 256-bit 8 252 4 60 512-bit 16 1020 8 252

1.1.7 A significant example: the evolution from P6 to Sunny Cove

Since the 4004 introduction, Intel represents the historical leader in the semiconductor market and, above all, in the CPU business. Its success depended predominantly on the ubiquitous diffusion of systems with the x86 instruction set developed by it (also by virtue of its backwards compatibility): since the choice made by IBM in 1980 of Intel 8088 as CPU of its first Personal Computer, Intel products have gained significant sales volumes, operating sometimes almost monopolistically in some market areas. The evolution of P6 microarchitecture is really an emblematic case, because it is a microarchitecture that has survived, obviously with profound changes, for over 25 years, through a dozen lithographic processes; performance of these processors have grown due to improved microarchitecture, elimination of bottlenecks, enlarged caches, higher clock and multicore.

Around 1990, Intel began in parallel the development of two new CPUs, P5 and P6. The first saw the light in 1993 with the name Pentium and represented a profound evolution of the previous 80486, above all for the adoption of a super-scalar architecture (with the two famous, albeit not fully symmetric, u and v pipes); moreover the fast, synchronous on-chip first-level cache doubled from 8 to 16 kB (with the adoption of a Harvard architecture, with separate caches and signal pathways for instructions and data); data cache used the MESI protocol to support more efficient write-back strategy in addition to the write-through previously used by the 80486 processor; branch prediction with an on-chip branch table was added to increase performance in looping constructs; there was a substantial improvement of the floating-point unit (FPU); external data bus was doubled from 32 to 64 bits; an APIC was integrated to support systems with multiple

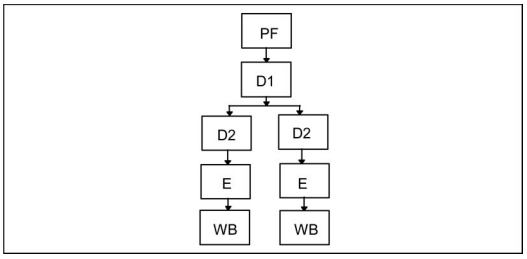


Figure 1.17: Intel P5 (Pentium) pipeline[Vv.Aa., 1997]

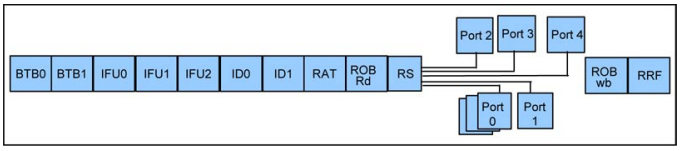


Figure 1.18: Intel P6 (Pentium Pro) pipeline[Vv.Aa., 1997]

processors, providing "glueless" support for up to two processors, enabling low-cost, two-way symmetric multiprocessing.

The P6 processor, that was sold commercially under the name Pentium Pro in 1995 (with simultaneous release of 600 nm and 350 nm variants, both with 5.5 milion transistors), represented an important turning point in the development of Intel CPUs: although it had insignificant architectural extensions compared to the Pentium, in the form of new conditional move and compare instructions, it was a beefier super-scalar micro-architecture, the first Intel CPU that involved the ("on the fly") translation of the CISC x86 instructions into RISC-like micro-ops (something already done in 1994 by NexGen with its Nx586 and that AMD was working on for its K5), the first one that used an out of order and speculative execution engine, the first with a very fast level two cache (connected by a special, dedicated bus: Dual Indipendent Bus was the commercial feature's name) and also provided "glueless" support for up to four processors.

P6 was received as an interesting and successful blend of architectures, dominating the high end PC market and spreading in the workstation and server areas. P6 executes the x86 CISC instruction set, maintaining the advantage of high code density, as well as backward compatibility with previous generations, but x86 instructions are fetched and decoded by a superscalar (three ways), in order, decoding stage, which translates any of them into one or more µops, i.e. simple, RISC-like instructions (the first decoder is

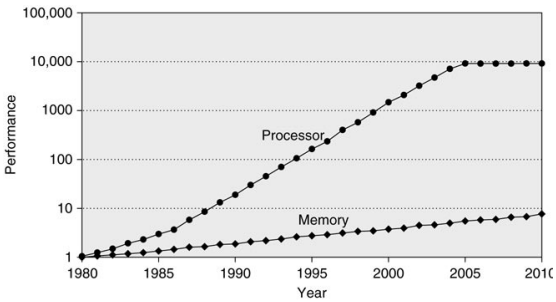


Figure 1.19: Comparison between the increase of CPU and RAM speed (1980-2010). While CPU speed increased substantially exponentially, the RAM speed has grown linearly.

complex and can manage x86 instructions which requires from 1 to 4 µops, while the other two decoder are simple and can translate only x86 instructions requiring a single µop). Then, a high performance "post-RISC" execution engine supports out-of-order execution through multiple, asymmetric (integer, floating point, load, store data and store address) pipelines, making use of five issue ports. While super-scalar versions of RISC processors relied on the compiler to order instructions for maximum performance and hardware checked the legality of multiple simultaneous instruction issue, post-RISC processors are much more aggressive at issuing instructions using hardware to dynamically perform the instruction reordering. The new processors find more parallelism by executing instructions out of program order, but the final stage retires the instruction in original program order.

In 1997, Klamath (the first Pentium II), with 7.5 milion transistors, doubled up the first level cache, both for instruction both for data (from dual separated 8 KB 2-way set associative to dual 16 KB 4-way set associative), it added MMX instruction set and it improved execution of legacy, 16-bit machine code (in particular, mitigating the negative effects in the partial writing of the 32 bit registers, introducing an internal flag to skip pipeline flushes whenever possible, and adding segment register caches). But all that glitters is not gold: one of the strengths of the P6 was the L2 cache operating at full processor frequency and with an access latency (additional to the L1 miss) of just 3 cycles (at least for basic 256KB version; there were even 512 KB L2 version at 166 MHz and 200 MHz and afterwards a special 1 MB L2 version at 200MHz). While P6 required a special dual or triple cavity package for the very fast L2 cache, Pentium II was implemented as a daughter board, i.e. a specific printed circuit board, with a pair of separate dies for 512KB L2 cache, operating at half frequency, with 18 cycle latency. This daughter board was packaged in a slot-based module within a plastic cartridge. Pentium II Xeon versions used up to 2MB of DDR SRAM (Static RAM) as L2 cache at half frequency, but full data rate burst transfer rate and only 14 cycle latency.

At the time Intel declared that it was essential to move the CPU into a sort of cartridge, believing it impossible to exceed 200 MHz with a conventional CPU installed in a socket; Intel asserted on the one hand that the heat produced was not disposable with conventional cooling systems, on the other specified that the cartridge contributed to isolate the processor from sources of electromagnetic noise. The fact that, to reduce production costs, Intel quickly switched from a SECC type cartridge to a thinner one called SECC2 and then introduced a shieldless SEPP format for the Celeron CPUs, amply demonstrated the groundlessness of the arguments inherent in the shielding to be electromagnetic waves. The introduction of the long-running Socet 370 platform just a year and a half after the presentation of Slot 1 completed the circle, emphasizing that there were no insurmountable difficulties in developing high frequency processors on the socket interface. In fact, the only real motivation behind the introduction of slot 1 was related to the second level cache. In fact, Intel needed a fast and connected L2 cache via a Back Side Bus, independent from Front Side Bus connection to the northbridge of system chipset, to differentiate its products from competitors' solutions, but clashed against the fact that the Klamath's 350 nm production process did not allow the integration of a significant amount of cache, while the strategy of the Pentium Pro dual cavity package was too expensive for a widely distributed product. In 1998, Mendocino Celeron at 250 nm introduced a very fast 128kB on-die L2 cache (with a load to use latency of just 4 cycles), while starting from 180 nm in 1999 also the high-end CPUs returned to a socket version, using on-die cache.

Deschutes was only a die shrink of Klamath at 250nm, but later it introduced 100 MHz FSB (Front Side Bus), a 50% increase and overcame a limitation of the early Pentium II; in fact, although they were able to address 4 GB of memory, on all the Klamath and on the first version of Deshutes, tag chips were mounted for the L2 cache that did not allow

to manage cache-accelerated access beyond the 512MB limit.

Katmai, the first Pentium III, introduced Processor Serial Number and floating-point SSE SIMD instruction set; later, Katmai-B intrudeced 133 MHZ FSB. In late 1999, Coppermine, instead, in addition to the 180 nm die shrink, introduced ATC (Advanced Transfer Cache), an L2 cache operating at full CPU frequency, with only 5 cycles latency and with a quadrupled bus, from 64 to 256 bits.

At the end of 2000, Willamette CPU, in origin P68, made its debut, as the first Pentium 4, based on the hyper-pipeline of the NetBurst microarchitecture. Despite Tualatin, the 130 nm Pentium III die shrink, was on roadmaps, the P6 microarchitecture seemed destined to be permanently shelved, as the 32-bit NetBurst processors would conquer PC market and the 64-bit Merced/Itanium platform would dominate the server and HPC sectors, only to land on high-end PCs in about 5 years.

Indeed, Intel was looking for a new architecture to replace the aging x86 architecture and to offer 64-bit computing, so in 1995, Intel and HP formed a joint venture to design a new architecture, Merced or IA-64; its roots lied in earlier attempts to build LIW and VLIW machines (Long Instruction Word and Very Long Instruction Word), especially those at Cydrome and Multiflow—and in a long history of compiler work that continued after these companies failed at HP, the University of Illinois, and elsewhere. VLIW try to exploit instruction level parallelism (ILP) in software, at compile time, avoiding the complexity inherent in some other chip designs. To remark that in their approach ILP is actively pursued, Intel called EPIC (Explicit Parallel Instruction Computing) its VLIW schema. Originally, Merced it was expected to be released in 1998, but was introduced three years later under the trade name Itanium; a profound revision, Itanium 2, made its debut in 2002.

Meanwhile, Intel's Israeli development section, after the abortion of the Timna project (low cost CPU with integrated graphic, project severally penalized by having invested in the Rambus memory), decided to carry out a profound revision of the P6 microarchitecture to develop a CPU, intended only for the mobile market, in which energy efficiency was more important than operating frequency. In 2003, Banias was released as Pentium M, the core of famous Centrino mobile platform. In comparison to Pentium IIII, Banias doubled both L1 instruction and data cache, doubled the shared L2 cache (with to respect to Tualatin), and borrowed from Pentium 4 SSE2 instruction set and very fast quad-pumped FSB. Micro-ops fusion, faster operations with the hardware stack, significantly improved branch prediction. Radical power saving technologies are introduced due to the orientation to mobile applications. There are a number of microarchitectural improvements, floating point multiplication/addition operations are divided between two separate execution ports, the delays due to partial register stall are fixed. Improved branch prediction and return address stack are among the most significant progress in Banias. Two ways to classify branch instructions are by whether the branch is conditional or unconditional,and whether the branch target is direct (can be calculated statically) or indirect (depends on a register value that is unknown until run time). Return instructions are indirect branches, so they can have multiple branch targets and can be difficult to predict. The multiple branch targets come from the fact that functions can be called from multiple sites (code reuse), and the return must branch back to the caller, which can be different every time. What makes function returns special is that in most sane programming languages, function calls and function returns are matched, and functions are nested in a last-in-first-out order. A hardware stack-like structure would be able to accurately predict function return target addresses by pushing the location of each call instruction that executed, then popped it off for each return instruction. This stack-like predictor structure is the return address stack, which works very well if the typical length of nested calls is not very long. Furthermore, for the first time Banias introduces direct

support to manage indirect jumps, an aspect of crucial importance for improving the efficiency in the use of polymorphism, in the context of Object Oriented Programming.

Later Dothan was a Banias die shrink at 90 nm, with double L2 cache and some minor tweaks, such as the execution speed of MMX addition instruction is doubled.

So, after a decade of honorable service, the P6 project seemed destined to play only for some time a niche role, in thin and light notebooks, waiting for it to be possible to reduce the consumption of the Netburst microarchitecture of the Pentium 4. Instead, the Pentium 4 Prescott arrived on the market with a strong delay, with operating frequencies significantly lower than expected, high energy consumption; furthermore, despite a long list of architectural improvements, the 55% increase in the misprediction pipeline, from 20 to 31 stages, resulted in a IPC reduction.

On May, 7th 2004, the ambitious Tejas project, a beefier Netburst processor slated to operate at frequencies of 7 GHz or higher, increasing the number of pipeline stages to between 40 and 50 stages, and with a new dynamical multithread support, was officially canceled and Intel announced the NGMA (Next-Generation Micro-Architecture) initiative: a dual core, low power 64 bit x86 CPU.

Yonah was then introduced as a sort of midterm solution (not a Conroe yet, not a Pentium III anymore), with the distinction of being the only true 32-bit dual core x86 CPU: in 2006, Core Duo was a dual core with shared L2 cache, it added SSE3 instruction set, but above all a significantly improved decoder with support for SSE micro-op fusion (to handle packed SSE instructions in all three decoder channels). It supports VT and improved power saving technologies, improved memory prefetch and faster execution of some instructions.

NGMA was developed as a sort of way out of the difficulty of frequency scaling with Prescott and derivatives and it was presented as a fusion of the micro-architectural characteristics of Pentium-M and Pentium 4, but in reality it was a dual core project, based exclusively on a profound revision of the Banias micro-architecture, with an enhancement of the execution of the SSE instructions and the adoption the so-called T* originally developed for Prescott and derivatives (CT, LT and VT, aka Clackamas, LaGrande and Vanderpool Technologies, respectively the 64-bit extension, security feature - TXT, Trusted Execution Technology - and virtualization hypervisor support).

NGMA was released in 2006 as Conroe (desktop) and Merom (mobile) CPU: for the first time, the fundamental P6 pipeline was modified, with the introduction of a wider execution engine, with a fourth decoder and a sixth issue port. Since AMD has long been the industry leader in the x86 CPU sector for the past seven years, with high-performance products such as Athlon K7 and Thunderbird, Athlon XP Palomino, Athlon 64 ClawHammer and Athlon 64 X2 Toledo and Windsor, which had introduced high IPCs, 64-bit instructions, integrated memory controllers, Intel decided to launch a new course to ensure rapid development of new solutions. With Conroe, Intel started indeed "Tick-Tock execution", a production model that included each year the introduction of a new product, with the alternation of the debut of a new microarchitecture on a proven lithographic process (a tock, such as Conroe, Nehalem, Sandy Bridge, Haswell and Skylake) or the introduction of a new production process by adopting at most minor revisions to an existing microarchitecture (a tick, such as Penryn, Westmere, Ivy Bridge, Broadwell). by virtue of the considerable delays in the introduction of the 14 nm lithographic process and the difficulties still existing with the 10 nm one, Intel has declared that tick-tock execution was no longer sustainable and it decided to change its strategy, coining the slogan "Process-Architecture-Optimization". Since 2016, under this three-phase model, every die shrink is followed by a microarchitecture change and then by one (or more!) optimization; indeed, on 14 nm litographic node, after the tick/process Broadwell and the tock/architecture Skylake, on optimization phase there were Kaby Lake, Kaby Lake

Refresh, Coffe Lake, Coffe Lake Refresh and Comet Lake.

In 2008, Nehalem introduced a very big IPC bump largely because it added an Integrated Memory Controller, five years after AMD K8; moreover, it backported Hyper Threading, Intel's marketing name for simultaneous multi-threading (SMT), i.e. the capabiliy the ability to fetch instructions from two threads at the same time instead of just a single one, from Netburst microarchitecture; it was the first CPU with Turbo Mode, hat is, the possibility of opportunistically increasing the operating frequency, when not all cores are active or the energy consumption and operating temperatures are under respective thresholds and the first with macro-ops fusion (despite having four decoders, in appropriate circumstances it can decode five x86 instructions per cycle); it included a Power Control Unit (PCU), which tunes voltage for a given frequency, operating conditions, and silicon characteristics.

In 2010, Westmere Tick at 32 nm introduced only new instructions to accelerate AES encoding; it was an atypical generation, because it included only Gulftown high end six-core solutions and low level Arrendale dual core CPU with integrated (on package) graphic, without offering quad-core solutions. Anyway, server versions of both Nehalem-EX and Westmere-EX introduced a ring bus core interconnection topology, which was later adopterd by Sandy Bridge.

In 2011, Sandy Bridge Tock, manufactured in the 32 nm process, was developed by Intel's Research and Development center in Haifa, from the same team that has baked Banias, and it was perhaps the the most profound revision of the P6 microarchitecture, for example for the adoption of a PRF-based (Phisical Register Files) renaming architecture (an aspect borrowed from Netburst), instead of centralized Retiremente Register File, which assures a single copy of every data and no movement after calculation (especially important for the large registers adopted by the SIMD units, such as the 128-bit XMM registers of the SSE and the 256-bit YMM registers of the AVX). Informally, in old roadmaps that product was denoted by Nehalem-C and later, in 2009, by Gesher, literally meaning bridge in Hebrew; however, Intel discarded the name when it was discussed that Gesher is also the name of a former political party in Israel. Anyway, Sandy Bridge introduced a profound revision to purse high performance and high power efficiency; it introduced a new µOP cache, containing 1536 already decoded instructions (but it is organized differently than the Netburst trace cache), higher branch prediction accuracy, new Zeroing and Ones idioms optimizations, AGUs capable to do both loads and stores dinamically, new 256 bit AVX instructions, improved performance for transcendental mathematics, AES encryption and SHA-1 hashing, a ring bus topology, a faster Last Level Cache, an on die integrated graphics and so on. For Sandy Bridge, a typical pipeline length of 14 stages has been estimated, in the case of hits in the µop cache, extended by 5 stages if it is necessary to use decoders.

In 2012, Ivy Bridge was a Tick, i.e. a die shrink to 22 nm of Sandy Bridge, based on FinFET 3D Tri-Gate transistors; it introduced a F16C Extension for performing half-precision/single-precision floating point conversion, a snew random number generator and the RDRAND instruction, codenamed Bull Mountain.

In 2013, Haswell was a Tock on 22 nm, it was an important redesign with the introduction of a pair of issue ports, a fourth ALU and the possibility of executing three-operand FMA instructions.

After significant problems and delays with the transition to 14nm, Broadwell brought marginal improvements, while Skylake has been the reference architecture for Intel processors for the past 5 years.

For example, the only difference between Skylake and Kabylake was that SkyLake's Speed Shift implementation is significantly improved, cutting responsiveness by as much as 66% (down to just 10-15ms to reach peak frequency, rather than 30 ms). Indeed,

with Kaby Lake, the hardware control around Speed Shift was tremendously improved, altought there wasn't a change in the OS driver; this could have an impact on latency limited interactions as well as situations where delays occur, such as asynchronous web page loading, so good Speed Shift implementation is a play for user experience in interactive enviroments.

This synthetic historical overview of the evolution of a microarchitecture over the course of about 25 years serves to testify how complex it is for conventional computers to be able to achieve generalized performance improvements.

For example, the first level cache of data and instructions was organized with two 8 kB blocks each for the 600nm production process, it was doubled two years later with the 350 nm process but took another six years and three lithographic processes, to reach the size of 32 kB each in 2003 in Banias; since then they have not changed, except for the 50% increase in the very recent Ice Lake, still a ghost product.

The second level cache, on the other hand, found a fluctuating trend: at the beginning it was very fast, but extremely expensive, available mainly with the capacity of 256 kB (although 512 and 1024 kB versions were contemplated, for those who needed speed at any cost), then its capacity was doubled at the expense of speed, especially in terms of latency, with Coppermine it was incorporated on die in the very fast ATC implementation and from that moment it doubled with each generation (Tualatin, Banias, Dothan), first to come across dual core solutions. In fact, despite the 90 to 65 nm die shrink, Yonah provided an L2 cache shared by both cores with overall capacity to that offered by Dothan for a single core. The L2 cache shared by a pair of cores has increased to 4 MB with Conroe at 65 nm and 6 MB with Penryn at 45nm, to then be reduced to 256 kB per core in the transition to the Nehalem generation at 45nm. From then on, the L2 cache has remained unchanged for multiple generations of products, to the Skylake-X / Cascade-Lake server variant (which uses 1 MB), and 512 kB for the problematic 10 nm products, CannonLake and Ice Lake. This oscillation in the size of the L2 waves cache has had the consequence that the access latency has gone from 4 cycles of the 128 kB of Mendocino from 250 nm, 5 cycles for the 256 kB of Coppermine to 180 nm to rise to about 10 cycles per 1MB of Banias, to go up to 15 cycles with the shared Conroe cache, go down to 10 cycles with Nehalem and oscillate between 11 and 12 cycles in subsequent generations, to land at 13 cycles with Ice Lake. Furthermore, as regards the bandwidth, it has gone from 8 bytes per cycles of the P6, to 4 for Klamath/Deshutes/Katmai, 16 for Coppermine/Tualatin, 32 for the CPUs from Banias up to Ivy, to reach 64 from Haswell to Ice Lake.

As for the front end, the decoders remained 3 for a dozen years, from 600 to 65 nm (from P6 to Yonah), to be expanded to 4 with Conroe and 5 with the Palm Cove core of Skylake. In 2011 Sandy Bridge introduced a 1536 instruction mop cache, which remained unchanged for a decade, until the introduction of the Sunny Cove core of Ice Lake.

The Out of Order window size, that dictate how many on flight instruction can be reordered to cover memory latency and maximize execution unit utilization, started at 40 µop for original P6 in 1995, the first time in 2003 it grew to 64 for Banias, then 96 for Conrore, 128 for Northwood (to better support two hardware thread per core via HyperThreading), 168 for Haswell in 2013, 192 for Broadwell, 224 for Skylake and Cannon Lake and 352 for Ice Lake.

An other example are AGU, special units devoted to address computation: since P6 to Westmere (from 600 nm in 1995 to 32 nm in 2010), there was only two AGU, one for loads and the other for store. Only in 2011, Sandy Bridge changed this structure, allowing the second AGU to generate addresses for load and store operations without distinction; in 2013 Haswell added a third AGU, to better support FMA3 operations, two dedicated to load and the other one to store operations; Ice Lake once again makes the AGU symmetrical, providing two for the loads and as many for the stores. Associated

with each address generated by an AGU, there is the possibility of accessing the L1 cache; from P6 to Yonah, at most one 64 bit load and store per cycle was possible; since Conroe up to Westmere, at most a 128 bit load and store were possible. With Sandy Bridge and Ivy Bridge, at most two loads or a 128 bit load and store were possible; In 2013, Haswell doubled its 256-bit ports, to better support AVX instructions, up to two loads and one store per cycle; Ice Lake, like Skylake-X, expands 512-bit ports to support AVX-512s, but also adds a second port for store operations.

Perhaps the parameter that indicates more than any other how difficult it is to significantly improve a microarchitecture over the years is the number of issue ports. The 600nm P6 had 5 issue ports and remained so until the 65nm Yonah introduced in early 2006, then the Conroe added a sixth to support a third ALU. In 2013, at 22 nm, Haswell added two to accommodate an additional AGU and a fourth ALU.

This roundup, without pretension of exhaustiveness, of generational differences during the evolution of the P6, shows how much more difficult it is to achieve fairly generalized increases in performance (not dependent, that is, on specific use cases or on the adoption of new ad hoc instructions). Furthermore, keeping the reference architecture relatively stable, although subject to periodic reinterpretations, perhaps has simplified life for some hacker activities.

For the past two years, indeed, we've seen multiple attacks based on microarchitectural vulnerabilities, such as like Meltdown, Spectre, Zombieload, Foreshadow/L1TF. All this security exploits can be used to probe or stole data. On March 2020, Load value injection was also dicovered, which can inject data values, and is resistant to the countermeasures so far used to mitigate the Meltdown vulnerability. The data injection can either be instructions or memory addresses, allowing the attacker to obtain data from the victim. This data injection bypasses even stringent security enclave environments, such as Intel's Software Guard Extensions (SGX), and the attackers claim that successful mitigation may result in a slowdown of 2x to 19x for any SGX code.

1.1.8 Multi-core

1174/5000 As a consequence of the difficulty of increasing the average CPI and increasing operating frequencies, multi-core solutions have proliferated in recent years. On the destop and portable computers side, dual core solutions spread very quickly, because they allowed significant performance increases with any workload: in fact, even in the hypothesis of running essentially single-threaded applications, a real multitask between two applications complex, frequent taskswitching between multiple applications or interacting with a single program, while in the background system updates, firewalls, antivirus, etc. are active, a dual core cpu offered obvious advantages. The transition to quadcore solutions has been slower, but due to AMD's technical and economic difficulties, Intel has found itself to be an undisputed monopolist in the medium-high end of the market for several years, leaving a stationary situation. With the introduction of the AMD Ryzen, based on Zen microarchitecture in 2017, the market has revived, allowing the rapid spread of 6-8 core solutions, with the availability of 10-12 core products under $ 500.

Anyway, solutions with a large number of cores have proliferated rapidly in the server, workstation and HPC sectors, such as the 28-core Skylake-EP, the 32-core AMD Naples and the 64-core AMD Rome, the latter also available on prosumer platforms under the aegis of Threadripper.

But over the past 15 years there has been a flourishing of interesting solutions, even outside of the x86 microprocessors, IBM POWER and UltraSparc.

The Teraflops Research Chip (also called Polaris) is a research manycore aka multicore processor, containing 80 cores developed by Intel Corporation's Tera-Scale Computing Research Program. The processor was officially announced February 11, 2007

and shown working at the 2007 International Solid-State Circuits Conference. Features of the processor include dual floating point engines, sleeping-core technology, self-correction, fixed-function cores, and three-dimensional memory stacking. The purpose of the chip is to explore the possibilities of Tera-Scale architecture (the process of creating processors with more than four cores) and to experiment with various forms of networking and communication within the next generation of processors. Along with 80 cores, the chip also contains 80 routers. Each of the cores on board the teraflops research chip contains two floating point engines. better load distribution and a decreased chance of overheating. If a core is overloaded then the heat produced by that core increases, which reflects a decrease in efficiency and a waste of energy. In the teraflops research chip, if some of the cores are being overloaded, that load can just be delegated to other cores, resulting in a load distribution which does not create as much heat The processor is constructed using a 65 nm CMOS process, the die is 12.64 mm by 21.72 mm (274.5 mm2) and contains 100 million transistors. Intel first disclosed it had built a prototype 80-core processor during 2007 Intel Developer Forum, when CEO Paul Otellini promised to deliver the chip within five years.

Intel used 100 million transistors on the chip, which measures 275 millimeters squared. By comparison, its Core 2 Duo chip uses 291 million transistors and measures 143 millimeters squared. The chip was built using Intel's 65-nanometer manufacturing technology, but any likely product based on the design would probably use a future process based on smaller transistors. A chip the size of the current research chip is likely too large for cost-effective manufacturing.

The computing elements are very basic and do not use the x86 instruction set used by Intel and Advanced Micro Devices' chips, which means Windows Vista can't be run on the research chip. Instead, the chip uses a VLIW (very long instruction word) architecture, a simpler approach to computing than the x86 instruction set.

There's also no way at present to connect this chip to memory. Intel is working on a stacked memory chip that it could place on top of the research chip, and it's talking to memory companies about next-generation designs for memory chips. Intel's researchers will then have to figure out how to create general-purpose processing cores that can handle the wide variety of applications in the world.

But the primary challenge for an 80-core chip will be figuring out how to write software that can take advantage of all that horsepower. The PC software community is just starting to get its hands around multicore programming, although its server counterparts are a little further ahead. Still, Microsoft, Apple and the Linux community have a long way to go before they'll be able to effectively utilize 80 individual processing units with their PC operating systems.

"The operating system has the most control over the CPU, and it's got to change. It has to be more intelligent about breaking things up"

Intel demonstrated the chip running an application created for solving differential equations. At 3.16GHz and with 0.95 volts applied to the processor, it can hit 1 teraflop of performance while consuming 62 watts of power. Intel constructed a special motherboard and cooling system for the demonstration in a San Francisco hotel.

The Single-Chip Cloud Computer (SCC) is a computer processor (CPU) created by Intel Corporation in 2009 that has 48 distinct physical cores that communicate through architecture similar to that of a cloud computer data center. Cores are a part of the processor that carry out instructions of code that allow the computer to run. The SCC was a product of a project started by Intel to research multi-core processors and parallel processing (doing multiple calculations at once). Additionally Intel wanted to experiment with incorporating the designs and architecture of huge cloud computer data centers (Cloud computing) into a single processing chip. They took the aspect of cloud computing

in which there are many remote servers that communicate with each other and applied it to a microprocessor. It was a new concept that Intel wanted to experiment with. The name "Single-chip Cloud Computer" originated from this concept. Intel developed this new chip architecture based on huge cloud data centers, the cores are separated across the chip but are able to directly communicate with each other. The chip contains 48 P54C Pentium cores connected with a 4x6 2D-mesh. This mesh is a group of 24 tiles set up in four rows and six columns. Each tile contained two cores and a 16 KB (8 per core) message passing buffer (MPB) shared by the two cores, essentially a router. The SCC naturally supports the message passing programming model, as it is not cache-coherent in hardware. [Corley, 2010]

As the number of cores per chip increases, load balancing becomes more important (and challenging) for efficient use of the available processing power.

For evaluating and comparing computer architectures, Amdahl's law is often used. It is named after computer scientist Gene Amdahl, and was presented at the AFIPS Spring Joint Computer Conference in 1967. For a task at a fixed workload, Amdahl's law expresses the theoretical speedup of the execution of the task with the respect to improvements of the computer: the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used

Amdahl's law can be formulated in the following way:

$$S_{overall}(p, s) = \frac{1}{(1-p) + \frac{p}{s}}$$

where

$S_{overall}$ is the theoretical speedup of the execution of the whole task; s is the speedup of the part of the task that benefits from improved system resources; p is the proportion of execution time that the part benefiting from improved resources originally occupied.

Furthermore,

$$S_{latency}(s) \leq \frac{1}{1-p} \lim_{s \to \infty} S_{latency}(s) = \frac{1}{1-p}$$

shows that the theoretical speedup of the execution of the whole task increases with the improvement of the resources of the system and that regardless of the magnitude of the improvement, the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement.

Amdahl's law applies only to the cases where the problem size is fixed. In practice, as more computing resources become available, they tend to get used on larger problems (larger datasets), and the time spent in the parallelizable part often grows much faster than the inherently serial work. In this case, Gustafson's law gives a less pessimistic and more realistic assessment of the parallel performance: this law gives the theoretical speedup in latency of the execution of a task at fixed execution time that can be expected of a system whose resources are improved. It was presented in the article Reevaluating Amdahl's Law in 1988.

Gustafson estimated the speedup S gained by using N processors (instead of just one) for a task with a serial fraction s (which does not benefit from parallelism) as follows:

$$S = N + (1 - N)s$$

Using different variables, Gustafson's law can be formulated the following way:

$$S_{latency}(s) = 1 - p + sp$$

where

S_latency is the theoretical speedup in latency of the execution of the whole task; s is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system; p is the percentage of the execution workload of the whole task concerning the part that benefits from the improvement of the resources of the system before the improvement. Gustafson's law addresses the shortcomings of Amdahl's law, which is based on the assumption of a fixed problem size, that is of an execution workload that does not change with respect to the improvement of the resources. Gustafson's law instead proposes that programmers tend to set the size of problems to fully exploit the computing power that becomes available as the resources improve. Therefore, if faster equipment is available, larger problems can be solved within the same time.

The impact of Gustafson's law was to shift[citation needed] research goals to select or reformulate problems so that solving a larger problem in the same amount of time would be possible. In a way the law redefines efficiency, due to the possibility that limitations imposed by the sequential part of a program may be countered by increasing the total amount of computation.

1.1.9 GPU

In the last fifteen years, there has been a growing multidisciplinary interest in the use of GPUs (Graphics Processing Units) for non-graphics, general purpose applications. With respect to conventional general purpose CPUs, GPUs usually invest more resource (logical gates, power usage) in arithmetic computation than in control operations; moreover, GPUs usually avoid or make little use of sophisticated techniques, such as out of order execution and speculative execution, furthermore they use simpler, more regular memory pattern access, which are more sensitive to high bandwidth than to low latency and not require complex memory disambiguation techniques (to avoid or at least mitigate aliasing issue). Most operations, even on "traditional" GPU, operate in a vectorized fashion: one operation can be performed on up to four values at once; for example, a pixel shader instruction could manipulate RGBA data (red, green, blu, alpha channels of pixel representation), while a vertex shader instruction could manipulate four-dimensional projective coordinates (x, y, z, w) of a triangle's vertex (projective or homogeneous coordinates are ubiquitous in computer graphics because they allow basic graphics manipulations, such as translation, rotation, scaling and perspective projection, to be each represented as a matrix, so that any arbitrary sequence of these transformations can be computed by evaluating the row-by-column product of the matrices corresponding to each transformation; by contrast, using simpler, tridimensional Cartesian coordinates, translations and perspective projection cannot be expressed as matrices). Moreover, combination of SIMD / MIMD techniques allow you to apply the same operation simultaneously to multiple vectors or vector components. GPUs have very large register files, which allow them to reduce context-switching latency. Register file size is also increasing over different GPU generations. Pioneer general-purpose computing on graphics processing units (GPGPU) started about 2002-2003, when DirectX 9 Shader Model 2.x suggested the support of two floating point types, full and partial precision, for pixel shader operations. is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). Full precision support could either be FP32 or FP24 (floating point 32- or 24-bit per component) or greater, while partial precision was at least FP16. ATI series of GPUs supported FP24 precision only in the programmable fragment pipeline, while Nvidia series supported both FP16 and FP32. A fundamental transition was found in 2007 with the transition to unified GPU architectures for vertex, pixels and other types of shaders (hull, computational, etc.).

As DirectX 9-capable GPUs became available, some researchers took notice of the

raw performance growth path of GPUs and began to explore the use of GPUs to solve complex parallel problems. DirectX 9 GPUs had been designed only to match the features required by the graphics API. To access the computational resources, a programmer had to cast their problem into native graphics operations.

Porting applications to GPGPUs is one of the most important issues in these highly-parallel architectures. There are millions of lines of existing legacy parallel code, which cannot exploit GPGPUs easily (for example, scientific communities have a lot of parallel code mostly written in MPI). In addition, the effort for tuning and writing new code is high for GPGPUs.

While computational power of GPU raised almost exponentially (for parallel tasks), there was a tremendous development in the programming languages and tools for deploying algorithms on GPU.
Among libraries and tools to deploy GPU programs, the usually preferred solution is NVIDIA CUDA, which is both a parallel computing platform and API (Application Programming Interface) model with support for NVIDIA GPUs, but even open-source or multiplatform solution such as OpenCL are spreading more and more. But recently Radeon Open Compute platforM (ROCm) was designed to be a universal platform for gpu-accelerated computing, to integrate multiple programming languages and makes it easy to add support for other languages. ROCm is focused on using AMD GPUs to accelerate computational tasks such as machine learning, engineering workloads, and scientific computing. In order to focus our development efforts on these domains of interest, ROCm supports a targeted set of hardware configurations. ROCm supports AMD GPUs that use following chips: GFX8 GPUs Fiji, Polaris 10, Vega 10, Vega 7nm

1.2 Quantum Computing

Electronics is approaching a major paradigm shift because silicon transistor scaling no longer yields historical energy-efficiency benefits, spurring research towards beyond-silicon nanotechnologies. In particular, carbon nanotube field-effect transistor (CNFET)-based digital circuits promise substantial energy-efficiency benefits, but the inability to perfectly control intrinsic nanoscale defects and variability in carbon nanotubes has precluded so far the realization of very-large-scale integrated systems [Hills et al., 2019].

Inspired by quantum mechanics laws, quantum computing [Deutsch, 1985; Feynman, 1982; Nielsen and Chuang, 2010] is a really promising computational paradigm aimed at overcoming classic computation in solving hard problems thanks to the exploitation of quantum properties such as superposition and entanglement [Deutsch et al., 1995; Finnila et al., 1994; Giovannetti et al., 2008a,b; Kadowaki and Nishimori, 1998; Lloyd, 1993, 1995, 1996; Menon and Ritwik, 2014; Rønnow et al., 2014; Simon, 1997]. Precisely, these features enable a so-called quantum massive parallelism that allows to evaluate simultaneously many computational paths and leads to a significant (usually exponential or quadratic) speed-up in approaching well-known hard problems such as the integer factorization or the search of items in unsorted databases.
However, quantum computers are subjected to stringent limits, both for the current shortage of quantum bits and gates, both for the ineluctable presence of various kinds of single and multi-bit errors, decoherence phenomena and so on [Chuang and Harrow, 2018; Chuang and Shor, 2018c; Nielsen and Chuang, 2010], which, in some ways, make quantum computers more similar to analogue computers than digital ones (l.

Already in the '80s, Feynman and Deutsch[Deutsch, 1985] proposed a novel computational paradigm, based on the idea that to effectively simulate the properties of a quantum system, in physics or chemistry, it could be advantageous to rely on a computer that works on quantum principles.

Today, some major quantum companies (I.B.M. and Rigetti above all) are strongly investing in designing and developing quantum processing unit, and providing users with cloud-based interfaces to enable a simple and fast implementation and analysis of quantum circuits. However, the proliferation of such interfaces is increasing the number of different quantum languages used to represent the quantum circuits, resulting in the definition of a set of heterogeneous cloud-based quantum platforms that do not have any capability of interacting among them. This diversification of schemes and paradigms partly mimics what happened during the development of the classic computers, where for each main programming paradigm different research bodies or companies have developed antagonistic languages, but risks creating an additional level of difficulty at a time in to which quantum computing is still taking its first concrete steps. As a consequence, according to me, the need for an abstract language emerges, capable of capturing all the essential features provided by the proprietary representation of quantum circuits and enable the definition of inter-operable quantum circuits.

Quantum computing, by its very nature, cannot be efficiently reproduced using classical computers. The resources required to simulate quantum hardware increases exponentially with the number of qubits. For the hardware developed in the next few years, even the world's largest classical supercomputers won't be enough.

Despite this, these near-term devices are still 'intermediate scale'. Though large enough that their simulation will be intractable, they won't yet be large enough to implement full-scale quantum error correction. This means that errors will occur during execution of any quantum computation. The longer our quantum program, the more these errors will accumulate.

These errors are unavoidable, and can take many forms depending on the physics of

the devices. To develop quantum algorithms that are robust against their effects, we need to know our enemy. This requires us to have an accurate model of the errors that occur, as well as the ability to simulate their effects. Then we'll be much better equipped to explore near-term quantum applications with noisy devices.

1.2.1 Quantum Mechanics in brief

While classical physics formulates effective models of interactions in nature at "macroscopic" scale, quantum mechanics tries to explain the aspects of nature at very small, atomic and subatomic, scales. It explains the behavior of matter and its interactions with energy on the scale of atoms and subatomic particles. Many aspects of quantum mechanics are counter-intuitive and can seem paradoxical because they describe behavior quite different from that seen at larger scales. For example, the uncertainty principle of quantum mechanics means that the more closely one pins down one measurement (such as the position of a particle), the less accurate another complementary measurement (i.e. conjugate variables) pertaining to the same particle (such as its speed) must become. Another example is entanglement, in which a measurement of any two-valued state of a particle (such as light polarized up or down) made on either of two "entangled" particles that are very far apart causes a subsequent measurement on the other particle to always be the other of the two values (such as polarized in the opposite direction).

Quantum mechanics has the curious distinction of being so far simultaneously the most successful and the most mysterious of scientific theories. Quantum mechanics is a fundamental theory in that important quantities (such as energy and angular momentum) of a bound system are restricted only to discrete values (quantization effect), small objects such as electrons have characteristics of both particles and waves (wave-particle duality).

Quantum mechanics was developed in fits and starts over a remarkable period from 1900 to the 1920s, when Planck's solution in 1900 to the black-body radiation problem, i.e. spectral distribution for electromagnetic radiation in thermodynamic equilibrium, when there is no net flow of matter or energy, introduced the the quantum of action, while Einstein offered a quantum-based explanation of the photoelectric effect (1905), the work for which he was awarded the Nobel prize in 1921. In the mid-1920s, quantum mechanics was developed to become the standard formulation for atomic physics. In the summer of 1925, Bohr and Heisenberg published results that closed the old quantum theory. Due to their particle-like behavior in certain processes and measurements, light quanta came to be called photons (1926). In 1926 Erwin Schrodinger suggested a partial differential equation for the wave functions of particles like electrons. And when effectively restricted to a finite region, this equation allowed only certain modes, corresponding to discrete quantum states – whose properties turned out to be exactly the same as implied by matrix mechanics. In the meantime, he presented the hypothesisThe de Broglie hypothesis holds for all types of matter: all matter exhibits properties of both particles and waves. The wavelength, λ, associated with any object is related to its momentum, p, through the Planck constant, h:

$$p = \frac{h}{\lambda}$$

The entire field of quantum physics emerged, leading to its wider acceptance at the famous Fifth Solvay Conference in 1927. Quantum mechanics matured then into its current form in the late 1920s and physicists had great success applying quantum mechanics to understand the fundamental particles and forces of nature, culminating in the development of the standard model of particle physics. Over the same period, physicists had equally great success in applying quantum mechanics to understand an astonishing range of phenomena in our world, from polymers to semiconductors, from superfluids to superconductors.

Quantum mechanics assert that the state space of a system is a Hilbert space (crucially, that the space has an inner product) and that observables of the system are Hermitian operators acting on vectors in that space – although they do not tell us which Hilbert space or which operators. These can be chosen appropriately in order to obtain a quantitative description of a quantum system. An important guide for making these choices is the correspondence principle, which states that the predictions of quantum mechanics reduce to those of classical mechanics when a system moves to higher energies or, equivalently, larger quantum numbers, i.e. whereas a single particle exhibits a degree of randomness, in systems incorporating millions of particles averaging takes over and, at the high energy limit, the statistical probability of random behaviour approaches zero. In other words, classical mechanics is simply a quantum mechanics of large systems. This "high energy" limit is known as the classical or correspondence limit. One can even start from an established classical model of a particular system, then try to guess the underlying quantum model that would give rise to the classical model in the correspondence limit.

The mathematical concept of a Hilbert space, named after David Hilbert, generalizes the notion of Euclidean space. It extends the methods of vector algebra and calculus from the two-dimensional Euclidean plane and three-dimensional space to spaces with any finite or infinite number of dimensions. A Hilbert space is an abstract vector space possessing the structure of an inner product that allows length and angle to be measured. Furthermore, Hilbert spaces are complete: there are enough limits in the space to allow the techniques of calculus to be used.

A Hilbert space H is a real or complex inner product space that is also a complete metric space with respect to the distance function induced by the inner product the states of a quantum mechanical system are vectors in a certain Hilbert space, the observables are hermitian operators on that space, the symmetries of the system are unitary operators, and measurements are orthogonal projections. The relation between quantum mechanical symmetries and unitary operators provided an impetus for the development of the unitary representation theory of groups, initiated in the 1928 work of Hermann Weyl

Hilbert spaces arise naturally and frequently in mathematics and physics, typically as infinite-dimensional function spaces. The earliest Hilbert spaces were studied from this point of view in the first decade of the 20th century by David Hilbert, Erhard Schmidt, and Frigyes Riesz. They are indispensable tools in the theories of partial differential equations, quantum mechanics, Fourier analysis (which includes applications to signal processing and heat transfer), and ergodic theory (which forms the mathematical underpinning of thermodynamics). John von Neumann coined the term Hilbert space for the abstract concept that underlies many of these diverse applications. The success of Hilbert space methods ushered in a very fruitful era for functional analysis. Apart from the classical Euclidean spaces, examples of Hilbert spaces include spaces of square-integrable functions, spaces of sequences, Sobolev spaces consisting of generalized functions, and Hardy spaces of holomorphic functions. In an inner product space, the concept of perpendicularity is replaced by the concept of orthogonality: two vectors v and w are orthogonal if their inner product $\langle \mathbf{v}, \mathbf{w} \rangle$ is zero. The inner product is a generalization of the dot product of vectors. The dot product is called the standard inner product or the Euclidean inner product. However, other inner products are possible

In 1925, Werner Heisenberg attempted to solve one of the problems that the Bohr model left unanswered, explaining the intensities of the different lines in the hydrogen emission spectrum. Through a series of mathematical analogies, he wrote out the quantum-mechanical analog for the classical computation of intensities. Shortly afterwards, Heisenberg's colleague Max Born realised that Heisenberg's method of calculating the probabilities for transitions between the different energy levels could best be expressed by using the mathematical concept of matrices. In the same year, building on de Broglie's

hypothesis, Erwin Schrödinger developed the equation that describes the behavior of a quantum-mechanical wave. The mathematical model, called the Schrödinger equation after its creator, is central to quantum mechanics, defines the permitted stationary states of a quantum system, and describes how the quantum state of a physical system changes in time. The wave itself is described by a mathematical function known as a "wave function". Schrödinger said that the wave function provides the "means for predicting probability of measurement results". Schrödinger was able to calculate the energy levels of hydrogen by treating a hydrogen atom's electron as a classical wave, moving in a well of electrical potential created by the proton. This calculation accurately reproduced the energy levels of the Bohr model. In May 1926, Schrödinger proved that Heisenberg's matrix mechanics and his own wave mechanics made the same predictions about the properties and behavior of the electron; mathematically, the two theories had an underlying common form. Yet the two men disagreed on the interpretation of their mutual theory. For instance, Heisenberg accepted the theoretical prediction of jumps of electrons between orbitals in an atom, but Schrödinger hoped that a theory based on continuous wave-like properties could avoid what he called (as paraphrased by Wilhelm Wien) "this nonsense about quantum jumps". In the end, Heisenberg's approach won out, and quantum jumps were confirmed.

1.2.2 Quantum Computing basic concepts

Even in classical computing, various kinds of errors than can affect computations or communications, indeed various encoding schemes are adopted to detect and possibly correct random errors. For example, both in the packets of information transferred over a network connection, and in the sectors recorded on a magnetic, optical or solid state medium, there are error detection codes next to the data. In common processors, parity schemes are typically used to protect instruction caches (which are used in read-only mode), while ECC (Error Correction Code) systems for other cache levels; as regards the main working memory, the systems that aspire to be used in mission critical environments make use of ECC schemes, as well as some professional graphics cards or ones specifically dedicated to computing. One of the most important challenges for the realization of quantum information tasks is the implementation of quantum logic gates that are robust in the presence of perturbations.

In quantum computing, furthermore there are both a quantum noise, which is a consequence of the interaction with the environment (a factor that is usually tried to limit with cryogenics) and a sort of classical noise, emerging from the interaction of the classical circuits that are used to experimentally control the quantum system and to make measurements.

The basic idea in quantum information science is that information can be encoded in the state of a quantum mechanical system. Hence, given a input state, which expresses the configuration of some quantum system in a pure state, a quantum algorithm is nothing more than the physical transformation that the system experiences. Hence, by the principles of quantum mechanics, a quantum algorithm is a unitary transformation which maps an input state into an output state:

Diffused misconception is that the potential — and the limits — of quantum computing must come substantially from hardware. After quantum processor with only 5-20 working qubits, the 50-qubit quantum machines now coming online from companies likes of Intel and IBM have inspired predictions that we are nearing "quantum supremacy" — a nebulous frontier where quantum computers begin to do things beyond the ability of classical machines.

But quantum supremacy is not a simple, clearly defined demarcation line, a sort of Rubicon to be crossed, but rather a series of not so small challenges. It will be established

problem by problem, quantum algorithm versus classical algorithm. "With quantum computers, progress is not just about speed. It's much more about the intricacy of the algorithms at play." (Michael Bremner, a quantum theorist at the University of Technology Sydney)

At room temperature, the computer may acquire errors due to thermal motion of the atoms in the computer's structure.

Before the dream of a quantum computer took shape in the 1980s, most computer scientists took for granted that classical computing was all there was. The field's pioneers had convincingly argued that classical computers — epitomized by the mathematical abstraction known as a Turing machine — should be able to compute everything that is computable in the physical universe, from basic arithmetic to stock trades to black hole collisions.

Classical machines couldn't necessarily do all these computations efficiently, though. Let's say you wanted to understand something like the chemical behavior of a molecule. This behavior depends on the behavior of the electrons in the molecule, which exist in a superposition of many classical states. Making things messier, the quantum state of each electron depends on the states of all the others — due to the quantum-mechanical phenomenon known as entanglement. Classically calculating these entangled states in even very simple molecules can become a nightmare of exponentially increasing complexity.

A quantum computer, by contrast, can deal with the intertwined fates of the electrons under study by superposing and entangling its own quantum bits. This enables the computer to process extraordinary amounts of information. Each single qubit you add doubles the states the system can simultaneously store: Two qubits can store four states, three qubits can store eight states, and so on. Thus, you might need just 50 entangled qubits to model quantum states that would require exponentially many classical bits — 1.125 quadrillion to be exact — to encode.

A quantum machine could therefore make the classically intractable problem of simulating large quantum-mechanical systems tractable, or so it appeared. "Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical," the physicist Richard Feynman famously quipped in 1981. "And by golly it's a wonderful problem, because it doesn't look so easy."

It wasn't, of course.

Even before anyone began tinkering with quantum hardware, theorists struggled to come up with suitable software. Early on, Feynman and David Deutsch, a physicist at the University of Oxford, learned that they could control quantum information with mathematical operations borrowed from linear algebra, which they called gates. As analogues to classical logic gates, quantum gates manipulate qubits in all sorts of ways — guiding them into a succession of superpositions and entanglements and then measuring their output. By mixing and matching gates to form circuits, the theorists could easily assemble quantum algorithms.

In the classical computation, the number of logic gates of n bit are in a finite number, indeed at most 2^{2^n}, as in in Tab. 2.1 (obviously, not all operators are equally interesting); in particular, the unary operators are technically four, but two are trivial (the output is always false or always true, regardless of the input), while the others are identity and negation (NOT). In the quantum case, instead, the possible one-qubit gates are a continuous set. That set is the unitary group U(2) for one qubit, or U(N) for n = log2 N qubits. Hence, a quantum logic gate can be engineered with in principle arbitrary high, but finite accuracy.

Richard Feynman, the physicist who came up with the idea for a quantum computer in the 1980s, quipped that "by golly, it's a wonderful problem, because it doesn't look so easy."

Conceiving algorithms that promised clear computational benefits proved more difficult. Until now, researchers had come up with only a few good candidates. Most famously, in 1994, a young staffer at Bell Laboratories named Peter Shor proposed a quantum algorithm that factors integers exponentially faster than any known classical algorithm — an efficiency that could allow it to crack many popular encryption schemes[Shor, 1994]; anyway, until now the only real implementations are able to work with very small numbers [Monz et al., 2016; Vandersypen et al., 2001].

Two years later, Shor's Bell Labs colleague Lov Grover devised an algorithm that speeds up the classically tedious process of searching through unsorted databases; its algorithm [Grover, 1996] allows a quadratic speed-up for this specific task.

"There were a variety of examples that indicated quantum computing power should be greater than classical," said Richard Jozsa, a quantum information scientist at the University of Cambridge.

The coherence time of the qubit should be larger than the time needed for the algorithm to compute.

Until recently, the pursuit of quantum power was largely an abstract one. "We weren't really concerned with implementing our algorithms because nobody believed that in the reasonable future we'd have a quantum computer to do it," Jozsa said. Running Shor's algorithm for integers large enough to unlock a standard 128-bit encryption key, for instance, would require thousands of qubits — plus probably many thousands more to correct for errors[Chuang and Harrow, 2018; Chuang and Shor, 2018c; Nielsen and Chuang, 2010].

But by 2011, things were starting to look up. That fall, at a conference in Brussels, Preskill speculated that "the day when well-controlled quantum systems can perform tasks surpassing what can be done in the classical world" might not be far off. Recent laboratory results, he said, could soon lead to quantum machines on the order of 100 qubits. Getting them to pull off some "super-classical" feat maybe wasn't out of the question. (Although D-Wave Systems' commercial quantum processors could now boast more than 2,000 qubits, they tackle only specific optimization problems; many experts doubt they can exhibit fully quantum effects and so outperform classical computers.)

"I was just trying to emphasize we were getting close — that we might finally reach a real milestone in human civilization where quantum technology becomes the most powerful information technology that we have," Preskill said. He called this milestone "quantum supremacy." The name — and the optimism — stuck. "It took off to an extent I didn't suspect."

A host of new computer technologies have emerged within the last few years, and quantum computing is arguably the technology requiring the greatest paradigm shift on the part of developers. The intuition behind quantum computing stemmed from what the fact that despite quantum mechanics greatest successes, even some of the simplest systems seemed to be beyond the human ability to model with quantum mechanics, because simulating carefully systems of even a few dozen interacting particles requires more computing power than any conventional computer can provide.

one of the main reasons why quantum mechanics is hard to simulate is because quantum theory says that matter, at a quantum level, is simultaneously in a linear superposition of different possible configurations, known as states. Unlike classical probability theory, these many configurations of the quantum state, which can be potentially observed, may interfere with each other like waves in a tidepool. This interference prevents the use of statistical sampling to obtain the quantum state configurations. Rather, we have to track every possible configuration a quantum system could be in if we want to understand the quantum evolution.

The fondational core of quantum computing is to store information in quantum states of matter and to use quantum gate operations to compute on that information, by harness-

ing and learning to "program" quantum interference. An early example of programming interference to solve a problem thought to be hard on our conventional computers was done by Peter Shor in 1994 for a problem known as factoring. Solving factoring brings with it the ability to break many of our public key cryptosystems underlying the security of e-commerce today, including RSA and Elliptic Curve Cryptography. Since that time, fast and efficient quantum computer algorithms have been developed for many of our hard classical tasks: simulating physical systems in chemistry, physics, and materials science, searching an unordered database, solving systems of linear equations, and machine learning.

Designing a quantum program to harness interference may sound like a daunting challenge, and while it is, many techniques and tools, including our Microsoft Quantum Development Kit, have been introduced to make quantum programming and algorithm development more accessible. There are a handful of basic strategies that can be used to manipulate quantum interference in a way useful for computing, while at the same time not causing the solution to be lost in a tangle of quantum possibilities. Quantum programming is a distinct art from classical programming requiring very different tools to understand and express quantum algorithmic thinking. Indeed, without general tools to aid a quantum developer in tackling the art of quantum programming, quantum algorithmic development is not so easy.

the challenges of quantum information processing; it is integrated in a software stack that enables a quantum algorithm to be compiled down to the primitive operations of a quantum computer. Before approaching the programming language, it's helpful to review the basic principles on which quantum computing is based. We will take the fundamental rules of quantum computing to be axioms, rather than detailing their foundations in quantum mechanics. Additionally, we will assume basic familiarity with linear algebra (vectors, matrices etc). If a deeper study of quantum computing history and principles is desired, we refer you to the reference section containing more information.

Decoherence is nothing but uncontrolled interactions between the system and its environment. The decoherence can lead to a quantum loss in the quantum processor and kills the advantage of a quantum algorithm. To avoid this decoherence time sets a limit on the number of operations that can be performed in our quantum algorithm.

Decoherence can be corrected using error-correction algorithms by encoding quantum state with redundancy over many qubits. But this can be achieved only when the individual quantum gates error rate is very small. Using this a full quantum algorithm that can run longer than the decoherence time can be built single fault-tolerant qubit requires thousands of physical qubits. A study shows that quantum computing can achieve a significant speedup, but this advantage diminished when classical processing required an error-correction scheme to be implemented

Designing higher fidelity qubits is an important hardware challenge as qubits must be considered as embedded in an open environment that requires classical simulation software packages. Quantum RAM unavailability. Complex computation problems like in machine learning or deep learning require huge amounts of data. Currently, we do not have quantum RAM (qRAM) that can efficiently encode this information as a quantum state and store it for a longer time. This is an important hardware challenge for quantum computing. Noisy Intermediate-Scale Quantum (NISQ) processors.

1.3 The main current development environments for quantum computing

Without any claim of exhaustiveness, below is a brief list of what in recent years have been the main initiatives to support the development of quantum computing, quantum

information and quantum machine learning. For each of these development environments, the main prerogatives and initiatives aimed at increasing the community of developers are indicated. All of these frameworks encourage the active participation in the quest for quantum advantage, supporting development of new and innovative software solutions, while the big companies continue to release hardare solutions that are faster, cheaper, or better quality with to respect to previous quantum devices.

1.3.1 IBM Quantum Experience

IBM Quantum Experience is an initiative, provided by the IBM, that offers everyone in the world, after a registration process, the ability to execute programs on quantum computers connected in the cloud. This online platform gives users in the general public access to a set of IBM's prototype quantum computers via the Cloud: currently, IBM offers a fleet of 18 quantum systems; privileged customers have access to premium quantum systems, with up to 53 qubits or up to 32 "quantum volume", while everyone else can take advantage of a simulator via the cloud, ibmq_qasm_simulator v0.1.547, which supports up to 32 qubits and a varied repertoire of basic gates (u1, u2, u3, cx, cz, id, x, y, z, h, s, sdg, t, tdg, ccx, swap, unitary, initialize, kraus), and a set of open quantum systems, i.e. a single qubit quantum processor (ibmq_armonk v1.1.0, online since October 16, 2019, with id, u1, u2, u3 basic gates and 1.136×10^{-3} single-qubit U2 error rate) and several multi-qubits quantum computers, as in Fig. 1.20, all available for free, trough a "fairshare" run mode. Until today, all quantum systems deployed by IBM Quantum are based on superconducting transmon qubit technology, using special cryogenic refrigerator, as IBM researchers think that this technology offers the control and scalability to pave a valid path to achieving better quantum systems.

Nowadays, the bigger IBM premium quantum system is locate at Rochester, Minnesota, in the facility that was designed by Saarinen: just the fact that this important building is covered with blue panels of various shades, under the inspiration of the sky, is at the origin of the nickname of "Big Blue" for multinational colossus of IT world. This quantum computer has a record breaking 53 qubits as shown in Fig. 1.21b. Since 2016, when it made the world's first quantum computer available through the IBM Cloud, IBM Quantum Experience has remained the premier place for researchers, industry professionals, developers, and students to access cutting edge quantum hardware.

Inside IBM Quantum Experience, developers can interact with a quantum processor through the quantum circuit model of computation, applying quantum gates on the qubits using a GUI called the quantum composer, writing quantum semi-assembly language code (QASM) or writing high-level code, usually in Python, to cooperate with Qiskit libraries.

When developing a circuit to solve a class of problems, one imagines being able to exploit arbitrary connections between the different qubits if necessary: indeed, various simulators support all-to-all connectivity; however, to date, each quantum computer is characterized by a specific coupling map. The coupling map is a sort of list that indicate those pairs of qubits that support two-qubit gate operations between them, to describe the device topology or connectivity. The coupling map can be viewed visually, with qubits represented as circles, and the supported two-qubit gate operations displayed as lines connecting the qubits, as in Fig. 1.20.

Another important parameter for real quantum computer is the maximum number of shots, i.e. the maximum number of times a single circuit can be executed on a hardware computer backend. The number of shots taken determines the precision of the output probability distribution over repeated executions. In Fig. 1.22, it is shown that the average error of a quantum computer increases with the duration of the computation, so a maximum number of iterations must be set to stem this difficulty. Indeed, the maximum number of shots (times) you can execute a single circuit on a IBM Quantum

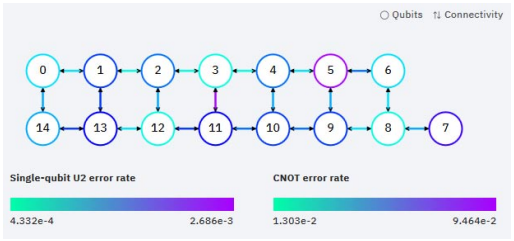
computer is usually 8192 (2^13). The number of shots taken determines the precision of the output probability distribution over repeated executions. By virtue of the intrinsically probabilistic nature of quantum computations, it is not possible to rely on the result of a single execution, or a limited number of executions, of a quantum circuit. The presence of errors due to decoherence and other quantum noise makes it rather preferable to repeat the execution of each circuit a large number of times, in order to be able to average the outcomes of the individual occurrences and infer a more reliable result. In theory, therefore, the best results should be obtained launching each circuit multiple times, a single shot at time, between two calibrations phases; however, for the same total number of executions, launching a circuit several times, reducing the number of shots for each occurrence, involves a dramatically significant dilation of execution times and significantly increases the risk of incurring failures due to exceeding the time-out or the onset of connection problems (at least as regards free access to simpler quantum computers).

1.3.2 Rigetti

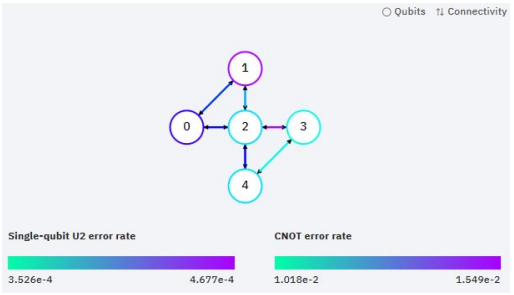
Rigetti Computing was a startup founded in 2013 by Chad Rigetti, a quantum computing physicist who take a bachelor's degree in physics from the University of Regina and a Ph.D. in applied physics from Yale. He worked in the quantum computing group at IBM before he decided to start an integrated systems company in Berkeley, California. Rigetti Computing is a full-stack quantum computing company: it designs and fabricates superconducting quantum processors, integrates them with a controlling architecture to build quantum computers, and develops software for programmers to use to build algorithms for the chips. In 2016, the company had begun testing a three-qubit chip made using aluminum circuits on a silicon wafer. In 2017, the company produced eight-qubit computers [Zeng et al., 2017], then it announced the public beta availability of a quantum cloud computing platform, called Forest, which allows developers to write quantum algorithms and the introduction of a quantum processor consisting of 20 superconducting transmon qubits with fixed capacitive coupling as shown in Fig. 1.25; due to a fabrication defect, qubit 3 is not tunable, consequently, Rigetti treat this as a 19-qubit processor, calling it Rigetti 19Q [Otterbach et al., 2017].

Rigetti has chosen for its system a quantum-classical hybrid architecture, based on co-location of a classic CPU with the quantum processing unit, to take advantage of the quantum chip as an accelerator for specific tasks while eliminating Internet transmission latency, using Quantum Cloud Service (QCS), as shown in Fig. 1.27. QCS use the QPUs Aspen (Quantum Processing Units) Aspen architecture 1.28b. The initial versions of Aspen used for public beta of QCS were based on a 16 qubit configuration, then evolved til 31 qubis 1.28a, while Rigetti announced that the Aspen family will extend to 128 qubits[10]. The QCS system is unique because each user receives a dedicated QMI (Quantum Machine Image), preconfigured with Rigetti's Forest SDK. Rigetti introduced some technologies such as parametric compilation, to allow to run the same quantum program with different parameters at run time, and an active qubit reset capability, to allow rapid execution of programs by eliminating any delays that would result in a return to the zero state of the qubit; according to Rigetti, these features together can provide up to 30x performance increase over a plain web API model. Through Quantum Cloud Services (QCS) platform, their machines can be integrated into any public, private or hybrid cloud. On Dec. 2, 2019 Rigetti announced a commercial agreement with Amazon, to offer its quantum computers through a fully managed Amazon Web Services (AWS) solution, named Amazon Braket, so to allow scientists, researchers, and developers to

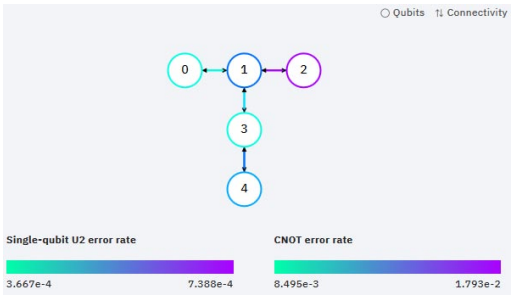
[10] https://medium.com/rigetti/the-rigetti-128-qubit-chip-and-what-it-means-for-quantum-df757d1b71ea



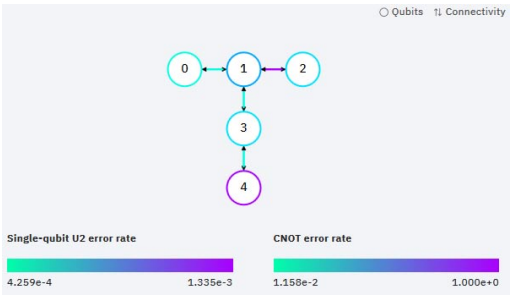
(a) ibmq_16_melbourne v2.1.0: online since November 06, 2018, it offers 16 Qubits and id, u1, u2, u3, cx as basis gates.



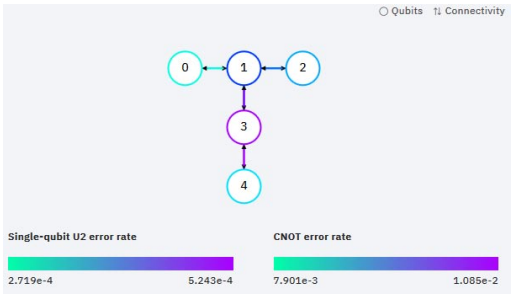
(b) ibmq_5_yorktown - ibmqx2 v2.0.5: online since January 24, 2017, it offers 5 Qubits and u1, u2, u3, cx, id as basis gates.



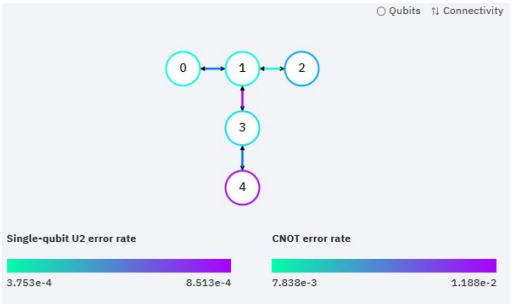
(c) ibmq_essex v1.0.1: online since September 13, 2019, it offers 5 Qubits and u1, u2, u3, cx, id as basis gates.



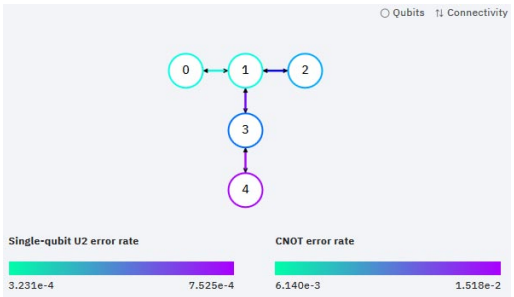
(d) ibmq_burlington v1.1.4: online since September 13, 2019, it offers 5 Qubits and u1, u2, u3, cx, id as basis gates.



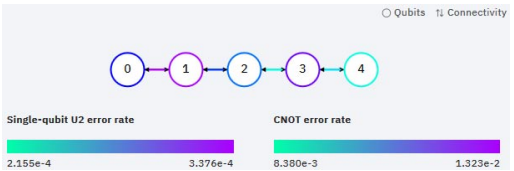
(e) ibmq_london v1.1.0: online since September 13, 2019, it offers 5 Qubits and u1, u2, u3, cx, id as basis gates.



(f) ibmq_vigo v1.0.2: online since July 03, 2019, it offers 5 Qubits and u1, u2, u3, cx, id as basis gates.

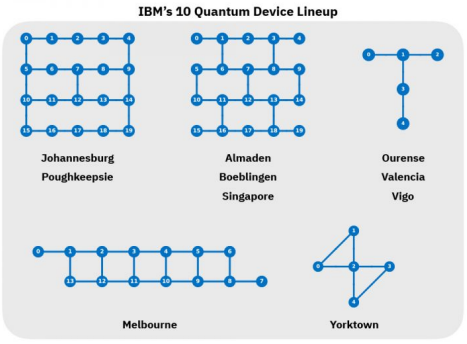


(g) ibmq_ourense v1.0.1: online since July 03, 2019, it offers 5 Qubits and u1, u2, u3, cx, id as basis gates.

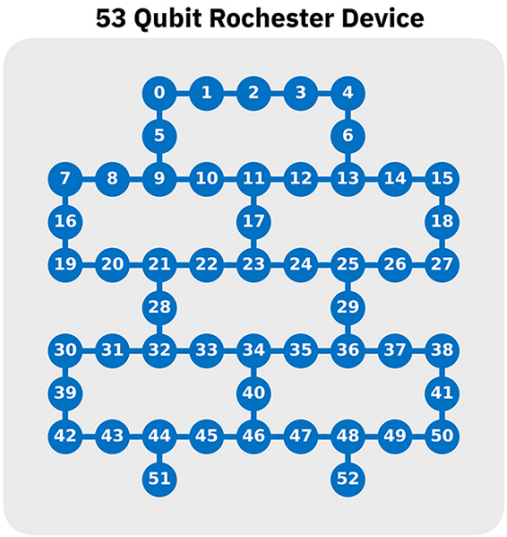


(h) ibmq_rome v1.0.0: online since March 23, 2020, it offers 5 Qubits and u1, u2, u3, cx, id as basis gates.

Figure 1.20: IBM multi-qubits quantum computers with free cloud access[6] (2020).



(a) IBM Quantum Systems: topologies for 2019 computers



(b) IBM Quantum Experience: 53 Qubit Rochester's topology

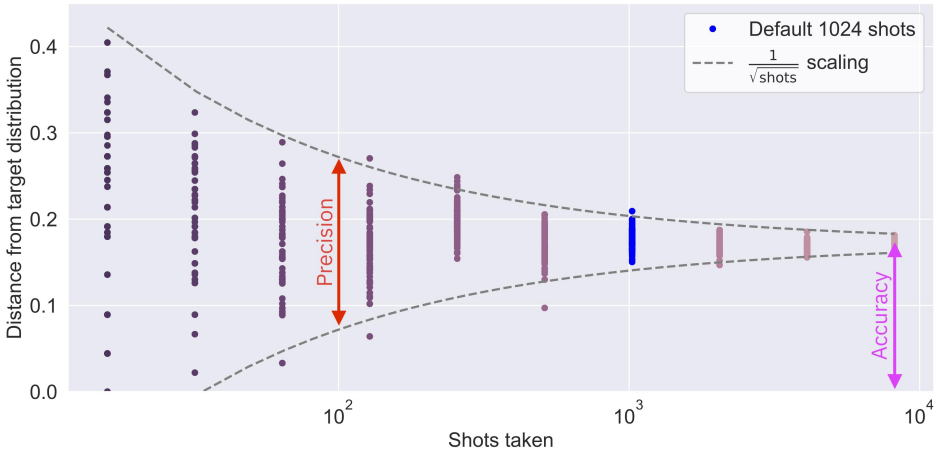


Figure 1.22: IBM Quantum computing sampling error: distance for a Bell state run on the IBM Quantum Boeblingen system from the theoretical answer (in terms of Hellinger distance) as a function of the number of shots taken.

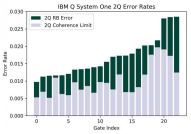


Figure 1.23: IBM Q System One two-quibts Error Rates[7]

Figure 1.24: IBM Quantum systems: fundamental metrics in four recent IBM Q systems[8]

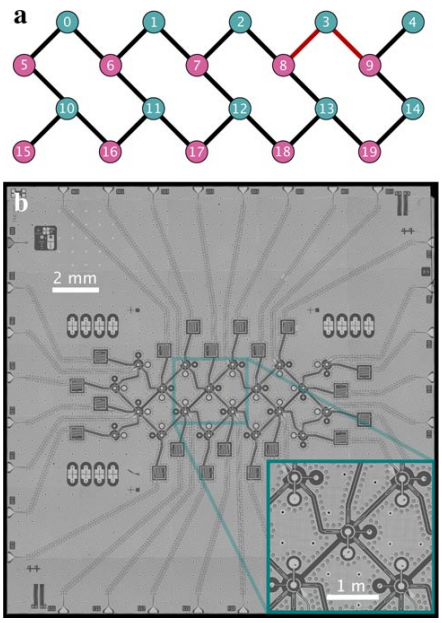


Figure 1.25: Rigetti 19Q quantum processor: a, Chip schematic showing tunable transmons (teal circles) capacitively coupled to
xed-frequency transmons (pink circles). b, Optical chip image. Note that some couplers have been dropped to produce a lattice with three-fold, rather than four-fold, connectivity [Otterbach et al., 2017].

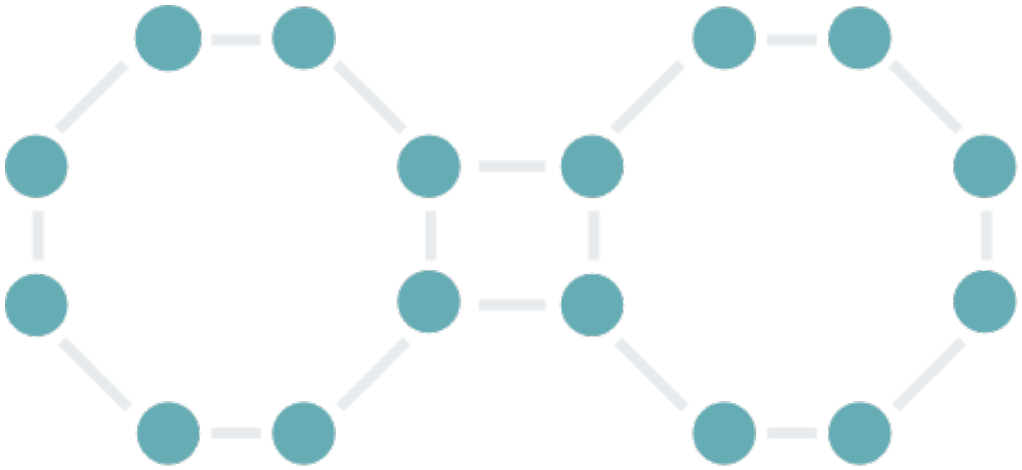


Figure 1.26: Rigetti developed a scalable 16-qubit form factor to pave the way to 128-qubit chip[9].

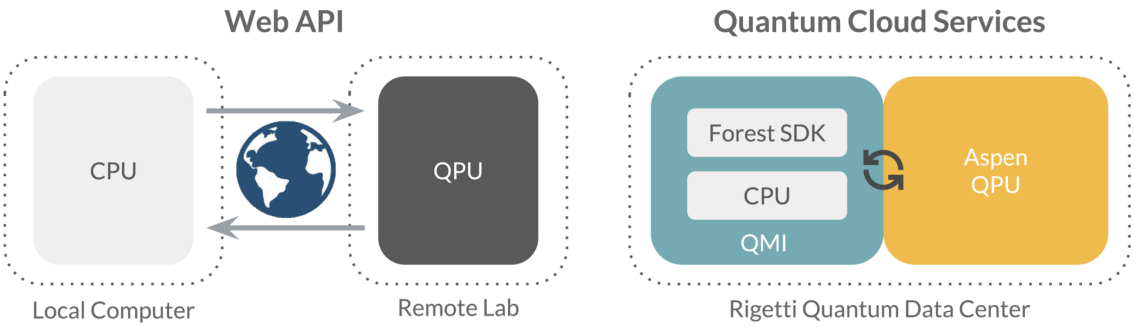


Figure 1.27: Rigetti Quantum Cloud Service[11]

begin experimenting with computers from quantum hardware providers in a single place. To boost enterprise customers relationships, Rigetti has also joined the AWS Partner Network (APN) as a solutions provider, to develop custom software solutions focused on simulation, optimization and machine learning for industry-leading organizations in finance, insurance, pharmaceuticals, defense, and energy.

Rigetti developed a quantum instruction language called Quil [Smith et al., 2017], which was the first to introduce a shared quantum-classical memory model. Quil supports quantum circuit model abstraction model and the superconducting quantum processors developed by Rigetti Computing through the Forest quantum programming API. The language also supports macro-like definitions of possibly parametrized quantum circuits and their expansion, qubit measurement and recording of the outcome in classical memory, synchronization with classical computers with the WAIT instruction which pauses the execution of a Quil program until a classical program has ended its execution, conditional and unconditional branching, pragma support, as well as inclusion of files for use as libraries (a standard set of gates is provided as one of the libraries.) A Python library called pyQuil was introduced to develop Quil programs with higher level constructs. Rigetti Computing developed a Quantum Virtual Machine, an abstract representation of a general-purpose quantum computing device, in Common Lisp that simulates the quantum machine on a classical computer and is capable of the parsing and execution of Quil programs with possibly remote execution via HTTP. The Quantum Abstract Machine (QAM) includes support for manipulating both classical and quantum state. As shown in Fig. 1.26, Rigetti developed a scalable 16-qubit form factor to pave the way to 128-qubit chip,

Aspen-8		Avg. Time Duration (μs)		Avg. Fidelity (per op.)	
Deployed	05.20.20	T1 Lifetime	29	Single-qubit gates	99.79%
Qubits	31	T2 Lifetime	18	Two-qubit gates	95.66%

(a) Rigetti Aspen 8

Aspen-7	Aspen-4	Agave	Acorn
Deployed 11.15.19	Deployed 03.10.19	Deployed 06.04.18	Deployed 12.17.17
Qubits 28	Qubits 13	Qubits 8	Qubits 19
Avg T1 41μs	Avg T1 30.47μs	Avg T1 13.38μs	Avg T1 20.3μs
Avg T2 35μs	Avg T2 20.13μs	Avg T2 15.05μs	Avg T2 10.9μs
Avg 1Q 99.23%	Avg 1Q 0.9988	Avg 1Q 0.96	Avg 1Q 0.9863
Avg 2Q 95.2%	Avg 2Q 0.9442	Avg 2Q 0.87	Avg 2Q 0.875
Avg 1Q gate time 80 ns	Avg 1Q gate time 60ns	Avg 1Q gate time 50ns	Avg 1Q gate time 50ns
Avg 2Q gate time 340 ns	Avg 2Q gate time 276ns	Avg 2Q gate time 169ns	Avg 2Q gate time 213ns

(b) Rigetti Aspen evolution

(c) Rigetti Quantum Processors[a]

[a] https://rigetti.com/

Chapter 2

Machine Learning

Machine learning is a very important subset of artificial intelligence (AI) research field. Machine Learning techniques are used in various research sectors, where it is necessary for a system to automatically learn a task or improve performance through experience, based on the data it can evaluate; with such techniques, a system can know how to recognize data or perform a task without having been explicitly programmed for the task to be performed. Nowadays machine learning is one of most rapidly growing technical fields, lying at the intersection of computer science and inferential statistics, and it consists of various models and techniques. Recent progress in machine learning has been driven both by the development of new learning algorithms and theory and by the ongoing explosion in the availability of online, often open access data-sets and relatively low-cost computational resources. The adoption of data-intensive machine-learning methods can be found throughout science, technology and even commerce, leading to more evidence-based decision-making across many walks of life, including health care, manufacturing, education, financial modeling, policing, and marketing. For example, when we interact with medical diagnostic centers and banks, buy at online stores or use social networks, machine learning algorithms are used to make our experience efficient, easy and safe (or, conversely, to deeper influence our choices).

2.1 Classical Artificial Intelligence

The seeds of modern AI were planted by classical philosophers who attempted to describe the process of human thinking as the syntactic, mechanical manipulation of symbols. This work culminated in the invention of the programmable digital computer in the 1940s, a machine based on the abstract essence of mathematical reasoning and syntactic manipulation of symbols. In the 1940s and 50s, digital computer and the ideas behind it inspired a handful of scientists from a variety of fields (mathematics, psychology, engineering, economics and political science) to begin seriously discussing the possibility of building a sort of artificial, probably electronic, brain. In the early 1950s, there was a variety of conceptual orientations for the field of the so called "thinking machines", particularly cybernetics, automata theory, and complex information processing. In particular, the research in neurology had shown that the brain was an electrical network of neurons that fired in all-or-nothing pulses, while Alan Turing's theory of computation suggested that any form of computation could be described by Turing's machines and implemented in digital computer.

In this period, Cybernetics emerged as a trans-disciplinary approach, comprising mechanical, physical, biological, cognitive, and social sciences, for study how humans, animals and machines control and communicate with each other, indeed Norbert Wiener defined cybernetics in 1948 as "the scientific study of control and communication in the

animal and the machine". Cybernetics is relevant to explore regulatory systems: their structures, constraints, and possibilities, in particular where action by the analysed system generates some change in its environment and that change is reflected in the system in some manner (feedback) that triggers a system change. The essential goal of the broad field of cybernetics is to understand and define the functions and processes of systems that have goals and that participate in complex, usually circular causal chains that move from action to sensing to comparison with desired goal, and again to action. Its focus is how anything (digital, mechanical or biological) processes information, reacts to information, and changes or can be changed to better accomplish the first two tasks. Therefore Cybernetics deals especially with learning, cognition, adaptation, social control, emergence, convergence, communication, efficiency, efficacy, and connectivity.

According to many experts, the field of AI research was founded, or at least acknowledged as an academic discipline, at a workshop held on the campus of Dartmouth College during the summer of 1956, which was essentially a sort of brainstorming session, according to the following proposal:
"We propose that a 2-month, 10-man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer."

Those who attended would become the leaders of AI research for decades. Many of them boldly predicted that a machine as intelligent as a human being would exist in no more than a generation and so they were given millions of dollars to make this vision come true. In retrospect, this pioneering phase has often been described as strong AI and it had set very high expectations, such as passing the famous Turing test: a machine and a human both converse sight unseen with a second human, who must evaluate which of the two is the machine, which passes the test if it can fool the evaluator a significant fraction of the time.

In the 1970s, AI was subject to critiques and financial setbacks. Some AI researchers seemed to understimate the difficulty of the problems they faced: their optimism had raised exaggeratedly high expectations, and when the promised results failed to realize, funding for AI projects were dramatically cut. Because of combinatorial explosion of paths to be explored to reach a solution, for many interesting problems there was not enough memory or processing speed to accomplish useful results. Moreover, many important artificial intelligence applications like vision or natural language processing require simply enormous amounts of information about the world: until relatively few years ago, researchers could not build a database so large and no one knew how a program might learn so much information.

Some sectors developed within AI are Knowledge Base, Symbolic Calculus, Logical Inference, Logical Reasoning Systems, Computer vision, Control systems, Conversation theory, Interactions of actors theory, Learning organization, Robotics, Cellular automaton, Decision support systems, Formal languages, Modal logic, Adaptive systems. Sometimes, symbolic AI, i.e. based on high-level symbolic representations of problems, is called "Good Old-Fashioned Artificial Intelligence" (GOFAI).

Usually, classical AI was based on the union of ad hoc studies for each class of problems, trying to build on abstract knowledge of each examined instance and to take into account the opinion of human experts, and uniform strategies, such as A* and backtrack algorithms.

For example, there are problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. Conceptually, in such cases the partial candidates can be represented as the nodes of a tree structure, i.e. the potential search tree. Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further. We can use breadth-first or depth-first (such as backtracking algorithm) approaches to find one solution: the breadth-first approach guarantees the identification of a solution, but usually requires a very high memory use, while the depth-first approach risks being trapped in a branch of infinite length, therefore a maximum exploration depth of the configuration tree is usually contemplated, so to force to evaluate even non-leaf nodes (usually through ad hoc heuristic functions). Anyway, for any problem that can be seen as combinatorial optimization task we can use this kind of algorithms. As another example, for two-player zero-sum game theory we can use minimax evaluation or alpha–beta pruning to find a solution in which each player minimizes the maximum payoff possible for the other. Chess and checkers are typical examples of games for which it is possible to develop strategies using the minimax algorithm; the alpha–beta pruning algorithm allows to discard entire branches that certainly do not contain better candidate solutions than the provisionally identified one, allowing to increase the maximum search depth.

A classical example is the n-queens puzzle: on a standard 8×8 chessboard, the "basic" eight queens puzzle is the problem of placing eight chess queens so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal, but the problem can be generalized to the case of $n \times n$ chessboards, for any natural number n. If n equals 1, there is obviously only a trivial solution, for n equals 2 or 3 there are no solutions, for n equals 4 there are two solutions, but each of them it is obtained from the other by reflection, so there is only a "fundamental" solution. There is no known formula for the exact number of solutions for each natural number n, or even for its asymptotic behaviour, but for n greater than 6 the number of solutions is growing very fast. The problem of finding all solutions to the n-queens problem can be quite computationally expensive, as there are $\binom{n^2}{n}$ possible arrangements of n queens on an $n \times n$ board; by applying a simple rule that constrains each queen to a single column (or row),it is possible to reduce the number of possibilities to n^2 possible combinations; generating permutations further reduces the possibilities to just $n!$ possibilities, which are then checked for diagonal attacks. As the factorial grows very rapidly, even taking into account symmetries, such as rotations of multiples of a right angle or reflection, the number of possible configurations to be tested is too high to proceed with a breadth-first exploration, therefore making search strategies preferable based on back-tracking.

Maybe one of the most famous, or at least spectacular, successes of classic artificial intelligence was when on 11 May 1997, IBM Deep Blue became the first computer chess-playing system to beat a reigning world chess champion, Garry Kasparov, the highest rated chess player in history (at the time), the World Champion for 15 years (1985–2000). Deep Blue was a supercomputer developed by IBM specifically for playing chess: its development started in 1985 as the ChipTest research project in Carnegie Mellon University led by Feng-hsiung Hsu. It eventually evolved into Deep Thought and it was later renamed to Deep Blue in 1989; even Joel Benjamin, a chess grandmaster, was added to the development team. It consisted of 30 nodes, with 120 MHz P2SC microprocessor (Power2 Single Chip) and, as coprocessors, 480 special-purpose VLSI chess chips per node, so it could elaborate an average of 200,000,000 moves per second. According to the TOP500 list, Deep Blue was the 259th most powerful supercomputer in 1997: it achieved 11.38 GFLOPS on the LINPACK benchmark. It used mostly brute-force strategies to overcome human champion.

2.1.1 Some considerations about Classical A.I. computational requests

If we examine the algorithms developed within classical artificial intelligence, we can identify some recurring traits: there is a predominance of symbolic processing and calculation with integer numbers; any computation with machine representations of decimal numbers is usually relegated to particular aspects (for example, the assignment of a score by an evaluation function); because there are many evaluations of conditional expressions, the code is usually really branch intensive; because there are complex data dependencies, a relative low Instruction Per Clock (IPC) is usually found even on complex, superscalar microarchitectures; there are often many jumps for functions' recursive calls; not regular data access make cache memory hierarchy and automatic, in hardware, data prefetching techniques quite ineffective; a high sensitivity to memory latency is revealed; moreover, this kind of algorithms not always is very thread friendly.

The adducted elements demonstrate how complicated it is for a microprocessor to efficiently perform artificial intelligence's classical algorithms.

For example, the high density of jump instructions, by virtue of conditional statements or calls to recursive functions, clashes with the high lengths of modern pipelines; moreover, unlike what usually happens with iterative structures, such as controlled loops, in which a jump usually follows a path many times, until the entry condition is falsified (therefore it is correctly predicted always or almost by a good branch prediction unit), in these circumstances even the best branch prediction units are thrown into crisis. Moreover, complex data dependencies tend to limit the IPC exploitation and the effectiveness of out-of-order execution engines, while they do not facilitate the use of SIMD or vector units; complex, difficult to predict memory access pattern limit the effectiveness of the cache memories and TLB, with frequent flushes, moreover put the hardware data prefetch units in difficulty (nowadays, access to the RAM memory can cost some hundreds of work cycles).

Ultimately, the most widely implemented techniques in modern processors to increase performance in the most frequently used programs, rarely cooperate effectively with most classical artificial intelligence algorithms.

2.1.2 Classical A.I. trouble: an example, working memory latency

In many classical A.I. algorithms, the load to use latency for access to information in working memory plays a crucial role, therefore a simple explanation of the access to the memory is provided. Generally a memory array is divided into banks or pages, than each block is addressed by rows and columns, so that each memory access consists of several stages. Memory speed can be characterized on the basis of two fundamental parameters: latency and bandwidth. Latency indicates how much time elapses between the need for information and its effective availability, while bandwidth expresses the maximum amount of information that can be transferred per unit of time, at least in the most favorable conditions (primarily sequential access, but even circumstances in which a pipelined access can conveniently mask the transfer latency of addresses for subsequent accesses). Even in the most favorable case, the burden of specifying the column containing the desired information block must be assumed at least, therefore the CAS (Column Access Strobe) latency plays a crucial role in determining the minimum latency of a memory access. Spatial locality principle dictate that if a particular memory location is referenced once, then it is likely that nearby memory locations will be referenced in the near future, so it is common to access several memory blocks in the same memory row (thus having to pay only once the duty in the form of RAS - Row Access Strobe - latency per multiple memory accesses): in such circumstances, the CAS latency alone determines the elapsed time (from the perspective of the memory module, at least). For over 50 years, JEDEC

(Joint Electron Device Engineering Council), which has over 300 members, including some of the world's largest computer companies, has been a global leader in developing open standards and publications for the microelectronics industry; its committees provide industry leadership in developing open standards for a broad range of technologies, which are the basic building blocks of the digital economy and form the bedrock on which healthy, high-volume markets are built. JEDEC play a hegemonic role especially for memory chip, modules and interfaces specifications, so as to develop several generations of SDRAM (Synchronous Dynamic Random Access Memory) memories, defining their operating parameters.

While the CPU speed has grown for long periods with an exponential rate, the working memory speed has grown linearly at most1.19. Furthermore, over the past 25 years, while RAM memory bandwidth has grown a couple of orders of magnitude, memory access latency has declined very little.

For example, in 1996 a Pentium or Pentium PRO 200 MHz had a working cycle of 5 ns, while the SDRAMs PC66 offered a CAS latency of 2 cycles at 66 MHz, corresponding therefore to 30 ns. As a result, the CAS latency corresponded to just 6 microprocessor cycles. Nowadays, a Core i9-9900 CPU offers a maximum frequency (Turbo Boost) of 5 GHz, therefore its work cycle lasts merely 0.2 ns, while it officially supports DDR4 memories up to 2666MHz. RAM DIMMs of this frequency, compliant with JEDEC (Joint Electron Device Engineering Council) standards, offer the minimum CAS latency of 17 cycles, or 12.75 ns, which means that the CAS latency corresponds to 63.75 CPU cycles. This example shows that the CAS latency perceived by the CPU has worsened by an order of magnitude (in terms of CPU clock cycles), while the overall bandwidth has increased by a factor of 80 (from 533 MB/s to 42.67 GB/s), with the average bandwidth for cycle increased by a factor of 3.2 (from 2.67 to 8.53 bytes/cycle).

Even violating the Intel specifications for this CPU, but respecting JEDEC's standard for memory, it is possible to adopt DDR4 3200 with CL 20, so despite a further 20% increase in bandwidth there is only an imperceptible reduction in CAS latency, i.e. at 12.50 ns (exactly the same absolute CAS latency value in nanoseconds offered in 2003 by the fastest memory compliant with JEDEC's standard, DDR-400 CL 2.5), therefore 62.5 CPU cycles (but only 25 CPU cycles for an AMD Athlon K8 Model Number 3200+ working at 2 GHz, one of top speed CPU in 2003).

By using expensive memories that circumvent the JEDEC specifications, it is possible to further reduce access latency; for example, with DDR4-2400 CL 12, DDR4-2800 CL 14, DDR4-3000 CL 15 or DDR4-3200 CL 16 the CAS latency is 10 ns (DDR4-2666 CL13 offers 2.5% lower latency, at 9.75 ns), exactly at the same level reached by the DDR-400 CL2, the fastest (but not strictly standard) memory in the year 2003. By opting for even more expensive devices (and agreeing to severely limit the maximum memory capacity), it is possible to go up to DDR4-4266 CL 17 (7.97 ns) or DDR4-4800 CL 18 (7.50 ns), that correspond to 39-40 CPU cycles at 5GHz. Over a period of 17 years therefore we cannot measure a latency reduction for standard JEDEC memory, but even regarding the best memories that do not comply with the JEDEC standards, we can point out that we are observing a latency reduction of just 20-25% , in the face of lower market availability, higher relative cost and more narrow capacity constraints.

It is also necessary to point out that, as the number of cores offered by modern CPUs increases, the connection topology becomes more complex and articulated (for example, multiple ring buses, multiple CCXs, i.e. core complexes, uniform mesh topology inside a single die, but there are even complex inter-die connections for multi-die solutions), the arbiter stage to memory access become more complicated and computationally expensive as well as the protocols aimed at ensuring the consistency of the information (MESI/-MOESI protocols, cache snooping, cache filter and so on). The resulting effect of these

important changes is that Intel CPUs with a low number of computational cores (Kaby Lake and Coffee Lake, with 4-6 cores) at most reach, when connected to memories that exceed JEDEC standards, the overall latency for memory access shown by the AMD Athlon 64 in 2003 (the first X86 CPU with memory controller and northbridge integrated in its die), i.e. approximately 40-45 ns (same latency, but with an effective bandwidth about 20 times higher, due to the transition from a single DDR memory channel at 400 MHz to a double DDR4 memory channel at around 4GHz). Even AMD Renoir, the Ryzen Mobile 4000 Series 7nm APU, with a monolithic design (so far the only "Zen 2" product not to use a multi-chip module package with IOD, a dedicated die for I/O), has a load to use memory latency of about 65 ns. All CPUs with a higher number of computational cores (such as Intel Haswell-E, Broadwell-E, Skylake-X and Cascade Lake-X or AMD Ryzen and ThreadRipper) demonstrate ineluctably higher memory access latency, sometimes even 2-3 times worse. Therefore, the most advanced processors, with the higher computing power, actually observe a worsening of latency compared to 17 years ago.

While JEDEC still has not published the DDR5 specification officially, since 2018 there are drafts of this new technology by JEDEC working groups and some provisional DDR5 IP (Intellectual Property), such as controllers and PHYs, have since been released commercially. At least on the server side, it is widely known that AMD's EPYC "Genoa" (based on Zen 4 microarchitecture, supposedly on 5nm litographic process) as well as Intel's Xeon Scalable "Sapphire Rapids" (based on Willow Cove microarchitecture, successor to the Ice Lake's Sunny Cove core: it should be the second refinement of the 10 nm process, which is supposed to include new security features and the cache subsystem redesign) will support DDR5 DRAM when they launch in the 2021 2022 time frame. Transition to DDR5 represents a major challenge for DRAM makers because the chips are set to simultaneously increase capacity, rise data transfer rates, increase effective performance (per clock and per channel), and lower power consumption all at the same time. DDR5 is indeed expected to bring in I/O speeds of 4800 to 6400 MT/s (Mega-Transfers per second), with a supply voltage drop to 1.1 Volt and 16-32 Gbit density; in addition, DDR5 is expected to make it easier to stack multiple DRAM devices, which will allow to further increase DRAM capacity, at least on the server side.

Industry sources expect that DDR5 ramp will begin with 16 Gb DRAMs at 4800 MT/s and that, from there, DDR5 will evolve in two directions: capacity and performance. Capacity wise, DDR5 will quickly grow from 16 Gb to 24 Gb and then to 32 Gb capacity per chip. As for performance, it is expected that DDR5 will evolve to 5200 MT/s data rate in 12 – 18 months after DDR5-4800 launch and then to 5600 MT/s in another 12 – 18 months, so performance progress of DDR5 will occur in a pretty much regular cadence, until the expected milestone of 6400 MT/s, i.e. double what is officially achieved by DDR4 almost a decade after its introduction. It is also expected that eache 64 bit DIMM will use two independent 32-bit channels per module (or 40 bit channels with ECC). Furthermore, DDR5 will have an improved command bus efficiency (because the channels will have their own 7-bit Address (Add)/Command (Cmd) buses), better refresh schemes, and an increased bank group for additional performance. In fact, Cadence Design Systems, one of the largest multinational companies involved in electronic design automation (EDA) software and engineering services, goes as far as announcing that improved functionality of DDR5 will enable a 36% higher real-world bandwidth when compared to DDR4 even at the same 3200 MT/s (obviously this bold claim will have to be put to a test).

However, the relevant fact is that none of the committees or companies involved into developing new RAM memory standards, such as JEDEC, Cadence, RAM memory manufacturers (Micron Technology, Samsung Semiconductor and SK Hynix), microprocessor manufacturers (Intel, AMD, ARM), venture to name a slight reduction in access latency as a goal to be pursued or at least a sort of collateral benefit of this new technology.

In general it is possible to profitably mask memory latency by exploiting locality of reference principle, with the introduction of one or more levels of cache memory; moreover, in a certain sense it is possible to buy latency by spending bandwidth, in the sense of being able to exploit any fraction of bandwidth that is not directly exploited by the software in use to preload information in one of the cache levels that have a high probability of being used in the near future, thus taking advantage of the hardware prefetching units present in the CPUs released in the last twenty years and constantly improved.

Anyway, for many "classic" artificial intelligence algorithms the assumptions of temporal locality and spatial locality are not always observed, therefore greater emphasis is reserved to the worst case scenario (full working memory, i.e. RAM, access). For example, rather than the simple traversal of elements in a one-dimensional array (or row/column scan in two-dimensional array, depending on whether it is stored in row major or column major order), from the base address to the highest element, exploiting the sequential locality of the array in memory, many A.I. algorithms use more sophisticated data structures, such as trees, hash tables and tries whose elements are scattered everywhere in memory, incurring often even in very expensive page table walk to react to TLB miss.

2.2 Machine Learning

The theoretical study and computer modeling of automatic learning processes in their multiple manifestations constitutes the subject matter of machine learning, one of most rapidly growing technical fields. The field of machine learning is organized around three primary research areas: the development and analysis of automatic learning systems to improve performance in a predetermined set of tasks, i.e. task-oriented studies; the investigation and computer simulation of vertebrates, mammals or human learning processes, i.e. cognitive simulation; the theoretical exploration of the space of possible learning methods and algorithms independent of application domain, i.e. theoretical analysis. An equally basic scientific objective of machine learning is the exploration of alternative learning mechanisms, including the discovery of different induction algorithms, the scope and limitations of certain methods, the information that must be available to the learner, the issue of coping with imperfect training data, and the creation of general techniques applicable in many task domains.

Machine learning algorithms can figure out how to perform important tasks by generalizing from examples. This is often feasible and cost-effective where manual programming is not. As more data becomes available, more ambitious problems can be tackled. As a result, machine learning is widely used in computer science and other fields.

Often the original data (raw data) cannot be used by the classifier to build a model, but an additional pre-processing is required which transforms the raw data to so called feature vectors which better describe the data, e.g., mean values and standard deviations, frequency power spectra, and amplitudes after a low or high pass filtering stage. When dealing with classification tasks of complex data, the generation of meaningful features is a major issue. This is due to the fact that the data often consists of a superposition of a multitude of signals, together with dynamic and observational noise. Hence, the data processing usually requires the combination of different pre-processing steps in addition to a classifier. In fact, the generation of good features plays a fundamental rule both to increase the goodness of the results of the training process both to converge towards an acceptable solution more quickly, so the choice of pre-processing is often at least as important as the selection of actual classification algorithm. Sometimes, for high-dimensional data, pre-processing includes some dimension reduction technique, performed prior to applying the learning algorithm, in order to avoid the effects of the so called curse of dimensionality, for example, PCA (Principal Component Analysis).

In the early 1900, Pearson was the PCA's artificer, then it was later independently invented and named by Harold Hotelling. PCA is mostly used as a tool in exploratory data analysis and it is very useful to better view complex datasets. Intuitively, if you have k experimental points in an n-dimensional space, but you have no a priori knowledge on their arrangement, the simplest operation is to evaluate a "best fit" line that passes through these experimental points, usually the one that minimizes the average squared distance from a point to the line. If the data followed an almost linear trend, this line would adequately represent them; in general, this assumption is obviously not satisfied, but the procedure can be repeated choosing the next best-fitting line in the orthogonal subspace to the first one. Accordingly PCA is conceived as an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by some scalar projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate (second principal component), and so on. To center the data around the origin, the mean of each variable is subtracted from the dataset: this is an extremely important step so that it is possible to identify the directions according to which the variability of the points is maximum, that is the dispersion with respect to a sort of center of mass. Anyway, PCA analysis is sensitive to the scaling of the data: theoretically it cannot be established in general how to best scale the data to obtain optimal results, usually each variable's variance is normalized (equal to 1). PCA can be done by eigenvalue decomposition or singular value decomposition of a data matrix, to speed up learning phase

From the most abstract possible viewpoint, machine learning can be summarized as learning a function (f) that maps input variables (X) to output variables (Y): $Y = f(x)$.

An algorithm learns this target mapping function from training data.

The form of the function is unknown, so we need to evaluate different machine learning algorithms and find which is better at approximating the underlying function.

Different algorithms make different assumptions or biases about the form of the function and how it can be learned.

Assumptions can greatly simplify the learning process, but can also limit what can be learned. Algorithms that simplify the function to a known form are called parametric machine learning algorithms: in this case, the learning model summarizes data with a set of parameters of fixed size (independent of the number of training examples). In this case, the model is generated with a combination of a scheme chosen a priori (number and type of elementary functions, for example) and a set of adaptive parameters, that is parameters are initialized with typical or random values and then changed according to an optimization procedure. For example, if we expect the data to follow a linear or parabolic trend, we can try to determinate the coefficients of, respectively, a first or second grade polynomial, according to some criterion (for example, maximum likelihood, i.e. least square technique). Often the assumed functional form is a linear combination of the input variables and as such parametric machine learning algorithms are often also called "linear machine learning algorithms"; then, applying a nonlinear output function could increase model flexibility. Some examples of parametric models are: logistic regression, LDA (Linear Discriminant Analysis), Perceptron, Naive Bayes, and even simple, fixed-size, neural networks.

Algorithms that do not make strong assumptions about the form of the mapping function are called non-parametric machine learning algorithms. Non-parametric methods are good when there have a lot of data and no prior knowledge, and when you don't want to worry too much about choosing just the right features. Non-parametric methods, instead, are good when there is a lot of data, but no prior knowledge. Some examples of non-parametric models are: Decision Trees, K-Nearest Neighbor, Support Vector Machines (with Kernel trick).

In conclusion, with parametric models to predict new data, we only need to determinate the parameters of the model, while in non-parametric methods there is more flexibility and for forecasting new data we need to know the parameters of the model and the state of the data that has been observed.

Some advantages of parametric models are: usually they are easier to understand and interpret results; they are typically very fast to learn from data and they do not require as much training data and can work well even if the fit to the data is not perfect. Obvious limitations of parametric Machine Learning algorithms are: by choosing a priori functional form, these methods are highly constrained to that specific form, so they are usually more suited to simpler problems; if we don't have strong and founded hypotheses on mapping function, in practice the methods give not so good results.

An easy to understand non-parametric model is the k-nearest neighbors algorithm that makes predictions based on the k most similar training patterns for a new data instance. The method does not assume anything about the form of the mapping function other than patterns that are close are likely have a similar output variable.

Some more examples of popular non-parametric machine learning algorithms are: Decision Trees like CART and C4.5, Support Vector Machines.

Benefits of Nonparametric Machine Learning Algorithms: Capable of fitting a large number of functional forms, No assumptions (or weak assumptions) about the underlying function, Can result in higher performance models for prediction. Limitations of nonparametric Machine Learning Algorithms: they usually require a lot more training data to estimate the mapping function; they can be slow to train as they often have far more parameters to train; there is more of a risk to overfit the training data and it is harder to explain why specific predictions are made.

The two main types of machine learning strategies currently used are supervised and unsupervised machine learning algorithms. The difference between these two types is defined by the way in which each algorithm learns the data to make predictions.

The supervised machine learning algorithms are the most used. With this model, a data scientist acts as a guide and teaches the algorithm the results to be generated. In supervised machine learning the algorithm learns from an already labeled data set with a predefined output (the expected output value for a function, the expected label for a classification problem).

Linear and logistic regression algorithms, multi-class classification algorithms and support vector machines are some examples of supervised machine learning.

Unsupervised machine learning uses a more independent approach, in which a computer learns to identify complex processes and patterns without the careful and constant guidance of an extern expert (usually human, but it could be another computer algorithm). Unsupervised machine learning involves training based on data without labels and for which a specific output has not been defined.

Unsupervised machine learning will look for the similarities between the dataset points and divide them into groups, assigning each group the new corresponding label. The k-means clustering algorithms, the analysis of main and independent components and the association rules are examples of unsupervised machine learning.

Machine learning has been adopted in a wide range of sectors to support various business objectives and use cases, including: classification of images, recommendation engines, calculation of the customer life cycle value, detection of anomalies, determination of dynamic prices, predictive maintenance [Bunks et al., 2000; Cline et al., 2017; Hashemian and Bean, 2011; Kaur et al., 2010; Liu et al., 2015b; Peng and Dong, 2011; Peng et al., 2010; Su et al., 2006; Susto et al., 2015; Wang et al., 2017b; Wu et al., 2007; Yam et al., 2001; Zeng et al., 2006; Zhou et al., 2010].

Among the reasons that have led to an ever wider expansion of machine learning in the

artificial intelligence sector, it cannot be overlooked that in recent decades there has been a synergetic development of CPU and GPU technologies on the one hand and machine learning algorithms on the other.

For example, compared to classic artificial intelligence algorithms, the density of jump instructions in machine learning is extremely low (a strongly penalizing element with the use of very long execution pipelines, dozens of stages on CPU, hundreds on GPU). Moreover, machine learning requires massive crunch number capability, with a predominant use of linear algebra, which is well suited to ample SIMD execution unit inside modern CPU (we have progressively witnessed the transition from 1997 K2-2 64-bit 3D Now units to 1999 Katmai 128-bit SSE, 2011 Sandy Bridge 256-bit AVX and, for a couple of years now, Skylake-X and recent Ice Lake 512-bit AVX-512)

Furthermore, machine learning algorithms present regular, predictable memory access patterns (which are relatively simple to manage from cache memories and hardware data prefetch units) and show more bandwidth sensitivity (it's decidedly simpler to augment bandwidth with respect to decrease latency).

Finally, machine learning tecniques usually make it relatively simple to partition the computation in the data domain, give multi-thread friendly and opens the doors to exploit multicore processor, graphic processor (GPU), FPGA and ASIC implementations.

A vulnerability of the machine learning techniques is that using faulty data or dirty data can lead to making bad predictions even if a very good model has been selected. For example, in supervised learning, we use data that has been previously labeled, but in many cases, this labeling is normally done by a human, which can lead to errors.

In theory, you can never have too much data (as long it's correct data). In practice, even though there have been tremendous advances in storage and computing costs and performance, we are still bound by physical constraints of time and space. So currently, one of the most important jobs a data scientist has is to judiciously pick out the data sources they think will have an impact in achieving accurate model predictions.

Not having enough data is possible because of cost or financial issues, privacy issues (for example, for health care), and so on.

There is a famous theorem in mathematics. The "No Free Lunch" (NFL) theorem states that there is no one model that works best for every problem. The assumptions of a good model for one domain may not hold for another, so it is not uncommon in data science to iterate using multiple models, trying to find the one that fits best for a given situation. This is especially true in supervised learning. Validation or cross-validation is commonly used to assess the predictive accuracy of multiple models with varying complexity to find the most suitable model. In addition, a model that works well could also be trained using multiple algorithms – for example, linear regression could be trained using normal equations or using gradient descent.

Depending on the use case, it is critical to ascertain the trade-offs between speed, accuracy, and complexity of different models and algorithms and to use the model that works best for a given domain. In computational complexity and optimization the no free lunch theorem is a result that states that for certain types of mathematical problems, the computational cost of finding a solution, averaged over all problems in the class, is the same for any solution method.

2.2.1 VC Dimension

The Vapnik–Chervonenkis (VC) dimension is a measure of the complexity of a space of functions that can be learned by a statistical classification algorithm. It is defined as the cardinality of the largest set of points that the algorithm can shatter. A classification model f with some parameter vector θ is said to shatter a set of data points $(x_1, x_2, \ldots, x_n)$

if, for all assignments of labels to those points, there exists a θ such that the model f makes no errors when evaluating that set of data points.

The VC dimension of a model f is the maximum number of points that can be arranged so that f shatters them. More formally, it is the maximum cardinal D such that some data point set of cardinality D can be shattered by f.

In general, VC dimension is a property of a set of functions which can be defined by various classes of function f. Let's consider a function that correspond to the binary pattern recognition problem

f we have n points then we can label them 2n possible way. Now for each labelling, a member of the set can be found which correctly assigns those labels then we say that set of the point is shattered by that set of function. The maximum number of the training points that can be shattered by is called the VC dimension for the set function . If for a function the VC dimension is h, then there exist at least one set of h points that can be shattered.

Let's assume that for a feature space R2, and the set of functions defined by a oriented straight line. Points of one side of the given line are labeled as class 1 and other points are as class -1. The orientation represents by the arrow in the figure 2.12 shows the points shattered by the line. Therefore it is possible to shatter three points. But not possible to find four points.

Intuitively, we may think that having higher parameters would result in higher VC dimension and few parameters will result very low VC dimension. But this intuition is proven wrong by E. Levin and J.S. Denker[Cortes and Vapnik, 1995]. It's been stated that: A learning machine with just one parameter, but with infinite VC dimension (a family of classifiers is said to have infinite VC dimension if it can shatter l points, no matter how large l). The definition of step function H(x);

Now lets consider one parameter family of functions defined by

Now we choose some number l number of points to be shattered:

Then we specify the labels as:

Then f(a) gives this labeling if we choose to be

2.3 Neural Networks

The complex and varied repertoire of behaviors expressed by living organisms in interactions with their respective environments is attributable to the computational processes performed by their nervous systems. Although there are significant differences in the constitution and organization of the nervous systems among the many forms of life, from the simplest invertebrates to the most complex mammals, it is possible to describe their basic functioning by virtue of interactions between special "computational" cells, called neurons. Biologically inspired by the structure and function of nervous tissue, particularly by the nervous system of vertebrates[Kandel et al., 2013], therefore it is possible to elaborate very general models of computation, characterised by the interactions between elementary computing units, called neurons.

Artificial neural networks are therefore made up of a set of computing units, usually of a simple and uniform type, connected to each other by making various kind of connections, modeled on the biological synaptic ones, so to allow the (unidirectional) transfer of the activation status from an afferent neuron to an efferent one.

Brief notes on biological neural networks

Nervous tissue is one of the four basic types of tissue present in Metazoa, that is, in almost all multicellular living organisms belonging to the Animalia kingdom, and constitutes a morpho-functional and structural unit responsible for the coordination of voluntary and involuntary actions that allow the individual to relate to his environment. This function is carried out by receiving, processing and transmitting the internal and external stimuli of the body. A peculiarity of the nervous tissue consists in the inclusion of cells equipped with special structures capable of sending signals, accurately and quickly, to cells of the organism even very distant.

Although the scientific study of the physiological properties of the nervous system can be traced back to the end of the eighteenth century, for example with the studies on the electrical nature of nerve communications conducted by Galvani [Kandel et al., 2013], only towards the end of the XIX century Golgi developed the so-called black reaction method, through which it was possible to analyze the nervous tissues, highlighting some constituent cells; in fact, in the central nervous system the cells are so densely interconnected that they are difficult to identify using optical microscopes. On the basis of his observations, Golgi was a supporter of a reticular theory, in which the presence of a functional syncytium of the nerve cells was hypothesized, i.e. that the latter were substantially fused in a single block (substantially a multinucleated cell).

Later Cajal exploited Golgi's analysis technique extensively and hu further perfected it, so that he was able to produce many accurate images of the cells of the nervous system; with its studies, Cajal argued, instead, that the individual nerve cells, called neurons, were anatomically and functionally distinct [Fields, 2006; Johnston et al., 2005]. Cajal also hypothesized that the distinction of cytoplasmic extensions in dendrites and axons played a crucial role in the transmission of information in a particular direction, formulating the law of dynamic polarization [Kandel et al., 2013].

Therefore, a controversy arose in neurobiology between the supporters of these two opposing interpretative hypotheses on the structure of the nervous system. Thus, in 1907 tissue culture began to resolve this dispute [Alberts et al., 2008]: small pieces of spinal cord were placed on coagulated tissue fluid in a warm and moist room and subsequently observed at intervals of regular time under a microscope. The American embryologist Harrison showed that long and thin cytoplasmic extensions are growing from the body of the individual nerve cells, even when each neuron is isolated from the others in specific tissue cultures [Kandel et al., 2013], thus proving that each nerve fiber derives from a

single nerve cell and is not the product of the fusion of many cells, thus confirming the so-called "neural doctrine ".

The introduction of electron microscopy sanctioned the definite proof of the validity of the neural doctrine [Kandel et al., 2013]: only in 1954 it was in fact possible to observe for the first time the synaptic cleft, that is the inter-synaptic space, about 20 nm, which results from the juxtaposition of the respective plasma membranes of the presynaptic axon and dendrites or of the cell body of the postsynaptic neuron [Johnston et al., 2005], thus providing convincing evidence that refuted the reticular theory.

In the nervous tissue, two main categories of cells are distinguished: glial cells and neuronal cells, or neurons.

Glial cells generally have a nutritive and supportive function for neurons, they guide their development during the growth of the organism, ensure the isolation of nerve tissues and protection from foreign bodies in case of injury. For over a century, it has been believed that they did not play a significant role in signal transmission and processing, but recent studies seem to suggest some type of involvement in cognitive processes [Franklin and Bussey, 2013; Ransom and Orkand, 1996; Turrigiano, 2006], although the mechanism by which they interact with neurons is not yet well understood.

Although neurons can exhibit significant differences between different animal species, as well as between distinct anatomical regions of the same organism, they have a significant set of common features. Neurons, in fact, consist of a cell body, called soma, that is the spherical part, usually with a diameter of the order of $10\mu m$, which includes the nucleus, which perform the synthesis of proteins necessary for cell development and repair. Cytoplasmic extensions, called neurites, originate from the soma; they vary enormously in length, thickness, branching mode and molecular structure. Despite this, most neurites can be attributed to one of the following functional categories: dendrites and axons.
The dendrites, characterized by a diameter of the order of $1\mu m$, have branches, similar to those of a tree, through which they receive signals from afferent neurons and propagate them in a centripetal direction, that is, towards the soma, while the axon, whose diameter is also usually of the order of $1\mu m$, conducts the signal in a centrifugal direction towards other cells, also located at a considerable distance. As an example, in a giraffe there are axons that travel several meters, avoiding to establish connections with inappropriate neuronal partners, to reach the right region and recognize the appropriate synaptic objectives [Kandel et al., 2013].

The complexity of the dendritic tree represents one of the main determinants of neuronal morphology and the number of signals received by the neuron. Unlike the axon, dendrites are not good conductors of nerve signals which tend to decrease in intensity. In addition, the dendrites thin up to the terminal point and contain polyribosomes. The axon, which starts from the so-called "axon hillock", has a uniform diameter and shows an excellent conductor thanks to the layers of myelin. The final part of the axon is an expansion called terminal button or synaptic button, through which it can make contact with the dendrites, or directly with the cell body, of other neurons. A neuron can react to external, chemical and electrical stimuli; it can reach a state of excitement, which triggers the production of an electrical impulse that propagates along the axon with minimal losses, until it reaches the synaptic button, where synaptic vesicles are present; they release neurotransmitters in the small synaptic cleft that stands between the axon and the dendrites; it is interesting to note that the release of neurotransmitters can only occur in a quantized manner [Katz, 1969]. The most common form of synapses, with axo-dendritic connection, has been described in a simplified way, but axosomatic, axo-axonic, dendro-dendritic synapses have also been observed.

It is not easy to determine how many cells make up the nervous system and how exactly they are distributed; for example, in [Kandel et al., 2013] and [Izhikevich, 2007] it is

specified that the human brain can be considered a network made up of over 10^{11} neurons, also in [Noctor et al., 2007] the presence of 10^{11} to 2 $cdot10^{11}$ neurons is estimated, with a number of glial cells higher by an order of magnitude. Before isotropic fractionation was developed, finding accurate cell numbers in the brain was more painstaking and susceptible to errors. Determining glial cell counts has been particularly challenging due to the small size of glia and the difficulty in telling them apart from other small cells. Although it was widely believed that the tiny glia outnumbered neurons, there was not a lot of hard evidence to prove this was the case, while in [Azevedo et al., 2009] the presence of $8.6 \cdot 10^{10}$ neurons and $8.4 \cdot 10^{10}$ glial cells is considered approximately.

A typical neuron receives afferent signals from over 10^4 other neurons via synaptic connections, i.e. contacts on its [Izhikevich, 2007] dendritic tree.

Biological-inspired neural models

Nicolas Rashevsky was a pioneer in the application of mathematics to biology as it proposed the ambitious goal of developing a quantitative theory[Rashevsky, 1933], analogous to mathematical physics, capable of dealing with the entire field of biology investigation [Rashevsky, 1938]. By developing a theory of the conduction of nerve signals, based on electrochemical gradients, Rashevsky managed to formulate one of the first models of neural excitability [Rashevsky, 1933], which expressed the relations between the intensity by means of differential equations of neural arousal and chemical concentrations, which play an excitatory or inhibitory role [Abraham, 2002].

Then Alan Hodgkin and Andrew Huxley described an accurate model to explain the dynamic of action potentials in the squid giant axon. They received the 1963 Nobel Prize in Physiology or Medicine for this work. The Hodgkin-Huxley computational model describes how the generation and propagation of action potential takes place in the squid giant axon, through nonlinear differential equations [Hodgkin and Huxley, 1952a,b]. It is a mathematical model strongly inspired by biological observations and therefore biophysically accurate, but presents a high computational complexity (usually prohibitive for large-scale simulations). The first classic experiments were conducted just on the squid giant axon because it is so large and robust so that it was possible to extrude its cytoplasm and therefore to perfuse it internally with artificial solutions of $Na^+, K^+, Cl^-, SO_4^{2-}$ [Alberts et al., 2008], as well as applying electrodes for accurate measurements of the potential differences already with the techniques available around the middle of the twentieth century. The model comprises four variables, described by the following system of differential equations

$$C_m \frac{\mathrm{d}V_m}{\mathrm{d}t} = I - g_K(V_m - V_K) - g_{Na}(V_m - V_{Na}) - g_l(V_m - V_l)$$

$$C_m \frac{\mathrm{d}V_m}{\mathrm{d}t} = I - \bar{g}_K n^4(V_m - V_K) - \bar{g}_{Na} m^3 h(V_m - V_{Na}) - \bar{g}_l(V_m - V_l)$$

$$\frac{dn}{dt} = \alpha_n(V_m)(1 - n) - \beta_n(V_m)n$$

$$\frac{dm}{dt} = \alpha_m(V_m)(1 - m) - \beta_m(V_m)m$$

$$\frac{dh}{dt} = \alpha_h(V_m)(1 - h) - \beta_h(V_m)h$$

Later, FitzHugh and J. Nagumo simplified the Hodgkin-Huxley model [FitzHugh, 1955; Izhikevich and FitzHugh, 2006; Nagumo et al., 1962].

$$\frac{dv}{dt} = v - \frac{v^3}{3} - w + I_{ext}$$

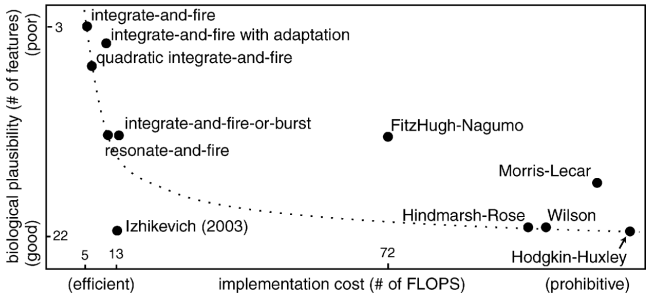


Figure 2.1: Comparison of neural models (relationship between biological plausibility and computational complexity)

Comparison between the neuro-computational properties of various neural models, in which the biological plausibility, expressed by counting the characteristics exhibited, is compared with respect to the implementation cost [Izhikevich, 2004].

$$\tau \frac{dw}{dt} = v + a - bw$$

Izhikevich neural model

It has been found that the models that accurately describe biological neurons usually present a high computational complexity, therefore, simpler models have been developed, suitable for carrying out simulations, such as the "integrate and shoot" model, which is computationally efficient, but proves unable to reproduce the rich dynamic repertoire expressed by cortical neurons, precisely because it is unrealistically simple. Izhikevich's neuronal model offers a discreet union between computational efficiency (similar to the model integrates and shoots) and biological plausibility, as it does not differ significantly from the dynamics expressed by Hodgkin-Huxley models [Izhikevich, 2003]. The model is based on two variables (v represents the membrane potential, while u is a "recovery" variable) and four parameters, resulting described by the following system of equations:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = 0.04v^2 + 5v + 140 - u + I$$

$$\frac{\mathrm{d}u}{\mathrm{d}t} = a(bv - u)$$

$$v \geq 30mV \Rightarrow v \leftarrow c, u \leftarrow u + d$$

Early models of artificial neural networks

The first attempt to formalize a hypothesis on how the brain can process information can be traced[Lisboa, 1992] in the paper of McCulloch and Pitts[McCulloch and Pitts, 1943], incidentally, the latter was a Rashevsky's student.

In the paper of McCulloch and Pitts, starting from the consideration that neuronal activity seems to be of the "all or nothing" type, they shaped the functioning of neurons as simple logical operators; by building networks with many interconnected units, they demonstrated that there was an equivalence between the model of neural networks they had developed and the propositional calculus, that is, that it was possible to perform arbitrary logic operations using appropriate neural networks. In their model, a linear combination of the afferent signals can be supplied as input (with ad hoc choice of coefficients, i.e. neural "weights") to each neuron, where it is compared with a threshold ("bias"), eventually applying a non linear output function, the Heaviside step function. It was quickly understood that the overall behavior of a circuit consisting of many neurons

Models	biophysically meaningful	tonic spiking	phasic spiking	tonic bursting	phasic bursting	mixed mode	spike frequency adaptation	class 1 excitable	class 2 excitable	spike latency	subthreshold oscillations	resonator	integrator	rebound spike	rebound burst	threshold variability	bistability	DAP	accommodation	inhibition-indiced spiking	inhibition-induced bursting	chaos	# of FLOPS
integrate-and-fire	-	+	-	-	-	-	-	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	5
integrate-and-fire with adapt.	-	+	-	-	-	-	+	+	-	-	-	-	+	-	-	-	-	+	-	-	-	-	10
integrate-and-fire-or-burst	-	+	+		+	-	+	+	-	-	-	-	+	+	+	-	+	+	-	-	-		13
resonate-and-fire	-	+	+	-	-	-	-	+	+	-	+	+	+	+	-	-	+	+	+	-	-	+	10
quadratic integrate-and-fire	-	+	-	-	-	-	-	+	-	+	-	-	+	-	-	+	+	-	-	-	-	-	7
Izhikevich (2003)	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	13
FitzHugh-Nagumo	-	+	+	-		-	-	+	-	+	+	+	-	+	-	+	+	-	+	+	-	-	72
Hindmarsh-Rose	-	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	120
Morris-Lecar	+	+	+	-		-	-	+	+	+	+	+	+	+		+	+	-	+	+	-	-	600
Wilson	-	+	+	+			+	+	+	+	+	+	+	+	+	+		+	+				180
Hodgkin-Huxley	+	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	1200

Figure 2.2: Comparison of the neuro-computational properties of spiking and bursting models

Comparison of the neuro-computational properties of various neural models, in which FLOPS's # denotes a number, at least indicative, of floating point arithmetic operations necessary to simulate the functioning of the model during a time span of 1 ms. Each empty square indicates that the model considered should be able to present the corresponding property, according to theoretical considerations, with an appropriate choice of parameters, but Izhikevich was unable to identify these parameters within a period of reasonable time. [Izhikevich, 2004].

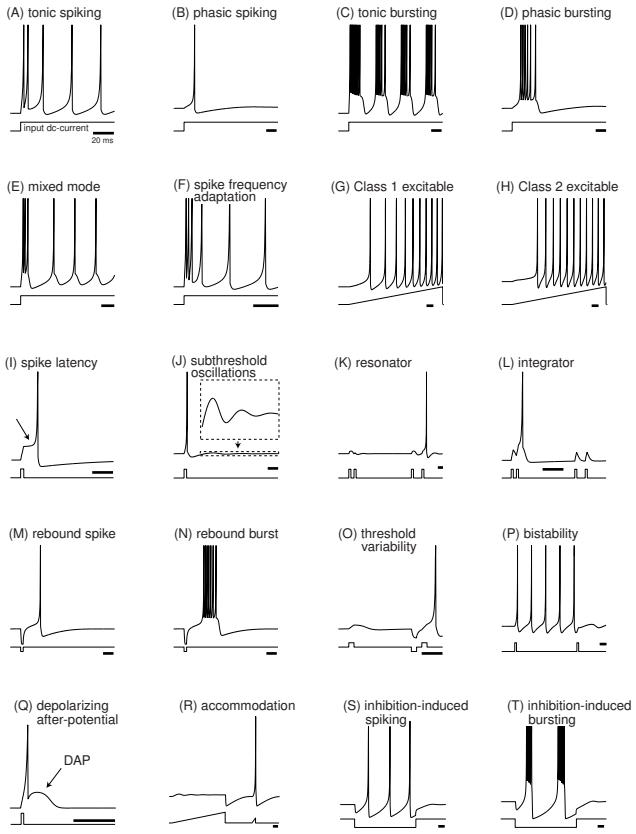


Figure 2.3: Summary of the neuro-computational properties of biological spiking neurons Synthesis of the neuro-computational properties of biological neurons. Each horizontal bar represents a time interval of 20 ms [Izhikevich, 2004].

depended more on the structure of the connections than on the minute details of the functioning of the individual neural units, therefore pioneer researchers began to wonder about the possibility to alter the connectivity of the neural circuits to obtain desirable correlations between nodes, considered as input to the system, and others considered to be output, in order to exploit neural networks for practical tasks, for example as statistical classifiers.

In 1949 the so-called Hebb [Hebb, 1949] rule was enunciated, according to which if a neuron repeatedly excites another neuron, a process of altering the effectiveness of the corresponding synaptic connection occurs.

In 1951, Minsky, who was pursuing his PhD, designed and built, together with Edmonds, the Stochastic Neural Analog Reinforcement Calculator (SNARC), which is considered the first computer based on neural networks, as well as the first machine capable of learning [Russell and Norvig, 1995]. This calculator, built using thermionic valves, was made up of randomly connected networks, in which learning was carried out according to Hebb's rule.

A milestone in the development of artificial neural networks was Rosenblatt's percetron [Rosenblatt, 1958, 1961], a neuron that can receive multiple input lines and has a single output line, with the intent to function as a binary classifier. Percetron works like a McCulloch and Pitts neuron: it calculates a linear combination of the input values, with coefficients considered as synaptic weights, it evaluates the difference with a threshold (bias) and finally applies a simple non-linear function, the Heaviside step function. The crucial step is that Rosenblatt defines a learning algorithm (usually called the delta rule) that allows to modify a vector of parameters (the synaptic weights), generally initialized randomly, presenting to the system predetermined inputs and evaluating the discrepancy between the output of the neuron and the expected one (thus constituting one of the first cases of supervised learning).

A few years later, Minsky and Papert elegantly demonstrated that there were classes of simple problems that went beyond the capabilities of the percetron [Minsky and Seymour, 1969]; in fact, the perceptron classifier works correctly only when the two classes of objects considered are linearly separable, that is if the separation surface between them is a hyperplane. This circumstance can be clarified by taking into consideration the Boolean logical operators (in the following, the false logical value is identified with 0 and the true with 1). If the minimum arity is selected, i.e. if the set of unary operators is taken into account, there are four possibilities: the identity, which leaves unchanged the operator's argument; the negation, which reverses its value; the "null" operator, which returns 0 regardless of the input and similarly the operator which always returns "one". In this very simple case, there is no problem of evaluating the decision boundary, therefore the percetron is able to learn all unary operators. Taking into consideration the arity 2, thera are 16 binary operators (in general, for an operator with n arguments, 2^{2^n} distinct truth tables are possible), including the conjunction (AND), the disjunction (OR) and their negations (NAND, NOR). McCulloch and Pitts showed how to make logical AND and OR gates with a neuron of their model, appropriately choosing the coefficients, and Minsky and Papert showed that percetron could learn those operators with delta rule, but they also showed the perceptron failed with the XOR (and obviously with its denial, XNOR). It is important to emphasize that the limitation imposed by linear separability becomes more and more burdensome with increasing dimensionality; if a percetron can learn 14 of the 16 possible binary operators, things quickly deteriorate with increasing arity, as shown in Tab. 2.1.

While Minsky showed that a single Rosenblatt's perceptron could not learn a logical XOR gate, there are various possibilities to overcome this limitation: it is possible to select a more complicated output function for the neuron, a non-linear combination of

n	Linearly separable op.	All Boolean op.(2^{2^n})	ratio
1	4	4	1
2	14	16	0.875
3	104	256	0.40625
4	1882	65536	0.028717
5	94572	$4.295e+09$	$2.2019e-05$
6	$1.5028e+07$	$1.8447e+19$	$8.1468e-13$
7	$8.3781e+09$	$3.4028e+38$	$2.4621e-29$
8	$1.7562e+13$	$1.1579e+77$	$1.5166e-64$

Table 2.1: Determination of the Boolean logical operators that are linearly separable, for each arity

the afferent signals can be supplied as input (for example, introducing quadratic terms [Hassoun, 1995]), or more neural units can be connected in cascade, possibly arranged in multiple layers.

For example, to solve the XOR problem you can use the Quadratic Threshold Gate (QTG) [Hassoun, 1995], described by the equation

$$y = H\left[\sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n}\sum_{j=i+1}^{n} w_{ij} x_i x_j - T\right]$$

where y represents the output of the neuron, x_i are the input parameters and H denotes the step function; to get an XOR, just choose: $y = H\left[x_1 + x_2 - 3x_1x_2 - \frac{1}{2}\right]$ In fig. 2.5, instead, there is a solution of the XOR problem with 2 neurons, in which there are no constraints on the topology (the network, that is, it is not feed-forward).

The difficulty in managing networks with multiple neurons is inherent in the training procedure, that is, how to modify the system parameters to reach the desired output. The computational burden of the learning algorithms was quadratic compared to the number of weights, so the interest in neural networks was reduced until the "rediscovery" of the backpropagation, an efficient way of computing the derivatives of the activation function of the neurons [Rumelhart et al., 1986].

Feed forward neural networks

If there are no cycles in the connection graph, the network is called feed forward [Anderson, 1995; Rojas, 1996], because information flows from inputs through the the intermediate nodes and finally to the output, but there are no feedback connections in which outputs of the model are feed back into the network. This implies that a feed-forward network cannot enjoy a memory, because the current output of the system does not depend on the previous values of the output and therefore neither of the previous internal states Feedforward neural networks can be represented by composing together many different functions to realize a black box global function, with units not linked to input or output (hidden neurons). Multilayer feed forward networks are often created, where each layer can receive information exclusively from the previous one. These networks are considered universal approximators [Cybenko, 1989; Hornik et al., 1989] and are widely used as automatic classifiers [Bishop, 2006; Bishop et al., 1995]; for this purpose, they are usually used through supervised learning algorithms, in which typical examples of input values and the desired outputs are provided. The network can try to learn to infer the relationship between inputs and outputs, modifying the effectiveness of connections, in an attempt to minimize an appropriate error function. The "rediscovery" of back-propagation, an efficient way to compute the derivatives of the neuron activation function

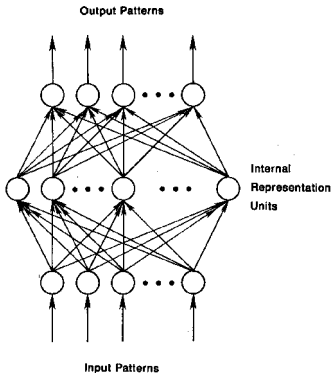


Figure 2.4: Multi-layer feed-forward neural network[Rumelhart et al., 1986].

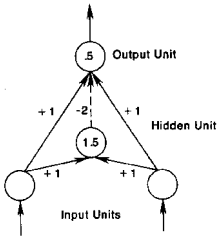


Figure 2.5: A MLFFNN which implements XOR operator [Rumelhart et al., 1986].

[Rumelhart et al., 1986], was of crucial importance to give an acceleration in the spread of neural networks feed forward because it is often used to reduce the computational burden of learning algorithms.

Recurrent neural networks

If in the topology of the interconnections of an artificial neural network the presence of one or more cycles is contemplated, recurrent neural networks are obtained [Amit, 1992]. The presence of a feedback mechanism allows to obtain a behavior that depends on time, because the output of the system depends not only on the current signals supplied at the input, but on a sort of memory, represented by the state of the network; for this reason, recurrent neural networks can be modeled as nonlinear dynamic systems [Strogatz, 2001].

There are multiple models of recurrent neural networks, which differ in topological aspects, for the selection of the activation or exit function of the neural units, for the choice of a synchronous or asynchronous update dynamic, for the use of a continuous or discrete time.

NARX is a really simple recurrent neural network, discrete time, in which the feedback is applied exclusively to the neuron to which the output of the system is assigned [Haykin, 1999]; from the states of the system in the previous p instants, as well as from the inputs in the last q instants, according to the general equation:

$$y_o[t+1] = F\left(y_o[t], y_o[t-1], \ldots, y_o[t-p+1], u[t], u[t-1], \ldots, u[t-q+1]\right)$$

The RMLP model (recurrent multilayer perceptron) is another example of a simple recurrent neural network: it can be conceived as a multilayer feed forward neural network, in which a feedback mechanism is added only at the last layer, the output one.

Hopfield networks are recurrent neural networks, made up of simple threshold neurons, characterized by symmetric connections (with null self-conference terms), used as

2.0

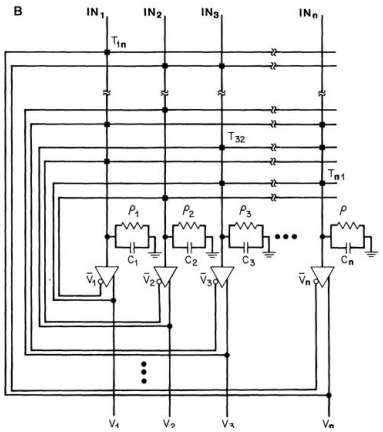


Figure 2.6: SPEC CPU Suite Growth across all of its iterations.

memories addressed on the basis of the content [Hopfield, 1982], offering robustness with respect to alterations of connections .

Continuous-time recurrent neural networks

The CTRNN model (Continuous-time recurrent neural networks) is one of the simplest computational models of non-linear time-continuous neural networks [Beer, 2006].

The CTRNN model was proposed in 1986, to describe the temporal evolution of the neuronal membrane potential as if it were a parallel R-C circuit, based on the observation that the cell membrane is permeable only to some ions[Hopfield et al., 1986]; the original model, with N constituent neurons, is described by the system of coupled differential equations:

$$C_i \dot{u}_i = -\frac{u_i}{R_i} + I_i + \sum_{j=1}^{N} T_{ij} f_j\left(u_j\right)$$

in which u_i represents the membrane potential, C_i, R_idenote the capacity and the resistance of the neuron, I_i indicates the input current, coming from outside the considered circuit, T_{ij} represents the strenght of the synaptic connection and f_j is a sigmoid type function, which varies from 0 to f_{max}.

Despite the simpleness of the model, CTRNNs are widely used because they have a biological plausibility, offer a limited computational burden and are universal approximators [Funahashi and Nakamura, 1993].

In the literature there are different formulations of the model, such as the following: in [Beer, 1995, 2006; Magg and Philippides, 2006; McHale and Husbands, 2004a,b]

$$\dot{y}_i = \frac{1}{\tau_i}\left\{-y_i + \sum_j w_{ij} f_j\left(y_j + \theta_j\right) + I_i^e\right\}$$

in [Buckley, 2008]:

$$\dot{y}_i = \frac{1}{\tau_i}\left\{-y_i + \tanh\left(\sum_j w_{ij} y_j + \theta_i\right)\right\}$$

in [Funahashi and Nakamura, 1993]:

$$\dot{u}_i = -\frac{u_i}{\tau_i} + \sum_j w_{ij}\sigma\left(u_j\right) + I_i^e$$

in [Donnarumma et al., 2015, 2012]

$$\dot{y}_i = \frac{1}{\tau_i}\left[-y_i + f_i\left(\sum_j w_{ij}y_j + \sum_j w_{ij}^e I_j + \theta_i\right)\right]$$

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

If this version of the model is selected and if the time dependency and the choice of the activation function are explicit, it is obtained:

$$\dot{y}_i(t) = \frac{1}{\tau_i}\left\{-y_i(t) + \sigma\left[\sum_j w_{ij}y_j(t) + \sum_j w_{ij}^e I_j(t) + \theta_i\right]\right\}$$

From this equation, by discretizing in the time domain with step Δt and resorting to integration using the Euler method, the discrete-time recurrent neural network equation can be obtained:

$$y_i[n] = y_i[n-1] + \frac{\Delta t}{\tau_i}\left\{-y_i[n-1] + \sigma\left[\sum_j w_{ij}y_j[n-1] + \sum_j w_{ij}^e I_j[n] + \theta_i\right]\right\}$$

Deep Learning

Although it is shown that two-layered feed-forward neural networks can constitute universal approximators [Cybenko, 1989; Hornik et al., 1989], in the sense that they can arbitrarily approximate any function from m input to n output, in the recent years there has been a growing interest in neural networks with multiple layers, that is, developed more in depth than in amplitude [Arnold et al., 2011; Chen and Lin, 2014; Joshi et al., 2017; LeCun et al., 2015; Ota et al., 2017; Szegedy et al., 2015] Modern deep learning provides a powerful framework for supervised learning, by adding more layers and more units within a layer, a deep network can represent functions of increasing complexity. Most tasks that consist of mapping an input vector to an output vector, and that are easy for a person to do rapidly, can be accomplished via deep learning, given sufficiently large models and sufficiently large datasets of labeled training examples [Goodfellow et al., 2016]. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics. Deep learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Deep convolutional nets have brought about breakthroughs in processing images, video, speech and audio, whereas recurrent nets have shone light on sequential data such as text and speech [LeCun et al., 2015].

The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data do not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f. Because the training data does not show the desired output for each of these layers, they are called hidden layers

Deep learning is a class of machine learning algorithms that uses multiple layers to progressively extract higher level features from the raw input. For example, in image processing, lower layers may identify edges, while higher layers may identify the concepts relevant to a human such as digits or letters or faces.

Deep learning architectures such as deep neural networks, deep belief networks, recurrent neural networks and convolutional neural networks have been applied to fields

including computer vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation, bioinformatics, drug design, medical image analysis, material inspection and board game programs, where they have produced results comparable to and in some cases surpassing human expert performance.

Deep learning is a set of machine learning algorithms that utilize deep neural networks, to power advanced applications, such as image recognition and computer vision, with wide-ranging use-cases across a variety of industries.

Deep learning has a long history and many aspirations. Modern deep learning provides a powerful framework for supervised learning.By adding more layers and more units within a layer, a deep network can representfunctions of increasing complexity. Most tasks that consist of mapping an input vector to an output vector, and that are easy for a person to do rapidly, can be accomplished via deep learning, given sufficiently large models and suffciently large datasets of labeled training examples. Other tasks, that cannot be described as associating one vector to another, or that are difficult enough that a person would require time to think and reflect in order to accomplish the task, remain beyond the scope of deep learning for now. This part of the book describes the core parametric function approximation technology that is behind nearly all modern practical applications of deep learning. We begin by describing the feedforward deep network model that is used to represent these functions. Next, we present advanced techniques for regularizationand optimization of such models. Scaling these models to large inputs such as high-resolution images or long temporal sequences requires specialization. We introduce the convolutional network for scaling to large images and the recurrent neuralnetwork for processing temporal sequences. Finally, we present general guide lines for the practical methodology involved in designing, building, and configuring an application involving deep learning and review some of its applications.

Deep feedforward networks, also called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f. For example, for a classifier, y=f(x) maps an input x to a category y. A feedforward network defines a mapping $y = f(x; \Theta)$ and learns the value of the parameters Θ that result in the best function approximation.

Neuromodulation

The connectomics represents a field of investigation of the neurosciences aimed at the realization of a connectome, that is an accurate description of the neural units and a complete map of the connections (through the realization of a detailed "wiring diagram") of the nervous system of a living organism. there is an ambitious goal of being able to enumerate all the synaptic connections of the brain, even of " complex " vertebrates, like a human being. Already "ultrastructural" analyzes of small brains or reduced portions of the brain of higher organisms are being carried out systematically, working on areas containing millions of synaptic connections (for the human brain, the most accurate estimate in the literature is 10^{15} synapse) [Bargmann, 2012].

The first complete description of the morphology and synaptic connectivity of an organism was made for the spasmidary nematode worm Caenorhabditis elegans, through reconstructions from the electron micro-graphs of the tissue sections [White et al., 1986]. This organism has just 302 neurons, yet offers a rich variety of them, so they have been cataloged in 118 distinct classes, according to their characteristics; approximately 5000 chemical synapses, 2000 neuro-muscular junctions and 600 electrical synapses [White et al., 1986] have been identified. The determination of the connectome of this nematode worm provided for the first time the possibility of attempting to identify and understand the neural basis of the whole behavioral repertoire of a living being [Dunn et al., 2004].

However, it has been highlighted that connectomics is presumably a necessary, but

not sufficient, element to determine the functioning of the nervous system [Brezina, 2010; Marder, 2012].

In fact, mechanisms of communication of nerve cells alternative to synaptic transmission (based on the action of molecules, which act as neurotransmitters, at the so-called chemical synapses, which are generally established between axons and dendrites) have been identified in living organisms. electrical synapses (or gap junctions) [Marder, 1998; Söhl et al., 2005], the mixed synapses [Rash et al., 1996], the ephaptic transmission [Kamermans and Fahrenfort, 2004; Krnjevic, 1986] and a plethora of volumetric transmission mechanisms [Agnati et al., 2006, 2010; Zoli et al., 1998].

In particular, alongside the role played by neurotransmitters in synaptic interactions, more and more emphasis is given to the study of neuromodulators [Bargmann, 2012], that is, a varied repertoire of chemicals that influence the functioning of the nervous system in a more articulated way compared to conventional neurotransmitters , as their effect may not be merely excitatory or inhibitory [Buckley, 2008].

Where neurotransmitters selectively act on a single neuron or rather on a single synaptic link, the neuromodulators intervene, instead, on larger areas, typically involving several neurons and possibly other cells of the nervous system.

Although the nervous systems of living organisms exploit a large number of neurotransmitters and neuromodulators, the first formal models of neural functionality completely ignored the nature of neuromodulators.

Neuromodulatory substances tend to behave as signaling agents with global action because they are released, both from neurons and from glial cells, typically more slowly at release sites consisting of " open " synapses [Zoli and Agnati, 1996] or from non-synaptic sites, from which they are able to spread for long times and wide distances through the intercellular space or even through the circulatory system, in the case of neuro-hormones [Brezina, 2010]. Neuromodulators therefore seem to play a crucial role for the volumetric transmission of signals [Brezina, 2010]. Many of the problems posed by the modeling of neuromodulators, but also the prerequisites for their considerable computational effectiveness, seem to derive from the spatial and temporal dissociation of their activity from that of the neural network on which they act [Brezina, 2010].

Neuromodulation seems to be a valid tool to increase or at least control the computational complexity of a neural network, without necessarily increasing its structural complexity [Fellous and Linster, 1998], understood as the number of constituent neurons and synaptic connections between of them constituted.

From a computational point of view, there is still much work to be done to understand the overall network architecture as well as the effects of the changes that the different neuromodulators produce in the individual elements present in these circuits. However, neuromodulators seem to be able to deal with problems related to the massively distributed architecture of the central nervous system, providing important information, both on the state of the organism and on the environment in which it is found, in a diffuse or selective way depending on the circumstances, as well as having significant effects on information processing by changing the dynamics properties of neurons, as well as the effectiveness of their synaptic connections [Dayan, 2012].

The speed of neuromodulatory processes in relation to neural computations plays a decisive role in determining the properties of a neural system. It is possible to distinguish two fundamental cases: the intervention of a slow modulation with respect to the functioning of the conventional neural network or a rapid modulation on a slow computation [Fellous and Linster, 1998].

Since the action of a neuromodulator is usually a slow process with a wide spatial diffusion, acting on some aspects of the properties of the membranes and synapses, we frequently fall back to the first case considered.

Rhythmic activation of neurons seems to play a crucial role in invertebrates as well as vertebrates, where the generation and synchronization of oscillatory activity can allow distinct neural subnets to communicate in a coordinated way. In recent years, there has been growing interest in the scientific community in the interaction of phenomena on different time scales in neural systems, in particular as regards the neuromodulatory phenomena [Buckley et al., 2005]. However, most research activities tend to consider neuromodulatory action as a slow and extrinsic influence on a conventional neural circuit: this kind of simplifications allows you to model neuromodulation as a dynamic change in the parameters of a neural system [Buckley, 2008].

Although the original Hodgkin-Huxley model does not include the intervention of neuromodulatory substances [Hodgkin and Huxley, 1952b], starting from a model of this type in [Bertram, 1993] we evaluate the effects of serotonin on the neuron R_{15} (burster type) of the abdominal ganglion of the Aplysia Californica mollusc, introducing a dependence of the maximum conductance on the concentration of the neuromodulator considered.

The exogenous application of serotonin on the neuron soma indirectly induces (through the production of intracellular cAMP) an increase of two "sub-threshold" currents, I_R and I_{NSR}, whose corresponding maximum conductances tend to increase up to reach a saturation (first one, then the other). The concentration of serotonin is analyzed as a fixed parameter, therefore any changes take place on a long time scale with respect to the dynamics of the considered neuron. Different values of this parameter allow to control the waveform of the membrane voltage, profoundly altering the behavior of the neuron. In fact, a change in the concentration of serotonin can change the number of shots in a single burst, the distance and the depth of the pause between two bursts, up to determine a steady state or lead to a rhythmic oscillation.

$$
\begin{aligned}
C_m \frac{\mathrm{d}V_m}{\mathrm{d}t} = I - & \\
& [\bar{g}_K n^2 j (V_m - V_K) + \bar{g}_{Na} m^3 h (V_m - V_{Na}) + \bar{g}_l (V_m - V_l) + \\
& \bar{g}_{Ca} x^2 (V_m - V_{Ca}) + \bar{g}_{K(Ca)} \frac{[Ca^{2+}]}{\bar{\mu}_\infty \ + [Ca^{2+}]} (V_m - V_K)] - \\
& [\bar{g}_{NSR}([5-HT]) q^4 y_\infty (V_m)(V_m - V_{Ca}) + I_D + \bar{g}_R([5-HT]) \\
& r_\infty (V_m)(V_m - V_K) + I_A]
\end{aligned}
$$

with:

$$
\bar{g}_{NSR}([5-HT]) = 0.12 + \frac{0.84}{1 + e^{-3(5[5-HT]-3)}}
$$

$$
\bar{g}_R([5-HT]) = 0.3 + \frac{1.8}{1 + e^{-1.6(10[5-HT]-3)}}
$$

$$
\frac{dn}{dt} = \alpha_n(V_m)(1-n) - \beta_n(V_m) n
$$

$$
\frac{dm}{dt} = \alpha_m(V_m)(1-m) - \beta_m(V_m) m
$$

$$
\frac{dh}{dt} = \alpha_h(V_m)(1-h) - \beta_h(V_m) h
$$

GasNet

Drawing inspiration from the biological behavior of gases such as nitrogen monoxide [Changeux, 1993; Gally et al., 1990; Hölscher, 1997; Montague and Sejnowski, 1994; Palmer et al., 1987; Philippides et al., 1998; Smith and Philippides, 2000], a computational

model of recurrent, discrete-time neural network was developed, GasNet [Husbands, 1998; Husbands et al., 1998a,b; Philippides et al., 2005]. This model implements an intrinsic neuromodulation process, in which the presence of gaseous neuromodulators can influence the properties of artificial neural units of the CTRNN type.

The output of each neuron is usually computed, applying a transfer function to the weighted sum of its inputs, but this sum is modulated by a multiplicative factor k_i which depends on the instantaneous concentration of the gases, diffused by the network nodes, in a neighborhood of the considered neuron. In the GasNet model, some neurons are classified as potential emitters, as they are capable of emitting one of the possible gases, if an appropriate condition is met, such as exceeding a predetermined level of the electrical potential or the concentration of one of gases, while in principle all neurons undergo the effects of neuromodulatory gas.

The original GasNet network model [Husbands et al., 1998a] contemplates the presence of 19 adaptive parameters for each neuron:

- two parameters x, y for the position in the Euclidean plane
- three parameters R_p, θ_{1p}, θ_{2p} to find the "positive" circular segment
- three parameters R_n, θ_{1n}, θ_{2n} to find the "negative" circular segment
- vis_{in} binary selector that indicates whether the neuron receives visual input; if it is active, it is followed by three further parameters:
  - the polar coordinates of the corresponding pixel in the image sensor (vis_r e vis_θ)
  - a threshold (vis_{thr})
- rec eventually a recurrent connection (inhibitory, excitatory, absent)
- TE represents under which circumstances the neuron can emit gas
- CE indicates the type of gas that may be emitted
- s determines the speed of diffusion
- R_e specifies the maximum emission radius
- $index^0$ is the default value of the index used to determine the transfer parameter k
- b the bias

In this version of the model, only synaptic connection values of $+1$ or -1 are allowed; furthermore, it is required that all outgoing excitatory connections ($+1$) belong to the "positive" circular segment and all the inhibitory ones (-1) to the "negative" one. If the two segments are partially overlapping, any neurons that lie in the intersection will have both excitatory and inhibitory connections. For the emission of gases, on the basis of the value of the TE parameter, it is assessed either the activation by the neuron of an electrical threshold or the achievement of a minimum gas concentration around the neuron; the threshold values are set a priori and the same for all neurons, as an electric type threshold equal to 0.5 and a threshold referred to gas concentrations equal to 0.1 are chosen.

The temporal dynamics of a GasNet neuron is described by the equation:

$$y_i[n] = \tanh\left[k_i[n]\left(\sum_j w_{ij} y_j[n-1] + I_i[n]\right) + b_i\right]$$

The multiplicative factor is determined by $k_i[n] = P[ind_i[n]]$, in which:

$$\begin{aligned} ind_i[n] &= f\left[ind_i[0] + \frac{C_{i,1}[n]}{C_0 K}\left(N - ind_i[0]\right) - \frac{C_{i,2}[n]}{C_0 K} ind_i[0]\right] \\ &= f\left[ind_i[0]\left(1 - \frac{C_{i,1}[n] + C_{i,2}[n]}{C_0 K}\right) + \frac{C_{i,1}[n]}{C_0 K} N\right] \end{aligned}$$

where $C0$ and K represent parameters, usually set equal to one, P is a vector of the possible values (discrete and symmetric with respect to zero) assumed by the multiplicative coefficient k_i and N indicates the length (typically $N = 11$ or $N = 13$):

$$P = \{k_{min}, \frac{1}{2}k_{min}, \ldots, 0, \ldots, \frac{1}{2}k_{max}, k_{max}\}$$

$$k_{min} = -4.0, k_{max} = 4.0, N = length(P)$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ \lfloor x \rfloor & 0 < x < N \\ N & x \geq N \end{cases}$$

The concentration of the l-th gas at the i-th neuron is expressed by:

$$C_i^l[n] = \sum_j \tilde{C}_{i,j}^l(d_{ij}, n), \text{ con } d_{ij} = |\vec{x_i} - \vec{x_j}|$$

where the contribution $\tilde{C}_{i,j}^l$ generated by the j-neuron is determined by:

$$\tilde{C}_{i,j}^l(d_{ij}, n) = \begin{cases} C_0 e^{-(\frac{d_{ij}}{r_j})^2} T_j[n] & d_{ij} < r_j \\ 0 & d_{ij} \geq r_j \end{cases}$$

with

$$T_i(t) = \begin{cases} H(\frac{t - t_{e,i}}{s_i}) & \text{if it is emitting gas} \\ H[H(\frac{t_{s,i} - t_{e,i}}{s_i}) - H(\frac{t - t_{s,i}}{s_i})] & \text{if it is not emitting gas} \end{cases}$$

and

$$H(x) = \begin{cases} 0 & x \leq 0 \\ x & 0 < x < 1 \\ 1 & x \geq 1 \end{cases}$$

in order to guarantee a saturation of the gas concentration (only in the emission phase) in the range $[0, 1]$.

In the literature, some variants of the GasNet model have been identified, based on 2 or 4 gases, with a Gaussian or exponential spatial diffusion trend, with different dependence of the multiplying factor on the gas concentrations, with limitations on the number of input synaptic connections to each neuron, with the velocities for the growth and decay of the concentration of each gas coinciding or not. However, these models share some simplifications:

- neurons are arranged in a two-dimensional Euclidean space;
- the weights of the connections can only take values ± 1;
- k_i can take values in a predetermined range (usually $[-4.0, 4.0]$), moreover, only discrete values are generally allowed;
- the gaseous diffusion is isotropic in space, decreases from the source and is canceled beyond a predetermined radius of influence;

- the gaseous diffusion shows a growth over time (during the emission) and a subsequent decrease with a linear trend in both cases, but with concentration saturation (usually in the interval $[0, 1]$);
- neuromodulation is slow compared to the characteristic times of synaptic communications.

Among the multiple variants of the GasNet model found in the literature, there is one in which the coefficients k_i that modulate the functioning of each neuron can assume continuous values [Buckley, 2008]:

$$k_i[n] = k_i[0] + C_{i,1}[n](k_{max} - k_i[0]) - C_{i,2}[n](k_i[0] - k_{min})$$

$$k_i[n] \in [k_{min}, k_{max}], k_{min} = -4, k_{max} = 4$$

$$C_{i,l}[n] = \sum_j \tilde{C}_{j,l}(d_{ij}, n)$$

$$\tilde{C}_{j,l}(d_{ij}, n) = \begin{cases} e^{-(\frac{d_{ij}}{r_j})^2} T_j[n] & d_{ij} < r_j \\ 0 & d_{ij} \geq r_j \end{cases}$$

$$\dot{T}_i(t) = H(y_i(t), C_i(t))G_i + (H(y_i(t), C_i(t)) - 1)D_i$$

$$C_i(t) = C_{i,1}(t) + C_{i,2}(t)$$

$$H_i(y, C) = \begin{cases} 1 & \text{se } y > \bar{\theta}_i \text{ o } C > \bar{C}_i \\ 0 & \text{otherwise} \end{cases}$$

$$GT_i \in \{-1, 0, 1\}$$

GasNet are mainly used for robotic tasks, such as discrimination of visual inputs [Husbands et al., 2010; Smith et al., 2002] or for the control of bipedal or quadrupedal locomotion [McHale and Husbands, 2004a,b], essentially using evolutionary learning algorithms [Floreano et al., 2008; Philippides et al., 2002].

In literature, an attempt has been made to evaluate the contributions of the elements characterizing the GasNet approach, adding or removing them in turn from the equations that regulate neuronal dynamics, so as to take into consideration all possible combinations [Buckley, 2008]. At least in the context of the specific task taken into consideration (pattern generation), it seems that the possibility of intervening on the synaptic inputs using a multiplicative factor, dependent on the concentration of gas, plays the crucial role in allowing GasNet to provide performance superior to other types of recurrent neural networks, lacking a neuromodulatory mechanism, even if the latter are equipped with a greater number of neurons. The other peculiar characteristics of the GasNet model, i.e. the spatial encapsulation (i.e. the effects of the spatial arrangement of neurons and limited diffusion rays) and the slow temporal dynamics of the action of the gases, on the other hand, do not seem to significantly improve the capacities. of the neural network. Until now, theoretical demonstrations justifying the apparent superiority of computational models such as GasNet compared to recurrent neural networks without neuromodulation have not been clarified [Smith et al., 2001]. In general, since the role of neuromodulation has not been fully clarified in the biological field, computational and robotic models temporarily deal with problems on a wide application spectrum, to identify the existing relationships between the characteristics of the model and its performance in fulfilling certain tasks [Soltoggio, 2008].

2.4 Machine Learning and recent computer industry development

The implementation of most of the algorithms in vogue in the machine learning sector involves a massive use of computationally intensive linear algebra routines, therefore it can significantly benefit from the adoption of hardware solutions that support the simultaneous processing of multiple vector or matrix operations.

This implies that, where it is possible to effectively separate the data to be processed into relatively independent blocks, such machine learning algorithms can receive significant benefits from the adoption of some sort of parallel computing systems. Depending on the specific algorithm, the dimensionality of the problem, the size of the dataset used in the training phase or the number of applications expected in the inferential phase, significant speedups can be found by adopting SMT (Simultaneous Multi Threading) techniques, adding more cores to each physical processor, adopting multi-socket systems or relying on a cluster of computers.

In addition, a significant speed improvement can be obtained by uniformly processing the elements of an array by adopting special instructions that support some kind of vector calculation. To this end, over the years various ISA (Instruction Set Architecture) extensions have been adopted, with the addition of S.I.M.D. (Single Instruction, Multiple Data) capabilities.

2.4.1 Increasing floating-point and vector capabilities

The presence of S.I.M.D. units provides a CPU with the ability to perform the same operation on multiple data points simultaneously, in fact resorting to multiple deeply connected processing elements. Such technique exploits data level parallelism, but not instruction concurrency: there are simultaneous calculations, but only a single instruction at a given moment is executed (for each SIMD unit). These additional instructions allow to improve the speed of execution of algorithms based on linear algebra techniques, or more generally that require the execution of uniform actions on the elements of an array, acting along at least those directions:

- more than one data is processed with a single instruction, making it possible to increase the actual ILP (Instruction Level Parallelism), moreover in a predictable way;
- conversely, because they allow to reduce the number of machine language instructions necessary to encode an algorithm (static instruction count) that solve a problem or compute a function, they increase the code density and thus they improve the instruction cache hit rate, i.e. the percentage of instruction fetches satisfied by cache hits (during instruction fetch stage) over a given time interval;
- the actual number of machine language instructions that must be executed by the CPU for the task (i.e., dynamic instruction count) decreases, this virtually corresponds to actually increasing the out of order execution windows size, in order to manage data flow analysis more effectively and to resolve data dependency issues;
- since the data are packaged in larger blocks, there are advantages related to the use of better alignment for the data structures, such as better exploitation of the modern memories' burst transfer modes are obtained and even the effectiveness of hardware data prefetching techniques is augmented.

In x86 world, MMX was the first SIMD ISA extension, developed by Intel, introduced in 1997 first in its P55C ("Pentium with MMX Technology"), then in AMD K6 (evolution

of Nexgen Nx686), Intel Klamath (first Pentium II), Cyrix 6x86MX ("M2"). Although officially MMX is not an acronym, it was often considered an abbreviation of MultiMedia eXtension, Multiple Math eXtension or Matrix Math eXtension, even on some old Intel's informative slides. Since it was the first set of SIMD instruction set adopted also on CPUs intended for the mass market, MMX suffered from considerable limitations: it only supports integer data types and, to avoid compatibility problems with the context switch mechanisms in existing operating systems (such as Chicago, alias Windows 95, Detroit, alias Windows 95B or OSR2, and Cairo, aka Windows NT 4.0), reuses the registers ST(0) to ST(7) already reserved to the x87 co-processor (NPU or FPU, i.e. Numeric or Floating-point Processing Unit). To reuse x87 FPU registers, MMX is forced to define eight 64-bit registers, called MM0 through MM7, which are used as aliases for the preexisting x87 registers.

Obviously, MMX defined 57 new operative codes for instructions that use those registers, most of which operate in a single clock. MMX supports four different data types: an eight-byte array (8x8 bits), a four-word array (4x16 bits), a two element double-word array (2x32 bits), and a quad-word object (1x64 bits), so that each MMX register can hold one of these four data types. Anyway, most operations are only supported on bytes, words and double-words: arithmetic operations (additions, subtractions, multiplications; "multiply and add", but only to multiply signed 16-bit words and then add the 32-bit results) and comparison instructions fall into this category. The packed logic (AND, OR, XOR, AND NOT) instructions, instead, are some examples of MMX instructions that actually operate on quad-word 64-bit values. Of course, there is no need for distinct packed byte, packed word, or packed double word versions of these bit-wise instructions, since they would all be equivalent to the provided 64-bit logic instruction. Shift instructions operate on word, double word, and quad word operands, but it is not provided a version of these instructions that operate on bytes. MMX includes, also, conversion instructions to pack and unpack data elements and data movement instructions.

In 1998, AMD introduced "Chompers", alias K6-2 or K6-3D, with "3DNow!", the first SIMD floating-point extension to x86 instruction set, which included 21 new instructions and enabled to perform vector processing on non-integer data, improving the performance of many graphic-intensive applications, at a time when 3D graphics were becoming mainstream in PC multimedia and games, but graphics controllers left the entire geometry computations to the main CPU, before hardware transform and lighting unit were introduced.

While SIMD instructions, such as MMXs, introduced by Intel have always found high support from compiler developers (moreover, Intel has historically developed excellent compilers for Fortran and C/C++, obviously optimized for its products) and they sooner or later have also been adopted by AMD CPUs, 3DNow! have never been supported by Intel, so this instruction set never gained much popularity with software developers and on August 2010 AMD announced that support for 3DNow would be dropped in its future processors, such as Bulldozer, Bobcat and Zen. As an enhancement to the MMX instruction set, the 3DNow instruction-set used the eight MMX SIMD registers to support common arithmetic operations (addition, subtraction, multiplication, but even max and min, reciprocal, square roots, reciprocal square root) on packed single-precision 32-bit floating-point data. The 3DNow instruction set also included operations for SIMD integer operations, data prefetch, and faster MMX-to-floating-point switching: 3DNow! allowed programs to mix integer code (MMX) and floating point code (3DNow!) at the same time without needing to switch context (necessary, however, to issue x87 instructions to conventional FPU).

In the execution of appropriate code sequences, the 3DNow instructions allowed a high performance increase, because they allowed to reach a peak speed of 2 additions

and 2 multiplications per cycle (but it cannot sustain that rate, because there were only two "simple" decoders), while with x87 code the K6 throughput is just 0.5 instructions for cycle; 3DNow usually raised the speed by about 2–4 times. AMD K7 (Athlon) CPU, instead, could sustain the peak of 2 additions and 2 multiplications per cycle, because it had three powerful symmetrical decoders, a beefier out of order execution engine, bigger caches and faster bus.

In response to the introduction of the 3DNow! instruction set, Intel, which was already considering the adoption of SIMD instructions capable of operating on floating-point data, in particular on on single precision data, designed the KNI (Katmai New Instructions), so called because Katmai, the first Pentium III, would have been the first microprocessor with such instructions. Apart from the controversial PSN (Personal Serial Number), the new instructions were the only difference between Katmai and Deschutes in 1999: to emphasize the usefulness of these new instructions, the KNI were commercially called ISSE (Internet Streaming SIMD Extensions), underlining their role to open the door and enter a new world, characterized by internet connectivity and video streams. Later ISSE was renamed SSE; anyway, it contained 70 instructions, most of which work on single precision floating point data: because it supports floating point math, it had wider applications than MMX and became more popular. Unlike MMX and 3DNow! extensions, which occupy the same register space as the normal FPU registers, SSE adds a separate register space to the microprocessor: Katmai added eight new 128-bit registers known as XMM0 through XMM7, four years later AMD K8's x86-64 doubled this space, adding a further eight registers XMM8 through XMM15 (this extension was backported to the Intel 64 architecture sush as Prescott with Yamhill project and CT, Clackamas Tecnology). Because of this, SSE can only be used on operating systems that preserve XMM registers during context switching. SSE used only a single data type for XMM registers: four 32-bit single-precision floating point numbers.

Therefore the main advantage of SSE compared to 3DNow! consisted in supporting 128-bit vectors, on which to act with vector or scalar instructions; however Katmai and subsequent CPU (until Conroe) divided each 128-bit operation into a pair of 64-bit operations (one ADD, one MUL), so that the maximum overall throughput was identical to that of 3DNow! solutions. One advantage of 3DNow, instead, is that it is possible to add or multiply the two numbers that are stored in the same register: this capability, known as horizontal computation, was the major addition to the SSE3 instruction set in 2004.

The AMD K7 Athlon introduced Extended 3DNow!: it added 5 new 3DNow instructions to boost DSP and 19 new MMX instructions (a subset of SSE instructions: all operations that not used XMM registers). Later, the K6-2+ and K6-III+ mobile CPU included only the 5 new 3DNow! instructions, leaving out the 19 new MMX instructions. These instructions provide some enhanced conversion and selection instructions, as well as some advanced cache management instructions.

In 2001, AMD Palomino (Athlon XP) was the first CPU to carry the 3DNow! Professional instruction set, i.e. Extended 3DNow! plus the complete SSE instruction set; on Palomino (and subsequent K7 class cores) and K8 CPU the speed of execution of vectorial operations with 3DNow and SSE is identical, but with SSEs a double number of values can be kept in the registers (or quadruple, in 64-bit mode); in principle, it is possible to combine 3DNow and SSE instructions to further reduce register pressure (to hold 48 single precision values, instead of 16 or 32, in 32-bit mode), but in practice it is difficult to find an appreciable speedup due to the instructions executing on shared functional units (anyway, the use of 3DNow, already very limited before Palomino's support of the SSE, was later completely shelved).

In 2000, Willamette New Instructions (WNI), introduced with the first Pentium 4, was released as SSE2 and it included 144 new instructions. It was a major enhancement

to SSE because it allowed considerably more flexibility in vector processing:

- it expanded the usage of the XMM registers to include, in addition to four 32-bit single precision floating-point: two 64-bit double-precision floating point numbers, two 64-bit integers, four 32-bit integers, eight 16-bit short integers, sixteen 8-bit bytes or characters;
- it introduced MMX-like integer operations on 128-bit XMM registers, so to make MMX largely redundant (though further performance increases theoretically can be attained in some circumstances by using MMX in parallel with SSE operations; on the other hand, another advantage of replacing MMX with SSE2 is avoiding the mode switching penalty for issuing x87 instructions present in MMX because it is sharing register space with the x87 FPU);
- it supported greater precision in the computation of numerical algorithms, by virtue of the adoption of the double precision format;
- it offered a complete set of instructions for dealing with all common data types;
- it included a set of cache control instructions, intended to minimize cache pollution when processing big streams of information.

Only in 2003, AMD included support for SSE2 instruction, starting with 64-bit K8 core: while Pentium 4 needed SSE2 instructions to achieve maximum performance operating in double precision, K8 achieved exactly the same peak and sustaind performance with x87 or SSE2 instructions (however, with the latter, it could take advantage of more registers space: for a double number of values in 32-bit mode and quadruple in 64-bit mode).

SSE3, alias PNI (Prescott New Instructions), was introduced by Intel in early 2004 with the third Pentium 4 core and it was an incremental upgrade to SSE2, beacause it added only 13 new instructions. SSE3 allowed to add or subtract two numbers that are stored in the same register (horizontal operation), which wasn't possible in SSE and SSE2, but only in 3DNow!, it supported a misaligned integer vector load instruction that has better performance for loads that cross cacheline boundaries, it introduced FISTTP (which allows to convert a floating-point value to integer by truncating, without having to change the status word) and it added also a couple process control instructions (MONITOR, MWAIT) to boost performance with Intel's HyperThreading Technology. AMD started supporting SSE3 in April of 2005, with K8 rev. E (Venice and San Diego), omitting these thread instructions, which are only useful for HyperThreading CPUs. The most notable change is just the capability to work horizontally in a register, as opposed to the vertical operation of all previous SSE instructions. These new instructions be used, for example, to simplify and speed up the implementation of scalar products. The graphic data are typically organized as AOS (Arrays Of Structures), which are not easily manipulated with the vertical SSE/SSE2 instructions, which lend themselves naturally to the management of SOA (Structure of Arrays). Previously, in order to take advantage of the SSE/SSE2 vector capabilities, it was therefore necessary to resort to auxiliary operations of shuffling and packaging of the XMM registers, while the new instructions eliminate this task, effectively reducing the number of instructions to be performed to obtain the same result. In essence it is a way of increasing the IPC that goes through the recompilation of the code with new optimized compilers; in an example provided by Intel, the scalar product with vectors in single-precision involves a threefold reduction in the number of instructions.

In May 2004, the Tejas CPU project was erased, but Tejas New Instructiosn (TNI) converged to Merom New Instructions (MNI) and as such they were introduced as SSSE (Supplemental SSE3) on June 26, 2006 with the "Woodcrest" Xeons. SSSE3 added 16

new instructions to accelerate computations on packed integers, which included horizontal addition or subtraction operations, absolute value, permuting the bytes in a word, multiplying 16-bit fixed-point numbers with correct rounding and within-word accumulate instructions, align data from the composite of two operands; because each instruction can act on 64-bit MMX or 128-bit XMM registers, they can be counted as 32 instructions.

Penryn New Instructions (PNI) was announced at the Fall 2006 Intel Developer Forum on September 27, 2006 (contextually to the announcement of the end of the development of SSE instructions, reserving the right to introduce only ATAs, Application Targeted Accelerators) then more precise details were provided at Spring 2007 IDF, but it was released on fall 2007 with SSE4.1 name: it was another enhancement to SSE, adding 47 new instructions, including a dot product instruction, additional integer instructions, a popcnt instruction. It features a number of instructions whose action is determined by a constant field and a set of instructions that take XMM0 as an implicit third operand; several of these instructions were enabled by the new single-cycle shuffle engine in Penryn. SSE4.1 became famous because it made it possible to improve the speed of encoding in very popular, at the time, Divx format, so Intel coined the marketing term HD Boost to refer to them. One of the stages of the video encoding process consists of motion estimation, i.e. identifying the differences between consecutive frames due to the movement of peoples and objects; it requires a lot of computation of sums of absolute differences, as well as finding the minimum values of the results of those computations. The SSE2 instruction PSADBW can compute two sums of differences from a pair of 16B unsigned integers; the SSE4 instruction MPSADBW can do eight; according to Intel, the same full search algorithm for motion estimation can take 71 cycles employing the SSE2 code path, compared to only 26 cycles using the SSE4 version (for each 16x16 pixel block).

AMD Barcelona (K10) in late 2007 supported only a small subset of SSE4.1 (the full SSE4 instruction set was supported with Bulldozer in 2011), but it added some instructions for bit manipulation, under the name SSE4a. In particular, SSE4a included POPCNT and LZCNT, instructions that came under the name Advanced Bit Manipulation (ABM) and which operate on integer rather than SSE registers. Morever, K10 introduced improvements for mis-aligned accesses; Intel later introduced similar speed improvements to unaligned SSE in their Nehalem processors, but it did not introduce misaligned access by non-load SSE instructions until Sandy Bridge.

The subsequent extension was constituted by the NNI, Nehalem New Instructions, released in 2008 as SSE4.2: seven instructions, considered as Application Targeted Accelerators. Indeed, Intel stated that the feedback received from the developers played a crucial role in developing this new instruction set. SSE4.2 added STTNI (String and Text New Instructions), several new instructions that perform character searches and comparison on two operands of 16 bytes at a time, instructions useful to speed up text processing, such as the parsing of XML documents. SSE4.2 added also a CRC32 instruction to compute cyclic redundancy checks as used in certain data transfer protocols, to speed up data integrity checking algorithms. Intel implemented also POPCNT with SSE4.2, postponing LZCNT support to Haswell era.

in March of 2008, WMI (Westmere New Instructions) was announced, a set of 7 instructions that assist with encryption and decryption with popular AES algorithm. In 2010, Westmere CPU introduced them under the name AES New Instructions (AES-NI).

After a stormy period, with AMD's proposal for a set of SSE5 instructions, with support for FMA3 (Fused Multiply-Add 3 operands) on 128-bit registers, and an AVX set by Intel, considered more advanced because it supported new 256-bit registers and 4-operand FMA4, the market has seen the progressive adoption of the AVX set, but with FMA3.

Gesher New Instructions (GNI), Advanced Vector Extensions (AVX) The AVX in-

struction set was announced by Intel in March of 2008. It departs from Intel's usual instruction encoding form in that it allows 3-operand instructions. It's also intended to address growing register sized in the future, as SIMD widths increase. Initially, plans are for 16 256-bit registers, but it also extends to 512-bit registers with AVX-512. Whereas SSE registers are called XMM0-XMM7, AVX's registers are called YMM0-YMM15. The XMM registers map to the bottom half of each of the larger YMM registers.

, is an advanced version of SSE announced by Intel featuring a widened data path from 128 bits to 256 bits and 3-operand instructions (up from 2). Intel released processors in early 2011 with AVX support.[5] AVX requires support from the operating system. AVX2 is an expansion of the AVX instruction set. All CPUs since AMD Carrizo or Intel Haswell support AVX2. AVX-512 (3.1 and 3.2) are 512-bit extensions to the 256-bit Advanced Vector Extensions SIMD instructions for x86 instruction set architecture.

Advanced Vector Extensions (AVX, also known as Sandy Bridge New Extensions) are extensions to the x86 instruction set architecture for microprocessors from Intel and AMD proposed by Intel in March 2008 and first supported by Intel with the Sandy Bridge[1] processor shipping in Q1 2011 and later on by AMD with the Bulldozer[2] processor shipping in Q3 2011. AVX provides new features, new instructions and a new coding scheme. AVX2 expands most integer commands to 256 bits and introduces fused multiply-accumulate (FMA) operations. AVX-512 expands AVX to 512-bit support using a new EVEX prefix encoding proposed by Intel in July 2013 and first supported by Intel with the Knights Landing processor, which shipped in 2016

AVX uses sixteen YMM registers to perform a Single Instruction on Multiple pieces of Data (see SIMD). Each YMM register can hold and do simultaneous operations (math) on: eight 32-bit single-precision floating point numbers or four 64-bit double-precision floating point numbers. The width of the SIMD registers is increased from 128 bits to 256 bits, and renamed from XMM0–XMM7 to YMM0–YMM7 (in x86-64 mode, from XMM0–XMM15 to YMM0–YMM15). The legacy SSE instructions can be still utilized via the VEX prefix to operate on the lower 128 bits of the YMM registers. AVX introduces a three-operand SIMD instruction format, where the destination register is distinct from the two source operands. For example, an SSE instruction using the conventional two-operand form a = a + b can now use a non-destructive three-operand form c = a + b, preserving both source operands. AVX's three-operand format is limited to the instructions with SIMD operands (YMM), and does not include instructions with general purpose registers (e.g. EAX). Such support will first appear in AVX2.[5] The alignment requirement of SIMD memory operands is relaxed.[6] The new VEX coding scheme introduces a new set of code prefixes that extends the opcode space, allows instructions to have more than two operands, and allows SIMD vector registers to be longer than 128 bits. The VEX prefix can also be used on the legacy SSE instructions giving them a three-operand form, and making them interact more efficiently with AVX instructions without the need for VZEROUPPER and VZEROALL. The AVX instructions support both 128-bit and 256-bit SIMD. The 128-bit versions can be useful to improve old code without needing to widen the vectorization, and avoid the penalty of going from SSE to AVX, they are also faster on some early AMD implementations of AVX. This mode is sometimes known as AVX-128

AVX-512 are 512-bit extensions to the 256-bit Advanced Vector Extensions SIMD instructions for x86 instruction set architecture proposed by Intel in July 2013, and are supported with Intel's Knights Landing processor.[3] AVX-512 instruction are encoded with the new EVEX prefix. It allows 4 operands, 7 new 64-bit opmask registers, scalar memory mode with automatic broadcast, explicit rounding control, and compressed displacement memory addressing mode. The width of the register file is increased to 512 bits and total register count increased to 32 (registers ZMM0-ZMM31) in x86-64 mode.

AVX-512 consists of multiple extensions not all meant to be supported by all processors implementing them. AVX-512 are 512-bit extensions to the 256-bit Advanced Vector Extensions SIMD instructions for x86 instruction set architecture (ISA) proposed by Intel in July 2013, and implemented in Intel's Xeon Phi x200 (Knights Landing)[1] and Skylake-X CPUs; this includes the Core-X series (excluding the Core i5-7640X and Core i7-7740X), as well as the new Xeon Scalable Processor Family and Xeon D-2100 Embedded Series.[2] AVX-512 is not the first 512-bit SIMD instruction set that Intel has introduced in processors: the earlier 512-bit SIMD instructions used in the first generation Xeon Phi coprocessors, derived from Intel's Larrabee project, are similar but not binary compatible and only partially source compatible. AVX-512 consists of multiple extensions that may be implemented independently. This policy is a departure from the historical requirement of implementing the entire instruction block. Only the core extension AVX-512F (AVX-512 Foundation) is required by all AVX-512 implementations.

The VEX prefix used by AVX and AVX2, while flexible, did not leave enough room for the features Intel wanted to add to AVX-512. This has led them to define a new prefix called EVEX.

Compared to VEX, EVEX adds the following benefits:[6] Expanded register encoding allowing 32 512-bit registers. Adds 8 new opmask registers for masking most AVX-512 instructions. Adds a new scalar memory mode that automatically performs a broadcast. Adds room for explicit rounding control in each instruction. Adds a new compressed displacement memory addressing mode. The extended registers, SIMD width bit, and opmask registers of AVX-512 are mandatory and all require support from the OS. SIMD modes The AVX-512 instructions are designed to mix with 128/256-bit AVX/AVX2 instructions without a performance penalty. However, AVX-512VL extensions allows the use of AVX-512 instructions on 128/256-bit registers XMM/YMM, so most SSE and AVX/AVX2 instructions have new AVX-512 versions encoded with the EVEX prefix which allow access to new features such as opmask and additional registers. Unlike AVX-256, the new instructions do not have new mnemonics but share namespace with AVX, making the distinction between VEX and EVEX encoded versions of an instruction ambiguous in the source code. Since AVX-512F only works on 32- and 64-bit values, SSE and AVX/AVX2 instructions that operate on bytes or words are available only with the AVX-512BW extension (Byte & Word support).

RISC-V vector extensions are finally nearing approval. They impose almost no limits on the size or number of data elements, and they allow mixed-width elements to execute in the same instruction stream with fixed-width elements and general-purpose instructions. They can operate on all data types, and compiled binaries will run on any implementation. CPU designs now under way range from microcontroller-class cores with 32-bit vectors to supercomputer-class cores with 16,384-bit vectors.

The 32-bit RISC-V Vector (RVV) specification will soon advance from v0.8 to v0.9. (The same instruction set will work with future 64- and 128-bit encodings. The V-extension task group expects to propose the v1.0 specification by the end of June, kicking off a 45-day review for final comments. Assuming no serious objections, the community board could adopt the spec in August. RISC-V vendors can then begin shipping production RTL for CPUs with the extensions. SiFive already has three cores in development.

RVV extensions depart from the single-instruction, multiple-data (SIMD) extensions for proprietary CPUs, which fix in hardware the vector widths and numbers of elements (lanes). As the need for more parallelism has grown, SIMD extensions have expanded to 512 bits or more. Each iteration adds dozens or hundreds of new instructions, and newer code usually won't run on older implementations.

By contrast, RVV extensions can adapt to different microarchitectures at run time. They target diverse CPU designs while maintaining binary compatibility, although code

compiled for a specific implementation will run faster. Programmers can write code in assembly language or in a high-level language by using intrinsic functions or a vectorizing compiler. The main competitors are Intel's AVX-512 and Arm's Scalable Vector Extension (SVE), but both are more conventional than RVV.

2.4.2 From dual core to multi core and GPU

A parallel system contains more than one processor having direct memory access to the shared memory that can form a common address space. Usually, a parallel system is of a Uniform Memory Access (UMA) architecture. In UMA architecture, the access latency (processing time) for accessing any particular location of a memory from a particular processor is the same. Moreover, the processors are also configured to be in a close proximity and are connected in an interconnection network. Conventionally, the interprocess processor communication between the processors is happening through either read or write operations across a shared memory, even though the usage of the message-passing capability is also possible (with emulation on the shared memory). Moreover, the hardware and software are tightly coupled, and usually, the processors in such network are installed to run on the same operating system. In general, the processors are homogeneous and are installed within the same container of the shared memory.

A multicomputer parallel system is another type of parallel system containing multiple processors configured without having a direct accessibility to the shared memory. Moreover, a common address space may or may not be expected to be formed by the memory of the multiple processors.

A multicomputer system in a Non-Uniform Memory Access (NUMA) architecture is usually configured with a common address space. In such NUMA architecture, accessing different memory locations in a shared memory across different processors shows different latency times.

Array processor exchanges information by passing as messages. Array processors have a very small market owing to the fact that they can perform closely synchronized data processing, and the data is exchanged in a locked event for applications such as digital signal processing and image processing. Such applications can also involve large iterations on the data as well.

Compared to the UMA and array processors architecture, NUMA as well as message-passing multicomputer systems are less preferred if the shared data access and communication much accepted. The primary benefit of having parallel systems is to derive a better throughput through sharing the computational tasks between multiple processors. The tasks that can be partitioned into multiple subtasks easily and need little communication for bringing synchronization in execution are the most efficient tasks to execute on parallel systems. The subtasks can be executed as a large vector or an array through matrix computations, which are common in scientific applications. Though parallel computing was much appreciated through research and was beneficial on legacy architectures, they are observed no more efficient/economic in recent times due to following reasons:

However, Amdahl's law is applicable only to scenarios where the program is of a fixed size. In general, on larger problems (larger datasets), more computing resources tend to get used if they are available, and the overall processing time in the parallel part usually improves much faster than the by default serial parts.

802.3ba is the designation given to the higher speed Ethernet task force which completed its work to modify the 802.3 standard to support speeds higher than 10 Gbit/s in

year	process (nm)	Product and codename	Instruction Set	100% ADD	100% MUL	50+50% Add/Mul	100% FMA
1995	350	Pentium Pro P6	x87	1	0.5	1	N/A
1997	350	AMD K6 (Nx686)	x87	0.5	0.5	0.5	N/A
1997	350	Pentium II Klamath	x87	1	0.5	1	N/A
1998	250	AMD K6-3D Chomper	3DNow!	2	2	4	N/A
1999	250	Pentium III Katmai	SSE	2	2	4	N/A
1999	250	AMD Athlon K7	x87	1	1	2	N/A
1999	250	AMD Athlon K7	3DNow!	2	2	4	N/A
2000	180	Pentium 4 Willamette	x87	1	0.5	1	N/A
2000	180	Pentium 4 Willamette	SSE	2	2	4	N/A
2001	180	AMD Athlon XP Palomino	SSE	2	2	4	N/A
2003	130	AMD Athlon 64 K8 Hammer	x87	1	1	2	N/A
2003	130	AMD Athlon 64 K8 Hammer	SSE	2	2	4	N/A
2006	65	Core 2 Duo Conroe	SSE	4	4	8	N/A
2007	65	AMD Phenom K10 Agena	SSE	4	4	8	N/A
2008	45	Core i7 Nehalem	SSE	4	4	8	N/A
2011	32	Core i7 Sandy Bridge	SSE	4	4	8	N/A
2011	32	Core i7 Sandy Bridge	AVX	8	8	16	N/A
2011	32	AMD FX Bulldozer	x87	2	2	2	N/A
2011	32	AMD FX Bulldozer	SSE	8	8	8	N/A
2011	32	AMD FX Bulldozer	AVX	8	8	8	16
2013	22	Core i7 Haswell	AVX	16	16	16	32
2017	14	AMD Ryzen Zen	SSE	8	8	16	N/A
2017	14	AMD Ryzen Zen	AVX	8	8	16	16
2017	14	Core i9 Skylake-X	AVX512	32	32	32	64
2017	7	AMD Ryzen 3 Zen2	AVX	16	16	32	32
2019	10	Core i7 Ice-Lake	AVX512	16	16	16	32

Table 2.2: This table shows some Intel and AMD flotating-point maximum throughput (in IEEE-754 single precision format) for each clock cycle, using common mix of basic arithmetic operations: all additions (or subtraction, nothing changes); all multiplications; one half additions/subtractions and half multiplications; special FMA (Fused Multiply Add) operations (more complex operations, such as reciprocal, division, square root and transcendent functions are more computational expensive, they are often not even fully pipelined, so that very limited use is made of them in critical paths code). (N/A means Not Applicable, because only some recent CPU had FMA units)

year	process (nm)	Product and codename	Instruction Set	100% ADD	100% MUL	50+50% Add/Mul	100% FMA
1995	350	Pentium Pro P6	x87	1	0.5	1	N/A
1997	350	AMD K6 (Nx686)	x87	0.5	0.5	0.5	N/A
1997	350	Pentium II Klamath	x87	1	0.5	1	N/A
1999	250	AMD Athlon K7	x87	1	1	2	N/A
2000	180	Pentium 4 Willamette	x87	1	0.5	1	N/A
2000	180	Pentium 4 Willamette	SSE2	1	1	2	N/A
2003	130	AMD Athlon 64 K8 Hammer	x87	1	1	2	N/A
2003	130	AMD Athlon 64 K8 Hammer	SSE2	1	1	2	N/A
2006	65	Core 2 Duo Conroe	SSE2	2	2	4	N/A
2007	65	AMD Phenom K10 Agena	SSE2	2	2	4	N/A
2008	45	Core i7 Nehalem	SSE2	2	2	4	N/A
2011	32	Core i7 Sandy Bridge	SSE2	2	2	4	N/A
2011	32	Core i7 Sandy Bridge	AVX	4	4	8	N/A
2011	32	AMD FX Bulldozer	x87	2	2	2	N/A
2011	32	AMD FX Bulldozer	SSE2	4	4	4	N/A
2011	32	AMD FX Bulldozer	AVX	4	4	4	8
2013	22	Core i7 Haswell	AVX	8	8	8	16
2017	14	AMD Ryzen Zen	SSE2	4	4	8	N/A
2017	14	AMD Ryzen Zen	AVX	4	4	8	8
2017	14	Core i9 Skylake-X	AVX512	16	16	16	32
2017	7	AMD Ryzen 3 Zen2	AVX	8	8	16	16
2019	10	Core i7 Ice-Lake	AVX512	8	8	8	16

Table 2.3: This table shows some Intel and AMD flotating-point maximum throughput (in IEEE-754 double precision format) for each clock cycle, using common mix of basic arithmetic operations. Same remarks as in Tab. 2.2

2010 The standard was announced in July 2007[86] and was ratified on June 17, 2010 IEEE 802.3's 400 Gb/s Ethernet Study Group started working on the 400 Gbit/s generation standard in March 2013. Results from the study group were published and approved on March 27, 2014. Accordingly, at the IEEE Industry Connections Higher Speed Ethernet Consensus group meeting in September 2012, 400 GbE was chosen as the next generation goal. Additional 200GbE objectives were added in January 2016. In 2016, several networking equipment suppliers were already offering proprietary solutions for 200G and 400G. 400 Gigabit Ethernet (400G, 400GbE) and 200 Gigabit Ethernet (200G, 200GbE) standards developed by the IEEE P802.3bs Task Force using broadly similar technology to 100 Gigabit Ethernet were approved on December 6, 2017. Distributed computing is the concurrent usage of more than one connected computer to solve a problem over a network connection. The computers that take part in distributed computing appear as single machines to their users.

Distributing computation across multiple computers is a great approach when these computers are observed to interact with each other over the distributed network to solve a bigger problem in reasonably less latency. In many respects, this sounds like a generalization of the concepts of parallel computing

While both distributed computing and parallel systems are widely available these days, the main difference between these two is that a parallel computing system consists of multiple processors that communicate with each other using a shared memory, whereas a distributed computing system contains multiple processors connected by a communication network. In parallel computing systems, as the number of processors increases, with enough parallelism available in applications, such systems easily beat sequential systems in performance through the shared memory. In such systems, the processors can also contain their own locally allocated memory, which is not available to any other processors.

In distributed computing systems, multiple system processors can communicate with each other using messages that are sent over the network. Such systems are increasingly available these days because of the availability at low price of computer processors and the high-bandwidth links to connect them.

The following reasons explain why a system should be built distributed, not just parallel: Scalability: As distributed systems do not have the problems associated with shared memory, with the increased number of processors, they are obviously regarded as more scalable than parallel systems. Reliability: The impact of the failure of any single subsystem or a computer on the network of computers defines the reliability of such a connected system. Definitely, distributed systems demonstrate a better aspect in this area compared to the parallel systems. Data sharing: Data sharing provided by distributed systems is similar to the data sharing provided by distributed databases. Thus, multiple organizations can have distributed systems with the integrated applications for data exchange. Resources sharing: If there exists an expensive and a special purpose resource or a processor, which cannot be dedicated to each processor in the system, such a resource can be easily shared across distributed systems. Heterogeneity and modularity: A system should be flexible enough to accept a new heterogeneous processor to be added into it and one of the processors to be replaced or removed from the system without affecting the overall system processing capability. Distributed systems are observed to be more flexible in this respect. Geographic construction: The geographic placement of different subsystems of an application may be inherently placed as distributed. Local processing may be forced by the low communication bandwidth more specifically within a wireless network. Economic: With the evolution of modern computers, high-bandwidth networks and workstations are available at low cost, which also favors distributed computing for economic reasons.

An important and key feature of distributed computing and the message-passing model

of communication is having no shared memory, which also infers the nonexistence of a common physical clock. The distributed system processors are loosely coupled so that they have their own individual capabilities in terms of speed and method of execution with versatile operating systems. They are not expected to be part of a dedicated system; however, they cooperate with one another by exposing the services and/or executing the tasks together as subtasks.

2.4.3 Low precision data formats

Another element that highlights the union between the hardware development of computation "agents" and the increasingly pervasive diffusion of machine learning techniques in recent years is the possibility of operating efficiently with reduced machine precision.

Do not use too much precision when it is not necessary. Single precision (32-bits) is faster on some operations and consumes only half the memory space as double precision (64-bits) or double extended (80-bits).

FPU (x87) instructions provide higher precision by calculating intermediate results with 80 bits of precision, by default, to minimise roundoff error in numerically unstable algorithms (see IEEE 754 design rationale and references therein). However, the x87 FPU is a scalar unit only whereas SSE2 can process a small vector of operands in parallel.

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point arithmetic established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating-point implementations that made them difficult to use reliably and portably. Many hardware floating-point units use the IEEE 754 standard.

The standard defines: arithmetic formats: sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values (NaNs) interchange formats: encodings (bit strings) that may be used to exchange floating-point data in an efficient and compact form rounding rules: properties to be satisfied when rounding numbers during arithmetic and conversions operations: arithmetic and other operations (such as trigonometric functions) on arithmetic formats exception handling: indications of exceptional conditions (such as division by zero, overflow, etc.) In 1976 Intel began planning to produce a floating point coprocessor. John Palmer, the manager of the effort, persuaded them that they should try to develop a standard for all their floating point operations. William Kahan was hired as a consultant; he had helped improve the accuracy of Hewlett-Packard's calculators. Kahan initially recommended that the floating point base be decimal[14] but the hardware design of the coprocessor was too far along to make that change. The work within Intel worried other vendors, who set up a standardization effort to ensure a 'level playing field'. Kahan attended the second IEEE 754 standards working group meeting, held in November 1977. Here, he received permission from Intel to put forward a draft proposal based on the standard arithmetic part of their design for a coprocessor. The arguments over gradual underflow lasted until 1981 when an expert hired by DEC to assess it sided against the dissenters. Even before it was approved, the draft standard had been implemented by a number of manufacturers.[15][16] The Intel 8087, which was announced in 1980, was the first chip to implement the draft standard. IEEE 754-1985 was an industry standard for representing floating-point numbers in computers, officially adopted in 1985 and superseded in 2008 by IEEE 754-2008, and then again in 2019 by minor revision IEEE 754-2019. During its 23 years, it was the most widely used format for floating-point computation. It was implemented in software, in the form of floating-point libraries, and in hardware, in the instructions of many CPUs and FPUs. The first integrated circuit to implement the draft of what was to become IEEE 754-1985 was the Intel 8087. The standard also recommends extended format(s) to be used to perform

internal computations at a higher precision than that required for the final result, to minimise round-off errors: the standard only specifies minimum precision and exponent requirements for such formats. The x87 80-bit extended format is the most commonly implemented extended format that meets these requirements.

IEEE 754-1985 represents numbers in binary, providing definitions for four levels of precision, of which the two most commonly used are: IEEE 754-2008, published in August 2008, includes nearly all of the original IEEE 754-1985 standard, plus the IEEE 854-1987 Standard for Radix-Independent Floating-Point Arithmetic. The current version, IEEE 754-2019, was published in July 2019. It is a minor revision of the previous version, incorporating mainly clarifications, defect fixes and new recommended operations. The standard defines five basic formats that are named for their numeric base and the number of bits used in their interchange encoding. There are three binary floating-point basic formats (encoded with 32, 64 or 128 bits) and two decimal floating-point basic formats (encoded with 64 or 128 bits). The binary32 and binary64 formats are the single and double formats of IEEE 754-1985 respectively The standard specifies optional extended and extendable precision formats, which provide greater precision than the basic formats.[12] An extended precision format extends a basic format by using more precision and more exponent range. An extendable precision format allows the user to specify the precision and exponent range. An implementation may use whatever internal representation it chooses for such formats; all that needs to be defined are its parameters (b, p, and emax). These parameters uniquely describe the set of finite numbers (combinations of sign, significand, and exponent for the given radix) that it can represent.

The standard recommends that language standards provide a method of specifying p and emax for each supported base b.[13] The standard recommends that language standards and implementations support an extended format which has a greater precision than the largest basic format supported for each radix b.[14] For an extended format with a precision between two basic formats the exponent range must be as great as that of the next wider basic format. So for instance a 64-bit extended precision binary number must have an 'emax' of at least 16383. The x87 80-bit extended format meets this requirement. IEEE 754-2008 (previously known as IEEE 754r) was published in August 2008 and is a significant revision to, and replaces, the IEEE 754-1985 floating-point standard, while in 2019 it got updated with a minor revision IEEE 754-2019.[1] The 2008 revision extended the previous standard where it was necessary, added decimal arithmetic and formats, tightened up certain areas of the original standard which were left undefined, and merged in IEEE 854 (the radix-independent floating-point standard). In a few cases, where stricter definitions of binary floating-point arithmetic might be performance-incompatible with some existing implementation, they were made optional.

Single-precision floating-point format is a computer number format, usually occupying 32 bits in computer memory; it represents a wide dynamic range of numeric values by using a floating radix point. A floating-point variable can represent a wider range of numbers than a fixed-point variable of the same bit width at the cost of precision.

In the IEEE 754-2008 standard, the 32-bit base-2 format is officially referred to as binary32; it was called single in IEEE 754-1985. IEEE 754 specifies additional floating-point types, such as 64-bit base-2 double precision and, more recently, base-10 representations. One of the first programming languages to provide single- and double-precision floating-point data types was Fortran. Before the widespread adoption of IEEE 754-1985, the representation and properties of floating-point data types depended on the computer manufacturer and computer model, and upon decisions made by programming-language designers. E.g., GW-BASIC's single-precision data type was the 32-bit MBF floating-point format. Single precision is termed REAL in Fortran, SINGLE-FLOAT in Common Lisp, float in C, C++, C#, Java, Float in Haskell, and Single in Object Pascal (Delphi),

Visual Basic, and MATLAB. However, float in Python, Ruby, PHP, and OCaml and single in versions of Octave before 3.2 refer to double-precision numbers. In most implementations of PostScript, and some embedded systems, the only supported precision is single.

Double-precision floating-point format is a computer number format, usually occupying 64 bits in computer memory; it represents a wide dynamic range of numeric values by using a floating radix point. Floating point is used to represent fractional values, or when a wider range is needed than is provided by fixed point (of the same bit width), even if at the cost of precision. Double precision may be chosen when the range or precision of single precision would be insufficient. In the IEEE 754-2008 standard, the 64-bit base-2 format is officially referred to as binary64; it was called double in IEEE 754-1985. IEEE 754 specifies additional floating-point formats, including 32-bit base-2 single precision and, more recently, base-10 representations. One of the first programming languages to provide single- and double-precision floating-point data types was Fortran. Before the widespread adoption of IEEE 754-1985, the representation and properties of floating-point data types depended on the computer manufacturer and computer model, and upon decisions made by programming-language implementers. E.g., GW-BASIC's double-precision data type was the 64-bit MBF floating-point format. Double-precision binary floating-point is a commonly used format on PCs, due to its wider range over single-precision floating point, in spite of its performance and bandwidth cost. As with single-precision floating-point format, it lacks precision on integer numbers when compared with an integer format of the same size. It is commonly known simply as double. The IEEE 754 standard specifies a binary64 as having: Sign bit: 1 bit Exponent: 11 bits Significand precision: 53 bits (52 explicitly stored) Using double-precision floating-point variables and mathematical functions (e.g., sin, cos, atan2, log, exp and sqrt) are slower than working with their single precision counterparts. One area of computing where this is a particular issue is for parallel code running on GPUs. For example, when using NVIDIA's CUDA platform, calculations with double precision take, depending on a hardware, approximately 2 to 32 times as long to complete compared to those done using single precision IEEE 754 quadruple-precision binary floating-point format The IEEE 754 standard specifies a binary128 as having: Sign bit: 1 bit Exponent width: 15 bits Significand precision: 113 bits (112 explicitly stored) In its 2008 revision, the IEEE 754 standard specifies a binary256 format among the interchange formats (it is not a basic format), as having: Sign bit: 1 bit Exponent width: 19 bits Significand precision: 237 bits (236 explicitly stored)

A roundoff error, also called rounding error, is the difference between the result produced by a given algorithm using exact arithmetic and the result produced by the same algorithm using finite-precision, rounded arithmetic.[3] Rounding errors are due to inexactness in the representation of real numbers and the arithmetic operations done with them. This is a form of quantization error.[4] When using approximation equations or algorithms, especially when using finitely many digits to represent real numbers (which in theory have infinitely many digits), one of the goals of numerical analysis is to estimate computation errors.[5] Computation errors, also called numerical errors, include both truncation errors and roundoff errors. When a sequence of calculations with an input involving roundoff error are made, errors may accumulate, sometimes dominating the calculation. In ill-conditioned problems, significant error may accumulate.[6] In short, there are two major facets of roundoff errors involved in numerical calculations: Digital computers have magnitude and precision limits on their ability to represent numbers. Certain numerical manipulations are highly sensitive to roundoff errors. This can result from both mathematical considerations as well as from the way in which computers perform arithmetic operations. Loss of significance is an undesirable effect in calculations using finite-precision arithmetic such as floating-point arithmetic. It occurs when an

operation on two numbers increases relative error substantially more than it increases absolute error, for example in subtracting two nearly equal numbers (known as catastrophic cancellation). The effect is that the number of significant digits in the result is reduced unacceptably. Ways to avoid this effect are studied in numerical analysis.

In computing, half precision is a binary floating-point computer number format that occupies 16 bits (two bytes in modern computers) in computer memory. In the IEEE 754-2008 standard, the 16-bit base-2 format is referred to as binary16. It is intended for storage of floating-point values in applications where higher precision is not essential for performing arithmetic computations. Although implementations of the IEEE Half-precision floating point are relatively new, several earlier 16-bit floating point formats have existed Sign bit: 1 bit Exponent width: 5 bits Significand precision: 11 bits (10 explicitly stored) The hardware-accelerated programmable shading group led by John Airey at SGI (Silicon Graphics) invented the s10e5 data type in 1997 as part of the 'bali' design effort. Nvidia and Microsoft defined the half datatype in the Cg language, released in early 2002, and implemented it in silicon in the GeForce FX, released in late 2002 The half data type makes use of the Partial Precision instruction modifier to request less precision. NVIDIA GPUs may use half-precision floating-point when the Partial Precision instruction modifier is specified. Half-precision floating-point is encoded with a sign bit, 10 mantissa bits, and 5 exponent bits (biased by 16), sometimes called s10e5. float The float data type corresponds to a floating-point representation with at least 24 bits. NVIDIA GPUs supporting vs_2_sw use standard IEEE 754 single-precision floating-point encoding with a sign bit, 23 mantissa bits, and 8 exponent bits (biased by 128), sometimes called s10e5. Older ATI GPUs use 24-bit floating-point.

If codes designed for x87 are ported to the lower precision double precision SSE2 floating point, certain combinations of math operations or input datasets can result in measurable numerical deviation, which can be an issue in reproducible scientific computations, e.g. if the calculation results must be compared against results generated from a different machine architecture. A related issue is that, historically, language standards and compilers had been inconsistent in their handling of the x87 80-bit registers implementing double extended precision variables, compared with the double and single precision formats implemented in SSE2: the rounding of extended precision intermediate values to double precision variables was not fully defined and was dependent on implementation details such as when registers were spilled to memory.

The bfloat16 is a truncated 16-bit version of the 32-bit IEEE 754 single-precision floating-point format that preserves 8 exponent bits, but reduces precision of the significand from 24-bits to 8 bits to save up memory, bandwidth, and processing resources, while still retaining the same range. The bfloat16 format was designed primarily for machine learning and near-sensor computing applications, where precision is needed near to 0 but not so much at the maximum range. The number representation is supported by Intel's upcoming FPGAs as well as Nervana neural network processors, and Google's TPUs. Given the fact that Intel supports the bfloat16 format across two of its product lines, it makes sense to support it elsewhere as well, which is what the company is going to do by adding its AVX512_BF16 instructions support to its upcoming Xeon Scalable 'Cooper Lake-SP' platform.

we actually won't see high core count Ice Lake CPUs for a while due to too low yields on 10nm. The bfloat16 (Brain Floating Point) floating-point format is a computer number format occupying 16 bits in computer memory; it represents a wide dynamic range of numeric values by using a floating radix point. This format is a truncated (16-bit) version of the 32-bit IEEE 754 single-precision floating-point format (binary32) with the intent of accelerating machine learning and near-sensor computing.[1] It preserves the approximate dynamic range of 32-bit floating-point numbers by retaining 8 exponent bits,

but supports only an 8-bit precision rather than the 24-bit significand of the binary32 format. More so than single-precision 32-bit floating-point numbers, bfloat16 numbers are unsuitable for integer calculations, but this is not their intended use.

The bfloat16 format is utilized in Intel AI processors, such as Nervana NNP-L1000, Xeon processors (AVX-512 BF16 extensions), and Intel FPGAs, Google Cloud TPUs, and TensorFlow. ARMv8.6-A also supports the bfloat16 format. As of October 2019, AMD has added support for the format to its ROCm libraries bfloat16 has the following format: Sign bit: 1 bit Exponent width: 8 bits Significand precision: 8 bits (7 explicitly stored), as opposed to 24 bits in a classical single-precision floating-point format

2.4.4 FPGA and ASIC

Evolvable Hardware, implementazioni di reti neurali shallow e deep su FPGA, aritmetica in precisione ridotta, ecc [Chakradhar et al., 2010; Gankidi and Thangavelautham, 2017; Guo et al., 2017; Gupta et al., 2011; Haddow and Tyrrell, 2018; Higuchi et al., 1996; Himavathi et al., 2007; Jeyanthi and Subadra, 2014; Kuon et al., 2008; Lacey et al., 2016; Liang et al., 2018; Liu et al., 2015a; Misra and Saha, 2010; Nazari et al., 2015; Nurvitadhi et al., 2017; Papadimitriou et al., 2011; Park and Sung, 2016; Perko et al., 2000; Schmit and Huang, 2016; Shafique et al., 2017; Sharma et al., 2017; Sipper et al., 1999; Vasicek and Sekanina, 2007; Wang et al., 2017a; Xiao et al., 2017; Yao and Higuchi, 1999; Zhang et al., 2015; Zhou et al., 2017] Achronix Semiconductor Corporation is a fabless semiconductor corporation based in Santa Clara, California, offering high-performance FPGA solutions. Achronix is the only supplier to have both high-performance and high-density standalone FPGAs and embedded FPGA (eFPGA) solutions in high-volume production. Achronix's FPGA and eFPGA IP offerings are further enhanced by ready-to-use PCIe accelerator cards targeting AI, ML, networking and data center applications. All of Achronix's products are supported by best-in-class EDA software tools. In 2019, Achronix announced the revolutionary Speedster7t FPGA product family based on TSMC 7nm FinFET technology — the first FPGA architecture to offer a 2D network-on-chip (NoC) and an array of machine learning processors optimized for AI/ML workloads. Built on TSMC's 7nm FinFET process, Speedster7t FPGAs feature a revolutionary new 2D network-on-chip (NoC), an array of new machine learning processors (MLPs) optimized for high-bandwidth and artificial intelligence/machine learning (AI/ML) workloads, high-bandwidth GDDR6 interfaces, 400G Ethernet and PCI Express Gen5 ports — all interconnected to deliver ASIC-level performance while retaining the full programmability of FPGAs. The Speedster7t FPGA family is optimized for high-bandwidth workloads and eliminates the performance bottlenecks associated with traditional FPGAs. Built on TSMC's 7nm FinFET process, Speedster7t FPGAs feature a revolutionary new 2D network-on-chip (NoC), an array of new machine learning processors (MLPs) optimized for high-bandwidth and artificial intelligence/machine learning (AI/ML) workloads, high-bandwidth GDDR6 interfaces, 400G Ethernet and PCI Express Gen5 ports — all interconnected to deliver ASIC-level performance while retaining the full programmability of FPGAs. Get started today with the VectorPath accelerator card, featuring the Speedster7t FPGA. 2D Networ on chip 20Tbps bandwidth, 4 Tbps 16 GDDR6 channels bandwidth, 385 Mb on chip memory, 2.6 Million 6-input luts

Chapter 3

Support Vector Machine

Support Vector Machines are supervised machine learning models, that can be used both for classification tasks and regression analysis.

In machine learning models used for regression or classification, such as neural networks, the form of the mapping from an input value, ordinarily encoded in a feature vector, to desired output is governed by a set, typically seen as a list or often as a vector, of adaptive parameters, such as neural network weights. During the deterministic or stochastic learning phase, a set of useful training data is usually used to obtain a point estimate of the parameter vector (or eventually to determine a "posterior" distribution over this vector). After the learning phase, the training data is then completely discarded, and predictions for new (typically unseen) inputs are based purely on the learned parameter vector. In particular, if there is a fixed finite number of parameters independent of dataset size, i.e. if the size of adaptive parameters is a priori determined, the models is called parametric.

However, there is a class of pattern recognition techniques, in which the training data points, or a subset of them, are kept and used also during the prediction phase: obviously these approaches are non-parametric machine learning models.

For example, there are simple techniques for pattern classification called nearest neighbours (NN) and its generalization K-nearest neighbors (KNN), which involved non-parametric methods to assigning to each new test vector, respectively, the same label as the closest example from the training set or the label determined by a plurality vote of its neighbors, with the test vector being assigned to the class most common among its k nearest neighbors. They are examples of memory based methods that involve storing the entire training set in order to make predictions for future data points. They typically require a metric to be defined that measures the similarity of any two vectors in input space, and are generally fast to "train" but slow at making predictions for test data points (if the train set is big and the learning phase retains a good part of it).

Support Vector Machines (SVM) are considered non-parametric supervised machine learning models, even if in the very basic case, i.e. the linear SVM with "hard" margin in its "primal" formulation, it's not really true: if feature space has D dimension, linear SVM needs indeed always D+1 parameters to determine uniquely a hyperplane and hence can be seen as parametric model.

In general, Support Vector Machines require the identification of a subset of the training set points, aimed at identifying a clear decision boundary that meets the principle of structural risk minimization. Precisely structural risk minimization is a pivotal point in the success of SVMs, because the learned model should not overfit the data, otherwise its generalization performance will be poor.

SVM have number of applications in several fields, such as hand-written characters recognition, face detection (i.e., classifieng parts of the image as a face or non-face), text

and hyper text categorization for both inductive and transductive models, and so on. The SVM algorithm has been widely applied also in the biometrics, like Protein fold and remote homology detection.

The high spread of support vector machines is linked to some advantages perceived in this machine learning model:

- SVM usually proves effective in high dimensional spaces;
- SVM has a simple geometric interpretation, while for other machine learning algorithms trying to interpret the parameters is decidedly complicated (in fact, they are mainly used as a black box);
- SVM uses structural risk minimization, so thie model is less prone to overfitting training data;
- SVM proves helpful in countering the curse of dimensionality, because it can be still efficacious even if size of training set is lower than space dimensionality;
- with to respect to other non parametric models, SVM could also be memory efficient, because it uses a subset of training points (i.e, the so called support vectors!) in the decision function;
- for the general case, i.e. non linear SVM, this model demonstrates remarkable versatility, because different Kernel functions can be specified for the decision function, both de facto standard kernels, such as polynomial of various degree or radial basis function, both ad hoc "custom" kernels;
- fixed a kernel function for SVM, the solution is guaranteed to be global and unique, which warrants the repeatability of the results and prevents the possibility of getting stuck in a local minimum, possibly much worse than the global one;
- because this machine learning approach is massively based on linear algebra, it is relatively simple to optimize its performance, improving its speed using two parallel computing strategies, i.e. by implementing multithreaded solutions and/or by adopting vector calculation instructions (for example, SIMD extensions), as well as implementing computation on GPU Salleh and Baharim [2015].

3.1 SVM for linearly separable data

For two-class, linearly separable training data sets, there are lots of possible linear separators. Intuitively, a decision boundary drawn in the middle of the void between data items of the two classes seems better than one which approaches very close to examples of one or both classes. While some learning methods such as the perceptron algorithm, find just any linear separator, others, like Naive Bayes [Russell and Norvig, 2016], search for the best linear separator according to some criterion. One conceptual problem is how to find a separating hyperplane that will generalize well: the dimensionality of the feature space could be very large, so not all hyperplanes that separate the training data will necessarily generalize well. The SVM is a lerning machine for binary classification problems and in particular it defines the criterion to be looking for a decision surface that is maximally far away from any data point. That criterion, just for the case of optimal hyperplanes for separable classes, was found by Vapnik in 1965. An optimal hyperplane is defined as the linear decision bundary with maximal margin between the vectors of the two classes. This distance from the decision surface to the closest data point determines the margin of the classifier. This method of construction necessarily means that the

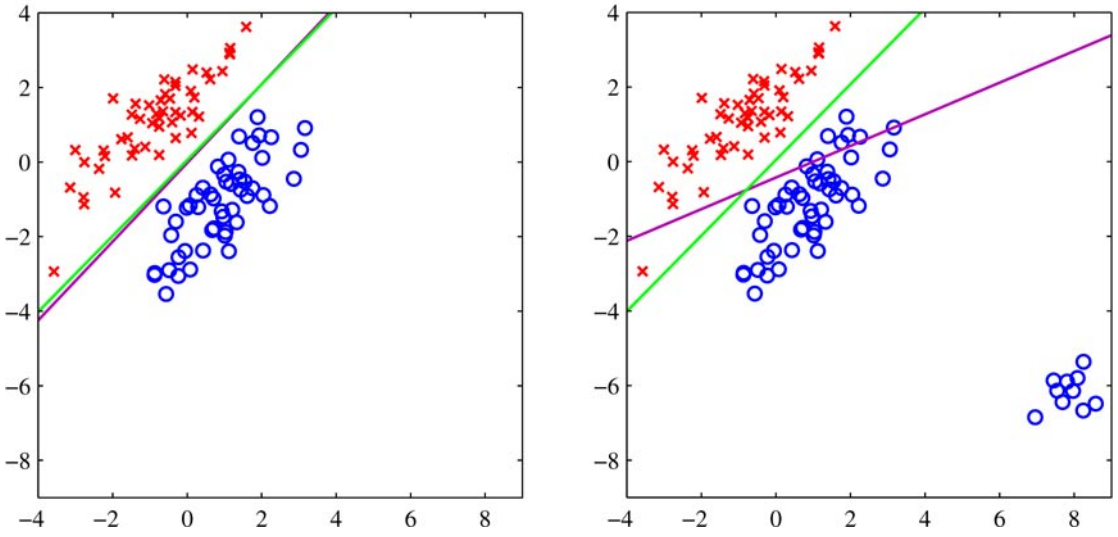


Figure 3.1: The left plot shows data from two classes, denoted by red crosses and blue circles, together with the decision boundary found by least squares (magenta curve) and also by the logistic regression model (green curve). The right-hand plot shows the corresponding results obtained when extra data points are added at the bottom left of the diagram, showing that least squares is highly sensitive to outliers, unlike logistic regression. (Fig. 4.4 in [Bishop, 2006]).

decision function for an SVM is fully specified by a (hopeful small) subset of the data points which define the position of the separator: precisely these points are referred to as the support vectors, so, you can think of these points as vectors "supporting" the margin and thus the decision boundary, keeping the boundary in a nice equilibrium.

Other data points play no role in determining the decision boundary that is chosen. So, the cool property that the decision boundary is solely determined by the support vectors, as its a linear combination of these vectors.

It was shown that if the training vectors are separated without errors by an optimal hyperptane the expectation value of the probability of committing an error on a test example is bounded by the ratio between the expectation value of the number of support vectors and the number of training vectors:

$$E[Pr(error)] \leq \frac{E[\#\, of\ support\ vectors]}{\#\, training\ vectors}$$

Note that this bound does not explicitly contain the dimensionality of the space of separation. It follows from this bound, that if the optimal hyperplane can be constructed from a small number of support vectors relative to the training set size the generalization ability will be high, even in an infinite dimensional space. In Section 5 we will demonstrate that the ratio (5) for a real life problems can be as low as 0.03 and the optimal hyperplane generalizes well in a billion dimensional feature space. Let

$$w_0 z + b_0 = 0$$

be the optimal hyperplane in feature space. We will show, that the weights w0 for the optimal hyperplane in the feature space can be written as some linear combination of support vectors

Lagrange multipliers

Con l'SVM è possibile creare un modello di classificazione. Ogni oggetto appartenente ad un insieme $X = x_1 \dots x_n$ verrà etichettato con una classe appartenente all'insieme $Y =$

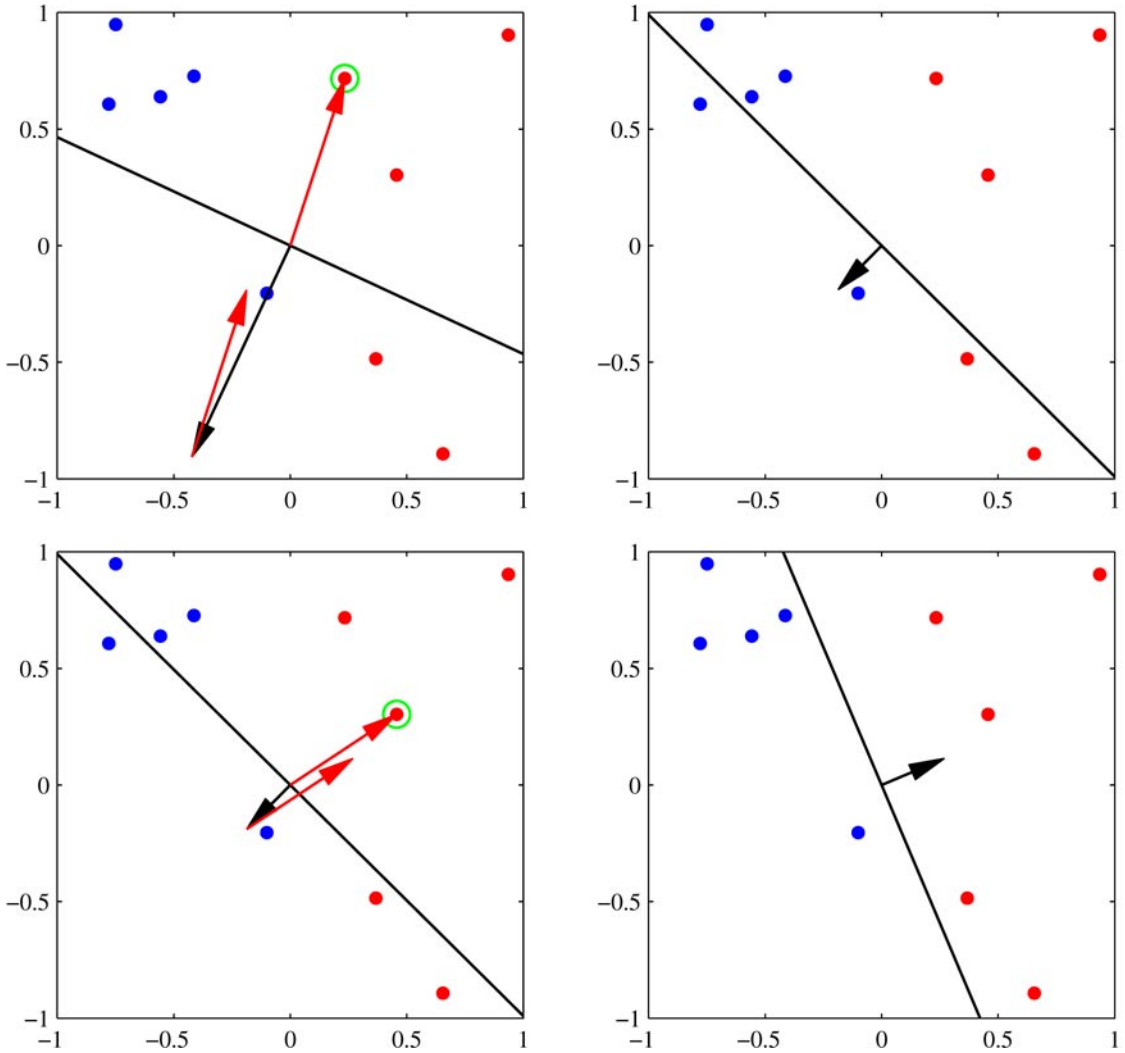


Figure 3.2: Illustration of the convergence of the perceptron learning algorithm, showing data points from two classes (red and blue) in a two-dimensional feature space.The top left plot shows the initial parameter vector w shown as a black arrow together with the corresponding decision boundary (black line), in which the arrow points towards the decision region which classified as belonging to the red class. The data point circled in green is misclassified and so its feature vector is added to the current weight vector, giving the new decision boundary shown in the top right plot. The bottom left plot shows the next misclassified point to be considered, indicated by the green circle, and its feature vector is again added to the weight vector giving the decision boundary shown in the bottom right plot for which all data points are correctly classified. (Fig. 4.7 in [Bishop, 2006]).

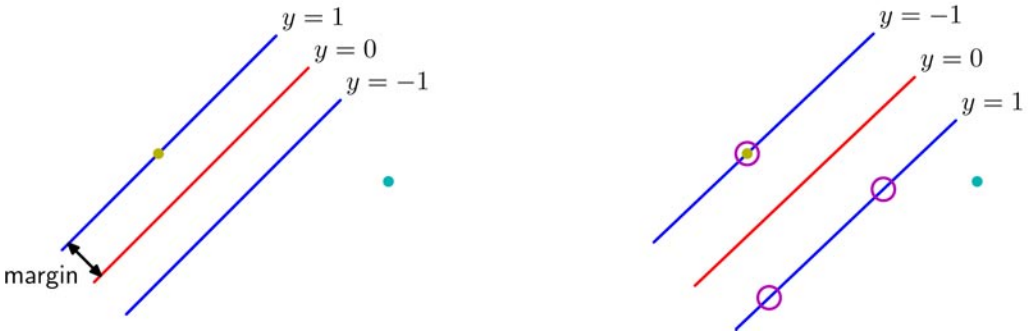


Figure 3.3: The margin is defined as the perpendicular distance between the decision boundary and the closest of the data points, as shown on the left figure. Maximizing the margin leads to a particular choice of decision boundary, as shown on the right. The location of this boundary is determined by a subset of the data points, known as support vectors, which are indicated by the circles. (Fig. 7.1 in [Bishop, 2006]).

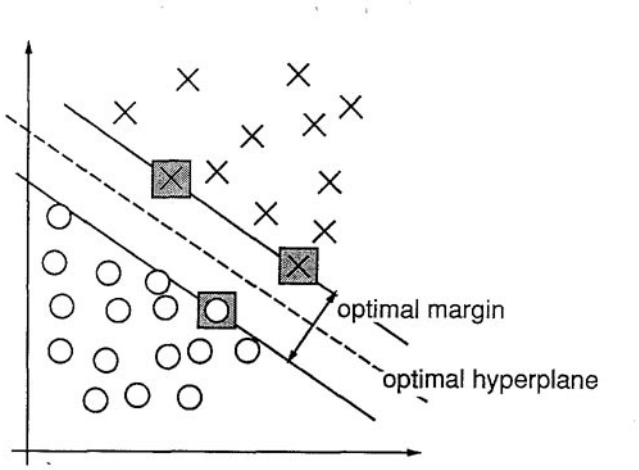


Figure 3.4: An example of a separable problem in a 2 dimensional space. The support vectors, marked with grey squares, define the margin of largest separation between the two classes. (Fig. 2 in [Cortes and Vapnik, 1995]).

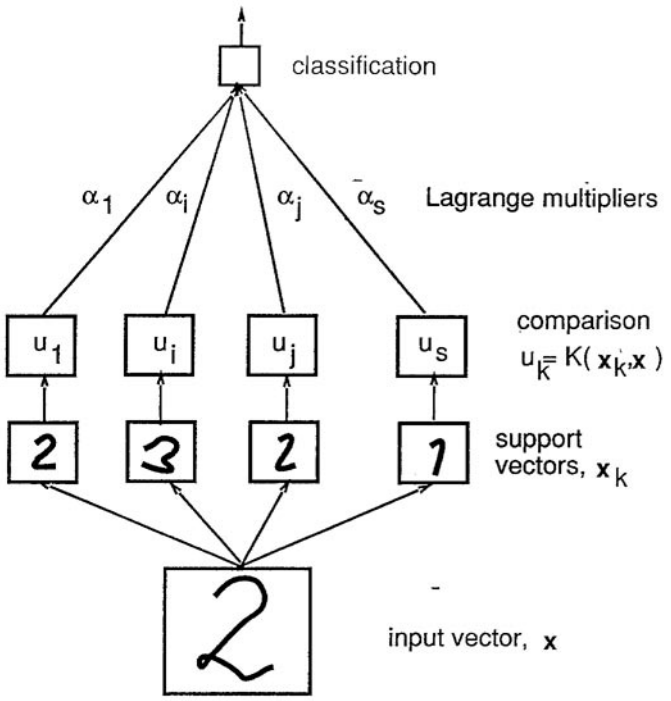


Figure 3.5: Classification of an unknown pattern by a support-vector network. The pattern is in input space compared to support vectors. The resulting values are non-linearly transformed. A linear function of these transformed values determine the output of the classifier. (Fig. 4 in [Cortes and Vapnik, 1995]).

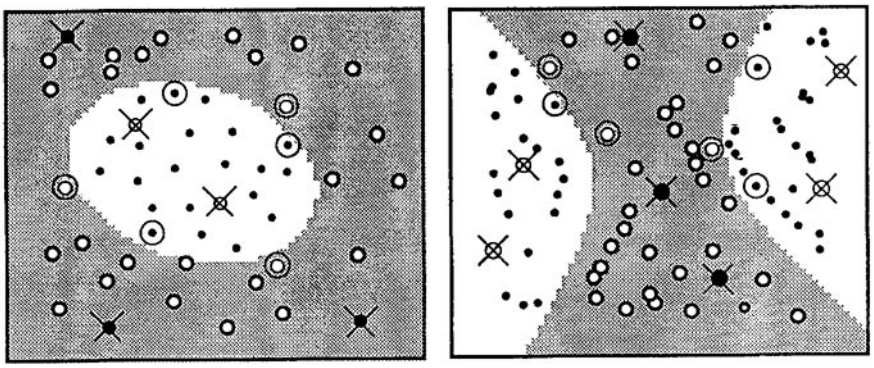


Figure 3.6: Examples of the dot-product (39) with d = 2, Support patterns are indicated with doable circles, errors with a cross. (Fig. 5 in [Cortes and Vapnik, 1995]).

$y_1 \dots y_m$. La classificazione è eseguita sugli attributi (features) dell'oggetto $x_i, 1 \leq i \leq n$. Fornendo un insieme di dati di addestramento, il modello SVM deve classificare elementi non analizzati precedentemente.

Dato un insieme di addestramento, in cui ogni elemento è una coppia nome e etichetta $(x_i, y_i), i = 1 \dots n$ dove $x_i \in R^n$ e $y \in [1, -1]^n$ la SVM trova la soluzione al seguente problema:

$$min_{w,b,\xi} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \xi_i$$

con $y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i, \xi \geq 0$

Il vettore x_i è mappato in uno spazio dimensionale più ampio dalla funzione ϕ. La SVM trova un iperpiano con il margine massimo nello spazio dimensionale. La mappatura del vettore x_i nella funzione ϕ è eseguita tramite una funzione definita kernel (K). In letteratura sono presenti quattro kernel di base:

- linear: $K(x_i, x_j) = x_i^T x_j$
- polinomiale: $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d, \gamma \geq 0$
- funzione radiale di base: $K(x_i, x_j) = exp(-\gamma |x_i - x_j|^2, \gamma \geq 0$
- funzione sigmoidea: $K(x_i, x_j) = tanh(\gamma x_i^T x_j + r)$

Sparse representation: the separating hyperplane f(x) is spanned those data points i where yi 6= 0,called Support Vectors. Both the estimation and the evaluation of f(x) only involve dot product.

3.2 Not linearly separable data

[Cortes and Vapnik, 1995]

The support-vector network implements the following idea: it maps the input vectors into some high dimensional feature space Z through some non-linear mapping chosen a priori. In this space a linear decision surface is constructed with special properties that ensure high generalization ability of the network.

the training problem is reformulated and represented in such away so as to obtain a (convex) quadratic programming (QP) problem. The solution to this QP problem is global and unique. In SVMs, it is possible to choose several types of kernel functions

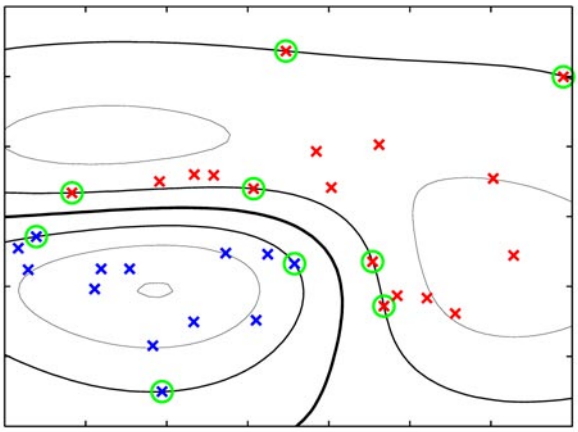


Figure 3.7: Example of synthetic data from two classes in two dimensions showing contours of constant y(x) obtained from a support vector machine having a Gaussian kernel function. Also shown are the decision boundary, the margin boundaries, and the support vectors. (Fig. 7.2 in [Bishop, 2006]).

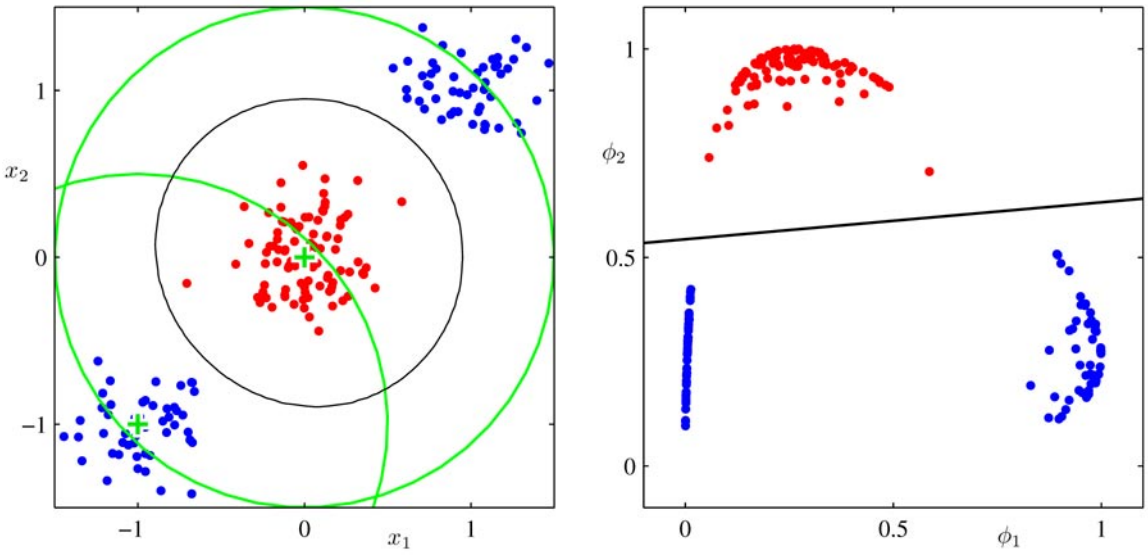


Figure 3.8: Illustration of the role of nonlinear basis functions in linear classification models. The left plot shows the original input space (x1, x2) together with data points from two classes labelled red and blue. Two 'Gaussian' basis functions φ1(x) and φ2(x) are defined in this space with centres shown by the green crosses and with contours shown by the green circles. The right-hand plot shows the corresponding feature space (φ1, φ2) together with the linear decision boundary obtained given by a logistic regression model of the form discussed in Section 4.3.2. This corresponds to a nonlinear decision boundary in the original input space, shown by the black curve in the left-hand plot. (Fig. 4.12 in [Bishop, 2006]).

including linear, polynomial, RBFs, MLPs with one hidden layer and splines, as long as the Mercer condition is satisfied.

Finally, what happens if one uses a kernel which does not satisfy Mercer's condition? In general, there may exist data such that the Hessian is indefinite, and for which the quadratic programming problem will have no solution (the dual objective function can become arbitrarily large). However, even for kernels that do not satisfy Mercer's condition, one might still find that a given training set results in a positive semidefinite Hessian, in which case the training will converge perfectly well. In this case, however, the geometrical interpretation described above is lacking. Burgess (1998)

Many linear parametric models can be re-cast into an equivalent 'dual representation' in which the predictions are also based on linear combinations of a kernel function evaluated at the training data points. As we shall see, for models which are based on a fixed nonlinear feature space mapping (x), the kernel function is given by the relation

From this definition, we see that the kernel is a symmetric function of its arguments so that . The kernel concept was introduced into the field of pattern recognition by Aizerman et al. (1964) in the context of the method of potential functions, so-called because of an analogy with electrostatics. Although neglected for many years, it was re-introduced into machine learning in the context of largemargin classifiers by Boser et al. (1992) giving rise to the technique of support Chapter 7 vector machines. Since then, there has been considerable interest in this topic, both in terms of theory and applications. One of the most significant developments has been the extension of kernels to handle symbolic objects, thereby greatly expanding the range of problems that can be addressed. Many linear models for regression and classification can be reformulated in terms of a dual representation in which the kernel function arises naturally.

Another idea would be hoping kernelizing would separate them in the non-linear higher dimension. introduction of non linear kernel function immersion in higher dimensionality space

SVM is a supervised machine learning algorithm. It can be used for classification or regression problems. It uses a method called the kernel trick to transform your data. A Support Vector Machine (SVM) performs classification by finding the hyperplane that maximizes the margin between the two classes. The vectors that define the hyperplane are the support vectors. The extreme points in the data sets that define the hyperplane are the support vectors

Kernel machines are algorithms in which kernels are employed to conceptually map data from an input space into a higher-dimensional feature space where the data can be processed using linear methods. The mapping is usually nonlinear and is implemented implicitly through the kernel trick. Many kernel methods have been developed by the machine learning community, such as support vector machines (SVMs) [51], kernel-based principal component analysis (KPCA) [36], kernel-based linear discriminant analysis (KLDA) [35], kernel-based independent component analysis (KICA) [2] and kernel-based nearest neighbour classifier [38]. SVM, a mostwidely used kernelmachine, was originally developed for two-class classification. Based on the principle of structural risk minimization, discriminative binary SVMs, referred to as Binary SupportVector Classifier (BSVC) in this chapter, have been reported to perform well in many real applications [9, 12, 37]. However, SVM also suffers from some fundamental problems in statistical pattern recognition, such as the imbalanced data problem [19], in which the size of the training data from one class is significantly larger than that of the other class in a two-class classification task. Such a problem is frequently encountered in many biomedical applications where data from both positive and negative diagnosis categories are not available equally.

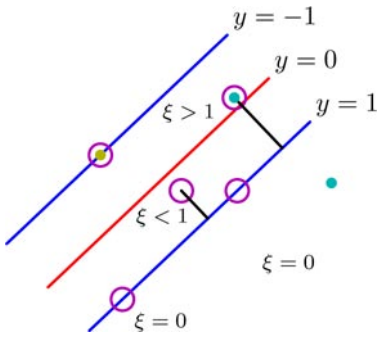


Figure 3.9: Illustration of the slack variables. Data points with circles around them are support vectors. (Fig. 7.3 in [Bishop, 2006]).

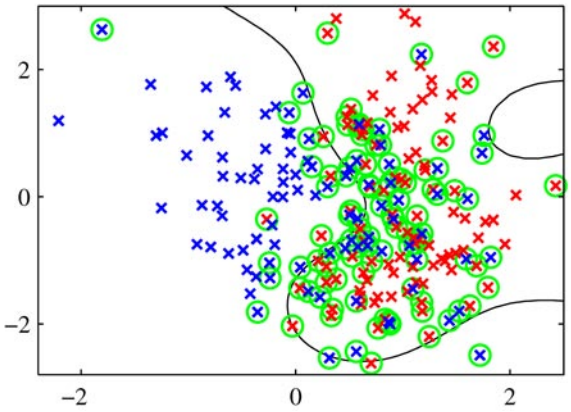


Figure 3.10: Illustration of the ν-SVM applied to a nonseparable data set in two dimensions with Gaussian kernels. The support vectors are indicated by circles. (Fig. 7.4 in [Bishop, 2006]).

3.3 Soft margin and least square support vector machine

Linear Separability of the SVM is relaxed by using something known as the Hinge Loss objective (i.e. a soft-margin SVM) instead of using the vanilla objective.

This involves recourse to Karush-Kuhn-Tucker conditions[Tucker and Kuhn, 1951]

The LS-SVM is the least squares SVM which minimizes a quadratic penalty on the slack variables as well. This allows the quadratic programming problem to be reduced to a set of matrix inversion operations in the dual space, which takes less time compared to solving the SVM quadratic problem. The flip side is the requirement that the matrices being invertible and/or well conditioned.

To demonstrate the KKT conditions and the derivation of support vectors (SVs) in SVM, we consider SVM developed by Corinna Cortes and Vladimir N. Vapnik in 1995. It can readily be extended to the case of non-linear SVM using the so-called kernel trick. The linear model of SVM is:

f(x)=wTx+b(1) where w and b are unkown and determined by the training samples. This is achieved by the following formulation:

mininize w,e subject to where C controls the trade-off between model complexity (first term) and empirical risk (second term).

Taking equality instead of inequality constraints in the problem formulation. As a result one solves a linear system instead of a QP problem. Due to this choice of a 2-norm one looses sparseness in the resulting LS-SVM Maximum Margin Classifier is

a choice when data is linearly separable but in many of the cases with no separating boundary possible and hence there is no solution with M>0. Hence, there is a need to extend the concept of a separating hyperplane in order to develop a hyperplane that ALMOST separates the classes, using so called SOFT MARGIN. This generalization is called "Support Vector Classifier". We want a hyperplane that does not perfectly separate two classes, in the interest of

LS-SVMs show greater robustness to individual observation and better classification of most of the training observations

In Support Vector Machines (SVMs), the solution of the classification problem is characterized by a (convex) quadratic programming (QP) problem. In a modified version of SVMs, called Least Squares SVM classifiers (LS-SVMs), a least squares cost function is proposed so as to obtain a linear set of equations in the dual space.

Anyway, LS-SVM is a variant of SVM with a "slightly" altered objective function:

The above formulation is actually an implicit consequence of a least square regression problem, that's why it is called least square-SVM.

Another difference is that SVM requires you to solve a quadratic programming problem while LS-SVM requires you to solve a linear system.

Since the formulation of both objective functions are different, their results would normally not be the same even when you apply them on the same dataset. However, it is demonstrated that, under specific conditions, they can be equivalent.

The conceptual problem is how to find a separating hyperplane that will generalize well: the dimensionality of the feature space will be large, and not all hyperplanes that separate the training data will necessarily generalize well

3.4 SVM vs ANN

One specific benefit that ANNs, such as multilayer feed-forward networks (FF nets for short), have over SVMs is that their size is fixed: they are parametric models, while SVMs are non-parametric. That is, in an ANN you have a bunch of hidden layers with sizes h1 through hn depending on the number of features, plus bias parameters, and those make up your model. By contrast, an SVM (at least a kernelized one) consists of a set of support vectors, selected from the training set, with a weight for each. In the worst case, the number of support vectors is exactly the number of training samples (though that mainly occurs with small training sets or in degenerate cases) and in general its model size scales linearly. In natural language processing, SVM classifiers with tens of thousands of support vectors, each having hundreds of thousands of features, is not unheard of.

Also, online training of FF nets is very simple compared to online SVM fitting, and predicting can be quite a bit faster.

Obviously all of the above pertains to the general case of kernelized SVMs. Linear SVM are a special case in that they are parametric and allow online learning with simple algorithms such as stochastic gradient descent.

A significant advantage of SVMs is that whilst ANNs can suffer from multiple local minima, the solution to an SVM is global and unique. Two more advantages of SVMs are that that have a simple geometric interpretation and give a sparse solution. Unlike ANNs, the computational complexity of SVMs does not depend on the dimensionality of the input space. ANNs use empirical risk minimization, whilst SVMs use structural risk minimization.

The reason that SVMs often outperform ANNs in practice is that they deal with the biggest problem with ANNs, SVMs are less prone to overfitting.

Another advanteg of SVM is that for two linearly separable classes SVM will draw the separating hyperplane halfway between the nearest points of the two classes (these

become support vectors), while a neural network would draw any line which separates the samples, which is correct for the training set, but might not have the best generalization properties.

Both random forests and SVMs are non-parametric models (i.e., the complexity grows as the number of training samples increases). Training a non-parametric model can thus be more expensive, computationally, compared to a generalized linear model, for example.

When data are not lineraly separable, we can end up with a lot of support vectors in SVMs; in the worst-case scenario, we have as many support vectors as we have samples in the training set. Although, there are multi-class SVMs, the typical implementation for mult-class classification is One-vs.-All; thus, we have to train an SVM for each class – in contrast, decision trees or random forests, which can handle multiple classes out of the box.

Chapter 4

Quantum Machine Learning

Quantum Machine Learning is a very new, promising interdisciplinary research area which lies in the intersection of quantum computing and information systems and machine learning[Biamonte et al., 2017; Da Silva et al., 2012; da Silva et al., 2016; Fard et al., 2018; Gupta and Zia, 2001; Havlíček et al., 2019; Havlivcek et al., 2018; Liu et al., 2013; Otterbach et al., 2017; Rebentrost et al., 2017; Ristè et al., 2017; Schuld et al., 2018; Schuld and Killoran, 2018; Schuld et al., 2014, 2015a,b; Verdon et al., 2017]. Said in a nutshell, Quantum Machine Learning deals with identifying and running suitable machine learning models on a quantum computer. One of the sources of inspiration is that many machine learning algorithms operate by performing specific matrix operations on vectors within high-dimensional feature spaces, which looks similar to what forms the basis of quantum mechanics.

Quantum Machine Learning takes on at least three stages: first stage involves converting classical data to suitable quantum data, second stage performs the computations on the quantum computer, trying to exploit a computational speed up, and finally third stage converts the quantum results back into the an understandable classical format.

Until now, the prevailing approach is to try to take inspiration from a consolidated classical machine learning model and try to convert it into a quantum format. In the future, it may be possible to directly develop effective quantum machine learning algorithms that could not have a classical counterpart.

In artificial intelligence, an evolutionary algorithm is a generic population-based metaheuristic optimization algorithm. An evolutionary algorithm uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the quality of the solutions. Evolution of the population then takes place after the repeated application of the above operators.

Evolutionary algorithms often perform well approximating solutions to all types of problems because they ideally do not make any assumption about the underlying fitness landscape. In most real applications of evolutionary algorithms, computational complexity is a prohibiting factor. In fact, this computational complexity is due to fitness function evaluation. It is possible to point out that even quantum evolutionary algorithms have been formulated and achieved some success[Han and Kim, 2000; Kumar and Kumar, 2018; Laboudi and Chikhi, 2012; Lahoz-Beltra, 2016; Layeb and Saidouni, 2007; Li et al., 2018; Ma and Jin, 2007; Narayanan and Moore, 1996; Nowotniak, 2010; Nowotniak and Kucharski, 2010; Platel et al., 2007; Sakurai and Katz, 2009; Sofge, 2008; Tkachuk, 2018; Udrescu et al., 2006; Wang et al., 2013; Zhang, 2011].

4.1 Quantum Computing

The first and most famous quantum algorithms [Chuang and Shor, 2018b; Nielsen and Chuang, 2010] exhibit considerable comptutational advantages with rispect to conventional computers: Deutsch-Jozsa algorithm [Deutsch and Jozsa, 1992] allows resolution of a specific class of "on promise" problems with just one function ("oracle") evaluation; Shor algorithm[Shor, 1994] provides an exponential speed-up for prime number factorization (anyway, until now the only real implementations are able to work with very small numbers [Monz et al., 2016]); Grover algorithm [Grover, 1996] allows a quadratic speed-up for searching an unstructured database.

Quantum computing can substantially speedup least-squares support vector machines. Least square SVMs translate an optimization problem to a set of linear equations. The linear equations require the quick calculation of the kernel function, or rather kernel matrix, so a potential advantage of quantum computing is inherent in the acceleration of the computation of kernel transformation. The other source of the quantum speedup is the efficient solution of the linear equations on quantum hardware.

By virtue of pressing technical and economic constraints, such as the modest number of qu-bits (quantum bits) made available by current or near-term hardware implementations, high production costs, the need to operate in cryogenics , of the high rate of errors[Chuang and Harrow, 2018; Chuang and Shor, 2018c], of the decoherence phenomena, the idea of adopting hybrid solutions is predominant, that is, in which classical and quantum computing agents co-exist profitably, in particular with the aim of using this last type of resource to accelerate only some portions of computation (assigning, therefore, the execution of specific small routine to quantum hardware solutions).

The two peculiar aspects of quantum systems that can play a decisive role in ensuring speed improvements ("quantum advantage") are superposition and entanglement: the superposition gives a quantum system the possibility to memorize (simultaneously) any linear combination of states, while the entanglement allows to establish a strong correlation between two quantum systems, so that the application of an interaction to the first has an effect also on the second one (as in the teleportation[Chuang and Shor, 2018b; Gottesman and Chuang, 1999; Rycerz et al., 2015]).

The development of quantum algorithms and quantum computers is mainly based on the computational model of quantum circuits [Chuang and Shor, 2018a; Nielsen and Chuang, 2010].
Because the temporal evolution of (perfectly closed) quantum systems is completely determined by unitary operators, it is necessary to design circuits based on reversible logic gates [Brylinski and Brylinski, 2002; DiVincenzo, 1995, 1998; Maslov et al., 2005, 2007; Sasao and Kinoshita, 1979; Shende et al., 2002], such as Toffoli and Fredkin gates [Fredkin and Toffoli, 1981; Toffoli, 1981], instead of classical dissipative gates (such as AND, OR, NAND, NOR, XOR, XNOR).

In these models, it is necessary to identify an adequate set of elementary reversible logic gates, capable of representing, or at least approximating, any quantum circuit [Barenco et al., 1995; Deutsch et al., 1995; DiVincenzo, 1995; Gottesman and Chuang, 1999; Lloyd, 1995; Shi, 2002].

4.1.1 Quantum Bits

Just as bits are the fundamental object of information in classical computing, qubits (quantum bits) are the fundamental object of information in quantum computing. To understand this correspondence, let's look at the simplest example: a single qubit.

While a bit, or binary digit, can have only two distinct values, such as true or false either 0 or 1, a qubit can have a value that is either of these or any quantum superposition

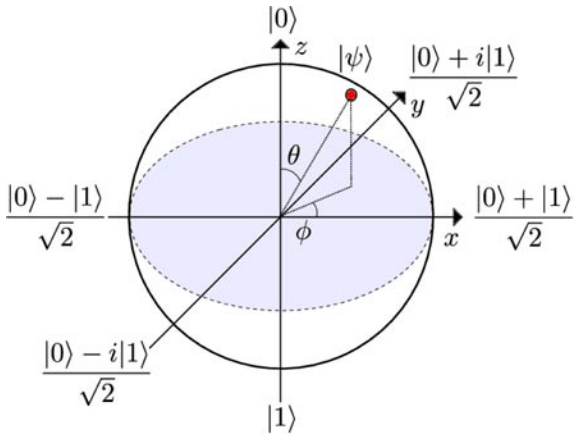


Figure 4.1: Bloch sphere is a geometrical representation of the pure state space of a two-level quantum mechanical system (qubit)

of the 0 and 1 states. The state of a single qubit so can be described by a two-dimensional column vector of unit norm (pure state), that is, the magnitude squared of its entries must sum to one. This vector, called the quantum state vector, holds all the information needed to describe the one-qubit quantum system just as a single bit holds all of the information needed to describe the state of a binary variable.

Any two-dimensional column vector of real or complex numbers with unitary norm represents a possible quantum state held by a qubit. Thus

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

represents a qubit state if α and β Because are complex numbers satisfying $\alpha^2 + \beta^2 = 1$ Some examples of valid quantum state vectors representing qubits include

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

In particular, the quantum state vectors

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

take a special role: these two vectors form a basis for the vector space that describes the qubit's state. This means that any quantum state vector can be written as a linear combination of these basis vectors. Specifically, the generic vector

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

can be written as

$$\alpha \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \beta \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

While any rotation of these vectors would serve as a perfectly valid basis for the qubit, we choose to privilege this one, by calling it the computational basis.

We take these two quantum states to correspond to the two states of a classical bit, namely 0 and 1. Thus, out of the infinite number of possible single-qubit quantum state vectors, only two correspond to states of classical bits; all other quantum states do not.

Now that we know how to represent a qubit, we can gain some intuition for what these states represent by discussing the concept of measurement. A measurement corresponds to the informal idea of observing a qubit status, which immediately collapses the quantum state to one of the two classical states 0 or 1. When a qubit given by the quantum state vector

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

is measured, we obtain the outcome 0 with probability α^2 and the outcome 1 with probability β^2. Obviously these probabilities sum up to 1 because of the normalization condition $\alpha^2 + \beta^2 = 1$.

The properties of measurement also mean that the overall sign of the quantum state vector is irrelevant. Negating a vector is equivalent to transform in

$$\begin{bmatrix} -\alpha \\ -\beta \end{bmatrix}$$

but because the probability of measuring 0 and 1 depends on the magnitude squared of the terms, inserting such signs does not change the probabilities whatsoever. More generally, the so called "global phase" of the system is not an observable physical quantity and therefore does not affect the state of the system.

A final important property of measurement is that it does not necessarily affect all quantum state vectors. If we start with a qubit in a pure state, which corresponds to the classical state 0 or 1, measuring this state, using the computational basis, will always yield the same outcome and leave the quantum state unchanged. In this sense, if we only have classical bits (i.e., qubits that are either 0 or 1) then measurement does not damage the system. This means that we can replicate classical data and manipulate it on a quantum computer just as one could do on a classical computer. The ability, however, to store information in both states at once is what elevates quantum computing beyond what is possible classically and further robs quantum computers of the ability to copy quantum data indiscriminately, see also the no-cloning theorem.

4.1.2 Quantum Gates

Since the temporal evolution of quantum systems (in the ideal case, i.e. of perfect isolation from the external environment) is completely determined by unitary type operators, it is essential to design circuits based on reversible logic gates[Brylinski and Brylinski, 2002; DiVincenzo, 1995, 1998; Maslov et al., 2005, 2007; Sasao and Kinoshita, 1979; Shende et al., 2002], such as Toffoli and Fredkin gates [Fredkin and Toffoli, 1981; Toffoli, 1981],instead of the classical logic gates (such as AND, OR, XOR), which are inevitably dissipative.

So quantum computers process data by applying a universal set of quantum gates that can emulate any rotation of the quantum state vector. This notion of universality is akin to the notion of universality for traditional (i.e., classical) computing where a gate set

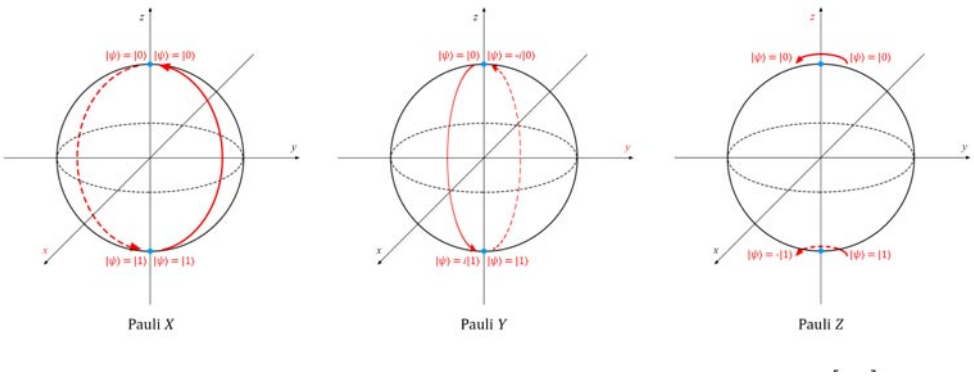


Figure 4.2: Pauli's gates

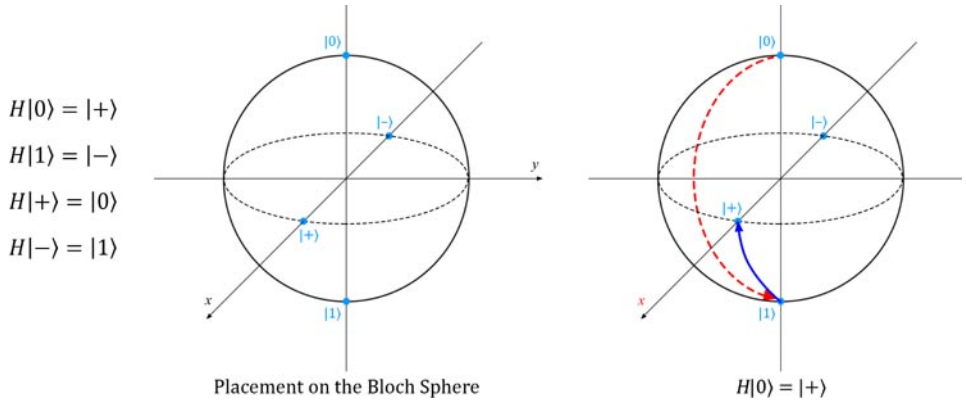


Figure 4.3: Hadamard's gate

is considered to be universal if every transformation of the input bits can be performed using a finite length circuit. In quantum computing, the valid transformations that we are allowed to perform on a qubit are unitary transformations and measurement. The adjoint operation or the complex conjugate transpose is of crucial importance to quantum computing because it is needed to invert quantum transformations.

Although this is a trivial example (as the H operation is self-adjoint), you can see how this becomes invaluable for more complicated qubit operations. For more information, see Operations and Functions.

On a classical computer, for each fixed arity n, it is possible to define 2^{2^n} logical connectives; for example, there are only four functions that map one bit to one bit. In contrast, there are an infinite number of unitary transformations on a single qubit on a quantum computer. Therefore, no finite set of primitive quantum operations, called gates, can exactly replicate the infinite set of unitary transformations allowed in quantum computing. This means, unlike classical computing, it is impossible for a quantum computer to implement every possible quantum program exactly using a finite number of gates. Thus quantum computers cannot be universal in the same sense of classical computers. As a result, when we say that a set of gates is universal for quantum computing we actually mean something slightly weaker than we mean with classical computing. For universality, we require that a quantum computer only approximate every unitary matrix within a finite error using a finite length gate sequence. In other words, a set of gates is a universal gate set if any unitary transformation can be approximately written as a product of gates from this set.

What does such a universal gate set look like in practice? The simplest such universal gate set for single-qubit gates consists of only two gates: the Hadamard gate H and the so-called T-gate (also known as the Pi/8 gate):

However, for practical reasons related to quantum error correction it can be more convenient to consider a larger gate set, namely one that can be generated using H and T. We can classify the quantum gates into two categories: Clifford gates and the T-gate. This subdivision is useful because in many quantum error correction schemes the so-called Clifford gates are easy to implement, that is they require very few resources in terms of operations and qubits to implement fault tolerantly, whereas non-Clifford gates are quite costly when requiring fault tolerance. The standard set of single-qubit Clifford gates include H, X, Y, Z, where the last three are used especially frequently and are named Pauli operators after their creator Wolfgang Pauli. Together with the non-Clifford gate (the T-gate), these operations can be composed to approximate any unitary transformation on a single qubit.

While the previous constitute the most popular primitive gates for describing operations on the logical level of the stack (think of the logical level as the level of the quantum algorithm), it is often convenient to consider less basic operations at the algorithmic level, for example operations closer to a function description level.

4.1.3 Multiple qubit gates

While single-qubit gates possess some counter-intuitive features, such as the ability to be in more than one state at a given time, if all we had in a quantum computer were single-qubit gates then we would have a device with computational power that would be dwarfed by even a calculator let alone a classical supercomputer. The true power of quantum computing only becomes evident as we increase the number of qubits and we resort to entaglment. This power arises, in part, because the dimension of the vector space of quantum state vectors grows exponentially with the number of qubits. This means that while a single qubit can be trivially modeled, simulating a fifty-qubit quantum computation would arguably push the limits of existing supercomputers. Increasing the

size of the computation by only one additional qubit doubles the memory required to store the state and roughly doubles the computational time. This rapid doubling of computational power is why a quantum computer with a relatively small number of qubits can far surpass the most powerful supercomputers of today, tomorrow and beyond for some computational tasks.

Why do we have exponential growth for quantum state vectors? Our goal in this section is to review the rules used to build multi-qubit states out of single-qubit states as well as discuss the gate operations that we need to include in our gate set to form a universal many-qubit quantum computer. These tools are absolutely necessary to understand the gate sets that are commonly used in code and also to gain intuition about why quantum effects such as entanglement or interference render quantum computing more powerful than classical computing.

The main difference between one- and two-qubit states is that two-qubit states are four dimensional rather than two dimensional. This is because the computational basis for two-qubit states is formed by the tensor products of one-qubit states.

It is easy to see that more generally the quantum state of n qubits is represented by a unit vector of dimension 2^n. Just as with single qubits, the quantum state vector of multiple qubits holds all the information needed to describe the system's behavior.

Such a two-qubit state, which cannot be written as the tensor product (or Kronecker product) of single-qubit states, is called an "entangled state"; the two qubits are said to be entangled. Loosely speaking, because the quantum state cannot be thought of as a tensor product of single qubit states, the information that the state holds is not confined to either of the qubits individually. Rather, the information is stored non-locally in the correlations between the two states. This non-locality of information is one of the major distinguishing features of quantum computing over classical computing and is essential for a number of quantum protocols including quantum teleportation and quantum error correction.

Measuring two-qubit states is very similar to single-qubit measurements.

It is also possible to measure just one qubit of a two-qubit quantum state. In cases where you measure only one of the qubits, the impact of measurement is subtly different because the entire state is not collapsed to a computational basis state, rather it is collapsed to only one sub-system. In other words, in such cases measuring only one qubit only collapses one of the subsystems but not all of them.

As in the single-qubit case, any unitary transformation is a valid operation on qubits. In general, a unitary transformation on n qubits is a matrix U of size 2^n (so that it acts on vectors of corresponding size).

We can also form two-qubit gates by applying single-qubit gates on both qubits, thus we can form two-qubit gates by taking the tensor product of some known single-qubit gates.

Note that while any two single-qubit gates define a two-qubit gate by taking their tensor product, the converse is not true. Not all two-qubit gates can be written as the tensor product of single-qubit gates. Such a gate is called an entangling gate. One example of an entangling gate is the CNOT gate.

The intuition behind a controlled-not gate can be generalized to arbitrary gates. A controlled gate in general is a gate that acts as identity (ie it has no action) unless a specific qubit is 1.

Building controlled unitaries in an efficient manner is a major challenge. The simplest way to implement this requires forming a database of controlled versions of fundamental gates and replacing every fundamental gate in the original unitary operation with its controlled counterpart. This is often quite wasteful and clever insight often can be used to just replace a few gates with controlled versions to achieve the same impact. For this

reason, we provide in our framework the ability to perform either the naive method of controlling or allow the user to define a controlled version of the unitary if an optimized hand-tuned version is known.

Gates can also be controlled using classical information. A classically controlled not-gate, for example, is just an ordinary not-gate but it is only applied if a classical bit is 1 as opposed to a quantum bit. In this sense, a classically controlled gate can be thought of as an if statement in the quantum code wherein the gate is applied only in one branch of the code.

As in the single-qubit case, a two-qubit gate set is universal if any unitary matrix can be approximated by a product of gates from this set to arbitrary precision. One example of a universal gate set is the Hadamard gate, the T gate, and the CNOT gate. By taking products of these gates, we can approximate any unitary matrix on two qubits.

We can follow exactly the same patterns explored in the two-qubit case to build many-qubit quantum states from smaller systems. Such states are built by forming tensor products of smaller states.

Quantum gates work in exactly the same way. In many qubit systems, there is often a need to allocate and de-allocate qubits that serve as temporary memory for the quantum computer. Such a qubit is called an ancilla. By default we assume the qubit state is initialized to 0 upon allocation. We can further assume that it is returned again to 0 before de-allocation. This assumption is important because if an ancilla qubit becomes entangled with another qubit register when it becomes de-allocated then the process of de-allocation will damage the ancilla. For this reason, we always assume that such qubits are reverted to their initial state before being released.

Finally, although new gates needed to be added to our gate set to achieve universal quantum computing for two qubit quantum computers, no new gates need to be introduced in the multi-qubit case. The gates H, T and CNOT form a universal gate set on many qubits because any general unitary transformation can be broken into a series of two qubit rotations. We then can leverage the theory developed for the two-qubit case and use it again here when we have many qubits.

4.1.4 Quantum circuits

This visual language for quantum operations can be more readily digestible than writing down its equivalent matrix once you understand the conventions for expressing a quantum circuit.

In a circuit diagram, each solid line depicts a qubit or more generally a qubit register. By convention, the top line is qubit register 0 and the remainder are labeled sequentially. Gates acting on one or more qubit registers are denoted as a box.

Quantum gates are ordered in chronological order with the left-most gate as the gate first applied to the qubits. In other words, if you picture the wires as holding the quantum state, the wires bring the quantum state through each of the gates in the diagram from left to right.

Matrix multiplication obeys the opposite convention: the right-most matrix is applied first. In quantum circuit diagrams, however, the left-most gate is applied first. This difference can at times lead to confusion, so it is important to note this significant difference between the linear algebraic notation and quantum circuit diagrams.

All quantum circuits have precisely the same number of wires (qubits) input to a quantum gate as the number of wires out from the quantum gate, because all quantum operations, save measurement, are unitary and hence reversible. If they did not have the same number of outputs as inputs they would not be reversible and hence not unitary, which is a contradiction. For this reason any box drawn in a circuit diagram must have precisely the same number of wires entering it as exiting it.

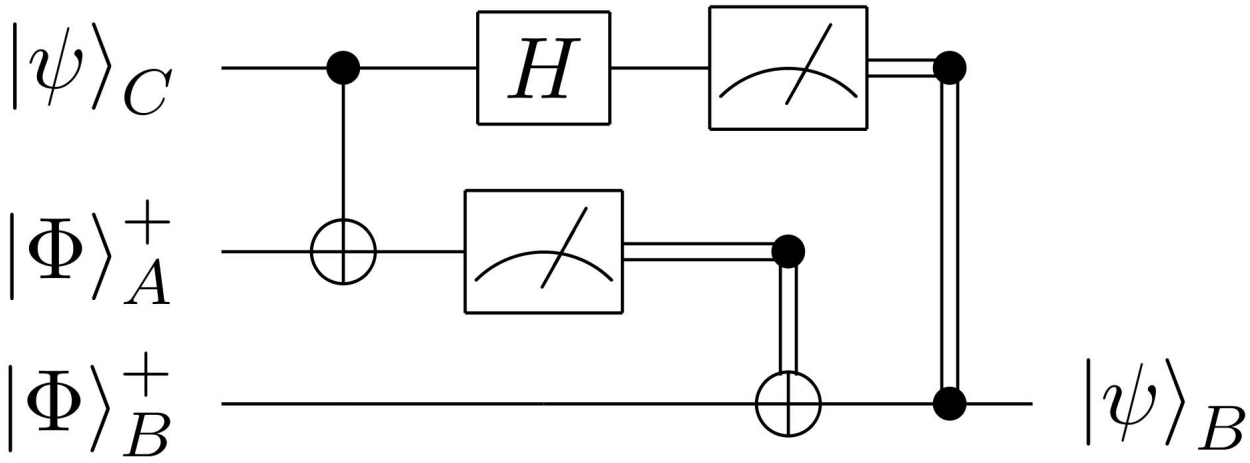


Figure 4.4: Quantum circuit representation of quantum teleportation

Multi-qubit circuit diagrams follow similar conventions to single-qubit ones.

Quantum teleportation is perhaps the best quantum algorithm for illustrating how to use quantum components to build a simple quantum circuit. Quantum teleportation is a method for moving data within a quantum computer (or even between distant quantum computers in a quantum network) through the use of entanglement and measurement. Interestingly, it is actually capable of moving a quantum state, say the value in a given qubit, from one qubit to another, without even knowing what the qubit's value is! This is necessary for the protocol to work according to the laws of quantum mechanics. The quantum teleportation circuit is given below; we also provide an annotated version of the circuit to illustrate how to read the quantum circuit.

4.2 An example: a quantum full adder

In order to highlight that quantum computing does not always present itself as the ideal platform to deal with a computational problem, we will introduce a simple example. The simplest and most intuitive arithmetic operation that acts on natural numbers is addition. The addition enjoys the associative property, admits an identity element and is commutative. By introducing the relative integers, the addition gives origin to an Abelian group, but in digital circuit implementations the adoption of signed numbers forces to devise adequate numerical representation schemes. That is, whereas to represent natural numbers, the conversion between positional systems characterized by different bases is rather simple, the management of the sign introduces a difficulty. For example, to convert a natural number from the decimal format into binary, octal or hexadecimal it is necessary to perform repeated divisions by the new base, taking care to collect the remainders and bring them back in reverse order (because the remainder of the first division provides the least significant figure in the new base and so on). The opposite conversion is even easier, because it is enough to multiply the value of each digit by the appropriate power of the adopted base, according to the position, and to add the collected products. However, if you want to represent relative numbers, you must somehow code the sign of the number, in addition to its module.

The simplest systems to represent relative numbers in binary format are signed magnitude representation, the ones' complement, the two-complement and the biased representation. The signed magnitude representation uses the most significant bit to represent the sign (0 indicates a non-negative number, 1 a negative number), which allows you to

A	B	C_in	Sum	C_out
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Table 4.1: Full adder truth table: A and B denote external inputs, C_in a carryover, where C_out and Sum represent the result.

easily partition the information: the first bit codes the sign, all the others the magnitude, so that two opposite numbers differ only in the first, most significant bit. Perhaps it is the simplest for a man to conceive, because it mimics the usual representation of the relative numbers with a decimal positional system The disadvantage of this system is that there are two distinct encodings for the neutral element, plus 0 (all zeros) and minus 0 (an one and all other bits equal zero). Moreover, addition, subtraction and comparison require different behaviour depending on the sign bit The ones' complement and two's complement use the same encoding as signed magnitude representation for non negative numbers. To encode a negative binary number, the ones' complement start with its opposite and then it flips every bit, i.e. the one's complement of a negative number is the bitwise NOT applied to its opposite. The disadvantage of this system is also that there are two distinct encodings for the neutral element, plus 0 (all zeros) and minus 0 (all ones). With this representation, even the simple addition requires specific attention depending on the sign of the oprandi, such as the addition of an appropriate offset (that is -1). The ones' complement of a negative number can be determined from the sign-magnitude representation flipping all bits of its magnitude. The two's complement of a negative number is calculated by adding the unit to the one's complement; in this representation, there is only one encoding of 0 (all zeros). Its merit is to simplify the hardware implementation of elementary arithmetic operations, which de facto is the reason for its widespread popularity. The biased representation is the only one (among typical representations, of course) in which all zero bits meaning is not zero! Biased representations are now primarily used for the exponent of floating-point numbers in IEEE-754 formats: arbitrarily fixed a non-negative bias value b, its coding in signed magnitude representation is used for encoding zero, the sequence of all zero corresponds to the value minus b and so on.

So, for sake of simplicity, we limit our attention to a very simple one bit binary adder, as shown in 4.5. In any positional numeric system, the sum of two numbers of m and n digits respectively may require at most $m + n$ digits, therefore the sum of two single bit numbers requires two bits for its representation. While the sum of two zero bits makes zero and the sum of a zero bit and a unit bit makes one, the sum of two unit bits requires indeed two bits to represent $(10)_2$. In the general case, it is possible that the carry-over of any addition of a previous pair of bits affects the sum of the two bits considered, therefore in general a full adder contemplates three inputs, as shown in Tab. 4.1.

So I developed a very simple quantum circuit (4.6)to implement full single bit adder and, at first, I I tested its operation both with Aer qasm_simulator and ibmq_qasm_simulator, getting 100% correct results. Then, I put real quantum computers to work.

A	B	C_in	'00'	'01'	'10'	'11'	Correct Pos.	Perc. of correct res.
0	0	0	39267	7902	26807	7944	1	47.93
0	1	0	18589	20444	18900	23987	2	24.96
1	0	0	19393	21104	17469	23954	2	25.76
1	1	0	29356	9819	33181	9564	1	40.50
0	0	1	18955	22698	17307	22960	2	27.71
0	1	1	32605	8478	32861	7976	1	40.11
1	0	1	34259	8194	31937	7530	2	38.99
1	1	1	18063	22121	19240	22496	1	27.46

Table 4.2: Full single bit adder results on real quantum device ibmq_burlington: for statistical purposes, the results of 10 executions were collected with the maximum number of 'shots', i.e. 8192, for a total of 81920 executions of each sum on the quantum computer. A and B denote external inputs, C_in a carryover. In the four columns from '00 'to' 11', the occurrences of each possible result are reported. In the last two columns are indicated: if correct results are in first or subsequent position; the percentage of correct results.

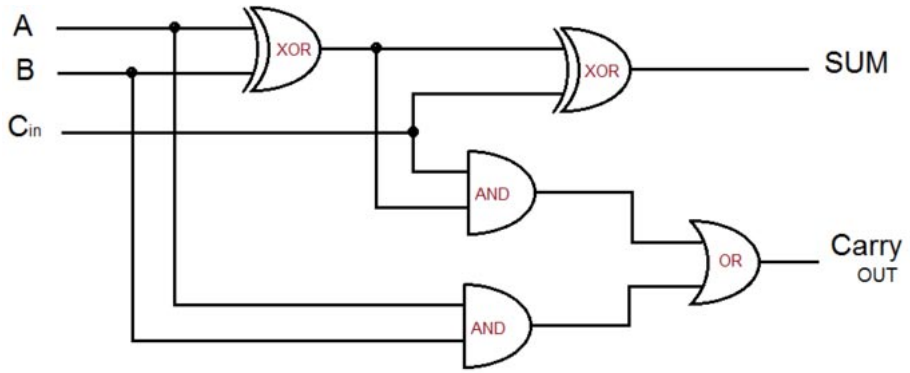


Figure 4.5: Digital circuit representation of a simple full adder

```
def my_adder(circuit, q, c):
  circuit.cx(q[0],q[3]); circuit.cx(q[1],q[3])
  circuit.cx(q[2],q[3]); circuit.ccx(q[0],q[1],q[4])
  circuit.ccx(q[0],q[2],q[4]); circuit.ccx(q[1],q[2],q[4])
  circuit.measure(q[3],c[0]); circuit.measure(q[4],c[1])
  return circuit
```

Listing 4.1: Python funtion to generate a quantum circuit capable of performing sigle bit addition, with carry-in and carry-out

```
def my_job(circuit, backend, num=10, shots=8192):
  counts ={}
  for a in range(num):
    job = execute(circuit, backend, shots=shots)
    result = job.result(); res = result.get_counts(circuit)
    for x in res:
      if x in counts: counts[x] = counts[x]+res[x]
```

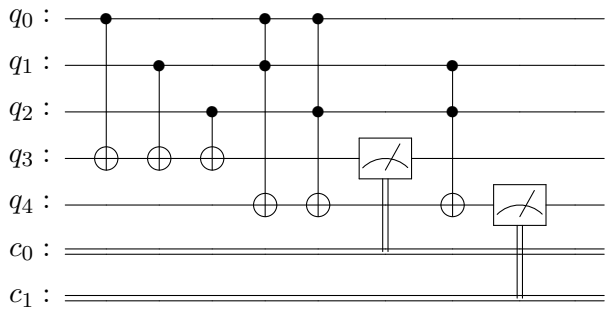


Figure 4.6: Quantum single bit full adder: q_0 and q_1 denote input, q_2 is carry-in

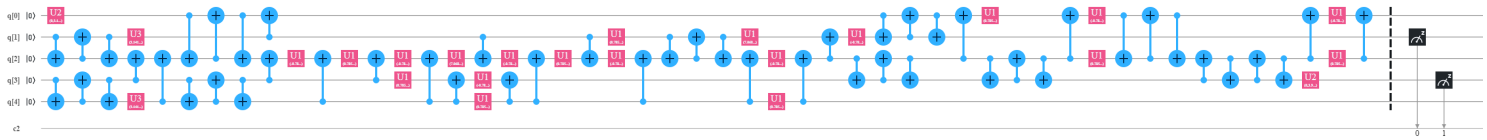
```
    else: counts[x] = res[x]
  return counts
```

Listing 4.2: Python function to issue a job on quantum computer multiple times

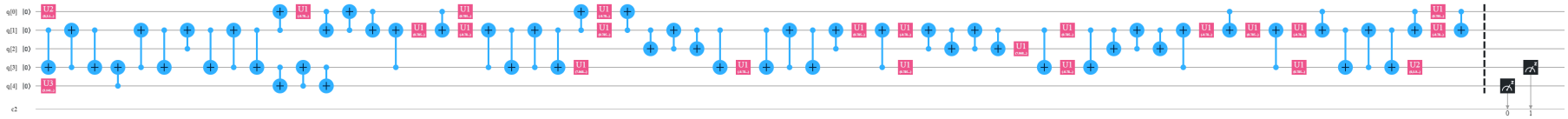
```
from qiskit import QuantumRegister, ClassicalRegister
from qiskit import QuantumCircuit, execute, IBMQ
provider = IBMQ.get_provider(hub='ibm-q')
backend = provider.get_backend('ibmq_burlington')
num=10; shots=8192; trials=num*shots

q = QuantumRegister(5,'q'); c = ClassicalRegister(2,'c');
circuit = QuantumCircuit(q,c)
correct='00';
circuit = my_adder(circuit, q, c);
res=my_job(circuit, backend, num, shots)
ind=1+len([x for x in res if res[x]>res[correct]]);
print('0+0', res, correct, ind, res[correct]/trials*100)

circuit = QuantumCircuit(q,c); circuit.x(q[0])
correct='01';
circuit = my_adder(circuit, q, c);
res=my_job(circuit, backend, num, shots)
ind=1+len([x for x in res if res[x]>res[correct]]);
print('1+0', res, correct, ind, res[correct]/trials*100)
....

circuit = QuantumCircuit(q,c); circuit.x(q[0]); circuit.x(q[1])
circuit.x(q[2]); correct='11';
circuit = my_adder(circuit, q, c);
res=my_job(circuit, backend, num, shots)
ind=1+len([x for x in res if res[x]>res[correct]]);
print(''Cin+1+1', res, correct, ind, res[correct]/trials*100)
```

Listing 4.3: Python sample code to test single bit adder

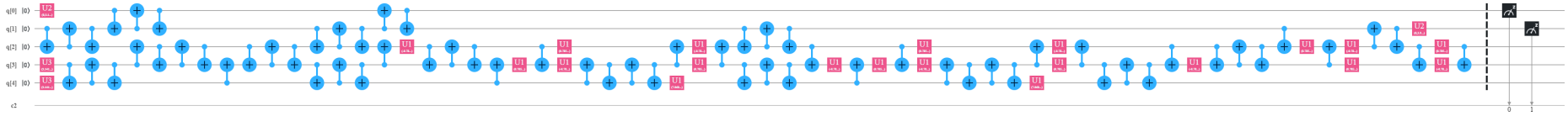
First I ran the circuit, with the maximum number of 'shots' allowed by qiskit (8192) on each quantum computer for which IBM allows free access, therefore on the providers:



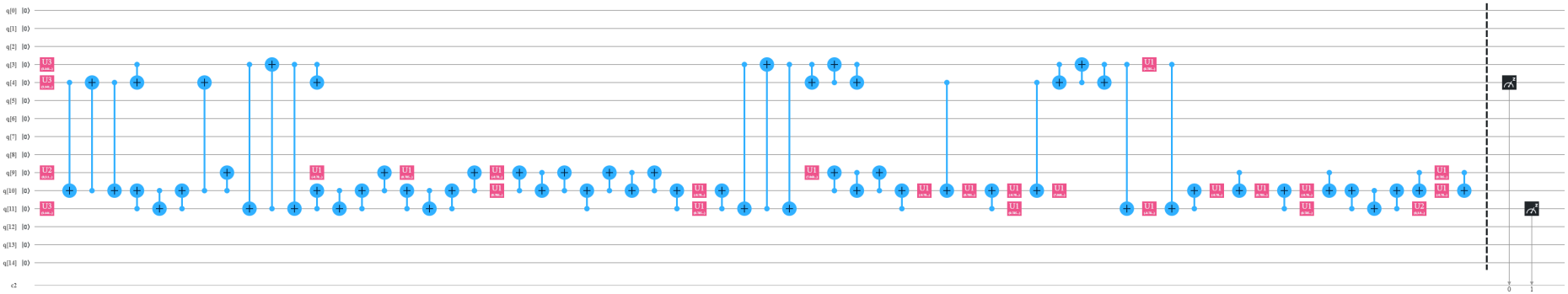
(a) $0 + 1 + C_{in}$ sum on ibmq_qx2 quantum computer



(b) $1 + 0$ sum on ibmq_london quantum computer



(c) $1 + 1$ sum on ibmq_rome quantum computer



(d) $1 + 1 + C_{in}$ sum on ibmq_16_melbourne quantum computer

(e) Quantum single bit full adder real circuits on different quantum computers

- ibmq_16_melbourne
- ibmqx2
- ibmq_vigo
- ibmq_ourense
- ibmq_london
- ibmq_burlington
- ibmq_essex
- ibmq_rome

Only ibmq_armonk was excluded because it has only one qubit. It can be shown in Fig. 4.7e that the same high level circuit, as depicted in Fig. 4.6, could be implemented in very different ways, according to qunatum computer's topology, i.e. the available physical connections and gates.

After ascertaining that the results of these preliminary tests seemed comparable on the different quantum computers, ibmq_burlington was chosen, because it seemed to respond faster at the time of the tests, to accumulate the results of 10 sessions of 8192 shots for each possible combination of the input values. As shown in Tab.4.2, even the simple addition of two bits, with eventual carry-in, is problematic on a quantum computer, by virtue of the probabilistic nature of the computation results. In particular, when an odd number of bits with 1 value occur at the input, the correct result is not the most frequent one, moreover the correct result presents itself with a percentage close to 25%: therefore the outcome of the circuit does not differ from that which would occur in a purely random event, for example by throwing two perfect coins with an honest toss, one coin for each bit of the result.

The results are better when there are an even number of bits with a unit value at the input. In fact, when a pair of bits with the value 1 occur at the input, about 40% of the results are correct, when all the input bits are null the zero result occurs almost in half of the cases. Therefore, even in these more favorable cases, quantum hardware proves to be inadequate for this simple arithmetic operation.

As further verification, I built an adder with two-bit addends, plus a possible carry-in. The result, this time, requires three bits to encode. I developed the circuit using 9 qubits, using one qubit for the possible carry over in the sum of the least significant bits, as shown in Fig. 4.8. Once I tested the correct functioning with a quantum simulator, I ran it with the only IBM quantum computer currently freely accessible with more than 5 qubits, i.e. ibmq_16_melbourne. As shown in Fig. 4.9d, due to topological constraints of the quantum processor of the IBM computer, not only does the circuit appear much longer and more complex due to the presence of many auxiliary operations, mainly swaps, but it must resort to an additional qubit, therefore using a total of 10 qubits.

The complete table for this adder contemplates 32 possible input configurations, all validated correctly on the simulator, but since the execution times on the quantum machine could be very long, it was preferred to test only a subset of the possibilities. In particular, 16 configurations with an even number of active bits and as many with an odd number of active bits are possible. It was therefore decided to test the only sequence with all zeros, two of the ten possible configurations with a pair of active bits and two of the five configurations with four active bits. Likewise, the only sequence with all one, two of the ten possible configurations with three active bits and two of the five configurations with a single active bit were selected. The possible results with 3 output bits are 8, half

A	B	C_in	Correct result	Correct Pos.	Perc. of correct res.
00	00	0	000	2	22.32
00	01	0	001	6	8.39
00	00	1	001	5	8.84
01	01	0	010	4	11.08
00	01	1	010	4	11.47
01	01	1	011	7	4.69
10	10	1	101	6	6.42
11	01	1	101	6	7.17
11	11	0	110	5	9.10
11	11	1	111	7	6.51

Table 4.3: Full two bits adder results on real quantum device ibmq_16_melbourne: for statistical purposes, the results of 10 executions were collected with the maximum number of 'shots', i.e. 8192, for a total of 81920 executions of each sum on the quantum computer. A and B denote external inputs, C_in a carryover. In the fifth column there are the correct results. In the last two columns are indicated: if correct results are in first or subsequent position; the percentage of correct results.

with an even number of active bits and half with an odd number, so that the ten input combinations have been chosen so as to generate 5 outputs with an odd number of active bits and as many with a even number. More precisely, three configurations were chosen with an even number of active input bits that would also generate a result with an even number of active bits, as many configurations with an odd number of active input bits that would generate a result with an odd number of active bits, and so on.

As shown in Tab. 4.3, on the real quantum processor, the arithmetically correct results had never been the most frequent outcomes. For all zero inputs, the zero result scored second place, with almost a fourth of the results, but for all other input possibilities the right answer scored among fourth and seventh place, furthermore, if the eight possible values encoded by three bits were considered as equally probable events, we would expect a probability of 12.5% for each outcome; for each combination of inputs, 81920 repetitions are carried out, which should constitute a sufficiently large number to guarantee an empirical feedback in line with the theoretical predictions. So the fact that, for all combinations of non-null inputs, the correct result occurs less frequently than a purely random event with eight possible outcomes (sometimes with frequencies even 2-3 times lower), proves unequivocally that, at least on current hardware quantum, it is not possible to attempt to perform even simple arithmetic operations, unless possibly resorting to robust detection and error correction schemes, which, however, would require the physical availability of a much higher number of qubits.

```
def my_2bit_adder(circuit, q, c):
  circuit.cx(q[0],q[6]); circuit.cx(q[1],q[6])
  circuit.cx(q[3],q[6]); circuit.ccx(q[0],q[1],q[5])
  circuit.ccx(q[0],q[3],q[5]); circuit.ccx(q[1],q[3],q[5])
  circuit.cx(q[2],q[7]); circuit.cx(q[4],q[7])
  circuit.cx(q[5],q[7]); circuit.ccx(q[2],q[4],q[8])
  circuit.ccx(q[2],q[5],q[8]); circuit.ccx(q[4],q[5],q[8])
  circuit.measure(q[6],c[0]); circuit.measure(q[7],c[1])
```

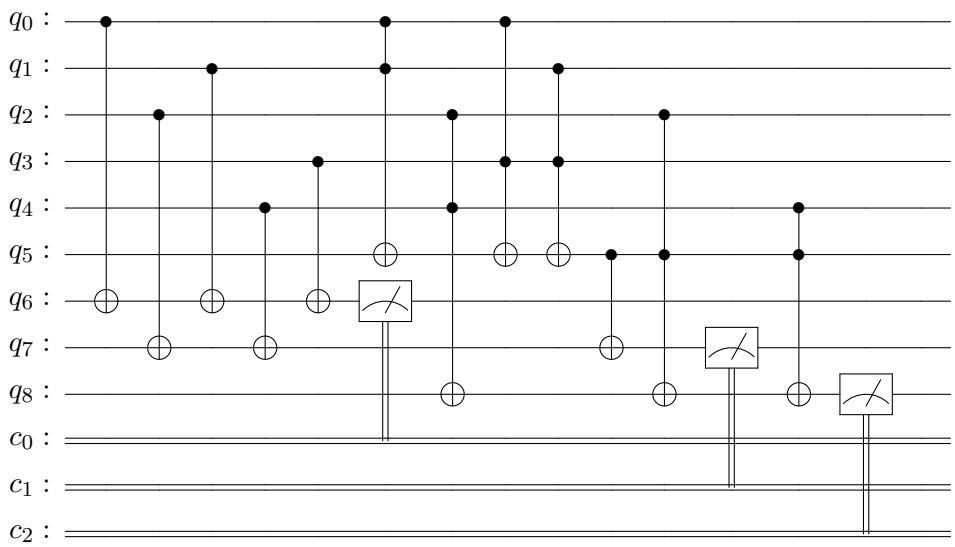


Figure 4.8: Quantum circuit for the sum of a pair of two bits numbers. q_0 is carry-in, q_1, q_2 denote first input (q_1 is the least significant digit), q_3, q_4 is the second addend (q_4 is the most significant digit)

```
    circuit.measure(q[8],c[2])
    return circuit
```

Listing 4.4: Python function to generate a quantum circuite to comute addition for a pair of two bits inputs)

```
from qiskit import QuantumRegister, ClassicalRegister
from qiskit import QuantumCircuit, execute,IBMQ
provider = IBMQ.get_provider(hub='ibm-q')
backend = provider.get_backend('ibmq_16_melbourne')
num=10; shots=8192; trials=num*shots

#00+00
q = QuantumRegister(9,'q'); c = ClassicalRegister(3,'c')
circuit = QuantumCircuit(q,c);
correct='000'
circuit = my_2bit_adder(circuit, q, c)
res=my_job(circuit, backend, num, shots)
ind=1+len([x for x in res if res[x]>res[correct]]);
print('00+00', res, correct, ind, res[correct]/trials*100)
...
#Cin+10+01
q = QuantumRegister(9,'q'); c = ClassicalRegister(3,'c')
circuit = QuantumCircuit(q,c);
circuit.x(q[0]); circuit.x(q[2]); circuit.x(q[3]);
correct='100'
circuit = my_2bit_adder(circuit, q, c)
res=my_job(circuit, backend, num, shots)
ind=1+len([x for x in res if res[x]>res[correct]]);
```

```
print('Cin+10+01', res, correct, ind, res[correct]/trials*100)
...
#Cin+11+11
q = QuantumRegister(9,'q'); c = ClassicalRegister(3,'c')
circuit = QuantumCircuit(q,c);
circuit.x(q[0]); circuit.x(q[1]); circuit.x(q[2])
circuit.x(q[3]); circuit.x(q[4])
correct='111'
circuit = my_2bit_adder(circuit, q, c)
res=my_job(circuit, backend, num, shots)
ind=1+len([x for x in res if res[x]>res[correct]]);
print('Cin+11+11', res, correct, ind, res[correct]/trials*100)
```

Listing 4.5: Python sample code to test the two bit full adder

4.3 Quantum Machine Learning

Quantum Machine Learning is a rapid growing scientific sector, born by the application of quantum computing to supervised or unsupervised machine learning techniques, such as SVM (Support Vector Machine) [Rebentrost et al., 2014].
Two main methods are proposed to exploit quantum computing with SVM: the quantum variational classifier and the quantum kernel estimator [Schuld et al., 2018; Schuld and Killoran, 2018]: these ideas were applied to quantum circuits with 2-qubits [Havlivcek et al., 2018]. It is an interesting open question what type of feature map circuits are classically intractable, but at the same time lead to powerful kernels for classical models such as support vector machines. [Schuld and Killoran, 2018].
Moreover, methods for training set selection were developed to improve (classical) SVM's scalability without deprecating its classification accuracy [Acampora et al., 2018].
I'll develop and test the application of suitable training set selection tecniques to quantum SVM, to improve classification's accuracy.

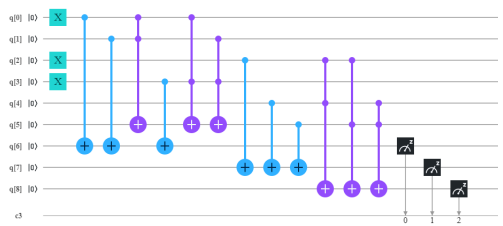
4.3.1 Quantum Machine Learning Perspectives

4.4 QSVM intro

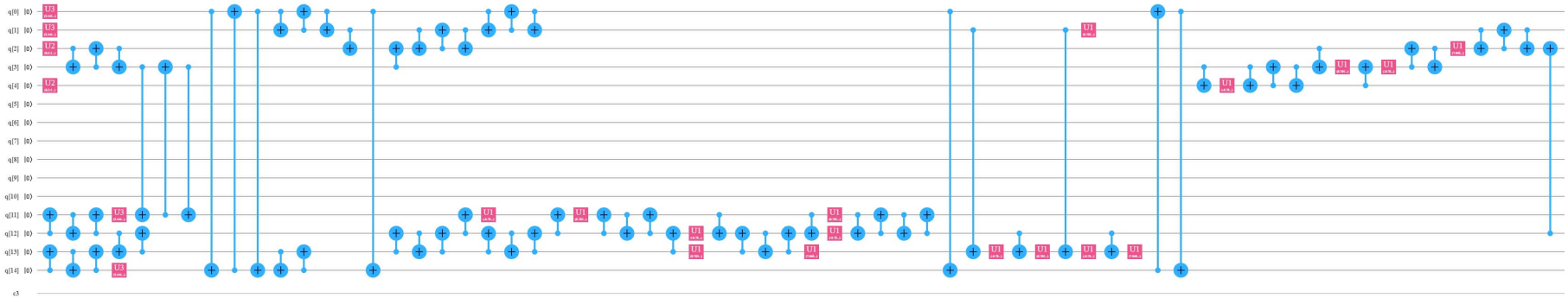
Over the years there has been a lot of demonstration regarding quantum computation. There are illustrations of how this emerging technology utilizes its effects to add a new dimension to this era of technological advancement. It opens new doors of immense possibilities for the researchers.

In 2016 Microsoft and Cambridge University researchers have implemented a few machine learning algorithms in a quantum environment and compared the complexity over a classical computer. It's been found a significant speedup[9] in quantum methods. The key factor behind this speedup was the exponential number of states in quantum computers.

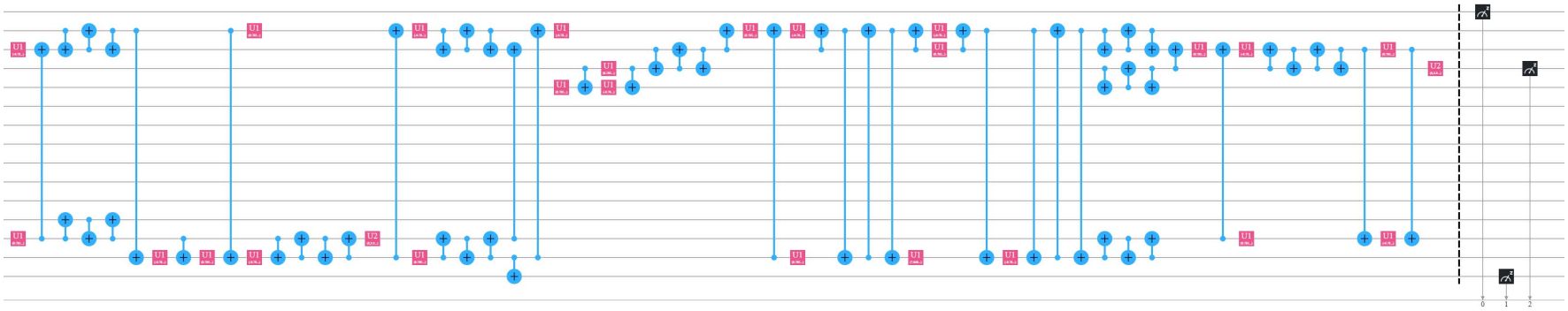
Noise in quantum operation and output is a big challenge. According to research on low depth circuits [Verdon et al., 2017], it has been proposed a method which employs the quantum approximate optimization algorithm as a subroutine in order to approximate sample from Gibbs states of Ising Hamiltonians. [Verdon et al., 2017] used this approximate Gibbs sampling to train neural networks for which they demonstrate training convergence for numerically simulated noisy circuits with depolarizing errors of rates of up to 4%. After analyzing all these research works we found out that there are the



(a) $10 + 01 + C_{in}$ sum on ibmq_qasm_simulator



(b) $11 + 11$ sum on ibmq_16_melbourne quantum computer (part 1)



(c) $11 + 11$ sum on ibmq_16_melbourne quantum computer (part 2)

(d) Quantum two bits full adder: abstract circuit on on ibmq_qasm_simulator versus real circuit on ibmq_16_melbourne quantum computer

enormous possibilities waiting for machine learning in quantum technology and there are a lot of scopes for improvement.

Chapter 5

Quantum Support Vector Machine

One of the fundamental task within supervised machine learning is the achievement of a classifier. For this kind of problem, we assume that a function exists that maps elements of a n-dimensional space onto a set of labels for all possible equivalence classes. Usually, this function is unknown and it could be very complex, maybe impossible to express in a mathematical closed form given data from a training set A and a test set B, both subset of the ensemble of all conceivable data.

Both training and test sets are assumed to be labeled by a map unknown to the algorithm. The training algorithm only receives the labels of the training data T. The goal is to infer an approximate map on the test set such that it agrees with high probability with the true map, not only on training data (overfitting) but above all on the never seen before test data. For such a learning task to be meaningful it is assumed that there is a correlation between the labels given for training and the true map. A classical approach to constructing an approximate labeling function uses so called support vector machines (SVMs).

Classical Support Vector Machine is hugely popular, in particular, for efficient binary data classification, whether it is directly linearly separable or it requires the use of an appropriate kernel function, the so called kernel trick: the data gets mapped non-linearly to a high dimensional space, the feature space, where a hyperplane is constructed to separate the labeled samples. For pattern recognition and image processing problem, a feature map starts from an initial set of measured data and incorporates features intended to be informative and non-redundant, facilitating the subsequent learning and generalization steps, and in some cases leading to better human interpretations. In general what we do is basically dimension reduction. It involves reducing the number of resources needed to explain an oversized set of information. once applying the analysis of complicated data, one in every of the most important issues stems from the number of variables concerned. Analysis with an oversized range of variables usually needs a large quantity of memory and computation power, and will even cause a classification algorithm to overfit to training samples and generalize poorly to new samples. Once the input data to an algorithm is just too large to be processed and is suspected to be redundant (for example, the identical measure is provided in each pound and kilograms), then it may be remodelled into a reduced set of features, named a feature vector. The methods of deciding a set of the initial features is named feature choice. The chosen features are expected to contain the relevant info from the input file, so the specified task may be performed by using the reduced illustration rather than the entire initial data. Instead of feature choise, often more sophisticated pre-processing techniques and feature extraction algorithms are used. One of them is PCA, Principal Component Analysis, but usually the original data is normalized before performing the PCA, at least to assure zero means, often to make unit variance.

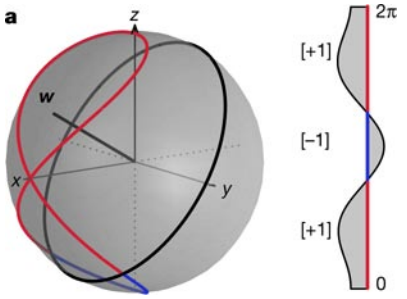


Figure 5.1: Feature map representation for a single qubit. (Fig. 1a in [Havlíček et al., 2019]).

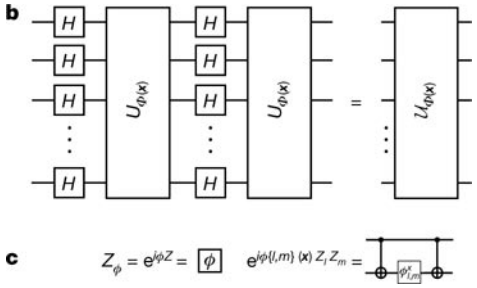


Figure 5.2: The general circuit is formed by products of single- and two-qubit unitaries that are diagonal in the computational basis. In our experiments, both the training and testing data are artificially generated to be perfectly classifiable using the feature map. (Fig. 1b in [Havlíček et al., 2019]).

Anyway, Support Vector Machine could suffer throubles on a classical machine when higher dimensions are involved or very large datasets are taken up, so, in order to enhance the efficiency of Support Vector Machine, the idea of running it on a quantum machine takes over.

Quantum Support Vector Machine[Ahmed, 2019; Arodz and Saeedi, 2019; Bishwas et al., 2018; Chatterjee and Yu, 2016; Ding et al., 2019; Havlíček et al., 2019; Havlivcek et al., 2018; Rebentrost et al., 2014; Schuld et al., 2018; Schuld and Killoran, 2018] is indeed a sort of quantum implementation of Support Vector Machine: in this algorithm's version, classical data should be transformed into quantum data and then analysed over a quantum computer.

The Quantum Support Vector Machine algorithm retrieves and processes the data in a classical way but training is done by the help of quantum state space, that is data model features are mapped in non linearly to a quantum state

A Quantum version of Support Vector Machine was developed even for annealing quantum computers produced by D-Wave[Willsch et al., 2019].

5.1 IBM QSVM

Qiskit is an open-source framework for quantum computing that was founded by IBM Research to allow software development for their cloud quantum computing service, IBM Q Experience, but nowadays contributions are also made by external supporters. In principle, Qiskit can be used for any quantum hardware that follows the reversible circuit model for universal quantum computation and it currently supports superconducting qubits and trapped ion, but obviously its primary target are quantum system offered by

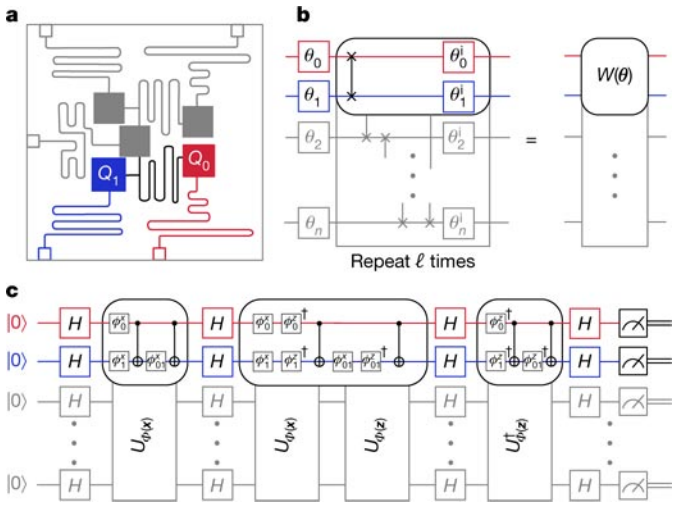


Figure 5.3: Experimental implementations. **a** Schematic of the five-qubit quantum processor. The experiment was performed on qubits Q0 and Q1, highlighted in the image. **b** Variational circuit used for our optimization method. The two top qubits depict the circuit implemented. **c**, Circuit to directly estimate the fidelity between a pair of feature vectors for data x and z as used for our second method. (Fig. 2 in [Havlíček et al., 2019]).

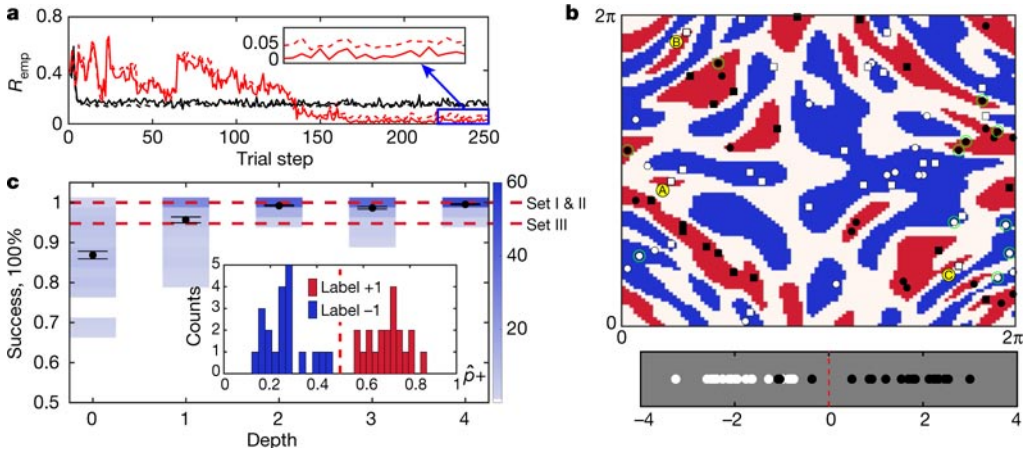


Figure 5.4: Convergence of the method and classification results. **a** Convergence of the cost function after 250 iterations of Spall's SPSA algorithm. Red (or black) curves correspond to $l = 4$ (or $l = 0$). We train three datasets per depth and perform 20 classifications per trained set. **b** Example data used for both methods in this work. The data labels (red for $+1$ label and blue for -1 label) are generated with a gap of 0.3 (white areas). The training set with 20 points per label is shown as white and black circles. For the quantum kernel estimation method we show the support vectors (green circles) and a classified test set (white and black squares). Three points are misclassified, labelled as A, B and C. **c** The classifications results are shown as blue histograms for all three randomly chosen unitaries (a total of 60 classifications per depth and 20 data points per classification per label), with mean values represented by black dots. The error bar is the standard error of the mean. The inset shows histograms as a function of the probability of measuring label $+1$ for one test set of 20 points per label obtained with an $l = 4$ classifier circuit, depicting classification of this set with 100% success. The dashed red lines show the results of our direct kernel estimation method for comparison, with Sets I and II yielding 100% success and Set III yielding 94.75% success. (Fig. 3 in [Havlíček et al., 2019]).

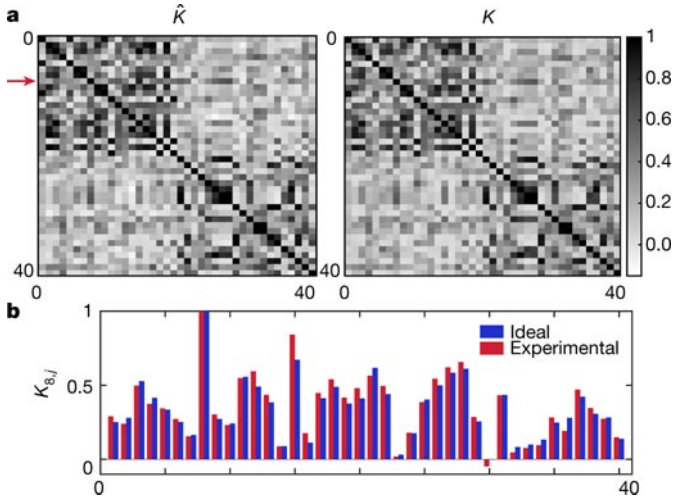


Figure 5.5: Kernels for Set III. a, Experimental (left) and ideal (right) kernel matrices containing the inner products of all data points used for training Set III (round symbols in Fig. 3b). A cut through row 8 (indicated by the red arrow in a) is shown in b, where the experimental (or ideal) results are shown as red (or blue) bars. (Fig. 4 in [Havlíček et al., 2019]).

IBM via cloud access.

Qiskit provides the ability to develop quantum programs with at least two principal approach. At a very low level, Qiskit supports the machine code-like of OpenQASM; in a certain sense, OpenQASM programming is even lower level than the machine language of a classic computer, because it requires the construction of a quantum circuit acting at the single qubit and gate level, where the machine language of a conventional CPU, even for an 8 bit, 40 years ago, commodity processor offered abstraction such as general and special purpose multibit registers, relatively complex logical and arithmetic operations, sophisticated indirect and indexed memory access systems, a flow control based on conditioned and unconditional jumps and so on.

Moreover, Qiskit provides abstract levels suitable for developers that prefer a development environment more suited to the expectations of a modern programming language, avoiding quantum circuits details. To support this needs, Qiskit contains four principal parts or elements, unfortunately named in an evocative way according to ancient prescientific theories, dating back to about 2500 years ago, thanks to the contribution of famous and influential Greek philosophers such as Anaximenes of Miletus, Empedocles and Plato. At least, until now, IBM has not released portions of the library dedicated to the quintessence contemplated by Aristotle.

Qiskit Terra is the core foundation and it provides tools to explicitly construct quantum circuits, assembling quantum gates, but it also provides tools to allow quantum circuits to be optimized for a particular quantum processor's topology, as well as managing batches of jobs, to be scheduled on local simulator or cloud access quantum devices and simulators.

Qiskit Aer is conceived to accelerate software development, offering both high-performance simulators hosted locally on the developer's device, as well as IBM High Performance Computing resources available through the cloud interface; these simulators can emulate a perfect quantum device but even reproduce several sophisticated noise models; in concert with Qiskit Terra, a highly configurable noise model for studying quantum computing in the NISQ regime is offered. More precisely, Aer includes the following simulators:

- state_vector Simulator is an auxiliary backend for Qiskit Aer. It simulates the

ideal execution of a quantum circuit and returns the final quantum state vector of the device at the end of simulation. This is useful for education, as well as the theoretical study and debugging of algorithms.

- qasm_simulator is considered the main Qiskit Aer backend; it emulates execution of a quantum circuits on a real device and returns measurement counts; to that end, it includes highly configurable noise models and can even be loaded with automatically generated approximate noise models based on the calibration parameters of actual hardware devices; there are multiple methods that can be used that simulate different circuits more efficiently, such as
  - statevector, which uses a dense statevector simulation;
  - stabilizer, which uses a Clifford stabilizer state simulator that is only valid for Clifford circuits and noise models;
  - extended_stabilizer, which uses an approximate simulator that decomposes circuits into stabilizer state terms, the number of which grows with the number of non-Clifford gates [Bravyi et al., 2019];
  - matrix_product_state, which uses a Matrix Product State (MPS) simulator;
- unitary_simulator allows simulation of the final unitary matrix implemented by an ideal quantum circuit; it is also conceived mostly for education and algorithm studies.
- pulse_simulator simulates continuous time Hamiltonian dynamics of a quantum system, with controls specified by pulse Schedule objects, and the model of the physical system specified by PulseSystemModel objects.

Qiskit Ignis is involved with addressing noise and errors: it contains tools for characterizing noise in near-term devices, as well as allowing computations to be performed in the presence of noise. This is includes tools for benchmarking near-term devices, error mitigation and error correction.

Qiskit Aqua (Algorithms for QUantum Applications) is projected to allow design algorithms and implement applications without without worrying too much about the details of the quantum circuits. Aqua provides high level tools, in the form of a library of quantum algorithms and components, that can already speedup application development in chemistry, machine learning, optimization and finance, favoring leverage near-term devices.

A key concept in classification methods is that of a kernel. Data cannot typically be separated by a hyperplane in its original space. A common technique used to find such a hyperplane consists on applying a non-linear transformation function to the data. This function is called a feature map, as it transforms the raw features, or measurable properties, of the phenomenon or subject under study. Classifying in this new feature space – and, as a matter of fact, also in any other space, including the raw original one – is nothing more than seeing how close data points are to each other. This is the same as computing the inner product for each pair of data in the set. In fact we do not need to compute the non-linear feature map for each datum, but only the inner product of each pair of data points in the new feature space. This collection of inner products is called the kernel and it is perfectly possible to have feature maps that are hard to compute but whose kernels are not.

The QSVM algorithm applies to classification problems that require a feature map for which computing the kernel is not efficient classically. This means that the required computational resources are expected to scale exponentially with the size of the problem.

QSVM uses a Quantum processor to solve this problem by a direct estimation of the kernel in the feature space. The method used falls in the category of what is called supervised learning, consisting of a training phase (where the kernel is calculated and the support vectors obtained) and a test or classification phase (where new data without labels is classified according to the solution found in the training phase).

Internally, QSVM will run the binary classification. If the data has more than 2 classes then a multiclass_extension is required to be supplied, but I preferred to "manually" encode a one-against-

I acknowledge the use of IBM Quantum services for this work. The views expressed are those of the author, and do not reflect the official policy or position of IBM or the IBM Quantum team.

Chapter 6

Analysis and results

6.1 Preliminary remarks

For this work, I choose free and preferably open-source hardware and software solutions. Among the development environments for quantum software, I selected IBM Quantum Experience[1], also by virtue of free cloud access to real quantum computers and extensive online community support. It is possible to use IBM Q services with Circuit Composer, QASM language (a sort of Quantum Assembly) [Bishop, 2017; Cross et al., 2017] or Qiskit library [Abraham et al., 2019], an open-source framework for quantum computing, based on the Python programming language[2]. Training sessions for quantum support vector machine were conducted using IBM's Qiskit Aqua framework, which contains a library of cross-domain quantum algorithms upon which applications for near-term quantum computing can be built. Aqua is designed to be extensible, and employs a pluggable framework where quantum algorithms can easily be added. It currently allows the developers to program chemistry simulations, machine learning sessions, optimization and finance applications on near-term quantum computers. The most relevant elements of Aqua used during these simulations are:

- the QSVM class, which run the binary classification based on QSVM algorithm;
- the SecondOrderExpansion class, which is a sub-class of PauliZExpansion where z_order is fixed at 2; it generate a feature map, according to the number of input features and the depth, i.e. the number of repeated circuits (obviously, at least 1; on current hardware, rarely can be grater than 2); entanglement is a very important parameter, because it generate the qubit connectivity according to a predefined topology; full entanglement allows to connect every qubit to each other, while linear entanglement connects each qubit only to the next one;
- the QuantumInstance class, which holds a Qiskit Terra backend as well as configuration for circuit transpilation and execution, including shots, i.e. the number of repetitions of each circuit, for sampling reasons; a QuantumInstance object have to be provided to an Aqua algorithm so the algorithm will run the circuits it needs using the instance;

From Qiskit Aer, I used mostly StatevectorSimulator, an ideal quantum circuit statevector simulator. In Qiskit IBM Q, there is the IBM Quantum Provider, which make it possibile to vary the back-end while running on different machine: I used both simulation on classical computer both remote, cloud access to IBM Q quantum processor.

[1] `http://quantum-computing.ibm.com/`
[2] `http://www.python.org/`

Because of its free and open-source nature, but even for for very rich libraries and its extensive community support, Python is by now a very popular programming language. Moreover, Python is often considered as the most powerful language that is still human readable. In the last few years, Python has conquered many followers for Machine Learning and Artificial Intelligence projects, thanks also to the diffusion of environments such as PyTorch and Tensorflow, but in general it is experiencing a considerable diffusion throughout the scientific community. So all algorithms in this thesis are implemented using Python; after a series of initial experiments conducted from the command line or using local development environments, such as the Python IDE PyCharm [3], I preferred to take advantage of the agility offered by Jupyter Notebook[4]. The Jupyter Notebook is an open-source interactive computing environment offered as a web application that allows developers to create and share documents that contain both "live" code (writing and running it interactively), both narrative text, graphics, images and equations. The Jupyter Notebook is increasingly popular for numerical simulation, statistical modeling and data visualization, and even machine learning, favoring sharing with developers communities using email, cloud storage services like Dropbox, version control systems like github. Jupyter Notebook supports a versatile system based on interaction with different kernels, that run executable code written in various programming languages and return output back to the notebook web application, but the core programming languages supported by Jupyter are Julia, Python, and R. Indeed, Jupyter started as a spin-off project of IPython (Interactive Python[5]), a command shell for interactive computing in multiple programming languages, originally developed for the Python programming language; even nowadays, Jupiter's default kernel is IPython, to efficiently run Python code. The Notebook support the interactive computing paradigm, offering a web-based application suitable for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results.

Therefore I used Python 3.6.9 [GCC 8.4.0] inside Google Colab, a free Jupyter notebook environment that requires no setup and runs entirely (writing, running, sharing code) on the cloud, but sometimes I run some code on Python 3.7.6 [GCC 7.3.0] inside IBM Quantum Experience Qiskit notebooks. For classical Support Vector Machine tests, in Python the reference library, or rather de facto standard, is scikit-learn library[6]; I used 0.22.2.post1 version inside Google Colab and 0.21.0 version inside Qiskit notebooks. Anyway, I always used the latest Qiskit 0.19.2 version, with qiskit-terra 0.14.1 version, qiskit-ignis 0.3.0 version, qiskit-aqua 0.7.1 version, qiskit-aer 0.5.1 version, qiskit-ibmq-provider 0.7.1 version.

6.2 First QSVM tests

During a preliminary phase, simple tests were carried out with the QSVM algorithm, both on the simulator and on real quantum systems, to verify the correct functioning of the classifier and to compare accuracy with classical SVM. In all selected cases, the performance of the QSVM has been evaluated with at least one simulator, the local simulator, statevector_simulator offered by Qiskit's Aer, which simulates the operation of an ideal quantum computer, free of topological limitations and not affected by any kind of error, or the ibmq_qasm_simulator, with free access via cloud, which offers a more accurate simulation, reproducing certain types of errors (in this regard, the default parameters have been confirmed, although probably a bit optimistic compared to the real

[3] `http:///www.jetbrains.com/pycharm/`
[4] `http://jupyter.org/`
[5] `https://github.com/ipython/ipython`
[6] `http://scikit-learn.org/`

performance of current quantum computers). For simple test, I choose prototypical two class classification task:

- one dimensional cases
  - linearly separable classes, for various noise levels
  - not linearly separable classes, for various noise levels
- two dimensional cases
  - linearly separable classes, for various noise levels
  - not linearly separable classes (XOR problem)
  - not linearly separable classes (circle)
- three dimensional cases
  - linearly separable classes, for various noise levels
  - not linearly separable classes (sphere)

Moreover, a simple three classes scenario was tested in two dimensions, using the vertices of an equilateral triangle. For the sake of brevity, I have not included in this report the data of all the synthetic cases carried out.

QSVM constructor requires a feature_map, to describe the quantum kernel, a training set and a test set, so for each test I created three dataset (each one balanced with to respect to the two classes):

- a training set, indispensable to perform any training algorithm;
- a test set, to check each trained model on data similar to that used for training
- a validation set, to try each trained model on more data, usually more difficult cases (for example, with applied bigger noise)

For validation set, I report separately success rates for each class; the success rates in recognizing the elements of each class correspond (or, at least, are proportional) to the elements on the main diagonal of a confusion matrix. Indeed, for two classes classification task often a confusion matrix is used: it is usually drawn as a square that allows simple visualization of the performance of a predictive model. Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class (depending on the authors, the matrix can be transposed). he name expresses the fact that this table considerably facilitates the possibility of ascertaining to what extent the supervised machine learning system, after the training phase, confuses the elements of the two classes or not, i.e. how often the decision algorithm mislabel one element as belonging to the other class. If we identify the first class with True and the second with False, then it makes sense to talk about True Positive (TP) as True elements recognized as True, True Negative (TN) as False elements considered False, while False Positive (FP) are False elements recognized as True and, finally, False Negative (FN) are True elements considered False. Common statistical measures of the performance of a binary classification model are Sensitivity and specificity. Sensitivity measures the proportion of actual true elements that are correctly identified as such, therefore it expresses the avoidance of false negatives and it is often called the true positive rate (TPR):

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN}$$

Specificity, instead, measures the proportion of actual negatives that are correctly identified as such, it expresses the avoidance of false positive and it is often called the true negative rate (TNR):

$$TNR = \frac{TN}{N} = \frac{TN}{TN + FP}$$

(where P denotes all positive and T all negative elements of available dataset). A perfect predictor would have 100% sensitivity and 100% specificity, so that it will leave only zeros outside the main diagonal of the confusion matrix, but this is an eventuality that does not happen concretely for non trivial classification problems. Anyway, for any statistical decision system, based on machine learning or not, there is usually a trade-off between the measures of specificity and sensitivity. So success rate for the two classes on validation set cam be considered as true positive rate (TPR) and true negative rate (TNR), the more relevant information in a binary classification task.

6.2.1 QSVM for a 1-dim feature space - linearly separable case

The first case examined was a binary classification problem in a single-dimensional feature space. In this toy example, I randomly generated numbers centered around zero and unity, with function in List. 6.3, as if I wanted to distinguish whether a digital electrical signal is taking on a value of 0 or 1.

The first time, I generated perfectly separated sets for training, because the minimum distance between the set of samples for 0 (true in negative logic, i.e. active-low) and 1 (false in negative logic) is $\frac{1}{2}$. I tried the QSVM, with code in List. 6.3 both with local statevector_simulator backend both with a real quantum device, ibmq_london. In both cases, the classifier generated obtained 100% accuracy both on the test set and on the validation set (both tests were generated using a radius doubled with to respect to that used for the training set, so that the sets corresponding to true and false could actually touch each other). By examining the classifiers generated in the two execution environments (local simulator and cloud quantum backend) starting from random sequences set for training, tests and validation sets, we find that the selected support vectors for the two classes are the same, while other parameters are very similar (alphas, bias).

During local simulator execution, the QSVM training generated this parameters:

```
'alphas': array([2.52204406, 0.92162106, 1.60042225])
'bias': array([0.11386612])
'support_vectors': array([[ 0.81237196], [ 0.20519846], [-0.22907846]])
'yin': array([-1.,  1.,  1.])
```

During cloud execution on a real quantum computer, the QSVM training generated this parameters:

```
'alphas': array([2.67232031, 1.05723112, 1.61503035])
'bias': array([0.13254928])
'support_vectors': array([[ 0.81237196], [ 0.20519846], [-0.22907846]])
'yin': array([-1.,  1.,  1.])
```

Later, I repeated the experiment, increasing the radius to $\frac{1}{2}$ to generate the points of the training set and $\frac{9}{16}$ for the test and validation sets. In this way, the samples of the two classes used for training are close and the samples used for testing and evaluation can be partially overlapped.

On local simulator, QSVM correctly predicted 91.67% of test elements, while on validation set accuracy was 86.67% for true set and 96.67% on false set.

backend	alg/kernel	test (%)	True P (%)	True N (%)
Aer statevector_simulator	QSVM	91.67	86.67	96.67
ibmq_london	QSVM	95.00	90.00	96.67
sklearn.svm.SVC	rbf	95.00	90.00	100.00
sklearn.svm.SVC	linear	95.00	96.67	96.67

Table 6.1: QSVM and SVM comparison on 1 dimension dataset with low noise: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. "rbf" denotes a radial basis function kernel.

On a real quantum computer, once again ibmq_london, QSVM has identified the same support vectors and it correctly predicted 95% of test elements, while on validation set accuracy was 90.0% for true set and 96.67% on false set.

During local simulator execution, the QSVM training generated this parameters:

```
'alphas': array([125.79289601, 5.49083558, 120.30206043])
'bias': array([1.56265853])
'support_vectors': array([[ 0.53301875], [-0.43020758], [ 0.4626269 ]])
'yin': array([-1.,  1.,  1.])
```

During cloud execution on a real quantum computer, ibmq_london, the QSVM training generated this parameters:

```
'alphas': array([104.94204362, 6.14408706,  98.79795647])
'bias': array([1.42964367])
'support_vectors': array([[ 0.53301875], [-0.43020758], [ 0.4626269 ]])
'yin': array([-1.,  1.,  1.])
```

Then I checked the classifier accuracy against classical SVM, using scikit-learn 0.23.1 pacage with code in List. 6.4; when I used linear classifier, I got 95.0% global accuracy on test set (96.67% on true set and 93.33% on false set) and 96.67% accuracy on both validation sets. After that, I used rbf kernel, i.e. radial basis functions, and I got 95.0% global accuracy on test set (93.33% on true set and 96.67% on false set), 90.0% on validation true set and lastly 100% accuracy on validation false set. For exploratory purposes only, the other kernels offered by scikit-learn for SVM have been evaluated and I report the results of accuracy only on the two validation sets in the format true / false: sigmoid kernel 66.67/56.67%, polynomial with degree 1 90.0/100.01%, polynomial with degree 2 96.67/96.67% and polynomial with degree 3 93.33/96.67%.

Noting that all the tests were carried out with sequences of values generated randomly but blocked to test the different classifiers in a uniform and repeatable way, it can be said that in this decidedly simple scenario the QSVM was able to provide performances similar to the classical SVM.

As a next step, I decided to increase the radii for the generation of sequences of random values, so that the two sets were no longer perfectly separable; more specifically, the radius to generate the training set had the value $\frac{9}{16}$, so that there was potentially an overlapping width $\frac{1}{8}$ between the two classes. Then I set the radius to generate the test and validation sets to the $\frac{5}{8}$ value, so that there was potentially an overlapping $\frac{1}{4}$ segment between the two classes.

On local simulator, QSVM correctly predicted 80.00% of test elements, while on validation set accuracy was 76.67% for true set and 90.00% on false set. On a real quantum computer, ibmq_london, QSVM has identified also in this case the same support vectors and it correctly predicted 80.00% of test elements, while on validation set accuracy was 83.33% for true set and 86.67% on false set, as shown in Tab. 6.2. Only for this case, I

backend	alg/kernel	test (%)	True P (%)	True N (%)
Aer statevector_simulator	QSVM	80.00	76.67	90.00
ibmq_london	QSVM	80.00	83.33	86.67

Table 6.2: QSVM and SVM comparison on 1 dimension dataset with high noise: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

also attempted the execution on single qubit ibmq_armonk quantum computer; in this platform, QSVM has identified five support vectors, that is two more; more precisely, two support vector are in common with respect to the previous executions, then three other vectors have been selected; it correctly predicted 85.0% of test elements.

During local simulator execution, the QSVM training generated this parameters:

```
'alphas': array([69.62656342,  6.91858324, 62.70798014])
'bias': array([1.32495047]),
'support_vectors': array([[ 0.57037937], [-0.47034937], [ 0.47247277]])
'yin': array([-1.,  1.,  1.])
```

During cloud execution on a real quantum computer, ibmq_london, the QSVM training generated this parameters:

```
'alphas': array([71.4176846 ,  6.6120571 , 64.80566182])
'bias': array([1.46213834])
'support_vectors': array([[ 0.57037937], [-0.47034937], [ 0.47247277]])
'yin': array([-1.,  1.,  1.])
```

During cloud execution on a real quantum computer, single qubit ibmq_armonk, the QSVM training generated this parameters:

```
'alphas': array([ 8.54823924, 12.97649675,  1.39973566,  4.34586523, 15.77913631])
'bias': array([0.5403182])
'support_vectors': array([[ 0.618765  ], [ 0.59224903], [ 0.41198377], [-0.47034937], [ 0
'yin': array([-1., -1.,  1.,  1.,  1.])
```

```
def add_noise_1d(x, r):
  for i in range(x.shape[0]):
    x[i] += random.random() * 2 * r - r
  return x
```

Listing 6.1: Python function to add random noise to one dimensional array

```
#training set creation
n=10; r=1/4
train_true=np.zeros((n,1))
train_true = add_noise_1d(train_true,r)
train_false=np.ones((n,1))
train_false = add_noise_1d(train_false, r)

#test and validation sets creation
```

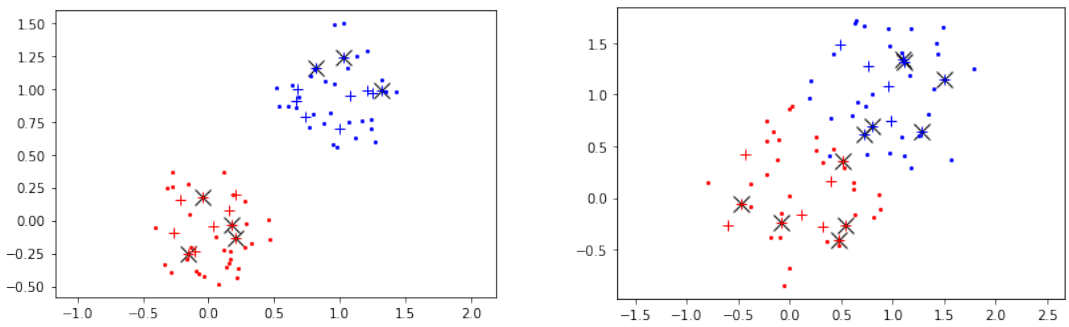
```
n2=3*n; r2=1/2
test_true=np.zeros((n2,1))
test_true = add_noise_1d(test_true, r2)
test_false=np.ones((n2,1))
test_false = add_noise_1d(test_false, r2)

val_true=np.zeros((n2,1))
val_true = add_noise_1d(val_true,r2)
val_false=np.ones((n2,1))
val_false = add_noise_1d(val_false, r2)

training_data = {'T': train_true,'F': train_false}
testing_data = {'T': test_true,'F': test_false}
```

Listing 6.2: Python sample code to generate training, test and validation sets

```
from qiskit import Aer
#backend = provider.get_backend('ibmq_qasm_simulator')
#backend = Aer.get_backend('statevector_simulator')
backend = provider.get_backend('ibmq_london')
shots=8192

num_qubits = 1
feature_map = SecondOrderExpansion(feature_dimension=num_qubits,
                                   depth=2,entanglement='full')
svm = QSVM(feature_map, training_data,testing_data)
quantum_instance = QuantumInstance(backend,shots=shots,
                                   skip_qobj_validation=False)
result = svm.run(quantum_instance)
print('Accuracy: ', result['testing_accuracy'],'\n')
print(result)
pred_true = svm.predict(val_true,quantum_instance)
print('Accuracy on true',
      np.count_nonzero(pred_true==1)/pred_true.size)
pred_false = svm.predict(val_false,quantum_instance)
print('Accuracy on false',
      np.count_nonzero(pred_false==0)/pred_false.size)
```

Listing 6.3: Python code to apply QSVM to generated data for 1-dim classification task

```
from sklearn import svm
clf = svm.SVC(kernel='linear') #then clf = svm.SVC(kernel='rbf')
X=np.concatenate((train_true, train_false))
y=np.concatenate((np.zeros((n,1)), np.ones((n,1)))).ravel()
clf.fit(X, y)
print(np.count_nonzero(clf.predict(test_true)==0)/
      test_true.shape[0]*100)
print(np.count_nonzero(clf.predict(test_false)==1)/
      test_false.shape[0]*100)
print(np.count_nonzero(clf.predict(val_true )==0)/
```



(a) Support vectors selected by QSVM for low noise datasets.

(b) Support vectors selected by QSVM for big noise datasets.

Figure 6.1: QSVM applied to a simple two dimension dataset; black crosses denote support vectors chosen by QSVM algorithm, plus markers represent training set datapoint, small dots are validation sets datapoint. Red and blu are used to distinguish classes.

```
        val_true.shape[0]*100)
print(np.count_nonzero(clf.predict(val_false)==1)/
        val_false.shape[0]*100)
```

Listing 6.4: Python code to apply classical SVM to generated data for 1-dim classification task

6.2.2 QSVM for a 1-dim feature space - not linearly separable case

6.2.3 QSVM for a 2-dim feature space - linearly separable case

The next step to test QSVM capabilities consisted in a test on very simple 2d feature space clasification task. I have assigned to the opposite vertices of a square, of coordinates (0; 0) and (1; 1) respectively, the values true and false. Then I generated training sets by applying random noise to these two values, selecting a radius equal to a quarter of the diagonal of the square, $\frac{\sqrt{2}}{4}$ To generate the test and the validation set, I expanded the radius to $\frac{1}{2}$, as shown in List. 6.6, obtaining data points shown in Fig. 6.1a.

I tried the QSVM, with code in List. 6.7 both with local statevector_simulator backend both with a real quantum device, ibmq_london. QSVM on local simulator chose three support vectors for each class and scored with 90% accuracy on test set; on validation set accuracy was 90% for true elements and 96.67% for false. QSVM on quantum computer chose four support vectors for true class and three support vectors for false class; it scored with 88.33% accuracy on test set; on validation set accuracy was 86.67% for true elements and 100.0% for false. For comparison, classical SVM scored 100.0% both on test set and validation sets, with several kernel choices (except sigmoid kernel, not reported here for convenience); it used three support vectors per class with linear kernel and polynomial kernel of degree 1, four support vector per class with radial basis functions kernel and only two support vectors per class with polynomial kernel of degree 2 or 3, as shown in Tab. 6.3.

During local simulator execution, the QSVM training generated this parameters:

```
'alphas': array([1.71994012, 2.31538821, 1.00775775, 1.4248689,
                 1.07434067, 2.54387655])
'bias': array([0.59862379])
'support_vectors': array([
       [ 1.02751184,  1.23596748], [ 1.31566984,  0.9830822 ],
```

backend	alg/kernel	test (%)	True P (%)	True N (%)
Aer statevector_simulator	QSVM	90.00	90.00	96.67
ibmq_london	QSVM	88.33	86.67	100.00
sklearn.svm.SVC	rbf	100.00	100.00	100.00
sklearn.svm.SVC	linear	100.00	100.00	100.00
sklearn.svm.SVC	poly 2 deg	100.00	100.00	100.00
sklearn.svm.SVC	poly 3 deg	100.00	100.00	100.00

Table 6.3: QSVM and SVM comparison on a simple 2 dimensions dataset: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

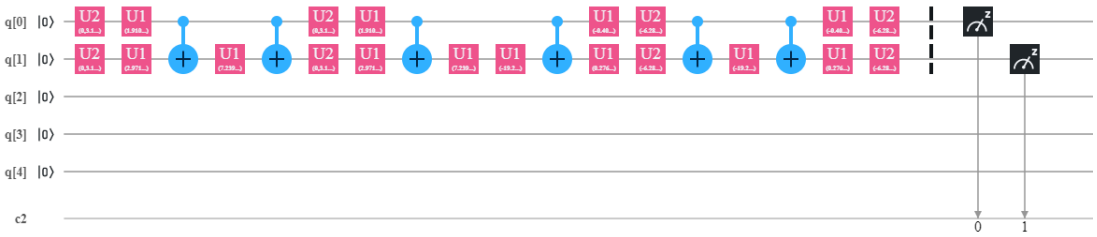


Figure 6.2: QSVM circuit for a linearly separable bi-dimensional feature space on ibmq_london quantum computer

```
       [ 0.82209995,  1.16294195], [ 0.20399493, -0.13820954],
       [-0.04519968,  0.17753551], [-0.15804101, -0.24959031]])
'yin': array([-1., -1., -1.,  1.,  1.,  1.])}
```

During cloud execution on a real quantum computer, the QSVM training generated this parameters:

```
'alphas': array([3.11537135, 2.79521063, 0.16353942, 1.51084673,
                 1.3310395, 0.07460427, 3.15763105])
'bias': array([0.71337578])
'support_vectors': array([
       [ 1.02751184,  1.23596748], [ 1.31566984,  0.9830822 ],
       [ 0.82209995,  1.16294195], [ 0.20399493, -0.13820954],
       [-0.04519968,  0.17753551], [ 0.17557884, -0.03350857],
       [-0.15804101, -0.24959031]])
'yin': array([-1., -1., -1.,  1.,  1.,  1.,  1.])
```

Afterwards I generated new training sets by applying random noise to the two reference values, selecting this time a radius equal to half of the diagonal of the square, $\frac{\sqrt{2}}{2}$ To generate the test and the validation set, I expanded the radius to 0.9, as shown in List. 6.6, obtaining data points shown in Fig. 6.1a.

Even this time, I tried the QSVM, with code in List. 6.7 both with local statevector_simulator backend both with a real quantum device, ibmq_london. QSVM on local simulator chose five support vectors for each class and scored with 71.67% accuracy on test set; on validation set accuracy was 70.00% for true elements and 56.67% for false. QSVM on quantum computer chose five support vectors for true class (four equals to that selected by the simulator) and six support vectors for false class (only two commons to simulator's choices); it scored with 71.67% accuracy on test set; on validation set accuracy

backend	alg/kernel	test (%)	True P (%)	True N (%)
Aer statevector_simulator	QSVM	71.67	70.00	56.67
ibmq_london	QSVM	71.67	76.67	53.33
sklearn.svm.SVC	rbf	98.33	100.00	96.67
sklearn.svm.SVC	linear	98.33	93.33	96.67
sklearn.svm.SVC	poly 2 deg	96.67	100.00	86.67
sklearn.svm.SVC	poly 3 deg	96.67	100.00	86.67
sklearn.svm.SVC	poly 4 deg	95.00	100.00	86.67
sklearn.svm.SVC	poly 5 deg	96.67	100.00	86.67

Table 6.4: QSVM and SVM comparison on a simple 2 dimensions dataset: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

was 76.67% for true elements and 53.33% for false. For comparison, classical SVM scored 98.33% on test set, 93.33% on validation true set and 96.67% on validation false set with linear kernel, using four support vectors for true classe and three for the false class; radial basis function used one more support vector for each class, obtaining 98.33% accuracy on test set, 100.0% on validation true set and 96.67% on validation false set; polynomial kernel with degree 1 replicated linear kernel results, with degree 2 or 3, it chose only two support vectors per class, with 96.67% accuracy on test set and 100.0% on validation true set but only 86.67% on validation false set; polynomial kernel with degree 4 chose two support vectors for true class and a single support vector for false class, with 95.0% accuracy on test set and 100.0% on validation true set but only 86.67% on validation false set; polynomial kernel with degree 5 chose a single support vectors for each class, with 96.67% accuracy on test set and 100.0% on validation true set but only 86.67% on validation false set.

Anyway, with this simple but very noisy data, QSVM was in trouble with to respect to classical SVM, as shown in Tab. 6.4.

During local simulator execution, the QSVM training generated this parameters:

```
'alphas': array([38.33259778, 2.61541404, 4.05983042,
                 5.34439088, 10.47660117, 16.87999588,
                 5.69595284, 28.82635007,  0.23659479,  9.18994133])
'bias': array([0.87647038])
'support_vectors': array([
       [ 1.10156387,  1.34357601], [ 0.71779101,  0.6201427 ],
       [ 1.28566177,  0.64410777], [ 0.49200567,  1.48838526],
       [ 0.96038797,  1.07865239], [ 0.51693332,  0.35894799],
       [-0.60513116, -0.27110868], [-0.46576562, -0.06375724],
       [-0.08524934, -0.2358175 ], [ 0.47684918, -0.40611562]])
'yin': array([-1., -1., -1., -1., -1.,  1.,  1.,  1.,  1.,  1.])
```

During cloud execution on a real quantum computer, the QSVM training generated this parameters:

```
'alphas': array([24.87595355, 14.46093403,  9.54782349,  2.05870143,
                 6.4200966, 4.21240539, 19.67110842,  0.02823584,
                 19.17509552, 13.87883043, 8.82264439])
'bias': array([1.57117708])
'support_vectors': array([
```

```
       [ 1.10156387,  1.34357601], [ 1.5070482 ,  1.14501997],
       [ 0.71779101,  0.6201427 ], [ 1.28566177,  0.64410777],
       [ 0.80031368,  0.69428676], [ 1.11264308,  1.31388827],
       [ 0.51693332,  0.35894799], [ 0.54524299, -0.26460588],
       [-0.46576562, -0.06375724], [-0.08524934, -0.2358175 ],
       [ 0.47684918, -0.40611562]])
'yin': array([-1., -1., -1., -1., -1., -1.,  1.,  1.,  1.,  1.,  1.])
```

```
def add_noise_2d(v,r):
  for i in range(v.shape[0]):
    a=True
    while a:
      x = random.random() * 2 * r - r
      y = random.random() * 2 * r - r
      if x*x+y*y < r*r:
        v[i,0] += x
        v[i,1] += y
        a=not a
  return v
```

Listing 6.5: Python function to add random noise to two dimensional array

```
n=10; r=math.sqrt(2)/4
train_true=np.zeros((n,2))
train_true = add_noise_2d(train_true,r)
train_false=np.ones((n,2))
train_false = add_noise_2d(train_false, r)

n2=3*n; r2=1/2
test_true=np.zeros((n2,2))
test_true = add_noise_2d(test_true,r2)
test_false=np.ones((n2,2))
test_false = add_noise_2d(test_false, r2)

val_true=np.zeros((n2,2))
val_true = add_noise_2d(val_true,r2)
val_false=np.ones((n2,2))
val_false = add_noise_2d(val_false, r2)

training_data = {'T': train_true,'F': train_false}
testing_data = {'T': test_true,'F': test_false}
```

Listing 6.6: Python code to generate datasets

```
backend = provider.get_backend('ibmq_london')
shots=8192
num_qubits = 2
feature_map = SecondOrderExpansion(feature_dimension=num_qubits,
                                   depth=2,entanglement='full')
```

```
svm = QSVM(feature_map, training_data,testing_data)
quantum_instance = QuantumInstance(backend,shots=shots,
                                   skip_qobj_validation=False)
result = svm.run(quantum_instance)
print('Accuracy: ', result['testing_accuracy'],'\n')
print(result)
pred_true = svm.predict(val_true,quantum_instance)
print('Accuracy on true',
      np.count_nonzero(pred_true==1)/pred_true.size)
pred_false = svm.predict(val_false,quantum_instance)
print('Accuracy on false',
      np.count_nonzero(pred_false==0)/pred_false.size)
```

Listing 6.7: Python code to apply QSVM to 2-dim feature space

6.2.4 QSVM for a 2-dim feature space - a three classes example

In a two dimensional feature space, the most simple three classes classification task could be conceived as to distinguish the vertices of an equilateral triangle.

```
n=10; r=1/4 #r=math.sqrt(2)/4
train_A=np.zeros((n,2)); train_A = add_noise_2d(train_A,r)
train_B=np.concatenate((np.ones((n,1)), np.zeros((n,1))), axis=1)
train_B = add_noise_2d(train_B, r)
train_C=np.concatenate((np.ones((n,1))/2, np.ones((n,1))*
                        math.sqrt(3)/2), axis=1)
train_C = add_noise_2d(train_C, r)
#similar code for test and validation sets
```

Listing 6.8: Python code to generate three classes, with points centered in the vertices of an equilateral triangle with a unitary side

```
backend = provider.get_backend('ibmq_essex')
num_qubits = 2
shots=8192
feature_map = SecondOrderExpansion(feature_dimension=num_qubits,
                                   depth=2,entanglement='full')

train_true=train_A; train_false=np.concatenate((train_B,train_C))
test_true=test_A; test_false=np.concatenate((test_B, test_C))
val_true=val_A; val_false=np.concatenate((val_B, val_C))

training_data = {'T': train_true, 'F': train_false}
testing_data = {'T': test_true, 'F': test_false}
print('A vs others')

svm = QSVM(feature_map, training_data,testing_data)
quantum_instance = QuantumInstance(backend,shots=shots,
                skip_qobj_validation=False)
print('Running....\n')
```

backend	alg/kernel	test (%)	True P (%)	True N (%)
statevector_simulator	QSVM	100.00	84.00	97.00
ibmq_qasm_simulator	QSVM	100.00	85.00	96.00
sklearn.svm.SVC	linear	100.00	100.00	100.00
sklearn.svm.SVC	rbf	100.00	100.00	100.00
sklearn.svm.SVC	sigmoid	62.50	0.00	99.00
sklearn.svm.SVC	poly 1	100.00	100.00	100.00
sklearn.svm.SVC	poly 2	100.00	100.00	90.00
sklearn.svm.SVC	poly 3	100.00	100.00	58.00
sklearn.svm.SVC	poly 4	100.00	100.00	42.00
sklearn.svm.SVC	poly 5	100.00	100.00	33.00

Table 6.5: QSVM and SVM comparison on a simple 3 dimensions dataset: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

```
result = svm.run(quantum_instance)
print('Accuracy: ', result['testing_accuracy'],'\n')
print(result)
print('Prediction')
pred_true = svm.predict(val_true, quantum_instance)
print('True positive', np.count_nonzero(pred_true==1)/
      pred_true.size)
pred_false = svm.predict(val_false ,quantum_instance)
print('True negative', np.count_nonzero(pred_false==0)/
      pred_false.size)
```

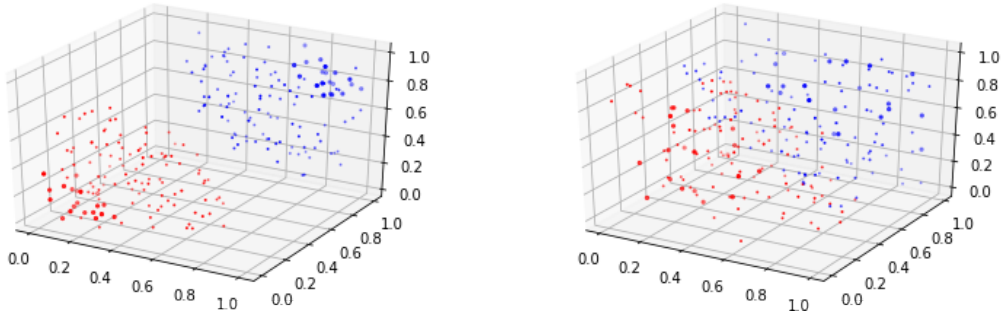
Listing 6.9: Python code to classify the first class versus the others, using QSVM

6.2.5 QSVM for a 3-dim feature space - linearly separable case

I decided to test QSVM capabilities on very simple 3d feature space clasification task. I have assigned to the opposite vertices of a cube, of coordinates (0; 0; 0) and (1; 1; 1) respectively, the values true and false. Then I generated training and test sets by applying random noise to these two values, selecting a radius equal to $\frac{1}{2}$ (20 elements for each class, both in training both in test set), while to generate the validetion set I choosed a radius $\frac{\sqrt{3}}{2}$, half of the diagonal of the cube. To generate the dataset, I applied the function in List. 6.10, obtaining data points (40 points for each class) as shown in Fig. 6.3.

I tried the QSVM both with local statevector_simulator backend both with the cloud ibmq_qasm_simulator backend. QSVM on both simulators chose five support vectors for a class and six for the other, scored with 100% accuracy on test set; on validation set, TPR was 84.00-85.00% for true elements and FPR was 96.00-97.00%. For comparison, classical SVM scored 100.0% both on test set and validation sets, with linear and RBF kernel, as shown in Tab. 6.5.

Then I increased the noise to half diagonal to generate train and test set and I choose $\frac{3}{5}$ of diagonal to generate noise for validation set. QSVM on local simulator chose ten support vectors for true class and eleven per false class, scored with 95.00% accuracy on test set; on validation set, TPR was 74.00% and FPR was 62.00%. QSVM on cloud quantum simulator chose ten support vectors for true class, but only nine support vectors for false class; anyway, it scored almost equals to noiseless statevector_simulator. For



(a) Two linearly separable datasets

(b) Datapoints with higher noise

Figure 6.3: Simple 3D dataset: on the left, traning and validation dataset are linearly separable, on the right training and test set is linerly separable, but validation set not (bigger points denote test set elements, smaller one validation set elements).

backend	alg/kernel	test (%)	True P (%)	True N (%)
statevector_simulator	QSVM	95.00	74.00	62.00
ibmq_qasm_simulator	QSVM	95.00	75.00	62.00
sklearn.svm.SVC	linear	100.00	85.00	74.00
sklearn.svm.SVC	rbf	100.00	79.00	71.00
sklearn.svm.SVC	sigmoid	0.00	17.00	33.00
sklearn.svm.SVC	poly 1	100.00	82.00	71.00
sklearn.svm.SVC	poly 2	100.00	88.00	67.00
sklearn.svm.SVC	poly 3	100.00	91.00	65.00
sklearn.svm.SVC	poly 4	100.00	94.00	64.00
sklearn.svm.SVC	poly 5	100.00	96.00	62.00

Table 6.6: QSVM and SVM comparison on a simple 3 dimensions dataset, with applied bigger noise: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. “poly n deg” denotes a polinomial kernel of degree n, “rbf” denotes a radial basis function kernel.

comparison, classical SVM scored 100.0% both on test set and validation sets, with several kernel choices (except sigmoid kernel, not reported here for convenience); it used three support vectors per class with linear kernel and polynomial kernel of degree 1, four support vector per class with radial basis functions kernel and only two support vectors per class with polynomial kernel of degree 2 or 3, as shown in Tab. 6.6.

As a third and final case, I increased noise to $\frac{5\sqrt{3}}{8}$ to generate training and test set (always 20 elements for each class) and to $\frac{3\sqrt{3}}{4}$ to generate training and test set (always 40 elements for each class), as shown in Fig. 6.4. QSVM on local simulator chose sixteen support vectors for true class and thirteen per false class, while cloud simulator chose one support vector less for true class and one more for false class; both simulators scored similar, with about half success rate on test set (i.e. no more accurate than tossing a perfect coin) and about the same on validation set. QSVM on cloud quantum simulator chose ten support vectors for true class, but only nine support vectors for false class; anyway, it scored almost equals to noiseless statevector_simulator. Curiously, with both simulators, QSVM has juggled relatively better in FPR, performing even better than most classic kernels. Except sigmoid kernel, classical SVM scored a little better on test set, with TPR similar to QSVM but with lower FPR, as shown in Tab. 6.7.

backend	alg/kernel	test (%)	True P (%)	True N (%)
statevector_simulator	QSVM	47.50	50.00	57.00
ibmq_qasm_simulator	QSVM	47.50	50.00	59.00
sklearn.svm.SVC	linear	62.50	50.00	32.00
sklearn.svm.SVC	rbf	62.50	45.00	31.00
sklearn.svm.SVC	sigmoid	27.50	55.00	72.00
sklearn.svm.SVC	poly 1	67.50	39.00	31.00
sklearn.svm.SVC	poly 2	65.00	45.00	29.00
sklearn.svm.SVC	poly 3	55.00	45.00	36.00
sklearn.svm.SVC	poly 4	52.50	40.00	33.00
sklearn.svm.SVC	poly 5	52.50	41.00	35.00

Table 6.7: QSVM and SVM comparison on a simple 3 dimensions dataset, with applied bigger noise: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. “poly n deg” denotes a polinomial kernel of degree n, “rbf” denotes a radial basis function kernel.

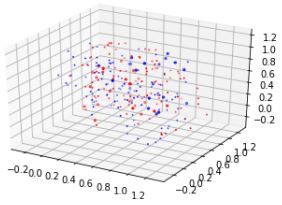


Figure 6.4: Simple 3D dataset with more noise; bigger points denote test set elements, smaller one validation set elements.

```
def add_noise_3d(v, r, s=1):
  for i in range(v.shape[0]):
    a=True
    while a:
      x = random.random() * r
      y = random.random() * r
      z = random.random() * r
      if x*x+y*y+z*z < r*r*r:
        v[i,0] += np.sign(s)*x
        v[i,1] += np.sign(s)*y
        v[i,2] += np.sign(s)*z
        a=not a
  return v
```

Listing 6.10: Python function to add 3d noise, remaining inside the unitary cube

6.2.6 QSVM for a 3-dim feature space - not linearly separable case-esphere case

As a last case in simple, artifical dataset, I conceived a sphere in 3d space. I think of a first class of points near the origin and a false class of points far from it; more precisely, I imagined that the first class included the points within a radius $\frac{1}{2}$ and the points outside belonged to the second class. In a first phase, I created a margin of $\frac{1}{10}$, in the sense that I considered belonging to the first class only points inside the sphere of radius $r - margin = \frac{2}{5}$, while for the second class I considered those outside the sphere

backend	alg/kernel	test (%)	True P (%)	True N (%)
statevector_simulator	QSVM	100.00	96.00	100.00
ibmq_qasm_simulator sklearn.svm.SVC	linear	62.50	96.00	48.00
sklearn.svm.SVC	rbf	97.50	100.00	100.00
sklearn.svm.SVC	sigmoid	55.00	80.00	48.00
sklearn.svm.SVC	poly 1	60.00	96.00	48.00
sklearn.svm.SVC	poly 2	97.50	100.00	96.00
sklearn.svm.SVC	poly 3	62.50	100.00	36.00
sklearn.svm.SVC	poly 4	90.00	100.00	92.00
sklearn.svm.SVC	poly 5	65.00	100.00	36.00

Table 6.8: QSVM and SVM comparison on a simple 3 dimensions dataset, with applied bigger noise: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

of radius $r + margin = \frac{3}{5}$. I created a train test with 30 elements for each class, then as shown in Tab. 6.8. 19 14 For classical SVM, there w

6.3 QSVM vs SVM comparison on standard datasets: some preliminary notes

In this section, I show comparison between classical and quantum SVM on several standard dataset, each of them publicly available and often used as machine learning benchmarks. In most cases, the performance of the QSVM has been evaluated with the local simulator, statevector_simulator offered by Qiskit's Aer, which simulates the operation of an ideal quantum computer, free of topological limitations and not affected by any kind of error, and with the ibmq_qasm_simulator, with free access via cloud, which offers a more accurate simulation, reproducing certain types of errors (in this regard, the default parameters have been confirmed, although probably a bit optimistic compared to the real performance of current quantum computers).

Only some specific configurations have been tested with real quantum computers, because very long processing times are required, because each session of QSVM training or inference requires issuing several jobs via cloud: moreover, each of them should be created, if necessary even "transpiled" (i.e. convert high level gates to basis gates effectively supported by selected quantum processor), then validated and eventually executed, after waiting in the queue. Above all, huge waiting times and uncertainties about correct completion are really big troubles. Furthermore, IBM's fairshare policies for the limited quantum computational resources offered free of charge via the cloud entail repeated cancellations of many jobs, followed by any attempts to restart them later; moreover, in many situations, after several hours from the initial launch of the processes, due to online access troubles or some errors with the IBM platform the execution of the training or test sessions ended, with a duplex deleterious effect: in addition to the lost time, the possibility of launching further processes via the cloud in the following hours worsened. For the majority of executions on real quantum computers (but also for many jobs on online simulator) the epilogue was one of them:

- "The user reached the maximum number of jobs running concurrently [1012]"
- "FAILURE: Can not get job id, Resubmit the qobj to get job id. Terra job error: Error submitting job: 400 Client Error: Bad Request for url: `https://api.`

`quantum-computing.ibm.com/api/Network/ibm-q/Groups/open/Projects/main/Jobs`. Reached maximum number (5) of concurrent jobs, Error code: 3458."

- "Job limit reached, waiting for job ### to finish before submitting the next one"
- "IBMQJobTimeoutError: Timeout while waiting for job ###."

Moreover, Google Colab Jupyter notebooks have an idle timeout of 90 minutes, so if user does not interact with his Google Colab notebook for more than 90 minutes, its istance is automatically aborted. Furthermore, Google Colab Jupyter notebooks have a maximum lifetime of 12 hours; these limitations make it difficult to automate large work sessions and often force training sessions to start over.

It must also be noted that the pressing topological limitations inherent in all current real quantum computers entail the need to resort to transpiling frequently when more than one pair of qubits is used: at least, indeed, there is at least the need for additional SWAP gates to match the abstract design to concrete circuit's topology and every SWAP gate gets decomposed into three controlled-not gates, so it can be seen that the actual circuit depth increases and then its output is more affected by noise. All this suggests that, in addition to the considerations supporting the reduction of the dimensionality of the feature space already ascertained in classic machine learning, in all likelihood in the case of quantum machine learning it is preferable to reduce the features to two to minimize the effects of noise, reduce the time of execution and increase the hope that a training session can be conducted without crashing, at least on current IBM free access systems.

Almost all test are executed with k-fold cross validation, a common model validation technique for assessing how the results of a statistical analysis will generalize to a never seen before dataset. With k-fold cross validation, the available dataset is divided into k equals (or almost equals) groups, than for k times a group is chosen as test dataset, while all the other k-1 groups are used as training dataset, as shown in Fig. 6.5. In general, the accuracy of each run are collected and then averaged, to obtain a more representative evaluation of the statistic model. In this work, I explicitly coded k-fold cross validation in Python, without resorting to sklearn's functions, such as train_test_split and cross_val_score.

This choice rewarded me by giving me considerable flexibility, allowing me to evaluate IBM Aqua's QSVM and sklearn's SVM uniformly, and giving me the opportunity to test, for each of the k training datasets configurations, each model with unseen datas, i.e. not involved in the k-fold selection. In fact, starting from each dataset identified in the literature, I first extracted random data from which to apply the k-fold cross validation, then I randomly extracted other data to be used as a further validation set. I resorted to this strategy because in many cases the data was too much to achieve results in a reasonable time with the simulator via the cloud and, least of all, with real quantum computers.

Some classification problems can exhibit a large imbalance in the distribution of the target classes: for instance there could be several times more negative samples than positive samples. In such circumstances, a simple sampling strategy could offer not enough sample of each class for the training model. Support Vector Machine, in particular, is a linear model whose performance can be severely impacted by the adoption of a highly unbalanced training set. In such cases, it is usually recommended to use appropriate strategies, such as stratified sampling, which sklearn offers in StratifiedKFold and StratifiedShuffleSplit. In general, a stratified k-fold cross-validation generates the folds as to preserve the percentage of samples for each class; in the case of binary classification, this technique assures that each partition contains roughly the same proportions of the two types of class labels.

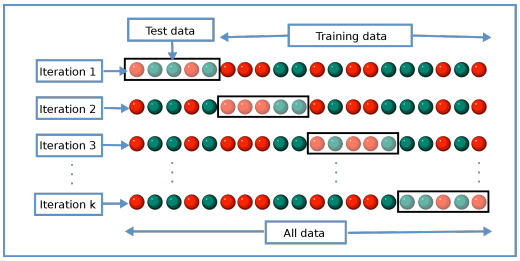


Figure 6.5: K-fold cross validation is both simple, computationally feasible and widely used in statistics and machine learning.

Moreover, I decided to write my own stratified k-fold cross-validation implementation. For binary classification, I have implemented the following strategy: from the available dataset, I choose a reasonable value for k and then randomly extract a certain number of representative elements of each class that is a multiple of k. So I divide each of the two sets into k equal parts, take a part from the first set, a part from the second set and make up the test set. I take the remaining $2k-2$ parts to make up the training set. Therefore, it's as if I separately had applied k-fold cross validation to each class.

Anyway, I could not use k-fold cross validation in the case of execution on real quantum processors, because the execution times were too long and many times the executions stopped due to errors. So, in some cases I have run QSVM algorithm only on a particular partition in training and test set; for this reason, being unable to mediate on k-folds, obviously a sample standard deviation is not indicated in the tables.

Furthermore, in some circumstances, after training the QSVM on a real quantum computer, problems occurred with access to cloud resources which prevented the QSVM from being applied to the validation set directly on the quantum hardware; in these circumstances, in the inferential phase the QSVM was run on a simulator, with the parameters learned on a real quantum computer, as indicated in List. 6.12. These cases are reported in the tables with computer type names_name/sim_name, for example ibmq_london/sv_sim denotes that the system has been trained on the ibmq_london computer, but operated in inferential mode on statevector_simulator.

```
tot_sz = 30; # train + test
k=5; s=int(tot_sz/k) # k-fold
# c1, c2 are np.array with all instance of each class
ind_true=random.sample(range(c1.shape[0]), tot_sz)
ind_false=random.sample(range(c2.shape[0]), tot_sz)
i_val_true=np.delete(np.array((range(c1.shape[0]))),ind_true)
i_val_false=np.delete(np.array((range(c2.shape[0]))),ind_false)

val_sz=60 # a new validation set
i_val_true = i_val_true[random.sample(range(
                i_val_true.shape[0]), val_sz)]
i_val_false = i_val_false[random.sample(range(
                 i_val_false.shape[0]), val_sz)]

for kernel in ['linear', 'rbf', 'sigmoid']:
  clf = svm.SVC(kernel=kernel)
  tp=np.zeros(k) # true positive on validation set
  tn=np.zeros(k) # true negative on validation set
```

```
acc=np.zeros(k) # test accuracy
val_tp=np.zeros(k)
val_tn=np.zeros(k)
for i in range(k):
  test_true=c1[ind_true[i*s:(i+1)*s]]
  train_true=np.concatenate((c1[ind_true[(i+1)*s:]],
             c1[ind_true[:(i*s)]]))
  test_false=c2[ind_false[i*s:(i+1)*s]]
  train_false=np.concatenate(( c2[ind_false[(i+1)*s:]],
              c2[ind_false[:(i*s)]]))
  Xtrain=np.concatenate((train_true, train_false))
  xclass=np.concatenate((np.zeros((train_true.shape[0],1)),
         np.ones((train_false.shape[0],1)))).ravel()
  clf.fit(Xtrain, xclass)

  acc[i]=(np.count_nonzero(clf.predict(test_true)==0)+
         np.count_nonzero(clf.predict(test_false)==1))/
         (test_true.shape[0]+test_false.shape[0])
  val_tp[i]=np.count_nonzero(clf.predict(c1[i_val_true])==0)/
            i_val_true.shape[0]
  val_tn[i]=np.count_nonzero(clf.predict(c2[i_val_false])==1)/
            i_val_false.shape[0]
accm=np.mean(acc)*100; accs=np.std(acc)*100
tpm=np.mean(val_tp)*100; tnm=np.mean(val_tn)*100
tps=np.std(val_tp)*100; tns=np.std(val_tn)*100
print('sklearn.svm.SVC & ', kernel) # and so on
```

Listing 6.11: Python code to apply k-fold cross validation

```
backend = provider.get_backend('ibmq_qasm_simulator')
qsvm.set_backend( backend  )
new_quantum_instance = QuantumInstance(backend, shots=shots)
# Predict using test data, inference on simulator
pred_true = qsvm.predict(test_true, new_quantum_instance)
# and so on...
```

Listing 6.12: Python code to compute the inferential phase on local or cloud simulator, after the training phase on a real cloud quantum computer

6.3.1 Preprocessing

For each machine learning system, performance depends greatly on the inputs considered in the training phase. It is often useful to extract relevant characteristics from the raw data available, reduce the number of dimensions to counteract the curse of dimensionality or perform some normalization procedure. One of the most simple techniques is standardization of datasets; it is a common requirement for many machine learning estimators, because they are often developed under the assumption that the distributions are Gaussian, so they might indeed behave badly if the individual features do not more or less look like standard normally distributed data. Moreover, to avoid that one feature does not affect the outcome of the learning process more than others, usually are preferable Gaussian with zero mean and unit variance. Anyway, one of the most popular and

simple techniques for reducing dimensionality is Principal component analysis (PCA). PCA is used to decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance; it can be used to apply linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space. In scikit-learn, PCA is implemented as a transformer object that learns n components in its fit method, and can be used on new data to project it on these components.

To correctly apply PCA algorithm, the input data have to be centered, i.e. with zero means for every features, otherwise it is clear, for example, that as the first main direction the algorithm could select the one that connects the experimental points more far away from with the origin, rather than the direction along which the data separate most. In addition, imposing a unit variance along each component prior to application of the PCA is strongly recommended for many cases, such as the K-Means clustering algorithm and Support Vector Machines with the RBF kernel. For instance, many elements used in the objective function of a learning algorithm assume that all features are centered around zero and have variance in the same order. If a feature has a variance that is orders of magnitude larger than others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

So, regardless of the shape of the expected distribution, I choose to just transform the data to center it by removing the mean value of each feature, then scale it by dividing non-constant features by their standard deviation, using StandardScaler object in sklearn.preprocessing. This object standardize features by removing the mean and scaling to unit variance; the standard score of a sample x is calculated as:

$$z = \frac{x - m}{s}$$

where m is the mean of the training samples (or zero if the flag with_mean is set to False), and s is the standard deviation of the training samples (or one if the flag with_std is set to False). Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using transform method.

However, even taking care of these measures, the QSVM often showed erratic learning: although the Qiskit Aqua library offers relatively high-level support, it is of paramount importance that the data provided in absolute value do not exceed the unit, because quantum circuits are composed of gates represented as unitary operators. To this end, after applying the PCA on standardized features, I resorted to MinMaxScaler object in sklearn.preprocessing library. This object is useful to transform features by scaling each feature individually to a given range, for example $[-1, 1]$

The transformation is given by:

$$X\_new = (X - X_{min})/(X_{max} - X_{min}) * (max - min) + min$$

where min, max represent range, X is the column vector of a specific feature, X_new is the new vector after transformation, while X_{min}, X_{max} represent minimum and amximum values for the feature X. Anyway, this transformation is often used as an alternative to zero mean, unit variance scaling.

It was assessed to further narrow the range of allowable values by 10%, so that each feature could span $[-0.9; 0.9]$. However, this kind of data transformation risks reducing the effectiveness of the preprocessing phase performed by the PCA; in fact, the projection on the principal directions ensures that the variance decreases when the direction index increases. By applying the MinMaxScaler instead, this difference between the selected main directions is canceled, a problem that could get worse with the increase in the

size of the feature space generated by the PCA, or by the number of selected principal components. So, for some datasets, I also tested the following strategy:

- fitting sklearn.preprocessing's StandardScaler object to raw data to transform them such as to guarantee zero means and unitary variance for each original feature
- fitting sklearn.decomposition's PCA object to data (generated from previous computation stage) and to transform them in a reduced dimensionality space item find the greatest absolute value of distance from the center, for each element of the dataset and for each direction considered by the PCA and then scaling all the values of the dataset, to make them fall within the desired range, for example $[-1, 1]$, but preserving the relationship between the variances.

As shown in List. 6.13,

```
#load datasets
rawdata = datasets.load_digits()

data=rawdata.data
y=rawdata.target
features=data.shape[1]

scaler = StandardScaler()
scaler.fit(data)
scaler.set_params(with_mean=True, with_std=True)
data=scaler.transform(data)

nfeat=4 # select principal components
data=PCA(n_components=nfeat).fit_transform(data)

mm=np.max((np.abs(np.min(data)), np.max(data)))
rad=0.9

for i in range(data.shape[0]):
  for j in range(nfeat):
    data[i, j] = (data[i, j]+mm)*rad/mm-rad
```

Listing 6.13: Python code to preprocess datasets

6.3.2 Evaluating results

In addition to all the considerations already explained regarding the comparison on simple artificial datasets of the Quantum Support Vector Machine, which adopts a kernel that exploits the quantum effect of the entanglement, with a classic version of the Support Vector Machine with various widely used kernels, in the case of more complex datasets widely used in the literature for the comparison of various classifiers, it was considered appropriate to carry out a more in-depth analysis. To this end, the receiver operating characteristic (ROC) analysis has been taken into consideration. This name is not very self-explanatory because the technique originates during World War II for the analysis of radar signals: it was conceived as a technique to increase the prediction of correctly detected aircraft from radar signals. So, for these purposes, ROC analysis measured the ability of a radar receiver operator to make these important distinctions, which was

just called the Receiver Operating Characteristic. Anyway, ROC analysis is based on evaluating ROC curve, a graphic representation that illustrates the diagnostic ability of a binary classifier system as a function of a threshold parameter, and Area Under Curve (AUC). If the two class subject to classification are considered as positive and negative elements, AUC express the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one (assuming 'positive' ranks higher than 'negative'). ROC Curve expresses the accuracy of a binary classifier and it is a curve parameterized according to a threshold parameter and drawn in an orthogonal reference system, where the False Positive Rate (FPR) is indicated on the abscissas and TPR is on the ordinates. Recalling the expressions for TPR and TNR:

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN}$$

$$TNR = \frac{TN}{N} = \frac{TN}{TN + FP}$$

equivalently FPR can be defined as:

$$FPR = \frac{FP}{N} = \frac{FP}{TN + FP} = 1 - TNR$$

The threshold parameter indicates where the boundary between the two classes can be put; usually, for a fixed classification system (for example, a trained SVM), if we want a zero FPR, we are forced to accept a low or even zero TPR value, while modifying threshold to increase TPR can indesirably increase FPR, too. So the steepness of ROC curves is very important, since it is ideal to maximize the true positive rate while minimizing the false positive rate. In the FPR/TPR plane, a perfect classifier would be represented by a vertical segment, from (0; 0) to (0; 1) and then an horizontal segment to (1; 1): the classifier always correctly predicts and the AUC takes the maximum value, that is, one. Conversely, the worst classifier never recognizes even a representative of the true class, regardless of the threshold parameter, therefore the AUC is zero. In a full random choice, such as a perfect coin toss, AUC equals 0.5, so a much higher value can be expected from a valid classification system. Although I am aware that it is subject to limitations, nevertheless AUC is one of the most important evaluation metrics for checking any classification model's performance, but when I choose to use ROC curve and AUC to compare QSVM and SVM, I ran into a small difficulty.

The sklearn library offers complete support for this kind of analysis, even providing the very convenient sklearn.metrics.plot_roc_curve function. This function receives as input an estimator instance, i.e. a trained classifier object, a set of input values and the correct corresponding labels; it outputs a ROC curve and it computes AUC. This function interacts perfectly with the implementation of SVM within the sklearn library, regardless of the selected kernel, but unfortunately it does not accept an instance of the QSVM class of the IBM Quantum Experience. After unsuccessfully attempting to build an effective wrapper for an instance of QSVM, I thought of getting around the problem by using the sklearn.metrics.roc_curve function. This function develops a ROC analysis generating three parallel arrays, in which, for the same index, the coordinates are found in the FPR/TPR plane and the corresponding threshold parameter; to this end, it receives an array containing true binary labels, a second array containing target scores and possibly a pos_label parameter that expresses the label of the positive class.

But a further complication arose, because the target scores can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by a decision function), but not predicted class labels! QSVM, indeed, offers a predict method; after the training phase, predict try to guess right class labels.

Therefore, I went to study the parameters learned by the QSVM, I verified that the expected relationship between the alpha coefficients and the y_i was respected fairly accurately and I manually obtained the weight vector of the model, so that I could apply explicitly the decision function to the input data, so as to obtain target scores suitable for the sklearn.metrics.roc_curve function. So I checked that the linear combination of α_i is (almost) zero (indeed, the sum is about $10^{-4} - 10^{-5}$:

$$\sum_{i=1}^{l} \alpha_i y_i = 0$$

The weights depend on a linear combination of chosen support vecors:

$$\vec{w} = \sum_{i=1}^{l} \alpha_i y_i \vec{x}_i$$

I remember that the decision function for an SVM classifier is expressed by

$$\vec{w}^t \vec{x}_i + bias$$

In List. 6.14, I reported the developed code to apply QSVM decision function to choosen data; by way of verification, I also ran similar code for the standard sklearn classifiers, to make sure that the outcomes matched those generated by sklearn.metrics.plot_roc_curve function.

```
# training QSVM on training_data
qsvm = QSVM( feature_map, training_data, testing_data)
result = qsvm.run(quantum_instance)

# extracting trained QSVM parameters
qsvm_res=result['svm']
bias=qsvm_res['bias']
alphas=qsvm_res['alphas']
s_v = qsvm_res['support_vectors']
yi = qsvm_res['yin']

# checking condition on alpha
test_acc=0
for l in range(yi.shape[0]):
  test_acc += alphas[l]*yi[l]
print('check zero ', test_acc)

# computing QSVM weights
qsvm_w=np.zeros((s_v.shape[1]))
for l in range(yi.shape[0]):
  qsvm_w = np.add(qsvm_w, alphas[l]*yi[l]*s_v[l,:])

# applying decision function to data
pred_val = np.zeros(datap.shape[0])
for l in range(datap.shape[0]):
  pred_val[l] = bias
  for feature in range(qsvm_w.shape[0]):
    pred_val[l] += qsvm_w[feature] * data[l, feature]
```

backend	alg/kernel	test (%)	True P (%)	True N (%)
statevector_simulator	QSVM	61.67	90.00	60.00
ibmq_qasm_simulator	QSVM	63.33	74.00	42.00
ibmq_burlington	QSVM	63.33	76.00	48.00
sklearn.svm.SVC	rbf	83.33	98.00	72.00
sklearn.svm.SVC	linear	61.67	52.00	66.00
sklearn.svm.SVC	sigmoid	51.67	24.00	76.00
sklearn.svm.SVC	poly 2 deg	70.00	72.00	82.00
sklearn.svm.SVC	poly 3 deg	48.33	56.00	44.00
sklearn.svm.SVC	poly 4 deg	68.33	60.00	88.00
sklearn.svm.SVC	poly 5 deg	66.66	42.00	88.00

Table 6.9: QSVM and SVM comparison on banana dataset with just raw data: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

```
fpr, tpr, thres = metrics.roc_curve(
          xvalclass, pred_val, pos_label=0)
roc_auc = metrics.auc(fpr, tpr)
```

Listing 6.14: Python code to apply ROC analysis and compute AUC for QSVM

6.4 QSVM vs SVM comparison on standard datasets

6.4.1 Banana dataset

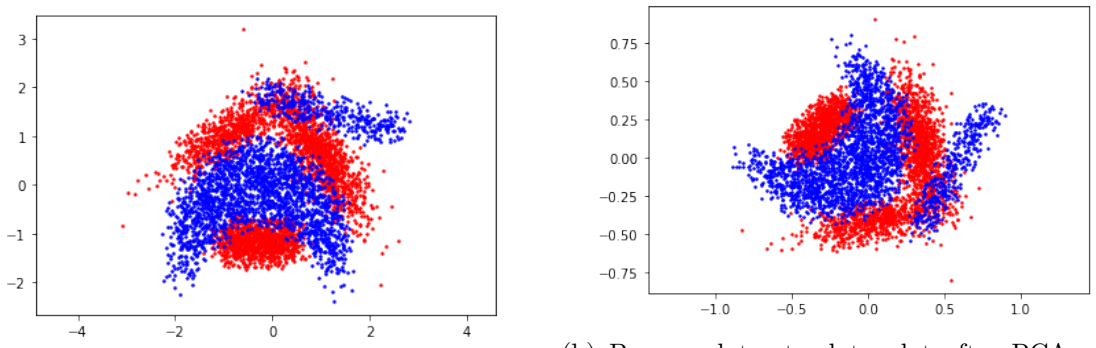
I used Banana dataset from Keel dataset repository[7]. It is declared as an artificial dataset (although a real world origin is indicated in its header file), where instances belongs to several clusters with a banana shape. There are only two attributes At1 and At2 corresponding to the x and y axis, respectively, where At1 contains real numbers in $[-3.09; 2.81]$ and At2 real numbers in $[-2.39; 3.19]$ The class label (-1 and 1) represents one of the two banana shapes in the dataset. So this dataset contains 5300 instances, 2924 in the first class and 2376 in the second.

In a preliminary phase, I proceeded to randomly extract two subsets of the same size, one with elements of the first class and the other with elements of the second class. I selected 20% of each set for training, 30% for the test and the remaining 50% for validation.

In Tab. 6.9, QSVM is compared with SVM with various kernel selections: their accuracy seems similar, only classical SVM with radial basis function scores best.

Then I applied the general procedure, as indicated in List. 6.13: anyway, it must be specified that the dataset already offers an almost flawless standardization of the data, as the means along the two features differ from zero for 10^{-5} and the standard deviations are away from the unit only for 10^{-4}, so the only effect of my preprocessing is a rotation due to PCA and a scaling, as shown in Fig. 6.6. So I applied k-fold cross validation, as in List. 6.11

[7] `http://sci2s.ugr.es/keel/dataset/data/classification/banana.zip`



(a) Banana dataset: raw data plot.

(b) Banana dataset: data plot after PCA and scaling.

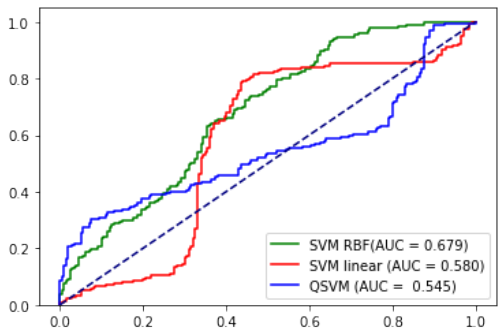
Figure 6.6: The banana dataset's graphic representation.

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
ibmq_rome/sv_sim	QSVM	50.00	55.00	80.00
statevector_simulator	QSVM	78.33 ± 8.50	72.00 ± 3.86	67.67 ± 4.03
ibmq_qasm_simulator	QSVM	63.33 ± 8.50	65.33 ± 4.14	56.00 ± 7.64
sklearn.svm.SVC	poly 1	40.00 ± 12.25	56.67 ± 6.41	42.00 ± 9.97
sklearn.svm.SVC	poly 2	68.33 ± 6.24	54.33 ± 9.35	64.67 ± 8.19
sklearn.svm.SVC	poly 3	55.00 ± 15.46	78.00 ± 5.21	63.67 ± 11.71
sklearn.svm.SVC	poly 4	68.33 ± 6.24	47.33 ± 2.71	72.33 ± 3.43
sklearn.svm.SVC	poly 5	55.00 ± 18.71	72.67 ± 9.23	65.33 ± 22.54
sklearn.svm.SVC	linear	46.67 ± 15.46	57.00 ± 6.62	39.67 ± 7.48
sklearn.svm.SVC	rbf	71.67 ± 10.00	88.67 ± 7.77	58.00 ± 3.56
sklearn.svm.SVC	sigmoid	41.67 ± 14.91	48.33 ± 5.87	34.67 ± 11.18

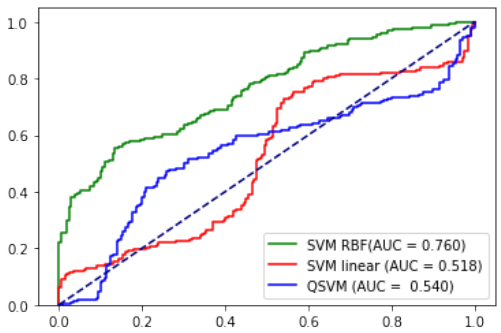
Table 6.10: QSVM and SVM comparison on banana dataset with just pre-processed data and 5-fold cross validation: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
statevector_simulator	QSVM	51.25 ± 4.18	56.20 ± 12.11	41.40 ± 17.47
sklearn.svm.SVC	poly 1	52.25 ± 4.96	53.60 ± 3.14	55.80 ± 8.08
sklearn.svm.SVC	poly 2	61.25 ± 7.25	45.60 ± 3.38	77.40 ± 1.02
sklearn.svm.SVC	poly 3	63.25 ± 8.01	83.20 ± 2.99	52.80 ± 1.60
sklearn.svm.SVC	poly 4	60.00 ± 4.11	54.80 ± 24.69	70.80 ± 23.94
sklearn.svm.SVC	poly 5	58.50 ± 8.04	88.80 ± 5.04	36.80 ± 5.81
sklearn.svm.SVC	linear	52.25 ± 3.66	53.60 ± 2.65	56.20 ± 8.23
sklearn.svm.SVC	rbf	88.50 ± 2.29	92.20 ± 1.60	83.60 ± 1.20
sklearn.svm.SVC	sigmoid	35.50 ± 6.74	25.00 ± 1.10	39.80 ± 1.60

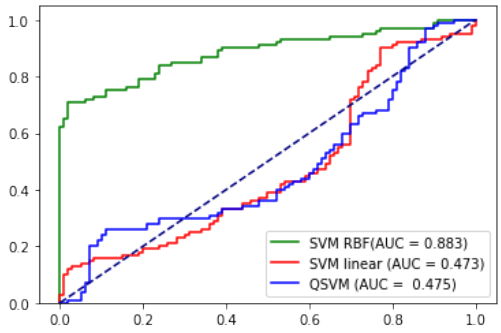
Table 6.11: QSVM and SVM comparison on banana dataset with just pre-processed data and 5-fold cross validation: increasing training set, with 200 data points for each class, do not improve learning. The last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.



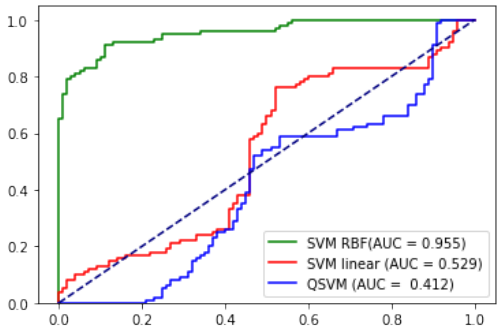
(a) Area under the ROC Curve for 10 elements for each class in training set.



(b) Area under the ROC Curve for 20 elements for each class in training set.



(c) Area under the ROC Curve for 50 elements for each class in training set.



(d) Area under the ROC Curve for 200 elements for each class in training set.

Figure 6.7: The banana dataset: Area under the ROC Curve, for different training sets..

6.4.2 Haberman dataset

I used Haberman's Survival data set from Keel dataset repository[8]. It is a real world dataset: it contains cases from a study that was conducted between 1958 and 1970 at the University of Chicago's Billings Hospital on the survival of patients who had undergone surgery for breast cancer. The patients who survived 5 years or longer are marked as positive, while if the patient died within 5 year is considered negative. There are three attributes, all expressed as integers: the first is age, with values in [30, 83], the second is the year in [58, 69], while the third is the number of positive axillary lymph nodes detected[9]. The class labels, respectively negative and positive, were transformed in -1 and 1. This dataset contains 306 instances, 225 in the first class and 81 in the second.

In a first phase, I proceeded to randomly extract two subsets of the same size, one with 70 elements of the first class and the other with 70 elements of the second class. I selected 20 elements of each set for training, other 20 for the test dataset and the remaining 30 elements for validation set, as in List. 6.15. In this scenario, with all 3 feature and none pre-processing I compared QSVM on both local simulator (without noise) and cloud simulator (with standard quantum noise), with lower score than conventional SVM with radial basis function.

Then I

```
import time, random
#data is an np.array with raw data
c1=data[(data[:,-1]==-1),:data.shape[1]-1]
```

[8] http://www.keel.es/

[9] http://www.kaggle.com/gilsousa/habermans-survival-data-set

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
statevector_simulator	QSVM	55.00	66.67	43.33
ibmq_qasm_simulator	QSVM	55.00	66.67	43.33
sklearn.svm.SVC	rbf	70.00	80.00	43.33
sklearn.svm.SVC	linear	62.50	60.00	50.00
sklearn.svm.SVC	sigmoid	50.00	70.00	20.00
sklearn.svm.SVC	poly 1 deg	70.00	80.00	40.00
sklearn.svm.SVC	poly 2 deg	65.00	96.67	23.33
sklearn.svm.SVC	poly 3 deg	62.50	63.33	53.33
sklearn.svm.SVC	poly 4 deg	55.00	56.67	56.67
sklearn.svm.SVC	poly 5 deg	62.50	70.00	63.33

Table 6.12: QSVM and SVM comparison on haberman dataset: the last three columns show the accuracy, respectively, on test dataset, then true positive on both validation datasets, the first one . "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

```
c2=data[(data[:,-1]==1), :data.shape[1]-1]

train_sz=20; test_sz=20; val_sz=30;
tot_sz = train_sz + test_sz + val_sz
ind_true=random.sample(range(c1.shape[0]), tot_sz)
ind_false=random.sample(range(c2.shape[0]), tot_sz)

train_true = c1[ind_true[0:train_sz], :]
train_false = c2[ind_false[0:train_sz], :]

test_true = c1[ind_true[train_sz:train_sz+test_sz], :]
test_false = c2[ind_false[train_sz:train_sz+test_sz], :]
val_true = c1[ind_true[train_sz+test_sz:], :]
val_false = c2[ind_false[train_sz+test_sz:], :]

training_data = {'T': train_true,'F': train_false}
testing_data = {'T': test_true,'F': test_false}
```

Listing 6.15: Python code to extract data from Haberman's dataset

6.4.3 Iris dataset

The famous Iris dataset is used as a classic example of statistical classification, so it is among the most used in the field of machine learning. It is a multivariate dataset introduced by Ronald Fisher in 1936. It consists of 150 instances of Iris measured by Edgar Anderson and classified according to three species: Iris silky, Iris virginica and Iris versicolor. There are exactly 50 members for each class. The four variables considered are the length and width of the sepal and petal. I used the version of the dataset embedded in the library sklearn. For all tests in this section, I used 5-fold cross validation.

First I normalized dataset, to make the mean zero and standard deviation unitary, then I applied a PCA analysis to project the dataset on the two main dimensions. Because there are three classes, I run SVM and QSVM for each possible binary scenario: first class versus the others, second class versus the others and eventually third class versus the other. In a first phase, I had thought of extracting the same number of samples from

backend	alg/kernel	test (%)	True P (%)	True N (%)
statevector_simulator	QSVM	65.00 ± 3.33	12.67 ± 25.33	90.33 ± 19.33
ibmq_qasm_simulator	QSVM	51.67 ± 11.06	44.67 ± 23.34	67.67 ± 14.97
sklearn.svm.SVC	poly 1	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	poly 2	90.00 ± 6.24	93.33 ± 0.00	82.33 ± 1.33
sklearn.svm.SVC	poly 3	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	poly 4	90.00 ± 3.33	86.67 ± 2.11	86.33 ± 1.94
sklearn.svm.SVC	poly 5	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	linear	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	rbf	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	sigmoid	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00

Table 6.13: QSVM and SVM comparison on on Iris dataset to discern the first class from the others with 5-fold cross validation: the last three columns show the accuracy, respectively, on test dataset, then true positive (for first class) and true negative on validation dataset. 20 elements were choosen for each class, so there was not a real balance for this task: 20 samples of class 1 and 40 samples of "not class 1". Indeed, QSVM presented an erratic behaviour because there was 16 training samples for first class and overall 32 training samples for other classes (ibmq_qasm_simulator was used with 1024 shots.)"poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

each class; therefore, to classify an object of the first class, I selected for example 20 elements from the first class and a total of 40 from the other two, to create a dataset to which apply The classic SVM algorithms managed this situation correctly, but the QSVM, performed both on the local simulator and on the IBM online simulator, exhibited erratic behavior: it was able to recognize the elements of the other two classes very well as not belonging to the first class but had difficulty recognizing first-class members, as shown in Tab. 6.13 Analyzing scrupulously the individual training sessions, in many cases the QSVM, at the end of the training, always answered 'other 'regardless of the input value, on the whole dataset. It was thus found that the general advice to correctly balance the training data before applying SVM classifier is of paramount importance with quantum implementation.

Then I trained both QSVM and SVM to classy element of class 2, but now balancing training, test and validation set, so that each of them contained one half of class 2 samples and one half of the other classes samples. In Tab. 6.14, results are reported, where quantum support vector machine don't seem able to convincingly exceed even the threshold of 50%, scoring worse thna even SVM with linear kernel.

Afterwards, I tried the procedure described in List. 6.13, extracting the two main components with the PCA, and selecting 35 points per class for the training set, 15 per class for the validation set. In Tab. 6.15 there are the results for the first class against the others, while in Tab. 6.16 there are the results for the second class against the others.

6.4.4 Breast cancer dataset

I used breast cancer dataset from sklearn libray. It is a diagnostic dataset from Wisconsin. It contains 569 istances with 30 attributes, distributed in two classes, respectvely with (212 and 357 elements. I generated 30 sample for each class for training and another unseen 40 sample for each class for validation set. In Tab. 6.17 I showed analysis when three principal components were chosen, while in Tab. 6.18 four components were selected. In the case of this dataset, performance tends to deteriorate by increasing the

backend	alg/kernel	test (%)	True P (%)	True N (%)
statevector_simulator	QSVM	50.00 ± 11.18	48.00 ± 20.07	44.33 ± 25.49
ibmq_qasm_simulator	QSVM	60.00 ± 12.25	55.33 ± 10.24	50.00 ± 10.95
sklearn.svm.SVC	poly 1	72.50 ± 14.58	67.33 ± 6.80	67.67 ± 2.91
sklearn.svm.SVC	poly 2	87.50 ± 7.91	95.33 ± 2.67	89.33 ± 1.33
sklearn.svm.SVC	poly 3	67.50 ± 12.75	92.67 ± 4.42	49.00 ± 2.49
sklearn.svm.SVC	poly 4	82.50 ± 6.12	98.67 ± 1.63	85.67 ± 3.59
sklearn.svm.SVC	poly 5	75.00 ± 11.18	100.00 ± 0.00	54.00 ± 3.74
sklearn.svm.SVC	linear	75.00 ± 17.68	65.33 ± 6.53	67.00 ± 3.86
sklearn.svm.SVC	rbf	82.50 ± 12.75	84.00 ± 6.46	89.67 ± 2.87
sklearn.svm.SVC	sigmoid	75.00 ± 15.81	60.00 ± 4.71	74.33 ± 6.72

Table 6.14: QSVM and SVM comparison on on Iris dataset to discern the second class from the others with 5-fold cross validation: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. “poly n deg” denotes a polinomial kernel of degree n, “rbf” denotes a radial basis function kernel. QSVM worked better on cloud simulator (ibmq_qasm_simulator was used with 1024 shots.)

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
ibmq_burlington	QSVM	57.14	33.33	93.33
statevector_simulator	QSVM	90.00 ± 7.28	100.00 ± 0.00	82.67 ± 5.33
ibmq_qasm_simulator	QSVM	85.71 ± 13.55	89.33 ± 10.83	78.67 ± 10.67
sklearn.svm.SVC	poly 1	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	poly 2	98.57 ± 2.86	93.33 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	poly 3	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	poly 4	98.57 ± 2.86	93.33 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	poly 5	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	linear	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	rbf	100.00 ± 0.00	94.67 ± 2.67	100.00 ± 0.00
sklearn.svm.SVC	sigmoid	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00

Table 6.15: QSVM and SVM comparison on on Iris dataset to discern the second class from the others with 5-fold cross validation: the last three columns show the accuracy, respectively, on test dataset, then true positive and true negative on validation dataset. “poly n deg” denotes a polinomial kernel of degree n, “rbf” denotes a radial basis function kernel. QSVM worked better on cloud simulator (ibmq_qasm_simulator was used with 1024 shots.)

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
ibmq_burlington	QSVM	64.29	86.67	20.00
statevector_simulator	QSVM	62.86 ± 15.25	89.33 ± 8.00	57.33 ± 9.04
ibmq_qasm_simulator	QSVM	82.86 ± 18.41	84.00 ± 5.33	74.67 ± 7.77
sklearn.svm.SVC	poly 1	75.71 ± 3.50	100.00 ± 0.00	40.00 ± 0.00
sklearn.svm.SVC	poly 2	88.57 ± 9.69	100.00 ± 0.00	86.67 ± 0.00
sklearn.svm.SVC	poly 3	77.14 ± 2.86	100.00 ± 0.00	40.00 ± 0.00
sklearn.svm.SVC	poly 4	88.57 ± 9.69	98.67 ± 2.67	80.00 ± 4.22
sklearn.svm.SVC	poly 5	74.29 ± 5.71	100.00 ± 0.00	38.67 ± 2.67
sklearn.svm.SVC	linear	74.29 ± 5.71	100.00 ± 0.00	40.00 ± 0.00
sklearn.svm.SVC	rbf	90.00 ± 10.69	100.00 ± 0.00	89.33 ± 5.33
sklearn.svm.SVC	sigmoid	72.86 ± 2.86	94.67 ± 2.67	49.33 ± 9.98

Table 6.16: QSVM and SVM comparison on Iris dataset (third class): the last three columns show the accuracy, respectively, on test dataset, then true positive on both validation datasets, the first one . "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
statevector_simulator	QSVM	85.00 ± 6.24	86.00 ± 5.15	85.50 ± 4.30
sklearn.svm.SVC	poly 1	90.00 ± 6.24	95.50 ± 1.87	98.00 ± 1.00
sklearn.svm.SVC	poly 2	53.33 ± 6.67	50.50 ± 7.31	89.00 ± 7.00
sklearn.svm.SVC	poly 3	83.33 ± 5.27	83.00 ± 6.00	100.00 ± 0.00
sklearn.svm.SVC	poly 4	58.33 ± 5.27	46.50 ± 4.64	94.00 ± 2.55
sklearn.svm.SVC	poly 5	70.00 ± 10.00	65.00 ± 3.87	100.00 ± 0.00
sklearn.svm.SVC	linear	90.00 ± 9.72	90.50 ± 2.92	97.00 ± 1.00
sklearn.svm.SVC	rbf	91.67 ± 7.45	97.00 ± 1.00	95.50 ± 1.00
sklearn.svm.SVC	sigmoid	93.33 ± 8.16	96.00 ± 1.22	97.00 ± 1.87

Table 6.17: QSVM and SVM comparison on breast cancer dataset (after pre-processing and with 5-fold cross validation): class 1 was compared against the other. With PCA, the three principal components were selected. The last three columns show the accuracy, respectively, on test dataset, then true positive on both validation datasets, the first one . "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

dimensionality of the feature space. The QSVM classifier proves to be able to discriminate the two classes, but with systematically lower performance than the classic SVM.

6.4.5 Wine dataset

scikit-learn contains a copy of the UCI ML Wine recognition datasets[10]. These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines. The attributes are :

- Alcohol
- Malic acid
- Ash

[10] `https://archive.ics.uci.edu/ml/datasets/wine`

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
ibmq_vigo/sv_simu	QSVM	75.00	65.00	95.00
ibmq_vigo/ibmq_qasm_sim	QSVM	75.00	65.00	95.00
statevector_simulator	QSVM	75.00 ± 17.48	71.00 ± 9.43	97.50 ± 3.16
ibmq_qasm_simulator	QSVM	75.00 ± 15.81	72.00 ± 6.78	94.00 ± 3.00
sklearn.svm.SVC	poly 1	95.00 ± 6.67	89.00 ± 1.22	97.00 ± 2.45
sklearn.svm.SVC	poly 2	76.67 ± 14.34	58.50 ± 4.06	95.00 ± 2.24
sklearn.svm.SVC	poly 3	90.00 ± 9.72	78.00 ± 1.87	97.50 ± 0.00
sklearn.svm.SVC	poly 4	71.67 ± 10.00	46.00 ± 4.64	100.00 ± 0.00
sklearn.svm.SVC	poly 5	81.67 ± 9.72	58.00 ± 4.00	98.00 ± 1.00
sklearn.svm.SVC	linear	96.67 ± 4.08	86.50 ± 2.00	97.50 ± 0.00
sklearn.svm.SVC	rbf	93.33 ± 6.24	88.00 ± 2.92	97.50 ± 1.58
sklearn.svm.SVC	sigmoid	93.33 ± 6.24	89.50 ± 1.00	94.00 ± 2.00

Table 6.18: QSVM and SVM comparison on breast cancer dataset (after pre-processing and with 5-fold cross validation): class 1 was compared against the other. With PCA, the four principal components were selected. The last three columns show the accuracy, respectively, on test dataset, then true positive on both validation datasets, the first one . "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

- Alcalinity of ash
- Magnesium
- Total phenols
- Flavanoids
- Nonflavanoid phenols
- Proanthocyanins
- Color intensity
- Hue
- OD280/OD315 of diluted wines
- Proline

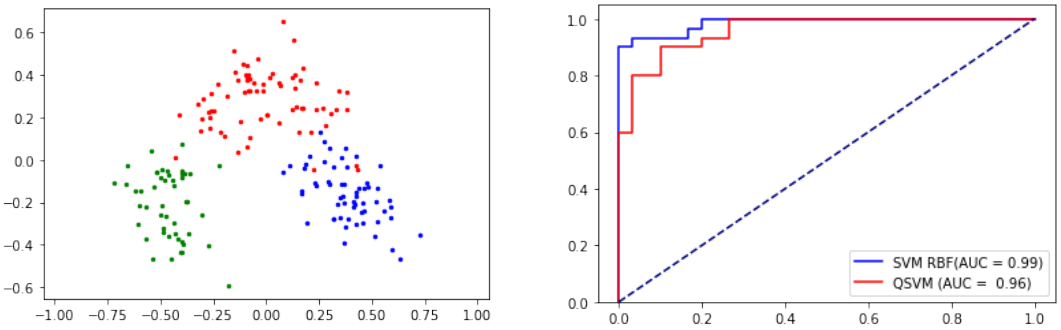
Wine dataset is usually considered a good data set for first testing of a new classifier, but not very challenging. In Tab. 6.19 and Tab. 6.20 I show the comparison between classifier when choosing, respectively class 1 versus the other two and then class 3 versus the other ones, selecting 3 principal components via PCA and using preprocessing as illustrated in List. 6.13. In Fig. 6.8b there is an example of ROC analysis and AUC computation for QSVM (on statevector_simulator) and SVM with radial basis function. In Tab. 6.21 e Tab. 6.22, the results are illustrated when MinMaxScaler is applied to the main components selected by the PCA to, losing the relative order of the features, in the sense that the difference in variance is faded away: in the first case, MinMaxScaler was applied to set all features in [-0.9; 0.9], in the second case MinMaxScaler was applied to set all features in [-1.0; 1.0]. At least when QSVM is executed on the simulator, the improvement in classifier performance seems evident when the feature values are reduced to values lower than the unit, while preserving the relative variances in the different main components.

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
statevector_simulator	QSVM	86.67 ± 13.54	98.67 ± 2.67	76.00 ± 9.04
sklearn.svm.SVC	poly 1	95.00 ± 6.67	97.33 ± 3.27	93.33 ± 0.00
sklearn.svm.SVC	poly 2	91.67 ± 5.27	97.33 ± 3.27	60.00 ± 4.22
sklearn.svm.SVC	poly 3	85.00 ± 6.24	81.33 ± 12.93	93.33 ± 4.22
sklearn.svm.SVC	poly 4	85.00 ± 9.72	96.00 ± 8.00	69.33 ± 5.33
sklearn.svm.SVC	poly 5	83.33 ± 7.45	66.67 ± 0.00	94.67 ± 4.99
sklearn.svm.SVC	linear	95.00 ± 4.08	98.67 ± 2.67	92.00 ± 2.67
sklearn.svm.SVC	rbf	98.33 ± 3.33	97.33 ± 3.27	100.00 ± 0.00
sklearn.svm.SVC	sigmoid	95.00 ± 6.67	93.33 ± 0.00	93.33 ± 0.00

Table 6.19: QSVM and SVM comparison on digits dataset (after pre-processing and with 5-fold cross validation): class 1 was compared against the others. With PCA, the three principal components were selected. The last three columns show the accuracy, respectively, on test dataset, then true positive on both validation datasets, the first one . "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
statevector_simulator	QSVM	81.67 ± 6.24	77.78 ± 3.51	63.33 ± 4.44
sklearn.svm.SVC	poly 1	98.33 ± 3.33	95.56 ± 2.22	100.00 ± 0.00
sklearn.svm.SVC	poly 2	90.00 ± 8.16	68.89 ± 4.44	92.22 ± 2.72
sklearn.svm.SVC	poly 3	95.00 ± 4.08	83.33 ± 3.51	98.89 ± 2.22
sklearn.svm.SVC	poly 4	86.67 ± 8.50	62.22 ± 4.16	94.44 ± 0.00
sklearn.svm.SVC	poly 5	93.33 ± 6.24	61.11 ± 0.00	100.00 ± 0.00
sklearn.svm.SVC	linear	96.67 ± 4.08	98.89 ± 2.22	98.89 ± 2.22
sklearn.svm.SVC	rbf	98.33 ± 3.33	95.56 ± 2.22	100.00 ± 0.00
sklearn.svm.SVC	sigmoid	96.67 ± 4.08	100.00 ± 0.00	96.67 ± 2.72

Table 6.20: QSVM and SVM comparison on digits dataset (after pre-processing and with 5-fold cross validation): class 3 was compared against the others. With PCA, the three principal components were selected. The last three columns show the accuracy, respectively, on test dataset, then true positive on both validation datasets, the first one . "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.



(a) Wine dataset: projection on first 2 principal components.

(b) Wine dataset: ROC and AUC comparison between QSVM and RBF.

Figure 6.8: Wine dataset's PCA representation and ROC analysis.

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
statevector_simulator	QSVM	86.67 ± 13.54	98.67 ± 2.67	76.00 ± 9.04
sklearn.svm.SVC	poly 1	95.00 ± 6.67	97.33 ± 3.27	93.33 ± 0.00
sklearn.svm.SVC	poly 2	91.67 ± 5.27	97.33 ± 3.27	60.00 ± 4.22
sklearn.svm.SVC	poly 3	85.00 ± 6.24	81.33 ± 12.93	93.33 ± 4.22
sklearn.svm.SVC	poly 4	85.00 ± 9.72	96.00 ± 8.00	69.33 ± 5.33
sklearn.svm.SVC	poly 5	83.33 ± 7.45	66.67 ± 0.00	94.67 ± 4.99
sklearn.svm.SVC	linear	95.00 ± 4.08	98.67 ± 2.67	92.00 ± 2.67
sklearn.svm.SVC	rbf	98.33 ± 3.33	97.33 ± 3.27	100.00 ± 0.00
sklearn.svm.SVC	sigmoid	95.00 ± 6.67	93.33 ± 0.00	93.33 ± 0.00

Table 6.21: QSVM and SVM comparison on digits dataset (with 5-fold cross validation): class 1 was compared against the others. With PCA, the three principal components were selected, then MinMaxScaler was applied to set all features in [-0.9; 0.9]. The last three columns show the accuracy, respectively, on test dataset, then true positive on both validation datasets, the first one . "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
ibmqx2	QSVM	78.57	80.00	70.00
statevector_simulator	QSVM	84.29 ± 2.86	98.00 ± 2.45	56.00 ± 7.35
ibmq_qasm_simulator	QSVM	72.86 ± 8.33	86.00 ± 5.83	65.00 ± 8.94
sklearn.svm.SVC	poly 1	90.00 ± 5.71	99.00 ± 2.00	93.00 ± 2.45
sklearn.svm.SVC	poly 2	84.29 ± 7.00	100.00 ± 0.00	61.00 ± 3.74
sklearn.svm.SVC	poly 3	81.43 ± 10.69	81.00 ± 4.90	98.00 ± 2.45
sklearn.svm.SVC	poly 4	77.14 ± 9.48	89.00 ± 9.70	63.00 ± 13.27
sklearn.svm.SVC	poly 5	80.00 ± 10.50	71.00 ± 4.90	100.00 ± 0.00
sklearn.svm.SVC	linear	90.00 ± 5.71	99.00 ± 2.00	93.00 ± 2.45
sklearn.svm.SVC	rbf	98.57 ± 2.86	100.00 ± 0.00	95.00 ± 0.00
sklearn.svm.SVC	sigmoid	91.43 ± 5.35	97.00 ± 2.45	93.00 ± 2.45

Table 6.22: QSVM and SVM comparison on digits dataset (with 5-fold cross validation): class 1 was compared against the others. With PCA, the three principal components were selected, then MinMaxScaler was applied to set all features in [-1.0; 1.0]. The last three columns show the accuracy, respectively, on test dataset, then true positive on both validation datasets, the first one . "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

6.4.6 Optical recognition of handwritten digits dataset

scikit-learn contains a copy of the optical recognition of handwritten digits dataset from the UCI ML datasets[11]. One of the simplest graphical representations adopted by computers for two-dimensional images is bitmap encoding. With this approach, the image is considered formed by a discrete number of points, divided into rows and columns; one or more contiguous bits in a specific portion of memory represent the color or light intensity information of each point, called pixel. In the case of perfectly monochromatic images, such as black and white images without shades of gray, a single bit is sufficient to encode the information of each pixel. Usually, if the pixel is on, the corresponding bit assumes the logical value 1, otherwise it assumes the value 0. This dataset includes a primary preprocessing made by programs developed by NIST to extract normalized bitmaps of handwritten digits from a preprinted form. The original images are 32x32 bitmaps, but they are divided into non-overlapping blocks of 4x4 and the number of on pixels are counted in each block. This generates an input matrix of 8x8 where each element is an integer in the range 0..16. In practice, a very simple case of bilinear filtering has been adopted. Bilinear filtering is indeed one of the basic resampling techniques in computer vision and image processing. This filtering is an extension of naive linear interpolation for functions of two variables on a rectangular bidimensional grid. This technique also became popular with the general public in the mid-90s, when the first graphics cards began to spread with the ability to accelerate three-dimensional rendering, such as the famous 3dfx Voodoo. One of the simplest expedients to achieve a satisfactory graphic rendering of the objects and characters consisted in covering the surfaces, approximated with strips of triangles, with textures, that is, appropriate chromatic descriptions. Obviously, depending on the relative distances and the observer's point of view, there is no one-to-one relationship between the points of a texture (texel) and the points displayed on the screen. The most basic method to overcome this problem simply chooses to draw on the texel whose coordinates are closer to those which would theoretically correspond to the pixel to be drawn on the screen; this method is called Nearest-neighbor filtering. Such simple texture mapping make the picture look pixelated, full of aliasing. Bilinear filtering, instead, prevents this by interpolating the points that are between texels.

NIST algorithm differ from conventional bilinear filtering in two ways: it acts as the sampling coordinate are always exactly in the center of each 2×2 square; it accumulates values, instead of average them. Anyway, this reduces dimensionality from 1024 to 64 and guarantees invariance with to respect to small distortions; each pixel is encoded with a first row of 4x4 pixel block counts in features from 0 to 7, then second row in features from 8 to 15 and so on, a row at time. To make this dataset, a total of 43 people participated: thirty of them contributed to the training set and different thirteen to the test set. Anyway, in the copy retrieved from scikit-learn, there are a total of 1797 entries, which is not a multiple of 43 or 10, either. Instead, were labeled about 180 entries for each decimal numeric digit, more precisely:

- 178 samples for 0
- 182 samples for 1
- 177 samples for 2
- 183 samples for 3
- 181 samples for 4
- 182 samples for 5

[11] `https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits`

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
ibmqx2/qasm_sim	QSVM	75.00	93.33	31.67
ibmqx2/sv_sim	QSVM	75.00	93.33	31.67
ibmqx2	QSVM	75.00	83.33	30.00
statevector_simulator	QSVM	85.00 ± 6.24	91.67 ± 4.08	84.67 ± 3.23
ibmq_qasm_simulator	QSVM	81.67 ± 6.24	88.33 ± 5.48	84.33 ± 4.03
sklearn.svm.SVC	poly 1	90.00 ± 8.16	99.00 ± 1.33	92.00 ± 3.40
sklearn.svm.SVC	poly 2	88.33 ± 8.50	93.00 ± 0.67	94.33 ± 1.33
sklearn.svm.SVC	poly 3	93.33 ± 6.24	89.00 ± 1.70	100.00 ± 0.00
sklearn.svm.SVC	poly 4	91.67 ± 7.45	83.67 ± 1.94	98.33 ± 0.00
sklearn.svm.SVC	poly 5	86.67 ± 4.08	84.00 ± 1.33	99.00 ± 0.82
sklearn.svm.SVC	linear	88.33 ± 11.30	100.00 ± 0.00	74.00 ± 4.55
sklearn.svm.SVC	rbf	96.67 ± 4.08	98.67 ± 1.25	97.00 ± 1.94
sklearn.svm.SVC	sigmoid	86.67 ± 11.30	100.00 ± 0.00	73.33 ± 2.79

Table 6.23: QSVM and SVM comparison on digits dataset (after pre-processing and with 5-fold cross validation): class 0 was compared against the others. With PCA, the three principal components were selected. The last three columns show the accuracy, respectively, on test dataset, then true positive on both validation datasets, the first one . "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

- 181 samples for 6
- 179 samples for 7
- 174 samples for 8
- 180 samples for 9

For this test, I report in Tab. 6.23 e 6.25 the results of classification test conducted selecting class 0 and comparing, in the training phase, 30 random istances of c0 against 30 (total) random instance of other classes. I applied 5-fold cross validation to this data, then checked TPR and TNR on validation set, with double number of elements with to respect to training set. In Tab. 6.24 the results are illustrated when MinMaxScaler is applied to the main components selected by the PCA, losing the relative order of the features, in the sense that the difference in variance is faded away. I am convinced that in higher dimension feature spaces my approach, as illustrated in List. 6.13 and which already for 3 features seems to benefit QSVM, on the local simulator at least, can guarantee more significant advantages.

6.5 Results

The preliminary conclusions, based on the executed tests, suggest that Quantum Support Vector Machine, in classification tasks, could prove, depending on the dataset, a lower accuracy than the classical Support Vector Machine. All this is not surprising at all, because it happens due to the ineluctable probabilistic nature of quantum computers and multiple noise issues in nowadays devices. Attenuating the noise in a quantum computer system is one of the biggest challenges so far, so all error-mitigation techniques present a route to accurate classification even with noisy intermediate-scale (NISQ) devices hardware. Certainly, on the basis of the scarce availability of qubits in the systems created so far, the adoption of effective error detection and correction schemes does not

-.9, .9 ibmqx2	QSVM	58.33	70.00	56.67
statevector_simulator	QSVM	91.67 ± 5.27	81.00 ± 2.26	80.67 ± 5.64
ibmq_qasm_simulator	QSVM	90.00 ± 6.24	81.67 ± 2.79	79.00 ± 6.72
sklearn.svm.SVC	poly 1	88.33 ± 8.50	93.00 ± 1.25	88.67 ± 4.14
sklearn.svm.SVC	poly 2	96.67 ± 4.08	87.67 ± 0.82	94.33 ± 1.33
sklearn.svm.SVC	poly 3	96.67 ± 4.08	88.67 ± 1.63	99.67 ± 0.67
sklearn.svm.SVC	poly 4	95.00 ± 4.08	85.00 ± 1.05	98.00 ± 1.94
sklearn.svm.SVC	poly 5	96.67 ± 4.08	84.00 ± 1.33	99.67 ± 0.67
sklearn.svm.SVC	linear	86.67 ± 6.67	96.33 ± 1.25	79.67 ± 2.87
sklearn.svm.SVC	rbf	96.67 ± 4.08	98.00 ± 0.67	93.33 ± 4.08
sklearn.svm.SVC	sigmoid	81.67 ± 6.24	100.00 ± 0.00	67.67 ± 1.70

Table 6.24: QSVM and SVM comparison on digits dataset (with 5-fold cross validation): class 0 was compared against the others. With PCA, the three principal components were selected, after using StandardScaler, then each features was normalized in [-.9, .9] with MinMaxScaler scaler.. The last three columns show the accuracy, respectively, on test dataset, then true positive on both validation datasets, the first one . "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

backend	alg/kernel	test (%)	True P c.1 (%)	True P. c2 (%)
statevector_simulator	QSVM	90.00 ± 9.72	89.67 ± 1.25	76.67 ± 6.67
ibmq_qasm_simulator	QSVM	85.00 ± 6.24	86.00 ± 3.09	74.67 ± 5.31
sklearn.svm.SVC	poly 1	93.33 ± 3.33	97.33 ± 1.33	88.33 ± 2.36
sklearn.svm.SVC	poly 2	96.67 ± 4.08	91.67 ± 0.00	94.33 ± 0.82
sklearn.svm.SVC	poly 3	98.33 ± 3.33	92.00 ± 0.67	96.00 ± 0.82
sklearn.svm.SVC	poly 4	96.67 ± 4.08	90.00 ± 0.00	95.67 ± 0.82
sklearn.svm.SVC	poly 5	96.67 ± 4.08	89.67 ± 0.67	96.67 ± 0.00
sklearn.svm.SVC	linear	88.33 ± 6.67	98.67 ± 0.67	77.00 ± 2.45
sklearn.svm.SVC	rbf	96.67 ± 4.08	96.67 ± 1.05	98.67 ± 0.67
sklearn.svm.SVC	sigmoid	90.00 ± 6.24	98.33 ± 0.00	79.67 ± 2.87

Table 6.25: QSVM and SVM comparison on digits dataset (after pre-processing and with 5-fold cross validation): class 0 was compared against the others. With PCA, the four principal components were selected. The last three columns show the accuracy, respectively, on test dataset, then true positive on both validation datasets, the first one . "poly n deg" denotes a polinomial kernel of degree n, "rbf" denotes a radial basis function kernel.

seem imminent; for example, the Shor code corrects arbitrary single-qubit errors, but to protect the qubit content both from a bit flip or a sign flip, it requires nine physical qubits to represent a single logical qubit. It can also be noted that the Shor code presupposes equivalent qubits, while in all the real systems created so far there are significant topological limitations.

Anyway, with Quantum Support Vector Machine any feature can be represented using a single qubit; it is possible to accomplish one dimensional classification task even on ibmq_armonk quantum computer, a single qubit quantum computer, while it's inconceivable to perform any non-trivial task on a classic computer with 1 bit or a few bits, let alone face a classification problem.

Therefore, there are solid preconditions for the application of machine learning techniques on quantum computers, including QSVM, to produce very interesting results, providing high time efficiency on high dimensional dataset with to respect to classical Support Vector Machine.

Chapter 7

Preliminary conclusion

For over three decades, starting from the pioneering speculations of Feynman, Benioff, Manin and Deutsch, the study of quantum systems has been very interesting, but mainly theoretical. Instead, in recent years there has been a thriving proliferation of projects and results, both on the hardware and on the software side, thanks to profound efforts made by academic and corporate research groups. In fact, the first prototypes of quantum computers were constructed and made available to the international community of researchers, often through highly flexible and totally free collaborative methods. We have thus witnessed on the one hand the development of solutions capable of implementing more complex circuits, making available a greater number of quantum digits (qubits), improving the accuracy of computation and the reliability of the systems (for example, in 2019 IBM quantum systems reached uptime of 95 percent), on the other hand the proliferation of software solutions characterized by a higher level of abstraction, such as IBM Qiskit library.

In some ways, this trend seems to mimic what has happened in the last few decades for digital electronic computers: on one side, we observed a progressive reduction in size (from entire buildings to a single room, then to a cabinet, after that to solutions that can be installed on desks or even used on the knees, until they land on hand-held devices and mobile phones) and power usage, accompanied by lower purchase costs and total costs of ownership, while simultaneously on the other side we have ascertained enormous progress for processing speed, volatile memory capacity, mass memory capacity, connection interfaces speed, and so on; furthermore, there was the widespread diffusion of ever more abstract and possibly simpler programming languages, as well as of human-machine interfaces that would allow use even to an audience without high mathematical and technical skills (from punched cards to the keyboard, from printing on paper to screens in text mode, from graphical user interfaces and mouse to voice commands, touch screens, etc.).

According to a widespread anecdotal, in 1943 IBM's president, Thomas J Watson, would have reputedly said: "I think there is a world market for about five computers". Nowadays there are definitely more than five functioning quantum computer, but I think that it is objectively difficult to foresee an ubiquitous diffusion of this technology even in a rather remote future. Certainly in the 1940s it was not possible to predict the invention of the transistor, the spread of microelectronics and the adoption of sophisticated lithographic processes, which made it possible to create, after just about 70 years, compact and inexpensive computers albeit millions of times faster than first prototypes, however, it seems very difficult to imagine the possibility of creating quantum computers that can spread in the consumer electronics sector or even in small and medium-sized companies.

On one side, there are multiple theoretical trobubles. For example, the so called no-cloning theorem states that it is not possible to copy of an arbitrary unknown quantum

state, so the single most frequently used assembler instruction (MOVE), or assignment instruction in high level languages, cannot be implemented in the case of quantum computers.

Holevo's theorem proves that although n qubits can use quantum superposition to encode information corresponding up to 2^n states, therefore generally requiring the storage of 2^n complex coefficients in the implementation through a simulator on a classic computer (therefore up to 2^{n+4} conventional bits for double precision real numbers type), the amount of classical information that can be retrieved can be only up to n classical bits. In fact, every time we measure the state of a qubit, we project its possible superposition of states in only one of the eigenstates of the selected measurement operator compatible with the considered state (Wave function collapse). So if we measure n qubits state, the state of the system is projected, in an intrinsically random way, in any of the eigenstates relating to the measurement operator present in the possible overlapping of quantum states.

On the other side, there are multiple hardware challenges. The decoherence phenomena comport the loss of quantum coherence, that is they disrupt a definite phase between different quantum states; a definite relationship is the conditio sine qua non to perform any quantum computation on quantum system, otherwise the quantum information encoded in quantum states are lost or damaged. For a perfectly isolated quantum system, the coherence is preserved under the laws of quantum physics, but then that system would be impossible to manipulate or investigate it, because during any measurement the quantum coherence is inevitably degraded. Decoherence due to unwanted interactions with the external environment can be reduced developing higher fidelity qubits and alleviated using error-correction algorithms by encoding quantum state with redundancy over many qubits; for example, Shor conceived a 9-qubit encoding, so to protect the content of a single logical qubit from any single qubit error using 9 phisically qubit. Just as over the years more sophisticated error detection and correction algorithms have been developed for classic computers (such as the different Reed – Solomon error-correcting codes for storage systems), there is the hope that encoding schemas can be developed with more robust error correction for quantum computing; it has been postulated that significantly increasing the number of physical qubits used to represent a single logical qubit (even reaching hundreds or thousands of qubits) could help in appreciably increasing the decoherence time, providing a quantum computer more time to carry out complex algorithms, which require circuits of greater depth than those implemented so far, but it is really difficult to imagine implementing such kind of expensive coding in a widespread way as long as there will be such a shortage of qubits in the hardware solutions available. Anyway, any multi phisical qubits encoding schema can effectively be pursued only when the individual quantum gates error rate is very small; as shown in Fig. 1.20, until now real quantum computers offer single qubit gate errors in the range $3 \times 10^{-4} - 3 \times 10^{-3}$ and CNOT (the simplest two-qubit gate) error rate $1.0 \times 10^{-2} - 7.5 \times 10^{-2}$, very high error rates compared to conventional electronic computers. For example, as shown in Tab. 4.2 e Tab. 4.3, even the simple addition between two qubits involves a considerable error rate and moreover in the sum of pairs of qubits it is easier to guess the correct result if we consider equiprobable the output binary digit sequences than launching execution on a quantum computer thousands of times and evaluating the frequencies of the possible outputs. The attempt to manage decoherence poses two major problems: the need for cryogenics, to try to minimize unwanted interactions with the external environment, seems to constitute an almost insurmountable limit for an ubiquitous diffusion of quantum computers; the need to minimize measurement operations and, consequently, any type of interaction with classic computers or with man-machine interaction tools (keyboards, mice, screens, microphones, etc.) does not make it possible to imagine a purely quantum interactive

computer. To the best of current theoretical and technological knowledge, therefore, it is possible to predict, even with an optimistic vision aimed at a distant future, at most a significant proliferation of quantum computer solutions as a component of mixed computing systems, in which interactions with the user and probably most of the operations will be carried out by classical digital electronic calculators, while the quantum component will be profitably used only as a specific accelerator for solving some classes of problems.

By now, several studies show that a quantum computer can offer significant speedups in some fields of application, for which many research activities are flourishing aimed at probing the applicability of quantum techniques to the solution of interesting problems. Certainly, as already postulated by Feynmann, functioning quantum computers, equipped with enough good quality qubits, will be indispensable for effectively simulating complex quantum systems. As Shor has shown, these computers can be used profitably to efficiently solve problems such as prime factorization and, consequently, to find important cryptographic applications. Anyway, any theoretical quantum advantage risks being significantly dissipated if coding systems that are too expensive must be adopted to protect the correctness of the application of an algorithm and provide guarantees regarding the quality of the computation.

Another big trouble is that currently, we do not have a quantum equivalent of Random Access Memory: altought there are theoretic studies on quantum RAM (qRAM) [Giovannetti et al., 2008a,b; Knill, 1996; Miszczak, 2011; Nagarajan et al., 2007], until now there is not a sort of cell that can efficiently encode this information as a quantum state and store it for a longer time. Indeed, the status of a qubit produced by the application of a gate must be immediately after being manipulated with the application of another gate or measured by the control system, in any case over a period of time sufficiently reduced to allow the containment of the effects of decoherence. So this is really a very important hardware challenge for quantum computing. if we cannot overcome this obstacle, we will never be able to hope to significantly extend the scope of quantum computation. If you allow me the coarse grain analogy, a current quantum computer is in a situation similar to the GPUs of twenty years ago: in some specific tasks, such as bilinear or trilinear filtering of textures with mipmapping, they achieved superior order performance larger than the CPUs (as an example, I remember a comparison with the Quake II graphics engine between an AMD Athlon Thunderbird CPU at 1400 MHz, with 37.5 million transistors, and a humble first generation Voodoo graphics card , released 5 years earlier and equipped with two specialized chips with one million transistors each, operating at 60 MHz: the video card reached higher frame rates, applying bilinear filtering, than those achieved with the software engine on the CPU with the application of the crudest Nearest-neighbor filter). However, the graphics chips were severely limited, so to apply effects not explicitly supported at the hardware level, one had to carry out complex processing techniques based on multiple passages in the graphics pipeline, with a noticeably deteriorating performance, or completely give up on these visual effects. Only when some flexibility was provided, with various releases of specifications for pixels and vertex shaders, was it possible to first free the creativity of the developers of professional graphic and video game applications and then inaugurate the thriving sector of the GP-GPU.

Probably, the challenge for the near future is to try to understand if, on the basis of rapid theoretical and technological progress, it will be possible to build computers equipped with units dedicated to quantum computation with costs, technical characteristics, maintenance needs, etc. such that they can be deployed ubiquitously in public and private research centers or if these solutions will be relegated forever to particular centers adequately equipped and accessible exclusively via remote interfaces, such as the cloud.

In this work, because of significant limitations found in current and near term quantum computers, clear advantages did not emerge for Quantum Support Vector Machine with to

respect to conventional Support Vector Machine, when applied on conventional dataset, not ad hoc cases designed to be difficult for classical computers. Despite this, it seems clear that in several study cases the QSVM is able to roughly replicate the performance of a classic SVM, which is very interesting in itself, and even more exciting just in light of the limitations of current quantum hardware. Anyway, there are valid reasons to expect good speed increases, applying quantum computation to solve machine learning problems, such as classification task with Quantum Support Vector Machine. The execution time of classical model should lag well behind the execution time of the quantum model, with big enough datasets. Indeed, the quantum model should performs better linear algebra computation than the classical model in terms of time complexity.

So if these physical limitations are mitigated, if accuracy and precision are improved, then the quantum computer and quantum machine learning, like Quantum Support Vector Machine, can be used as the basis for important future developments.

List of Figures

List of Tables

Bibliography

Abraham, H., AduOffei, Akhalwaya, I. Y., Aleksandrowicz, G., Alexander, T., Alexandrowics, G., Arbel, E., Asfaw, A., Azaustre, C., AzizNgoueya, Barkoutsos, P., Barron, G., Bello, L., Ben-Haim, Y., Bevenius, D., Bishop, L. S., Bolos, S., Bosch, S., Bravyi, S., Bucher, D., Burov, A., Cabrera, F., Calpin, P., Capelluto, L., Carballo, J., Carrascal, G., Chen, A., Chen, C.-F., Chen, R., Chow, J. M., Claus, C., Clauss, C., Cross, A. J., Cross, A. W., Cross, S., Cruz-Benito, J., Culver, C., Córcoles-Gonzales, A. D., Dague, S., Dandachi, T. E., Dartiailh, M., DavideFrr, Davila, A. R., Dekusar, A., Ding, D., Doi, J., Drechsler, E., Drew, Dumitrescu, E., Dumon, K., Duran, I., EL-Safty, K., Eastman, E., Eendebak, P., Egger, D., Everitt, M., Fernández, P. M., Ferrera, A. H., Frisch, A., Fuhrer, A., GEORGE, M., Gacon, J., Gadi, Gago, B. G., Gambella, C., Gambetta, J. M., Gammanpila, A., Garcia, L., Garion, S., Gilliam, A., Gomez-Mosquera, J., de la Puente González, S., Gorzinski, J., Gould, I., Greenberg, D., Grinko, D., Guan, W., Gunnels, J. A., Haglund, M., Haide, I., Hamamura, I., Havlicek, V., Hellmers, J., Herok, Ł., Hillmich, S., Horii, H., Howington, C., Hu, S., Hu, W., Imai, H., Imamichi, T., Ishizaki, K., Iten, R., Itoko, T., JamesSeaward, Javadi, A., Javadi-Abhari, A., Jessica, Johns, K., Kachmann, T., Kanazawa, N., Kang-Bae, Karazeev, A., Kassebaum, P., King, S., Knabberjoe, Kovyrshin, A., Krishnakumar, R., Krishnan, V., Krsulich, K., Kus, G., LaRose, R., Lambert, R., Latone, J., Lawrence, S., Liu, D., Liu, P., Maeng, Y., Malyshev, A., Marecek, J., Marques, M., Mathews, D., Matsuo, A., McClure, D. T., McGarry, C., McKay, D., McPherson, D., Meesala, S., Mevissen, M., Mezzacapo, A., Midha, R., Minev, Z., Mitchell, A., Moll, N., Mooring, M. D., Morales, R., Moran, N., MrF, Murali, P., Müggenburg, J., Nadlinger, D., Nakanishi, K., Nannicini, G., Nation, P., Navarro, E., Naveh, Y., Neagle, S. W., Neuweiler, P., Niroula, P., Norlen, H., O'Riordan, L. J., Ogunbayo, O., Ollitrault, P., Oud, S., Padilha, D., Paik, H., Perriello, S., Phan, A., Piro, F., Pistoia, M., Pozas-iKerstjens, A., Prutyanov, V., Puzzuoli, D., Pérez, J., Quintiii, Raymond, R., Redondo, R. M.-C., Reuter, M., Rice, J., Rodríguez, D. M., RohithKarur, Rossmannek, M., Ryu, M., SAPV, T., SamFerracin, Sandberg, M., Sargsyan, H., Sathaye, N., Schmitt, B., Schnabel, C., Schoenfeld, Z., Scholten, T. L., Schoute, E., Schwarm, J., Sertage, I. F., Setia, K., Shammah, N., Shi, Y., Silva, A., Simonetto, A., Singstock, N., Siraichi, Y., Sitdikov, I., Sivarajah, S., Sletfjerding, M. B., Smolin, J. A., Soeken, M., Sokolov, I. O., SooluThomas, Steenken, D., Stypulkoski, M., Suen, J., Sun, S., Sung, K. J., Takahashi, H., Tavernelli, I., Taylor, C., Taylour, P., Thomas, S., Tillet, M., Tod, M., de la Torre, E., Trabing, K., Treinish, M., TrishaPe, Turner, W., Vaknin, Y., Valcarce, C. R., Varchon, F., Vazquez, A. C., Vogt-Lee, D., Vuillot, C., Weaver, J., Wieczorek, R., Wildstrom, J. A., Wille, R., Winston, E., Woehr, J. J., Woerner, S., Woo, R., Wood, C. J., Wood, R., Wood, S., Wood, S., Wootton, J., Yeralin, D., Young, R., Yu, J., Zachow, C., Zdanski, L., Zoufal, C., Zoufalc, a matsuo, adekusar drl, azulehner, bcamorrison, brandhsn, chlorophyll zz, dan1pal, dime10, drholmie, elfrocampeador, faisaldebouni, fanizzamarco, gadial, gruu, jliu45, kanejess, klinvill, kurarrr, lerongil, ma5x, merav aharoni, michelle4654, ordmoj, sethmerkel, strickroman, sumitpuri, tigerjack, toural, vvilpas, welien, willhbang,

yang.luh, yelojakit, and yotamvakninibm (2019). Qiskit: An open-source framework for quantum computing. *Accessed on: Mar*, 16.

Abraham, T. H. (2002). (physio)logical circuits: the intellectual origins of the mcculloch-pitts neural networks. *Journal of the History of the Behavioral Sciences*, 38(1):3–25.

Acampora, G., Herrera, F., Tortora, G., and Vitiello, A. (2018). A multi-objective evolutionary approach to training set selection for support vector machine. *Knowledge-Based Systems*, 147:94–108.

Adachi, S. H. and Henderson, M. P. (2015). Application of quantum annealing to training of deep neural networks. *arXiv preprint arXiv:1510.06356*.

Agnati, L., Leo, G., Zanardi, A., Genedani, S., Rivera, A., Fuxe, K., and Guidolin, D. (2006). Volume transmission and wiring transmission from cellular to molecular networks: history and perspectives. *Acta Physiological Society*, 187(1-2):329–344.

Agnati, L. F., Guidolin, D., Guescini, M., Genedani, S., and Fuxe, K. (2010). Understanding wiring and volume transmission. *Brain research reviews*, 64(1):137–159.

Ahmed, S. (2019). *Pattern recognition with Quantum Support Vector Machine (QSVM) on near term quantum processors.* PhD thesis, Brac University.

Akkary, H. and Driscoll, M. A. (1998). A dynamic multithreading processor. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–236. IEEE.

Alberts, B., Johnson, A., Lewis, J., Raff, M., Roberts, K., and Walter, P. (2008). *Molecular Biology of the Cell 5th*. Garland Science.

Amit, D. J. (1992). *Modeling brain function: The world of attractor neural networks*. Cambridge University Press.

Anderson, J. A. (1995). *An introduction to neural networks*. MIT press.

Arnold, L., Rebecchi, S., Chevallier, S., and Paugam-Moisy, H. (2011). An introduction to deep learning. In *European Symposium on Artificial Neural Networks (ESANN)*.

Arodz, T. and Saeedi, S. (2019). Quantum sparse support vector machines. *arXiv preprint arXiv:1902.01879*.

Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J. C., Barends, R., Biswas, R., Boixo, S., Brandao, F. G., Buell, D. A., et al. (2019). Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510.

Azevedo, F. A., Carvalho, L. R., Grinberg, L. T., Farfel, J. M., Ferretti, R. E., Leite, R. E., Lent, R., Herculano-Houzel, S., et al. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, 513(5):532–541.

Barenco, A., Bennett, C. H., Cleve, R., DiVincenzo, D. P., Margolus, N., Shor, P., Sleator, T., Smolin, J. A., and Weinfurter, H. (1995). Elementary gates for quantum computation. *Physical review A*, 52(5):3457.

Bargmann, C. I. (2012). Beyond the connectome: how neuromodulators shape neural circuits. *Bioessays*, 34(6):458–465.

Beer, R. D. (1995). On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3(4):469–509.

Beer, R. D. (2006). Parameter space structure of continuous-time recurrent neural networks. *Neural Computation*, 18(12):3009–3051.

Bertram, R. (1993). A computational study of the effects of serotonin on a molluscan burster neuron. *Biological Cybernetics*, 69(3):257–267.

Biamonte, J., Wittek, P., Pancotti, N., Rebentrost, P., Wiebe, N., and Lloyd, S. (2017). Quantum machine learning. *Nature*, 549(7671):195.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Bishop, C. M. et al. (1995). *Neural networks for pattern recognition*. Clarendon press Oxford.

Bishop, L. S. (2017). Qasm 2.0: A quantum circuit intermediate representation. *Bulletin of the American Physical Society*, 62.

Bishwas, A. K., Mani, A., and Palade, V. (2018). An all-pair quantum svm approach for big data multiclass classification. *Quantum Information Processing*, 17(10):282.

Bowden, C. M., Chen, G., Diao, Z., and Klappenecker, A. (2002). The universality of the quantum fourier transform in forming the basis of quantum computing algorithms. *Journal of mathematical analysis and applications*, 274(1):69–80.

Bravyi, S., Browne, D., Calpin, P., Campbell, E., Gosset, D., and Howard, M. (2019). Simulation of quantum circuits by low-rank stabilizer decompositions. *Quantum*, 3:181.

Brezina, V. (2010). Beyond the wiring diagram: signalling through complex neuromodulator networks. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 365(1551):2363–2374.

Brylinski, J.-L. and Brylinski, R. (2002). Universal quantum gates. *Mathematics of Quantum Computation*, 79.

Buckley, C. L. (2008). *A systemic analysis of the ideas immanent in neuromodulation*. PhD thesis, University of Southampton.

Buckley, C. L., Bullock, S., and Cohen, N. (2005). Timescale and stability in adaptive behaviour. In *Advances in Artificial Life*, pages 292–301. Springer.

Bunks, C., McCarthy, D., and Al-Ani, T. (2000). Condition-based maintenance of machines using hidden markov models. *Mechanical Systems and Signal Processing*, 14(4):597–612.

Burges, C. J. (1998). A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167.

Care, C. (2010). Introduction: Analogue computers in the history of computing. In *Technology for Modelling*, pages 3–16. Springer.

Carleo, G., Troyer, M., Torlai, G., Melko, R., Carrasquilla, J., and Mazzola, G. (2018). Neural-network quantum states. *Bulletin of the American Physical Society*.

Chakradhar, S., Sankaradas, M., Jakkula, V., and Cadambi, S. (2010). A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 247–257. ACM.

Changeux, J.-P. (1993). Chemical signaling in the brain. *Scientific American*, 269(5):58–58.

Chatterjee, R. and Yu, T. (2016). Generalized coherent states, reproducing kernels, and quantum support vector machines. *arXiv preprint arXiv:1612.03713*.

Chen, J., Wang, L., and Charbon, E. (2017). A quantum-implementable neural network model. *Quantum Information Processing*, 16(10):245.

Chen, X.-W. and Lin, X. (2014). Big data deep learning: challenges and perspectives. *IEEE access*, 2:514–525.

Chien, A. A. and Karamcheti, V. (2013). Moore's law: The first ending and a new beginning. *Computer*, 46(12):48–53.

Chuang, I. L. and Harrow, A. (2018). Mitx 8.371.1x: Quantum information science ii, part 1; quantum states, noise and error correction.

Chuang, I. L. and Shor, P. (2018a). Mitx 8.370.1x: Foundations of quantum and classical computing; quantum mechanics, reversible computation, and quantum measurement.

Chuang, I. L. and Shor, P. (2018b). Mitx 8.370.2x: Simple quantum protocols and algorithms; teleportation and superdense coding, the deutsch jozsa and simon's algorithm, grover's quantum search algorithm, and shor's quantum factoring algorithm.

Chuang, I. L. and Shor, P. (2018c). Mitx 8.370.3x: Foundations of quantum communication; noise and quantum channels, and quantum key distribution.

Cline, B., Niculescu, R. S., Huffman, D., and Deckel, B. (2017). Predictive maintenance applications for machine learning. In *Reliability and Maintainability Symposium (RAMS), 2017 Annual*, pages 1–7. IEEE.

Corley, A.-M. (2010). Intel lifts the hood on its single-chip cloud computer. *IEEE Spectrum Online (9 feb 2010) report from ISSCC-2010, spectrum. ieee. org/semiconductors/processors/intel-lifts-the-hood-on-itssinglechip-cloud-computer*.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

Cross, A. W., Bishop, L. S., Smolin, J. A., and Gambetta, J. M. (2017). Open quantum assembly language. *arXiv preprint arXiv:1707.03429*.

Curnow, H. J. and Wichmann, B. A. (1976). A synthetic benchmark. *The Computer Journal*, 19(1):43–49.

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

Da Silva, A. J., De Oliveira, W. R., and Ludermir, T. B. (2012). Classical and superposed learning for quantum weightless neural networks. *Neurocomputing*, 75(1):52–60.

da Silva, A. J., Ludermir, T. B., and de Oliveira, W. R. (2016). Quantum perceptron over a field and neural network architecture selection in a quantum computer. *Neural Networks*, 76:55–64.

Dayan, P. (2012). Twenty-five lessons from computational neuromodulation. *Neuron*, 76(1):240–256.

Deutsch, D. (1985). Quantum theory, the church–turing principle and the universal quantum computer. In *Proc. R. Soc. Lond. A*, volume 400, pages 97–117. The Royal Society.

Deutsch, D., Barenco, A., and Ekert, A. (1995). Universality in quantum computation. *Proc. R. Soc. Lond. A*, 449(1937):669–677.

Deutsch, D. and Jozsa, R. (1992). Rapid solution of problems by quantum computation. *Proc. R. Soc. Lond. A*, 439(1907):553–558.

Ding, C., Bao, T.-Y., and Huang, H.-L. (2019). Quantum-inspired support vector machine. *arXiv preprint arXiv:1906.08902*.

DiVincenzo, D. P. (1995). Two-bit gates are universal for quantum computation. *Physical Review A*, 51(2):1015.

DiVincenzo, D. P. (1998). Quantum gates and circuits. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 454, pages 261–276. The Royal Society.

Donnarumma, F., Prevete, R., Chersi, F., and Pezzulo, G. (2015). A programmer-interpreter neural network architecture for prefrontal cognitive control. *International Journal of Neural Systems*.

Donnarumma, F., Prevete, R., and Trautteur, G. (2012). Programming in the brain: a neural network theoretical framework. *Connection Science*, 24(2-3):71–90.

Dunn, N. A., Lockery, S. R., Pierce-Shimomura, J. T., and Conery, J. S. (2004). A neural network model of chemotaxis predicts functions of synaptic connections in the nematode caenorhabditis elegans. *Journal of computational neuroscience*, 17(2):137–147.

Fard, E. R., Aghayar, K., and Amniat-Talab, M. (2018). Quantum pattern recognition with multi-neuron interactions. *Quantum Information Processing*, 17(3):42.

Fellous, J.-M. and Linster, C. (1998). Computational models of neuromodulation. *Neural computation*, 10(4):771–805.

Feynman, R. P. (1982). Simulating physics with computers. *International journal of theoretical physics*, 21(6-7):467–488.

Fields, R. D. (2006). Beyond the neuron doctrine. *Scientific American Mind*, 17(3):20–27.

Finnila, A., Gomez, M., Sebenik, C., Stenson, C., and Doll, J. (1994). Quantum annealing: a new method for minimizing multidimensional functions. *Chemical physics letters*, 219(5-6):343–348.

FitzHugh, R. (1955). Mathematical models of threshold phenomena in the nerve membrane. *The bulletin of mathematical biophysics*, 17(4):257–278.

Floreano, D., Dürr, P., and Mattiussi, C. (2008). Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62.

Franklin, R. J. and Bussey, T. J. (2013). Do your glial cells make you clever? *Cell stem cell*, 12(3):265–266.

Fredkin, E. and Toffoli, T. (1981). Conservative logic. In *Collision-based computing*, pages 47–81. Springer.

Funahashi, K.-i. and Nakamura, Y. (1993). Approximation of dynamical systems by continuous time recurrent neural networks. *Neural networks*, 6(6):801–806.

Gally, J. A., Montague, P. R., Reeke, G. N., and Edelman, G. M. (1990). The no hypothesis: possible effects of a short-lived, rapidly diffusible signal in the development and function of the nervous system. *Proceedings of the National Academy of Sciences*, 87(9):3547–3551.

Gankidi, P. R. and Thangavelautham, J. (2017). Fpga architecture for deep learning and its application to planetary robotics. In *Aerospace Conference, 2017 IEEE*, pages 1–9. IEEE.

Giovannetti, V., Lloyd, S., and Maccone, L. (2008a). Architectures for a quantum random access memory. *Physical Review A*, 78(5):052310.

Giovannetti, V., Lloyd, S., and Maccone, L. (2008b). Quantum random access memory. *Physical review letters*, 100(16):160501.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press. `http://www.deeplearningbook.org`.

Gottesman, D. and Chuang, I. L. (1999). Demonstrating the viability of universal quantum computation using teleportation and single-qubit operations. *Nature*, 402(6760):390.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM.

Guo, K., Zeng, S., Yu, J., Wang, Y., and Yang, H. (2017). A survey of fpga based neural network accelerator. *arXiv preprint arXiv:1712.08934*.

Gupta, S. and Zia, R. (2001). Quantum neural networks. *Journal of Computer and System Sciences*, 63(3):355–383.

Gupta, V., Mohapatra, D., Park, S. P., Raghunathan, A., and Roy, K. (2011). Impact: imprecise adders for low-power approximate computing. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, ISLPED '11, pages 409–414, Piscataway, NJ, USA. IEEE Press, IEEE Press.

Haddow, P. C. and Tyrrell, A. M. (2018). Evolvable hardware challenges: Past, present and the path to a promising future. In *Inspired by Nature*, pages 3–37. Springer.

Han, K.-H. and Kim, J.-H. (2000). Genetic quantum algorithm and its application to combinatorial optimization problem. In *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No. 00TH8512)*, volume 2, pages 1354–1360. IEEE.

Hashemian, H. M. and Bean, W. C. (2011). State-of-the-art predictive maintenance techniques. *IEEE Transactions on Instrumentation and measurement*, 60(10):3480–3492.

Hassoun, M. H. (1995). *Fundamentals of artificial neural networks*. MIT press.

Havlíček, V., Córcoles, A. D., Temme, K., Harrow, A. W., Kandala, A., Chow, J. M., and Gambetta, J. M. (2019). Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209.

Havlivcek, V., Córcoles, A. D., Temme, K., Harrow, A. W., Chow, J. M., and Gambetta, J. M. (2018). Supervised learning with quantum enhanced feature spaces. *arXiv preprint arXiv:1804.11326*.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*. Printice-Hall.

Hebb, D. O. (1949). The organization of behavior.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Higuchi, T., Iwata, M., Kajitani, I., Iba, H., Hirao, Y., Furuya, T., and Manderick, B. (1996). Evolvable hardware and its application to pattern recognition and fault-tolerant systems. *Towards evolvable hardware*, pages 118–135.

Hills, G., Lau, C., Wright, A., Fuller, S., Bishop, M. D., Srimani, T., Kanhaiya, P., Ho, R., Amer, A., Stein, Y., et al. (2019). Modern microprocessor built from complementary carbon nanotube transistors. *Nature*, 572(7771):595–602.

Himavathi, S., Anitha, D., and Muthuramalingam, A. (2007). Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888.

Hodgkin, A. L. and Huxley, A. F. (1952a). The components of membrane conductance in the giant axon of loligo. *The Journal of physiology*, 116(4):473–496.

Hodgkin, A. L. and Huxley, A. F. (1952b). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500.

Hölscher, C. (1997). Nitric oxide, the enigmatic neuronal messenger: its role in synaptic plasticity. *Trends in neurosciences*, 20(7):298–303.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558.

Hopfield, J. J., Tank, D. W., et al. (1986). Computing with neural circuits- a model. *Science*, 233(4764):625–633.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

Hu, W. (2018). Towards a real quantum neuron. *Natural Science*, 10(03):99.

Husbands, P. (1998). Evolving robot behaviours with diffusing gas networks. In *Evolutionary robotics*, pages 71–86. Springer.

Husbands, P., Philippides, A., Vargas, P., Buckley, C. L., Fine, P., Di Paolo, E., and O'Shea, M. (2010). Spatial, temporal, and modulatory factors affecting gasnet evolvability in a visually guided robotics task. *Complexity*, 16(2):35–44.

Husbands, P., Smith, T., Jakobi, N., and O'Shea, M. (1998a). Better living through chemistry: Evolving gasnets for robot control. *Connection Science*, 10(3-4):185–210.

Husbands, P., Smith, T., O'Shea, M., Jakobi, N., Anderson, J., and Philippides, A. (1998b). Brains, gases and robots. In *ICANN 98*, pages 51–63. Springer.

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572.

Izhikevich, E. M. (2004). Which model to use for cortical spiking neurons? *IEEE transactions on neural networks*, 15(5):1063–1070.

Izhikevich, E. M. (2007). *Dynamical systems in neuroscience*. MIT press.

Izhikevich, E. M. and FitzHugh, R. (2006). Fitzhugh-nagumo model. *Scholarpedia*, 1(9):1349.

Jeyanthi, S. and Subadra, M. (2014). Implementation of single neuron using various activation functions with fpga. In *Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on*, pages 1126–1131. IEEE.

Johnston, D., Fields, R. D., Bullock, T. H., Bennett, M. V., Josephson, R., and Marder, E. (2005). The neuron doctrine, redux. *Science*, 310(5749):791–793.

Jordan, M. I. and Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260.

Joshi, B., Stewart, K., and Shapiro, D. (2017). Bringing impressionism to life with neural style transfer in come swim. *arXiv preprint arXiv:1701.04928*.

Kadowaki, T. and Nishimori, H. (1998). Quantum annealing in the transverse ising model. *Physical Review E*, 58(5):5355.

Kamermans, M. and Fahrenfort, I. (2004). Ephaptic interactions within a chemical synapse: hemichannel-mediated ephaptic inhibition in the retina. *Current opinion in neurobiology*, 14(5):531–541.

Kandel, E. R., Schwartz, J. H., Jessell, T. M., et al. (2013). *Principles of neural science, 5th edition*. McGraw-Hill New York, 5th edition.

Katz, B. (1969). *The release of neural transmitter substances*, volume 10. Liverpool University Press.

Kaur, A., Kaur, K., and Malhotra, R. (2010). Soft computing approaches for prediction of software maintenance effort. *International Journal of Computer Applications*, 1(16).

Knill, E. (1996). Conventions for quantum pseudocode. Technical report, Los Alamos National Lab, NM.

Krnjevic, K. (1986). Ephaptic interactions: A significant mode of communications in the brain. *Physiology*, 1(1):28–29.

Kumar, S. and Kumar, T. V. (2018). A novel quantum-inspired evolutionary view selection algorithm. *Sādhanā*, 43(10):166.

Kuon, I., Tessier, R., and Rose, J. (2008). Fpga architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253.

Laboudi, Z. and Chikhi, S. (2012). Comparison of genetic algorithm and quantum genetic algorithm. *Int. Arab J. Inf. Technol.*, 9(3):243–249.

Lacey, G., Taylor, G. W., and Areibi, S. (2016). Deep learning on fpgas: Past, present, and future. *arXiv preprint arXiv:1602.04283*.

Lahoz-Beltra, R. (2016). Quantum genetic algorithms for computer scientists. *Computers*, 5(4):24.

Layeb, A. and Saidouni, D.-E. (2007). Quantum genetic algorithm for binary decision diagram ordering problem. *International Journal of Computer Science and Network Security*, 7(9):130–135.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Li, L., Cui, G., Lv, X., Sun, X., and Wang, H. (2018). An improved quantum rotation gate in genetic algorithm for job shop scheduling problem. In *2018 International Conference on Information Systems and Computer Aided Education (ICISCAE)*, pages 322–325. IEEE.

Liang, S., Yin, S., Liu, L., Luk, W., and Wei, S. (2018). Fp-bnn: Binarized neural network on fpga. *Neurocomputing*, 275:1072–1086.

Lisboa, P. G. (1992). *Neural networks: current applications*. Chapman & Hall, Ltd.

Liu, C.-Y., Chen, C., Chang, C.-T., and Shih, L.-M. (2013). Single-hidden-layer feed-forward quantum neural network based on grover learning. *Neural Networks*, 45:144–150.

Liu, L., Luo, J., Deng, X., and Li, S. (2015a). Fpga-based acceleration of deep neural networks using high level method. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2015 10th International Conference on*, pages 824–827. IEEE.

Liu, N. and Rebentrost, P. (2018). Quantum machine learning for quantum anomaly detection. *Physical Review A*, 97(4):042315.

Liu, Q., Dong, M., Lv, W., Geng, X., and Li, Y. (2015b). A novel method using adaptive hidden semi-markov model for multi-sensor monitoring equipment health prognosis. *Mechanical Systems and Signal Processing*, 64:217–232.

Lloyd, S. (1993). A potentially realizable quantum computer. *Science*, 261(5128):1569–1571.

Lloyd, S. (1995). Almost any quantum logic gate is universal. *Physical Review Letters*, 75(2):346.

Lloyd, S. (1996). Universal quantum simulators. *Science*, pages 1073–1078.

Lloyd, S., Mohseni, M., and Rebentrost, P. (2013). Quantum algorithms for supervised and unsupervised machine learning. *arXiv preprint arXiv:1307.0411*.

Lloyd, S., Mohseni, M., and Rebentrost, P. (2014). Quantum principal component analysis. *Nature Physics*, 10(9):631.

Ma, S. and Jin, W. (2007). A new parallel quantum genetic algorithm with probability-gate and its probability analysis. In *International Conference on Intelligent Systems and Knowledge Engineering 2007*. Atlantis Press.

Magg, S. and Philippides, A. (2006). Gasnets and ctrnns–a comparison in terms of evolvability. In *From Animals to Animats 9*, pages 461–472. Springer.

Marder, E. (1998). Electrical synapses: beyond speed and synchrony to computation. *Current biology*, 8(22):R795–R797.

Marder, E. (2012). Neuromodulation of neuronal circuits: back to the future. *Neuron*, 76(1):1–11.

Markov, I. L. (2014). Limits on fundamental limits to computation. *Nature*, 512(7513):147.

Maslov, D., Dueck, G. W., and Miller, D. M. (2005). Toffoli network synthesis with templates. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(6):807–817.

Maslov, D., Dueck, G. W., and Miller, D. M. (2007). Techniques for the synthesis of reversible toffoli networks. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(4):42.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

McHale, G. and Husbands, P. (2004a). Gasnets and other evolvable neural networks applied to bipedal locomotion. *From Animals to Animats*, 8:163–172.

McHale, G. and Husbands, P. (2004b). Quadrupedal locomotion: Gasnets, ctrnns and hybrid ctrnn/pnns compared. In *Proceedings of the 9th International Conference on the Simulation and Synthesis of Living Systems (Alife IX)*, pages 106–112.

Menon, P. S. and Ritwik, M. (2014). A comprehensive but not complicated survey on quantum computing. *IERI Procedia*, 10:144–152.

Minsky, M. and Seymour, P. (1969). Perceptrons.

Misra, J. and Saha, I. (2010). Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, 74(1):239–255.

Miszczak, J. (2011). Models of quantum computation and quantum programming languages. *Bulletin of the Polish Academy of Sciences: Technical Sciences*, 59(3):305–324.

Montague, P. R. and Sejnowski, T. J. (1994). The predictive brain: temporal coincidence and temporal order in synaptic learning mechanisms. *Learning & Memory*, 1(1):1–33.

Monz, T., Nigg, D., Martinez, E. A., Brandl, M. F., Schindler, P., Rines, R., Wang, S. X., Chuang, I. L., and Blatt, R. (2016). Realization of a scalable shor algorithm. *Science*, 351(6277):1068–1070.

Moore, G. E. et al. (1965). Cramming more components onto integrated circuits.

Nagarajan, R., Papanikolaou, N., and Williams, D. (2007). Simulating and compiling code for the sequential quantum random access machine. *Electronic Notes in Theoretical Computer Science*, 170:101–124.

Nagumo, J., Arimoto, S., and Yoshizawa, S. (1962). An active pulse transmission line simulating nerve axon. *Proceedings of the IRE*, 50(10):2061–2070.

Nam, Y., Su, Y., and Maslov, D. (2018). Approximate quantum fourier transform with $o(nlog(n))$ t gates. *arXiv preprint arXiv:1803.04933*.

Narayanan, A. and Moore, M. (1996). Quantum-inspired genetic algorithms. In *Proceedings of IEEE international conference on evolutionary computation*, pages 61–66. IEEE.

Nazari, S., Amiri, M., Faez, K., and Amiri, M. (2015). Multiplier-less digital implementation of neuron–astrocyte signalling on fpga. *Neurocomputing*, 164:281–292.

Nielsen, M. A. and Chuang, I. L. (2010). *Quantum Computation and Quantum Information*. Cambridge University Press.

Noctor, S. C., Martínez-Cerdeño, V., and Kriegstein, A. R. (2007). Contribution of intermediate progenitor cells to cortical histogenesis. *Archives of neurology*, 64(5):639–642.

Nowotniak, R. (2010). Survey of quantum-inspired evolutionary algorithms. In *Materiały konferencyjne Forum Innowacji Młodych Badaczy*.

Nowotniak, R. and Kucharski, J. (2010). Building blocks propagation in quantum-inspired genetic algorithm. *arXiv preprint arXiv:1007.4221*.

Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Ong Gee Hock, J., Liew, Y. T., Srivatsan, K., Moss, D., Subhaschandra, S., et al. (2017). Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 5–14. ACM.

Ota, K., Dao, M. S., Mezaris, V., and De Natale, F. G. (2017). Deep learning for mobile multimedia: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 13(3s):34:1–34:22.

Otterbach, J., Manenti, R., Alidoust, N., Bestwick, A., Block, M., Bloom, B., Caldwell, S., Didier, N., Fried, E. S., Hong, S., et al. (2017). Unsupervised machine learning on a hybrid quantum computer. *arXiv preprint arXiv:1712.05771*.

Palmer, R. M., Ferrige, A., and Moncada, S. (1987). Nitric oxide release accounts for the biological activity of endothelium-derived relaxing factor. *Nature*, 327(6122):524–526.

Papadimitriou, K., Dollas, A., and Hauck, S. (2011). Performance of partial reconfiguration in fpga systems: A survey and a cost model. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(4):36:1–36:24.

Park, J. and Sung, W. (2016). Fpga based implementation of deep neural networks using on-chip memory only. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 1011–1015. IEEE.

Peng, Y. and Dong, M. (2011). A prognosis method using age-dependent hidden semi-markov model for equipment health prediction. *Mechanical Systems and Signal Processing*, 25(1):237–252.

Peng, Y., Dong, M., and Zuo, M. J. (2010). Current status of machine prognostics in condition-based maintenance: a review. *The International Journal of Advanced Manufacturing Technology*, 50(1):297–313.

Perko, M., Fajfar, I., Tuma, T., and Puhan, J. (2000). Low-cost, high-performance cnn simulator implemented in fpga. In *Cellular Neural Networks and Their Applications, 2000.(CNNA 2000). Proceedings of the 2000 6th IEEE International Workshop on*, pages 277–282. IEEE.

Philippides, A., Husbands, P., and O'Shea, M. (1998). Neural signalling: it'sa gas! In *ICANN 98*, pages 979–984. Springer.

Philippides, A., Husbands, P., Smith, T., and O'Shea, M. (2002). Fast and loose: biologically inspired couplings. *Artificial Life VIII, ed. by Standish, R., Abbass, HA and Bedau, MA (MIT Press, Cambridge MA, 2002) pp*, pages 293–301.

Philippides, A., Husbands, P., Smith, T., and O'shea, M. (2005). Flexible couplings: Diffusing neuromodulators and adaptive robotics. *Artificial Life*, 11(1-2):139–160.

Platel, M. D., Schliebs, S., and Kasabov, N. (2007). A versatile quantum-inspired evolutionary algorithm. In *2007 IEEE Congress on Evolutionary Computation*, pages 423–430. IEEE.

Ransom, B. R. and Orkand, R. K. (1996). Glial-neuronal interactions in non-synaptic areas of the brain: studies in the optic nerve. *Trends in neurosciences*, 19(8):352–358.

Rash, J. E., Dillman, R. K., Bilhartz, B. L., Duffy, H. S., Whalen, L. R., and Yasumura, T. (1996). Mixed synapses discovered and mapped throughout mammalian spinal cord. *Proceedings of the National Academy of Sciences*, 93(9):4235–4239.

Rashevsky, N. (1933). Outline of a physico-mathematical theory of excitation and inhibition. *Protoplasma*, 20(1):42–56.

Rashevsky, N. (1938). *Mathematical biophysics. Physicomathematical foundations of biology*, volume 111. Chicago.

Rebentrost, P., Bromley, T. R., Weedbrook, C., and Lloyd, S. (2017). A quantum recurrent neural network. *arXiv preprint arXiv:1710.03599*.

Rebentrost, P., Mohseni, M., and Lloyd, S. (2014). Quantum support vector machine for big data classification. *Physical review letters*, 113(13):130503.

Ristè, D., Da Silva, M. P., Ryan, C. A., Cross, A. W., Córcoles, A. D., Smolin, J. A., Gambetta, J. M., Chow, J. M., and Johnson, B. R. (2017). Demonstration of quantum advantage in machine learning. *npj Quantum Information*, 3(1):16.

Rojas, R. (1996). *Neural networks: a systematic introduction*. Springer.

Rønnow, T. F., Wang, Z., Job, J., Boixo, S., Isakov, S. V., Wecker, D., Martinis, J. M., Lidar, D. A., and Troyer, M. (2014). Defining and detecting quantum speedup. *Science*, 345(6195):420–424.

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

Rosenblatt, F. (1961). Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, DTIC Document.

Ruan, Y., Chen, H., Tan, J., and Li, X. (2016). Quantum computation for large-scale image classification. *Quantum Information Processing*, 15(10):4049–4069.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. *Parallel Distributed Processing*, pages 318–362.

Russell, S. and Norvig, P. (1995). *Artificial intelligence: a modern approach*. Prentice Hall.

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Rycerz, K., Patrzyk, J., Patrzyk, B., and Bubak, M. (2015). Teaching quantum computing with the quide simulator. *Procedia Computer Science*, 51:1724–1733.

Sakurai, A. and Katz, P. S. (2009). State-, timing-, and pattern-dependent neuromodulation of synaptic strength by a serotonergic interneuron. *The Journal of Neuroscience*, 29(1):268–279.

Salleh, N. S. M. and Baharim, M. F. (2015). Performance comparison of parallel execution using gpu and cpu in svm training session. In *2015 4th International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, pages 214–217. IEEE.

Sasao, T. and Kinoshita, K. (1979). Conservative logic elements and their universality. *IEEE Transactions on Computers*, 28(9):682–685.

Schmit, H. and Huang, R. (2016). Dissecting xeon+ fpga: Why the integration of cpus and fpgas makes a power difference for the datacenter. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, pages 152–153. ACM.

Schuld, M., Bocharov, A., Svore, K., and Wiebe, N. (2018). Circuit-centric quantum classifiers. *arXiv preprint arXiv:1804.00633*.

Schuld, M. and Killoran, N. (2018). Quantum machine learning in feature hilbert spaces. *arXiv preprint arXiv:1803.07128*.

Schuld, M., Sinayskiy, I., and Petruccione, F. (2014). The quest for a quantum neural network. *Quantum Information Processing*, 13(11):2567–2586.

Schuld, M., Sinayskiy, I., and Petruccione, F. (2015a). An introduction to quantum machine learning. *Contemporary Physics*, 56(2):172–185.

Schuld, M., Sinayskiy, I., and Petruccione, F. (2015b). Simulating a perceptron on a quantum computer. *Physics Letters A*, 379(7):660–663.

Shafique, M., Hafiz, R., Javed, M. U., Abbas, S., Sekanina, L., Vasicek, Z., and Mrazek, V. (2017). Adaptive and energy-efficient architectures for machine learning: Challenges, opportunities, and research roadmap. In *VLSI (ISVLSI), 2017 IEEE Computer Society Annual Symposium on*, pages 627–632. IEEE.

Sharma, H., Park, J., Suda, N., Lai, L., Chau, B., Kim, J. K., Chandra, V., and Esmaeilzadeh, H. (2017). Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. *arXiv preprint arXiv:1712.01507*.

Shende, V. V., Prasad, A. K., Markov, I. L., and Hayes, J. P. (2002). Reversible logic circuit synthesis. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 353–360. ACM.

Shi, Y. (2002). Both toffoli and controlled-not need little help to do universal quantum computation. *arXiv preprint quant-ph/0205115*.

Shor, P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134. Ieee.

Simon, D. R. (1997). On the power of quantum computation. *SIAM journal on computing*, 26(5):1474–1483.

Sipper, M., Mange, D., and Sanchez, E. (1999). Quo vadis evolvable hardware? *Communications of the ACM*, 42(4):50–56.

Smith, R. S., Curtis, M. J., and Zeng, W. J. (2017). A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355v2*.

Smith, T., Husbands, P., and O'Shea, M. (2001). Not measuring evolvability: Initial investigation of an evolutionary robotics search space. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 9–16. IEEE.

Smith, T., Husbands, P., Philippides, A., and O'Shea, M. (2002). Neuronal plasticity and temporal adaptivity: Gasnet robot control networks. *Adaptive Behavior*, 10(3-4):161–183.

Smith, T. and Philippides, A. (2000). Nitric oxide signalling in real and artificial neural networks. *BT Technology Journal*, 18(4):140–149.

Sofge, D. A. (2008). Prospective algorithms for quantum evolutionary computation. *Proceedings of the Second Quantum Interaction Symposium (QI-2008), College Publications, UK, 2008*.

Söhl, G., Maxeiner, S., and Willecke, K. (2005). Expression and functions of neuronal gap junctions. *Nature Reviews Neuroscience*, 6(3):191–200.

Soltoggio, A. (2008). *Evolutionary and Computational Advantages of Neuromodulated Plasticity*. PhD thesis, University of Birmingham, UK.

Strogatz, S. H. (2001). *Nonlinear dynamics and chaos: with applications to physics, biology and chemistry*. Perseus publishing.

Su, Y.-C., Cheng, F.-T., Hung, M.-H., and Huang, H.-C. (2006). Intelligent prognostics system design and implementation. *IEEE Transactions on Semiconductor Manufacturing*, 19(2):195–207.

Susto, G. A., Schirru, A., Pampuri, S., McLoone, S., and Beghi, A. (2015). Machine learning for predictive maintenance: A multiple classifier approach. *IEEE Transactions on Industrial Informatics*, 11(3):812–820.

Suykens, J. A., Lukas, L., and Vandewalle, J. (2000). Sparse approximation using least squares support vector machines. In *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No. 00CH36353)*, volume 2, pages 757–760. IEEE.

Suykens, J. A. and Vandewalle, J. (1999). Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

Tkachuk, V. (2018). Quantum genetic algorithm on multilevel quantum systems. *Mathematical Problems in Engineering*, 2018.

Toffoli, T. (1981). Bicontinuous extensions of invertible combinatorial functions. *Mathematical Systems Theory*, 14(1):13–23.

Tucker, A. W. and Kuhn, H. (1951). Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics nd Probability, Univ. of California Press*, pages 481–492.

Turrigiano, G. G. (2006). More than a sidekick: glia and homeostatic synaptic plasticity. *Trends in molecular medicine*, 12(10):458–460.

Udrescu, M., Prodan, L., and Vlăduţiu, M. (2006). Implementing quantum genetic algorithms: a solution based on grover's algorithm. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 71–82. ACM.

Vandersypen, L. M., Steffen, M., Breyta, G., Yannoni, C. S., Sherwood, M. H., and Chuang, I. L. (2001). Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414(6866):883.

Vasicek, Z. and Sekanina, L. (2007). An evolvable hardware system in xilinx virtex ii pro fpga. *International Journal of Innovative Computing and Applications*, 1(1):63–73.

Verdon, G., Broughton, M., and Biamonte, J. (2017). A quantum algorithm to train neural networks using low-depth circuits. *arXiv preprint arXiv:1712.05304*.

Vv.Aa. (1997). *Intel Architecture Optimization Manual*. Intel.

Wang, C., Gong, L., Yu, Q., Li, X., Xie, Y., and Zhou, X. (2017a). Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517.

Wang, H., Liu, J., Zhi, J., and Fu, C. (2013). The improvement of quantum genetic algorithm and its application on function optimization. *Mathematical problems in engineering*, 2013.

Wang, H., Xu, Z., and Pedrycz, W. (2017b). An overview on the roles of fuzzy set techniques in big data processing: Trends, challenges and opportunities. *Knowledge-Based Systems*, 118:15–30.

White, J., Southgate, E., Thomson, J., and Brenner, S. (1986). The structure of the nervous system of the nematode caenorhabditis elegans: the mind of a worm. *Phil. Trans. R. Soc. Lond*, 314:1–340.

Willsch, D., Willsch, M., De Raedt, H., and Michielsen, K. (2019). Support vector machines on the d-wave quantum annealer. *arXiv preprint arXiv:1906.06283*.

Wilson, C., Otterbach, J., Tezak, N., Smith, R., Crooks, G., and da Silva, M. (2018). Quantum kitchen sinks: An algorithm for machine learning on near-term quantum computers. *arXiv preprint arXiv:1806.08321*.

Wu, S.-j., Gebraeel, N., Lawley, M. A., and Yih, Y. (2007). A neural network integrated decision support system for condition-based optimal predictive maintenance policy. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 37(2):226–236.

Xiao, Q., Liang, Y., Lu, L., Yan, S., and Tai, Y.-W. (2017). Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 62:1–62:6. ACM.

Yam, R., Tse, P., Li, L., and Tu, P. (2001). Intelligent predictive decision support system for condition-based maintenance. *The International Journal of Advanced Manufacturing Technology*, 17(5):383–391.

Yao, X. and Higuchi, T. (1999). Promises and challenges of evolvable hardware. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 29(1):87–97.

Zeng, W., Johnson, B., Smith, R., Rubin, N., Reagor, M., Ryan, C., and Rigetti, C. (2017). First quantum computers need smart software. *Nature*, 549:149–151.

Zeng, Y., Jiang, W., Zhu, C., Liu, J., Teng, W., and Zhang, Y. (2006). Prediction of equipment maintenance using optimized support vector machine. *Computational Intelligence*, pages 570–579.

Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., and Cong, J. (2015). Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM.

Zhang, G. (2011). Quantum-inspired evolutionary algorithms: a survey and empirical study. *Journal of Heuristics*, 17(3):303–351.

Zhao, Z., Pozas-Kerstjens, A., Rebentrost, P., and Wittek, P. (2018). Bayesian deep learning on a quantum computer. *arXiv preprint arXiv:1806.11463*.

Zhou, Y., Redkar, S., and Huang, X. (2017). Deep learning binary neural network on an fpga. *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 281–284.

Zhou, Z.-J., Hu, C.-H., Xu, D.-L., Chen, M.-Y., and Zhou, D.-H. (2010). A model for real-time failure prognosis based on hidden markov model and belief rule base. *European Journal of Operational Research*, 207(1):269–283.

Zoli, M. and Agnati, L. F. (1996). Wiring and volume transmission in the central nervous system: the concept of closed and open synapses. *Progress in neurobiology*, 49(4):363–380.

Zoli, M., Torri, C., Ferrari, R., Jansson, A., Zini, I., Fuxe, K., and Agnati, L. F. (1998). The emergence of the volume transmission concept. *Brain research reviews*, 26(2):136–147.