POST-DIGITAL - European Training Network on Post-Digital Computing [GA860360]

Document Details

Title	Deliverable 1.1 Report on critical gaps in existing formal methods to capture computation in unconventional substrates
Deliverable number	D1.1
Deliverable Type	Report (public)
Deliverable title	Report on critical gaps in existing formal methods to capture computation in unconventional substrates
Work Package	WP1- New concepts and theory
Description	Report on critical gaps in existing formal methods to capture computation in unconventional substrates
Deliverable due date	30/09/2021
Actual date of submission	19/10/2021
Lead beneficiary	Groningen Authors: ESR1 Steven Abreu, ESR2 Guillaume Pourcel Co-authors: PI Herbert Jaeger, ESR7 Diego Arguello Ron, ESR10 Benedikt Vettelschoss
Version number	V1.0
Status	Final

Dissemination level

PU	Public	X
CO	Confidential, only for members of the consortium (including Commission Services	

Project Details

Grant Agreement	860360
Project Acronym	POST-DIGITAL
Project Title	POST-DIGITAL - European Training Network on Post-Digital Computing
Call Identifier	H2020-MSCA-ITN-2019
Project Website	https://postdigital.astonphotonics.uk/
Start of the Project	1 April 2020
Project Duration	48 months

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 860360.

Table of Contents

1. Introduction 4
2. Existing work 6
2.1 Existing approaches 6
2.2 Existing hardware 9
3. Selected Themes 9
3.1 Analog and Digital 9
3.2 Timescales 12
3.3 Stochasticity 14
3.4 Robustness 16
3.5 Programming 19
3.6 Information and representation in cognitive systems 36
3.7 Inspiration from the brain 39
3.8 Mathematical modeling: requirements and deficits 45
4. Conclusion 57
References 58

1 Introduction

The pervasiveness and power of digital computing can hardly be overstated; it has lead the way into a new digital age and shaped our societies. The power of digital computing comes from a rich and unified framework of interrelated theories that describe digital computation. This theoretical framework has served as a guiding light for hardware manufacturers, software designers, and everyone in between - and their joint contributions have constructed the monumental tower of digital computing.

But not all is well in digital computing. A number of issues and limitations have lead to the reinvestigation of the very foundations on which digital computing so firmly rests [Copeland et al., 2016]. The exploding investment costs for microchip fabrication make it harder to further downscale transistors, hinting at fundamental limits [Waldrop, 2016] and suggesting that future progress may come from sources other than manufacturing and miniaturization, like computer architecture [Hennessy and Patterson, 2019] or software development [Larus, 2009]. With this, the clear hardware/software separation may cease to be tenable - ESR1 discusses this issue in Section 3.5.

At the same time, an increasing diversity in computer hardware for specialized demands is emerging, as evidenced by the GPU for graphics processing (and follow-up applications, like training machine learning models), the FPGA for embedded devices and replacing the need for ASICs in some cases, the TPU for Google's machine learning purposes, and an increasing number of more unconventional hardware systems - some of which are investigated and developed in the Post-Digital consortium (cf. Deliverable 2.1). The price in efficiency that we pay for the convenience of general-purpose computers, termed "Turing Tariff" [Edwards, 2021], is growing too high. The result may be a new era of closer collaboration between hardware and software research, leading to the co-development of hardware and software. This will likely enable many new research ideas to flourish which have failed in the past because they were incompatible with the hardware available. As an example, Hooker [2020] argues that deep learning had to wait decades until the wide availability of a new computing device, the GPU, in order to be widely accepted as a promising research direction. Unconventional computers may enable us to go where present digital computing cannot take us.

In terms of theory, many alternative approaches to computing have emerged in past decades. We use unconventional computing as an umbrella term for all such theories of computation. Most notably, and of particular relevance to the Post-Digital consortium is neuromorphic computing. Fueled by the deep learning revolution [LeCun et al., 2015] and the demonstrated power of such cognitive-style computing [Silver et al., 2018], neuromorphic computing has recently moved towards the spotlight of unconventional computing paradigms. With the ability to break through the energy barrier that digital computing cannot penetrate, neuromorphic computing technology promises to deliver brain-like energy efficiency [Boahen, 2017, Sarpeshkar, 1998]. Of course, taking inspiration from the brain to build computers is difficult if we do not understand how and what computation takes place in the brain. ESR2 outlines this issue in Section 3.7.

However, it may be argued that taking inspiration from the brain to build next-generation computers has its limits. We are not building computers from "wet" materials like neuronal tissues. Instead, we are already working on exploiting physical effects that cannot occur in biological tissues, like the massive parallelism and unmatched speed of photonics that is exploited by our partners in the Post-Digital consortium. So perhaps what we should learn from the brain is that it exploits the computational capabilities of its underlying physical substrate. If we apply this lesson to the materials in which we intend to build next-generation computers, this leads us

to considerably widen our research agenda and investigate how physical phenomena in whatever material can be harnessed for computation [Jaeger, 2021]. This is not a new idea, but has already been investigated for decades under terms like "physical computing", "in-materio computing", and many more (cf. Section 2.1).

As we venture out to work with unconventional computers exploiting novel material effects which deviate more and more from the digital computing paradigm, the lack of support from a guiding theoretical pillar manifests in many ways. It is precisely this lack of theoretical support and guidance in the development of unconventional computers which motivates the present report. Jaeger [2021] describes how the theory of digital computing has guided the development of the field:

> The formal theory of DC has crystallized into canonical textbooks which are taught to computer science students worldwide in essentially always the same coverage and terminology. I find it important to point out that the powers of the DC paradigm do not emerge from a single formal theory or model. Besides the theory of Turing machines, which could be mistaken as the theory of DC, there are other formalisms, models and theories that are just as essential for the practical manifestations of DC. They include the theories of automata, formal grammars and languages, programming languages and compiler design, computability and complexity theory, Boolean and first-order logic, other logics and metalogical frameworks. These theories, models and formalisms are tightly and transparently interrelated. Their totality can be likened to an orchestra which instruments all professional DC activities from device engineering to microchip fabrication technologies, from circuit design to computer architectures and communication networks, from programming language development to human–computer interfacing, from databases to internet services, from beginners' programming exercises to software engineering and use-case specification frameworks and all the rest.

In this report, we aim to give an overview of the existing theories that can be used to capture computation in unconventional substrates. Our main goal is to depict the current landscape of unconventional computing research and to point the most promising directions to be pursued by the theoretically oriented ESRs in the Post-Digital project. We begin by giving a brief (and necessarily incomplete) overview of existing approaches in Section 2.1 to extend our current notion of "computing". For completeness, we also give a (very) brief overview of some of the existing hardware that is mentioned in this report in Section 2.2, but the reader is referred to the Post-Digital deliverable D2.1 for a more detailed report on the hardware systems that are used and developed in the Post-Digital consortium.

The main part of the report is organized in different themes which we have selected to reflect some of the areas in which critical gaps in our theoretical understanding are evident. The fragmented, yet also overlapping, organization of our report is a reflection of the similarly fragmented, overlapping nature of the theories which are needed to drive the understanding and development of the field. For each of the selected themes, we give an overview of the problems and conclude each section by outlining some of the critical gaps that have been identified.

At this point it may be pointed out that the main authors of this deliverable are ESR1 and ESR2. Their research is directly related to the content of this deliverable and describes the environment in which they work, as well as the environment in which the visiting ESRs (3, 5, 8, 9, and 10) will be embedded for their secondments. ESR1 is working on developing new concepts, formalisms and methods for programming unconventional computers and will be working on solving real-world relevant learning tasks on the DYNAP-SE2 to exploit a well

characterized set of nonlinear phenomena of this analog microprocessor. Naturally, he wrote Section 3.5 on programming, as well as Sections 2.2, 3.1, and contributed to Sections 3.2, 3.3, 3.4. ESR2 wrote Sections 3.6 and 3.7 as this is related to his PhD project, and contributed to Section 3.2. He is working with formal and algorithmic procedures to observe/measure, stabilize and/or control neuromorphic systems. His starting point is conceptor [Jaeger, 2014] which are being studied in various ways in his host group. His work will also contribute to shield neuromorphic systems against parameter drift, device mismatch, aging, and outside perturbations. ESR7 also contributed to Sections 3.3 and 3.4 as he is working on the role of noise and robustness in his research and experimental setup. Lastly, PI Jaeger contributed to this deliverable by writing Sections 2.1 and 3.8.

2 Existing work

2.1 Existing approaches

A wide range of physical, biological, chemical, social systems have been considered as “computing”, “cognitive”, “intelligent”, “processing information”, “processing signals” in one way or other in numerous scientific communities and subcommunities, for instance computer science and AI (of course), analog computing, machine learning and neural computation, computational and cognitive neuroscience, behavior-based robotics, statistical physics, information theory, signal processing and control, network theory, materials science, dynamical systems theory, cybernetics and theoretical biology, immunology, and within the manifold approaches that have been variously called unconventional computing, physical computing, in-materio, neuromorphic computing (et cetera — we have a private list of about 20 names that have been forwarded in the last five decades). Often, but not always, the respective “computing” (wide sense) systems have been formally modeled in various degrees of rigor and detail. There are two main motivations why formalizations have been worked out. The first motivation is scientific understanding of the respective real-world systems in the spirit of exact natural science: formal models can generate hypotheses which are submitted to empirical tests. The second motivation is to obtain innovative procedures for carrying out “computations” for relevant practical tasks — procedures which in one way or other expand the canon of digital/symbolic programming. All of these modeling efforts add to the richness of our understanding of “computing”, extending and sometimes complementing the standard digital/symbolic paradigm in many directions. A comprehensive survey would be a monumental task. All we can do here is to point out, in random order, a number of modeling approaches which have been formally worked out to a certain depth and which, taken together, illustrate the conceptual and methodological richness of the wider “computing” landscape.

- According to the classical work by Ashby [1952], the main evolutionary drive for biological brains is that they must enable adaptation to qualitatively changing exernal conditions. This led (at least in “higher” animals) to **ultrastable** information processing mechanisms which go beyond the customary stabilization mechanisms known from control theory. An ultrastable brain commands on discontinuous search-and-discover mechanisms to quickly cope with novel external challenges. The theory is formalized in the mathematical language of cybernetics and control theory.
- Mathematical models in (human and animal) **motion science** describe complex motor patterns and analyze how they can be perceived and controlled [Hogan and Flash, 1987, Thoroughman and Shadmehr, 2000, d’Avella et al., 2003]. This research is potentially

instructive for generalizing digital, discrete-time computing models to continuous-valued, continuous-time materials and mechanisms, in that it provides many worked-out ideas how one can identify symbol-like invariants in continuously evolving high-dimensional signals.

- Grenander's **pattern theory**, especially in the transparent rendering of Fields medalist David Mumford [Mumford, 1994, 2002], offers a thoroughly formal account of how (primarily spatial / visual) "patterns" which are emerging in complex physical systems can be generated, compounded, transformed and encoded.
- A classical subfield of AI, **qualitative physics** [Forbus, 1988] (closely related: *naive physics*, *qualitative reasoning*) explores logic-based formalisms which capture the everyday reasoning of humans about their mesoscale physical environment.
- The **free energy principle** [Friston et al., 2010] gives a first-principles account of hierarchical information processing in biological brains, starting from the premise that the central purpose of brain function is to make the predictions and decide for the actions which are most relevant for continual survival. This approach has been worked out in connected formal theories which range from explanations of (spiking) neural feedback mechanisms to highly abstracted, information-theoretic models of predictive, adaptive autonomous agents.
- **Reservoir computing** [Maass et al., 2002, Jaeger and Haas, 2004] is a model of a learning architecture that uses a random, high-dimensional, non-adaptive, nonlinearly excitable medium as computing substrate. First defined for recurrent neural networks [Maass et al., 2002, Jaeger and Haas, 2004], reservoir computing has become one of the leading enablers for computing based on unconventional materials [Tanaka et al., 2019]. It is also a main paradigm for most research in Post-Digital.
- **Stochastic computing** [von Neumann, 1956] and **hyperdimensional computing** [Kanerva, 2009] are two related formal theories where the carriers of information are random (long or even infinite) bitstrings, which can be combined into new such bitstrings by operations which can be interpreted as logical or numerical.
- Insights gained in the fields of **emergent computation** [Forrest, 1990] steer attention to the powers of collective phenomena in dissipative systems, where macrolevel phenomena "self-organize" from the interactions of microlevel components.
- Machine learning and data mining methods for detecting **concept drift** [Gama et al., 2013] offer statistical characterizations of how data streams change qualitatively over time, including recent methods which exploit hierarchical structuring of distributions [Hammoodi et al., 2018].
- Recent propositions to develop a theory of **stream automata** [Endrullis et al., 2019] aim at extending the classical theory of finite-state automata to infinite data stream processing.
- The **neural engineering framework** of Eliasmith [2005], Eliasmith et al. [2012a] is a compendium of rigorously defined algorithmic procedures which allow to realize complex signal processing and control architectures in substrates of spiking neurons. This framework has become quite influential in the design and programming of spiking neuromorphic hardware microchips and architectures [Neckar et al., 2019].
- The **dynamic field theory** [Schöner, 2019] models neural processing through nonlinearly interacting fields and solitons on hierarchies of two-dimensional, spatially continuous neural sheets.

- **Heteroclinic channels** [Rabinovich et al., 2008] provide an explanation of conceptual sequencing in neural dynamics in terms of neural state trajectories wandering between saddle nodes — this approach is one of the few that give a mathematical account of symbolic reasoning (though of restricted kinds) in terms of dynamical systems theory.
- According to **neural sampling** models [Buesing et al., 2011], ensembles of random spike events can be interpreted as sampling from probability distributions which in turn are cognitively interpretable and are organized and processed as in Bayesian networks. In Section 3.8 we discuss these models in more detail.
- Under the banner line of **neuro-symbolic integration**, in recent years an interdisciplinary community has emerged, where logicians from theoretical computer science collaborate with neural network and machine learning researchers. A variety of formal models have been developed which on the one hand, aim at explaining how logical-symbolic inferences can arise in neural dynamics, and on the other hand how neural networks can be used to efficiently implement logical inference mechanisms [Besold et al., 2017].
- **Constructor theory** [Deutsch and Marletto, 2015] is an exemplary instance of a number of approaches in foundational physics to reconstruct/explain the laws of physics from first principles of information processing, and the temporal evolution of physical reality in terms of “computing”.
- Wolfram [2020] has started a large-scale community project where local graph transformation rules are studied which yield growing graphs, some of which can simulate Turing machines. Wolfram’s vision is to find a set of simple rules whose associated growing graphs can explain all known physical phenomena — another instance of the idea that the physical universe can be understood as a computing system.
- In their formalization based on **commuting diagrams** where physical systems and abstract models of computing tasks are connected through semantic and procedural relations, Horsman et al. [2014] give a general, abstract, and formal account of what it means that a physical system “computes”. Their view is far more general than the digital computing perspective.
- [Zhang et al., 2020] present a principled strategy which allows users of current industrial spiking neuromorphic microchips to program the latter, compiling abstract task specifications to machine code through a **hierarchy of formal compilation procedures**. The intermediate representation formalisms can be regarded as modeling formalisms for neuromorphic computing.

Modeling approaches like the ones listed here illustrate that the concept of “computing” has many dimensions beyond the canonical framing of digital computing. However, so far we do not yet possess a theory framework which

1. on the one hand is general enough to encompass all the currently explored venues to non-digital computing, and
2. on the other hand is so specific that it can practically guide hardware engineers, system architecture designers, and programmers alike to let them systematically design, build, program and use unconventional computing systems.

The models and formalisms listed above fall short of meeting these two conditions, for a number of reasons — they are too inexpressive, or too abstractly “meta”, or restricted to specific material substrates or sets of physical phenomena, or too premature, or too closely connected to the

digital-symbolic paradigm. In Section 3.8 we will analyse in more detail which requirements have to be met for a comprehensive theoretical modeling of “computing”.

2.2 Existing hardware

In this section, we give a short introduction to some of the hardware systems and experimental platforms that will be mentioned in this deliverable. This choice coincides largely with the range of hardware platforms that are being used in the Post-Digital consortium. For a more detailed treatment, please refer to the deliverable D2.1 (by CSIC-IFISC): "Report on the design of photonic- and electronic- based analogue computing".

A first distinction is to be made in the signal processing mode of the hardware systems. Here we differentiate between digital, analog, and hybrid (mixed) signal processing. The details of digital, analog, and mixed-signal computation are discussed in Section 3.1.

Electronics is the most mature technology in which computing hardware is built. Conventional computer chips are generally built using digital CMOS technology. However, CMOS technology can also be used to build analog or mixed-signal computers [Mead, 1990, Moradi et al., 2018]. Memristors, which are fabricated using non-CMOS materials, have been shown to be compatible with CMOS technology, leading to hybrid memrisive-CMOS circuits [Li et al., 2018]. Spintronic oscillators exploit magnetization dynamics and have been demonstrated to be compatible with CMOS technology, providing a simple, ultra-compact, low-power element that can emulate collections of neurons [Torrejon et al., 2017]. In our Post-Digital consortium, photonic computing plays a major role. Photonic computers work in the optical, instead of the electronic, domain. We also distinguish between fully photonic [Duport et al., 2012] and photonic-electronic [Paquot et al., 2012, Ying et al., 2020] computers in which the signal is converted between the optical and electronic domain. Other materials exist in which computation has been demonstrated: van Noort et al. [2002] used DNA to solve combinatorial optimization problems, Adamatzky [2007] used slime mould to implement a general-purpose computer, and Cucchi et al. [2021] used organic electro-chemical transistors to classify arrhythmic heartbeats in real time.

3 Selected Themes

3.1 Analog and Digital

Analog computation is commonly understood as computation using the continuum, i.e. real numbers $x \in \mathrm{R}$ (and/or real time $t \in \mathrm{R}$) whereas digital computing is concerned only with sets that can be put into isomorphism with natural numbers $n \in \mathrm{N}$. However, the term "analog computing" historically stems from computing using analogy, i.e. building computers that evolve in the same was as the system that they are used to model. Assuming a continuous model of physical reality, the two views are largely identical.

The prevalent model of digital computation has been discovered several times in different, yet equivalent, forms - most notably as the Turing machine [Turing, 1936], recursive functions, and the lambda-calculus. Approaches to formalize analog computation exist, yet are not easily connected and present rather independent views and models [Bournez and Pouly, 2018, Shannon, 1941, Rubel, 1993, Moore, 1996, Blum et al., 1989]. A common result in the theory of analog computation is that access to continuous values can lead to super-Turing computation, i.e. the ability to solve problems that a digital computer cannot solve. Siegelmann and Sontag [1995]

proved that a neural network with rational weights is able to compute any Turing-computable function. Further, Siegelmann and Sontag [1994] proved that recurrent neural networks with real weights are super-Turing, and so are evolving recurrent neural networks, even with rational weights [Cabessa and Siegelmann, 2011]. More recently, George et al. [2016] developed the Field-Programmable Analog Array (FPAA), a reconfigurable mixed-mode computing device that was inspired by the FGPA. This represents an important step towards more programmable, general-purpose analog (or mixed-mode) hardware.

Digital computing is rooted in logic which is reflected in the training of computer scientists and results in a clear preference of computer scientists to use discrete-algebraic and logic-based formalisms. This stands in stark contrast to the continuous, dynamical systems based modeling languages that physicists use, and which are a natural choice for describing analog computers. Because of this language difference, analog computers are often used in domains where the computation or modeling is phrased in terms of differential equations. Indeed, the first analog computers which were used for the analysis of ballistic trajectories computed differential equations, and so does Shannon's differential analyzer [Shannon, 1941].

The dichotomy of digital and analog computing devices is well-established in common parlance. Upon closer inspection, it is observed that there is no clear line that separates digital computers from analog ones. It can be argued that everything that is physically real, moves continuously through time and state space and physical discrete-state machines therefore do not exist except as formal abstraction. This was already acknowledged by Turing [1950]. At the same time, the computer's (and the user's) ability to discriminate between values is limited in precision, therefore allowing the machine to move only within a finite set of states. Giving the analog-digital divide more ontological significance than it has leads to contradictions and non-sensical questions of whether a certain machine is analog or digital. There are no intrinsic physical properties that divide machines into either digital or analog [Moor, 1978]. Classifying a machine as digital or analog is an *interpretation* of the computing machine on a symbolic level, and therefore depends on the modeler's purpose and preference. Although a physical machine can be interpreted as either analog or digital, it is not clear how the two models of computation are related.

The benefits of digital computing are clear; they are easy to program, there is a very productive unified area of research that investigates their limits and complexity, and exponential scaling laws have been providing us with continuous performance improvements (the continuation of which is not guaranteed). In contrast, analog computers do not have a unified research community and progress cannot keep up with digital scaling laws. However, as digital computing is approaching fundamental limits regarding energy and size [Waldrop, 2016, Hennessy and Patterson, 2019], it may be worth to re-investigate analog computing. Hasler and Marr [2013] show that computational efficiency, defined as computation per energy, has not scaled as well as expected, and predict an energy efficiency wall with marginal returns using classical digital techniques. Sarpeshkar [1998] (Figure 3) shows that analog computation is more efficient than digital (in MOS technology) when the output signal-to-noise ratio is below a certain value, which is the case for many applications, e.g. cognitive-style computation using neural networks of limited precision.

It is also possible to merge analog and digital computing with so-called mixed-signal processing. Sarpeshkar [1998] describes such hybrid approaches as the "best of both worlds" and proposes a distributed computing architecture with continuous-signal, continuous-time analog computation with discrete-signal communication and restoration blocks.

3.1.1 Physical computing

Recently, interest in analog computing has resurfaced as physical computing [Jaeger, 2021, Hasler and Black, 2021], an area that aims to exploit physical nanoscale phenomena of many sorts for computation, thereby leading to more energy efficient computation. Much drive has come from artificial intelligence, where cognitive-style computation using neural networks present a natural opportunity for analog computers as accelerators [Musisi-Nkambwe et al., 2021]. Two fields connected to physical computing deserve special mention in this context: natural computing, which was already mentioned in the introduction, aims to take inspiration from nature [de Castro, 2007], and neuromorphic computing aims to take inspiration from the brain in particular [Indiveri, 2021].

Quantum computation may also be seen as a model of analog computation, allowing more powerful computation by accessing complex-values amplitudes. The computational power of quantum computers are well-explored in theory. While quantum computers do not lead to an exponential speedup for all problems, they do lead to a significant speedup for a certain class of problems. It is interesting to note that quantum computing has also been exploited and demonstrated in classical analog hardware, using a paradigm called Hilbert Space Computing [Kish, 2003]. As such, it seems like a key benefit of quantum computers lies in their access to analog values. Note that this has been proposed only as an experimental platform to test quantum algorithms without the practical difficulties inherent with current quantum computers - the argument is not that quantum computers can be made obsolete through analog computers, see Kish [2003] for details.

Analog and physical computation has also been widely explored through the paradigm of reservoir computing, which originates in the reservoir of the Echo State Network [Jaeger, 2001] and the Liquid State Machine [Maass et al., 2002]. Tanaka et al. [2019] give an overview of recent applications of reservoir computing in physical systems.

A key problem in physical computation is the fundamental difference between physical models and computing models. Horsman et al. [2014] argue that the physical and computational models of devices must correspond exactly in order to compute and they propose their framework of "commuting diagrams" to this end. Marković et al. [2020] argue that including more physics in algorithms and materials for neuromorphic computing can have a major beneficial impact on the development of new devices and the use of existing ones. This alignment between physics and computing is a common theme in this report, and investigated further in Section 3.5.

3.1.2 Critical gaps

Evidently, analog and mixed-signal computation is suffering from the lack of theoretical unification similar in breadth, depth and precision as the theory of Turing computability provides for digital computation. Such a unification of analog computing approaches and an accompanying formalism to describe analog computation is missing and highly desirable, see Jaeger [2021], Stepney and Hickinbotham [2018]. Such a formalism would serve as a bridge to tie together models of computation with physical models of the underlying materials. Hasler and Black [2021] advocate the development of an "Analog Turing Machine" as a fundamental model of analog computation as well as the development of an "analog complexity theory" which would allow a clear comparison of time and space complexity (in the theoretical computer science sense) between digital and analog. For a detailed discussion on such a unifying theoretical basis for a general theory of computation that includes both digital and analog computation, see Section 3.8.

Aside from a general theory of analog computation, another critical gap is found in the way analog computers are currently programmed. That is, they are not programmed at all, in the conventional sense of building programs top-down, from the high-level conceptual ideas towards the low-level hardware implementation. Instead, programming/designing analog computers is seen as a bottom-up "art", rather than an engineering problem. One way forward may be to compile a library of "analog standard cells", i.e. a set of computational primitives that are used as standardized building blocks in analog circuit design [Hasler, 2020b]. A significant step forward is given by the recent development of FPAAs and the accompanying toolkit [Hasler, 2020a]. The problems with programming unconventional computing devices, including analog ones, is further explored in Section 3.5.

3.2 Timescales

Timescales in unconventional hardware are already under investigation in the European project called MemScales in the research group of ESR1 and ESR2. Both ESRs will be involved and benefit from these investigations. This section briefly illustrates the importance of timescales in the brain and outlines some of the critical gaps that are identified and analyzed in more detail in a report from the MemScales project [Jaeger et al., 2021].

Biological brains exhibit dynamical processes on many timescales, and different processes affect different physical elements in brains in different ways. This leads to a entangled maze of dynamical phenomena in which it is hard to not get lost. A coarse orientation is provided by the conceptual sequence *inference* → *adaptation* → *learning* → *development* → *evolution*. These terms denote bundles of dynamical phenomena which manifest themselves on increasingly long timescales. None of them has a precise definition, but all of them are used in computational science, cognitive science and neural-networks based machine learning with more or less similar semantic intuitions:

- *Inference* processes refer to the fast operations of sensor processing, motor control and “reasoning” which do not essentially rely on structural or parametric changes of the neural processing system, using the system “as is”. In machine learning one often speaks of “inference” when a ready-trained neural network (or other ML model) is used to process task instances for which it has been trained.
- *Adaptation* is a particularly broad and vague concept. A common denominator of its uses seems to be that adaptation works on slower timescales than inference, and is in principle reversible. It often describes processes when a cognitive / neural system re-calibrates, or re-focusses itself when the environmental context of operation changes. In formal models, adaptation processes often are expressed through changes of control parameters in neural subsystems, induced by “top down” regulatory mechanisms or subsystem-inherent homeostatic self-stabilization mechanisms. While this seems to us the most common intuition connected to the word “adaptation”, it is also used in a much more generalized way to denote any change of any sort of system (from a single synapse to a biological population in an ecological niche) that improves the system’s “performance” or “viability”. In those cases, adaptation is not usually reversible.
- *Learning* refers to processes which expand the functionality of a cognitive system on the basis of experience. Learning processes are usually considered irreversible (“forgetting” are processes in their own right which cannot be understood as time-reversed learning). Learning processes are commonly associated with irreversible changes in system parameters — in neural networks typically “synaptic weights”. Structural changes (like deletion

of neural connections or adding neurons to a network) may also result from learning, though this aspect seems less central to the "learning" concept than mere parametric change.

- *Development* is a notion which is much more common in the cognitive and neurosciences than in machine learning. It refers to the life-long history of an individual, autonomous cognitive system (animal, human, or generalized "agent"). The development history is often segmented into life periods like pre-natal development, stages of infancy, youth, adolescence, old age which are in turn associated with specific structure-changing processes in the agent's brain. We foresee that developmental change will also become an important theme in neuromorphic computing systems based on non-digital hardware which cannot be "programmed" and whose physical substrate is subject to aging.
- *Evolution* is the longest-timescale item in our list of process categories. It describes the adaptive change of entire populations, across generations, to fit a (possibly changing) environmental "niche".

Mathematical models of cognitive systems describe inference and adaptations processes (typically) through changes in the values of system variables (dynamical state variables and/or control parameters). The system equations do not structurally change. In contrast, models of development and evolutionary processes must account for structural changes in the system equations. Formal tools for *effecting* and *simulating* structural change in system equations exist in the form of genetic / evolutionary algorithms. However, mathematical theories that can be used to *characterize* and *analyse* structural change *in qualitative terms* are scarce, heuristic, and generally still under-developed. We find that certain tools in mathematical logic ("non-monotonic logic") come closest. However, these formalisms are not connected yet to dynamical systems modeling.

3.2.1 Critical gaps

More complex cognitive processing needs more timescales. A task's cognitive complexity seems closely linked to the spectrum of memory timescales needed for it. This indicates that for a systematic development of neuromorphic technologies it is helpful to work out a complexity hierarchy of task types and initially not "reach for the stars" but concentrate on tasks of modest complexity that require to integrate information across a few timescales only (or even a single one). This is illustrated, for example, in the work by He et al. [2019] (described in more detail in Section 3.5) that was done in the research group of ESR1, ESR2, and PI Jaeger, where the timescale of the learning task was slower than the timescale of the processing units in the unconventional hardware system.

Delays in unconventional computing. Signal travel delays in unclocked analog neuromorphic microchips become a problem when delay times are not well separated from the fastest timescales demanded by the processing task (in which case delays can be ignored). For high-frequency online processing tasks (for instance in future neuromorphic low-energy communication nodes), an explicit modeling and algorithmic compensation for physical delays is needed. For multi-timescale offline tasks, an upper limit for task throughput rates is given by the necessity to separate physical delays from the fastest task timescale. We note that delays are no mathematical or algorithmical problem in digital computing as long as physical on-chip delay times are much shorter than clock cycle times.
If one would find a way to physically realize tapped delay lines (by traveling waves or solitons, maybe skyrmionic?), multiple timescale dynamics (with longest scale given by longest signal

travel time on the delay line) might become explicitly designable. In order to achieve this, we need mathematics and algorithms for embedding tapped delay lines in analog computing architectures.

Life history timescale. If the motto of brain-like computing is taken seriously, the “lifespan” timescale of an individual hardware system becomes relevant. Digital microchips don’t age and don’t have an individuation history: if they start processing 0’s and 1’s differently from when they were sold, they are called “broken” and are replaced by an unbroken identical twin. Analog neuromorphic microchips will likely be individual from the moment when they leave the fab (due to device mismatch); they will often exhibit slow parameter drift and physical aging; and they cannot be “programmed” in the traditional sense but will likely have to be trained. This will lead to individual lifelong learning and adaptation histories.
What is needed are novel mathematical tools to describe qualitative change and continual/lifelong learning schemes (algorithms and training schemes) that are appropriate for physically aging systems, which in particular will require a collaboration between learning and homeostatic self-stabilization mechanisms.

3.3 Stochasticity

Looking at the theory of digital computation and its origins in logic, it is clear that the operation of a digital computer heavily depends on the reliable operation of all of its components. A single bit flip in any of the computers’ registers can lead to its failure. Because the standard theory of digital computation only works with perfect precision, this high precision requirement has been put on hardware engineers who have to ensure that the memory and operation of every single component is reliable.

This stands in stark contrast to the brain, whose neurons are unreliable and noisy [Rolls and Deco, 2010], with neural spike output changing from trial to trial in identical experiments [Maass, 2014]. Yet, somehow, the brain is able to generate reliable behavior from unreliable components. This has fascinated the research community since the early days of computers and led to models of computing with probabilistic logic [von Neumann, 1956] and the inception of stochastic computing, where information is represented and processed in probability distributions [Alaghi and Hayes, 2013]. As computing technology moves towards circuits with increased uncertainty in their behavior (cf. Section 3.4), we need to better understand the use of probabilities in computation.

At this point it may be helpful to point out the difference between stochastic computing and randomized (or stochastic) algorithms. The former is a model of computation that differs from digital computation in interesting ways. The latter is a standard approach in digital computing to design algorithms that make use of a (pseudo-)random number generator in order to achieve good expected performance, where the expectation is taken over the distribution of possible inputs on which the algorithm is run. Such randomized algorithms give rise to complexity classes like the bounded-error probabilistic polynomial time class (**BPP**) which includes all decision problems that can be solved by a probabilistic Turing machine (which runs such randomized algorithms) in polynomial time with an error probability no greater than $1/3$ for all problem instances. This is comparable to a computation in quantum computing, where a quantum circuit is inherently probabilistic: the final measurement, or readout, from a quantum algorithm "collapses" each qubit state, which might be in a superposition of 0 and 1, into a single bit of either 0 or 1 with a probability that depends on the quantum state. This makes randomized algorithms a natural choice for analyzing quantum computation, and for comparing classical and quantum computers. The class of problems that can be solved by a quantum circuit in

polynomial time with an error probability no greater than $1/3$ is **BQP**. It is known that **BPP** is contained in **BQP**, that is, quantum circuits are at least as powerful as probabilistic Turing machines, but it has not been shown whether **BPP** = **BQP** [Aaronson, 2013]. We have evidence that gives us reason to believe that indeed **BPP** = **BQP**, for example the existence of a quantum factoring algorithm [Shor, 1994] but no classical one. Evidently, randomized algorithms are important and immensely useful in practice, but as they are immersed in the prevailing digital paradigm of computing as discrete symbol manipulation processes, we will not further discuss randomized algorithms in this form, and instead turn to more general aspects of stochastic computation.

The following two paragraphs lean heavily on the review paper by PI Jaeger [Jaeger, 2021]. In digital computation, symbols are primary objects which are manipulated according to rules of logical inference to yield a computation. In stochastic models of computation, probability distributions can be considered as the primary objects, analog to symbols in digital computing [Jaeger, 2021]. These probability distributions need not be represented as closed formulas, but can also be represented using stochastic sampling (which has natural applications in optimization problems) for example through Monte Carlo [Neal, 1993] or particle swarm methods [Dellaert et al., 1999]. Such sampling-based representations of probability distributions can represent distributions which can not be described analytically. The role of logical inference can then be replaced by probabilistic inference. According to the Bayesian view on probability, probability theory is seen as an extension of logic [Jaynes, 2003], which further extends the analogy between logic as a standard formalism for digital computing and probability theory as a formalism for stochastic computing.

Probabilistic inference is widely used as a framework in cognitive science to describe predictions about the outcome of an agents' actions (see Bayesian brain [Tenenbaum et al., 2006], predictive brain [Clark, 2013], free energy model [Kiebel et al., 2008]). In mathematics and machine learning we find various computational frameworks that formalize some aspects of the predictive brain hypothesis [Jaeger, 2021], for example the above-mentioned computation with probabilistic logic [von Neumann, 1956], observable operator models and predictive state representations of probability distributions [Jaeger, 2000, Littman et al., 2001], Boltzmann machines [Ackley et al., 1985], or the neural engineering framework [Eliasmith et al., 2012b].

Stochasticity and noise does not seem to be a problem in the brain, but instead might be considered an essential feature [Kitajo et al., 2003]. There are many sources of stochasticity in the brain, such as unreliable transmission in synapses or the stochastic opening and closing of membrane channels [Maass, 2014]. In theoretical neuroscience, the stochasticity of neural spike events has been interpreted as a sampling operation, both in time and in space, through which information about underlying probability distributions is represented [Buesing et al., 2011, Pecevski and Maass, 2011]. In another view, the precise firing time patterns can also be considered to carry information [Thorpe et al., 2001, Izhikevich, 2006, Deneve, 2008], instead of representing an underlying distribution through samples.

Such sampling-based processing has also been used productively in non-digital physical computing systems, for example in DNA computing [van Noort et al., 2002] for solving optimization and search problems, and with wider application range in analog spiking neuromorphic hardware [Indiveri et al., 2011, Haessig et al., 2018, Moradi et al., 2018, Neckar et al., 2019, He et al., 2019]. These experimental results demonstrate the usefulness of exploiting stochasticity in unconventional computing systems, as many of these systems are inherently noisy [Zhou et al., 2020, Semenova et al., 2019]. Even in digital computers, decreasing device sizes has lead to manufacturing errors that result in increased variability in physical circuit characteristics. Thus, the development of stochastic computing may be a deciding factor for taking full advan-

tage of future computing hardware, while simultaneously allowing lower precision requirements from hardware engineers.

In order to create computing systems with characteristics similar to those of the biological neuronal networks, e.g power efficiency, learning capability, and robustness, it is necessary to leave behind the concept of artificial neural networks that use precise weights to compute [Olin-Ammentorp et al., 2021]. The noise tolerance and robustness of the brain has already served as inspiration to create neuromorphic computing based on single-electron circuits where thermal noise is used to carry out neural computation [Oya et al., 2007].

Moreover, Spiking Neural Networks (SNNs) have been implemented using analogue hardware where noise improves the resilience to synaptic inaccuracies [Olin-Ammentorp et al., 2021]. Finally, noise has also been used as a tool to increase the convergence speed of SNNs when recalling information stored in complex probability distributions. It has also been demonstrated to be useful when creating heuristic solutions for hard computational problems, and it is a key element in self-organization and learning in SNNs [Maass, 2014].

3.3.1 Critical gaps

While stochasticity obviously plays important roles in neural and physical systems, both beneficial and detrimental, we are still lacking a full understanding of the mechanisms by which noise impacts information processing in all its different roles. Practical theory guidance for hardware system engineers, architecture designers, and programmers or users is rarely available.

A promising start to better understand the nature of stochastic computation are studies similar to the one by Semenova et al. [2021] which identify what kind of noise can be tolerated in different models of computation. A possible next step is to go beyond noise tolerance, and identify the kind of noise that can actually benefit the computation, e.g. by acting as a regularizer in machine learning applications.

3.4 Robustness

A key difference between digital and natural computation is their robustness. While a single bit flip can lead to the complete failure of a computer program, the brain works reliably despite the ongoing death and re-generation of neurons. Where human (animal) vision is robust to a wide variety of changes in lighting conditions, programmed computer vision programs struggle and exhibit brittleness in unforeseen situations. Our current theories of digital computation work only because this computation unfolds in a hardware that is perfectly known in advance - thanks to the precision requirements that digital hardware designers comply with.

Such precision engineering does not exist in natural systems, which use "robust adaptive procedures instead of optimizing strategies that work well only when finely tuned to precisely known environment" [Simon and Laird, 2019]. We are only beginning to understand the mechanisms that provide for such physical and functional robustness that natural systems exhibit. Kitano [2004] provides an architectural explanatory framework for the robustness of biological systems and a first step to a broader understanding which would then also enable the translation of these findings into engineering practices. This is not merely a problem for unconventional computing, however. With increasing miniaturization and shrinking dimensions of circuit components, even conventional digital VLSI circuits are subject to increased levels of variation and noise [Constantinescu, 2003] that need to be faced if this miniaturization is to continue.

Moreover, robustness is not restricted to the physical implementation of computing devices but also extends to the software and programming of these devices. Reliability, security, and dependability are core concepts and any software engineer is trained in methods to increase or guarantee the reliability of software systems.

We consider a system to be robust if it is tolerant to faults, where faults may be introduced from several different fronts. In digital hardware, the faults may be broadly classified as static (physical defects that originate in the design or manufacturing phase) or dynamic (e.g. latch up, a condition in which a CMOS gate gets stuck at a fixed value) [Woods and Lightbody, 2008]. Other sources of faults include: data noise (in the input), internal/thermal noise, physical damage, programming errors (bugs), and modeling errors. Under robustness, we understand an overall framework for dealing with faults. Some of the methods that have been identified in robust systems are: redundancy, compensation, diversity, mechanical robustness, granularity, restoration (cognitive), cohesion and coupling, recovery, self-healing, self-repair (physical).

In this section, we give a brief outline of some of the work on making unconventional computing hardware robust. Much of this section concerns the internal noise of these devices because this is closely connected to the project of ESR7, who contributed to this section, as well as the mismatch and limited observability which is a key issue for ESR1, who is working with the Dynap-se analog neuromorphic hardware [Moradi et al., 2018] and also contributed to this section. Finally, a more general discussion on theoretical proposals to deal with imprecision and uncertainty from an algorithmic and knowledge-representation view is given.

Several implementations of analog neurons have been proposed using physical components as for example lasers, memristors or spin-torque oscillators, among others. Moreover, optical and electronic concepts for parallel networks have demonstrated that it is feasible to achieve fully parallel networking. In spite of the characteristics that make such systems so promising, they suffer from intrinsic noise generated by their analog components [Semenova et al., 2019, Zhou et al., 2020].

Electronic implementation of ANNs is the most mature technology and a common approach in this context is in-memory computing, which typically uses non-volatile memory (NVM) crossbar arrays to encode the network weights as analogue values. Here, noise is a consequence of the limited precision of memristive NVM cells as well as of the fact that the stored values are prone to change over time between read and write operations [Zhou et al., 2020]. The noise on this type of hardware can be modeled by an additive zero-mean Gaussian noise that is applied on the parameters. The variance of this Gaussian noise is a function of the effective number of bits of the output of an analogue computation or proportional to the range of values that the device can represent [Rekhi et al., 2019].

In neural network with spiking neuron models (SNNs), noise has been shown to play a beneficial role, acting as a regularizer that increases the network's robustness [Olin-Ammentorp et al., 2021]. In order to take advantage of this positive effect of noise, Stromatias et al. [2015] propose an adapted training mechanism that improves the network's performance by over 30%. It is important however, to take into account that software simulations of hardware noise is not always accurate. Petrovici et al. [2017] characterize several of these distortive effects and show how a hierarchical topology shapes the information flow in a way that makes them largely resilient to these effects.

In analog hardware built using photonics, only limited efforts have been made to characterize the noise present. A very recent study can be found in Semenova et al. [2021]. When working with this type of hardware, noise is simulated by perturbing the output of the neurons using a noise operator that has an additive and a multiplicative component. Moreover, these compo-

nents are composed of correlated and uncorrelated noise. The first one is a consequence of the components' internal processes, in this case the activity of the neurons. The second one is generated in central components that influence the overall circuit's state, for example the voltage source that feeds the circuit or its temperature [Semenova et al., 2019].

In many unconventional computing devices, device mismatch is drastically higher than in conventional electronics and leads to problems that do not need to be addressed in digital computing. In organic materials, current technology does not enable us to produce multiple identical devices; instead, every device is unique and behaves differently from others. This makes apparently identical circuits respond in different ways to the same input [Weis et al., 2020, Andraud and Verhelst, 2018]. A source of these mismatches are the variations and defects found in the manufacturing process. These variations are known as static because they are constant after affecting the circuit only once. Nevertheless, these are not the only ones, as aging or stress lead to variations with a slow and varying impact on the circuit over time, and because of this, they are known as quasi-static. Finally, it is possible to find variations that affect the circuit in a rapid way and therefore are known as dynamic. These are a consequence of temperature and voltage changes, among others [Andraud and Verhelst, 2018].

For example, in the organic electrochemical networks by Cucchi et al. [2021], the dendritic trees are grown separately on every device and yield distinct connectivity and functionality which makes programming impossible. A well-established solution in material computing is to treat this network as a reservoir and train only the readout layer. The authors are able to report good performance, yet to better exploit the material's computational power, a more refined way to handle this device mismatch may be desirable. This device mismatch is not limited to organic materials, however, but has also been reported across the literature on unconventional computing, for example in the nano-electronic dopant network processing unit [Bose et al., 2015, Chen et al., 2020] where each unit is treated separately, essentially as a blackbox, and trained using an evolutionary optimization procedure. Even in analog electronic hardware, device mismatch prevents users from directly copying functionality across different chips as one can do with a program on digital computers. Instead, novel mathematical methods need to be developed in order to transfer algorithms between devices [He et al., 2019].

Strategies known as self-calibration (compensate static variations), self-healing (compensate static and quasi-static variations) and self-adaptation (compensate static, quasi-static and dynamic variations) are promising approaches to counteract this problem by letting circuit compensate for these variations itself. To achieve this, the circuit must have sensors to monitor the existence of variations, tuning knobs to adjust the circuit's performances and an algorithm that makes the decision of which tuning knob must be adjusted [Andraud and Verhelst, 2018].

The term *soft computing* has been coined by Zadeh [1994] to denote a set of algorithms that exploit their tolerance for imprecision and uncertainty. While traditional (hard) computing needs precision, certainty and rigor, soft computing uses modes of reasoning that are approximate and robust, in the sense that is introduced in this section. Since the inception of soft computing, neural networks were a prominent example of this approach. Another tool is fuzzy logic, a form of many-valued logic (in fact, infinite-valued logic) in which a truth value is represented by a real number $x \in [0, 1]$. Fuzzy logic has found applications in control theory and artificial intelligence because of its ability to reason with vagueness. As such, it differs from probability theory and probabilistic reasoning, which deals with inference made under uncertainty. Valiant [2013] calls such approximate algorithms, when applied to learning tasks, *ecorithms* (cf. Section 3.5). As opposed to algorithms, which exist only in the computer, ecorithms constantly interact with the real-world environment, and learn from this interaction. As such, they include machine learning algorithms as well as evolutionary algorithms. The defining feature of ecorithms is

their interactiveness and their robustness to a wide range of possible environments.

Until now, we have interpreted robustness in light of hardware variations and noise. Another domain in which robustness is a well-known issue is that of reasoning and knowledge databases. In the traditional approach (see Section 3.6), knowledge was programmed into a database by human programmers, and a formal logical framework was used to reason about that knowledge and draw inferences [Lenat, 1995]. These systems have found successful applications in many domains, whereas in other domains they often fail when used in situations not foreseen by the human programmer. This failure is often called *brittleness*. In order to reason with imprecise concepts, from fuzzy logic or generated by learning algorithms from real-world interactions, various approaches have been proposed. Robust logic [Valiant, 2000] is a way to overcome the brittleness of traditional real-world reasoning programs that are based on mathematical logic. Robust logic provides a sound and efficient proof procedure for reasoning and is more robust because its rules are learnable.

3.4.1 Critical gaps

Since much inspiration for building robust systems comes from nature and, particularly, from biology, a better understanding of how biological systems achieve robustness is desirable. Such a research agenda is outlined and exemplified by Kitano [2004]. Of course, this must also be translated into a set of practical guidelines and principles that enable hardware (and software) engineers to design robust computing systems.

3.5 Programming

Regardless of the substrate underlying the computation, our ability to exploit unconventional computing is limited by the methods of programming these substrates. Similar to the concept of computing itself, the notion of programming will need to be re-investigated from the perspective of unconventional computing hardware. Some issues of programming such unconventional devices have already been investigated in conventional programming, such as the complexity of interactive large-scale software systems, while others are novel because of their irrelevance in digital computing, such as noise, limited read/write access, device mismatch, always-on capability, or multiple timescale dynamics.

Research in digital computing has brought us increasingly sophisticated programming tools that facilitate the development and maintenance of complex software systems. We need to reach a comparable level of sophistication in the software tools for unconventional computing if we wish to build systems of comparable complexity. Luckily, we do not have to start from scratch; decades of research into programming languages and software engineering are ready to be adapted for unconventional hardware.

This section will first give a brief overview of programming research, identify major trends in programming digital computers and take a closer look at the programming problem itself. We proceed to clear up some terminology that is frequently used in programming unconventional computers. We then survey some existing approaches and paradigms for programming unconventional computers. Finally, we point to remaining challenges and gaps in the research on programming unconventional computers.

3.5.1 Programming trends

In the early days of computing, programming was done by manually configuring the computer's internal wiring. The representation of a program was very close to the underlying hardware, translating Boolean circuits into electric diagrams and rewiring the computer to represent those. Stored-program computers eliminated the need to mechanically rearrange wires, instead enabling programmers to load the program into memory and execute it directly. However, before the PC revolution, computers were operated in batch mode so that programmers would submit their programs (e.g. as staples of punched cards) to the computer operator, who executed these programs in a FIFO (first-in, first-out) manner. It was common to have to wait a day to get the results from running simple (by today's standards) programs.

One of the first major trends in programming was to tighten the feedback loop between the programmer and the computer. This started with simple end-user interfaces that could submit jobs directly to a powerful computer, but only really took off when personal computers became powerful enough to run most programs. The cost of making (simple) mistakes, e.g. misspelling a variable name, has decreased dramatically. While it would have cost a programmer in the 1980s a whole day, today compilers (or even automatic syntax checkers embedded in the development environment) will detect the mistake in a fraction of a second. On the upside, this has enabled programmers to write more complex software faster, while on the downside it has arguably decreased code quality if programmers can write and execute faster than they can think.

Another major breakthrough was the abstraction of low-level hardware details from the programmer. The history of programming languages can be seen as that of continuously raising the level of abstraction at which the programmer works. From machine code (1st generation) to assembly code (2nd generation) to high-level programming languages (3rd generation), the level of abstraction has increased further. Because this leads to a reduction in code needed to specify certain (high-level) operations, such higher programming languages can also be seen as more "automated". In fact, raising the level of abstraction in programming languages has often been euphemized as "automated programming" [Parnas, 1985].

A more recent trend towards automation in programming has emerged from research on artificial intelligence, where a program is not fully specified by the programmer, but instead the program (or part of the program) is *learned* by the computer. This is often done with a supervised machine learning setup, where labeled training data is used to infer a program. In such a setup, recently termed Software 2.0 [Karpathy, 2017], much of the programming work shifts to collecting, curating, augmenting and maintaining a good dataset. More will be said about these techniques of *differentiable programming* (a superset of deep learning) and *program induction* later.

The industry has moved away from pure programming towards software engineering, which includes programming as a sub-task, but goes significantly further. The key distinction here is that software systems are complex, interactive systems. In computer science research, programs are often treated as equivalent to algorithms, in the sense that they are free of side effects. In practice, however, algorithms are only building blocks that make up a complex software system full of side effects in the form of interactions with the environment (either the user or other software systems). In effect, this is the distinction in functional programming between pure and impure functions. Much research has been focused on the pure parts of software, whereas in practice most software is impure. This trend towards more complex, interactive software systems has fueled the software complexity crisis and is the source of much technical debt in the industry.

3.5.2 Views on programming

Programming is the principled way of going from an intention (a desired program or behavior) to a system (usually a computer) that implements and satisfies this intention. In this broad conception of the term, it is perfectly reasonable to speak of programming a thermostat, or even (aspects of) behavior of an animal (conditioning) or another human being (social engineering). Here, we are concerned with programming unconventional computers, so we will restrict ourselves to programming computers (in a very general sense of the word, going beyond Turing).

Figure 1 shows a diagrammatic representation of the programming process. The programming process starts with an *intention*, which can be thought of as the desired behavior, the desired function, or the problem which ought to be solved. This intention can be formalized into a *specification* in some formal language, which in turn can be used to automatically generate a *program* that satisfies this specification. This can be done through synthesis programs and is widely used, e.g. in FPGAs. Instead of formalizing the intention, one can also come up with an informal *idea* for a program. This idea comes one step closer to the implemented *program*, which can be obtained by simply coding (and thereby formalizing) the idea in some programming language. There is also a third way which does not require the formalization of the problem at all. Natural Language Programming (NLP) is an approach in which a computer can generate (and execute) a desired program from natural language input. This is exemplified in voice assistants like Siri, which take natural language input and act on the information that is given. Upon saying "Hey Siri, set an alarm for 8AM tomorrow", the device will execute an alarm-setting program. In this case, the alarm-setting program was likely pre-programmed by a human programmer as a routine that takes a single variable (the time for which to set the alarm), but this can be further extended to work for more generic programs. Once the *program* has been generated (regardless of how it was generated), it can be compiled into different programming languages. Usually, the program is first generated in a high-level programming language and then compiled "down" into a hardware-compatible language which is then executed. The execution of the program then yields some *behavior*. Sometimes (or most times, in the case of human programmers), the program will not yield the desired behavior and needs to be debugged - the behavior is observed and it is determined in what way the behavior deviates from the desired behavior, and finally the program itself is manipulated in order to decrease the difference between expected and desired behavior. This can also be done automatically, e.g. through reinforcement learning, if the difference between expected and desired behavior is formalized as an error function.

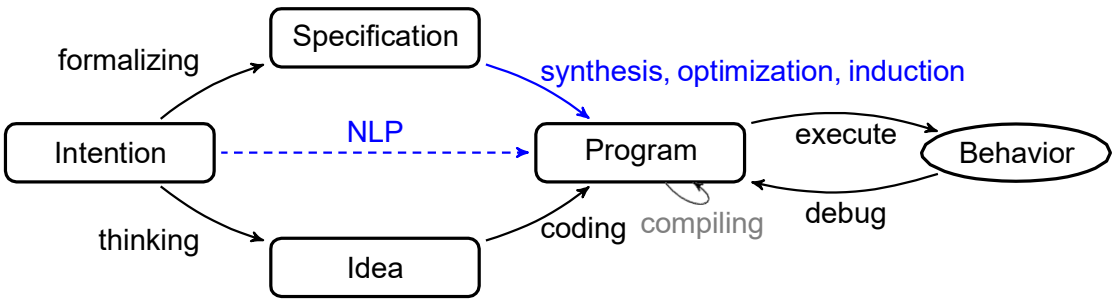

Figure 1: Diagram for the process of programming a computer. Adapted and extended from Grünert [2017]. Inspiration (and some terms) taken from Primiero [2016]. Edges highlighted in blue reflect automated processes which themselves require the execution of a computer program to be realized.

The diagram further shows that programming involves multiple problems that can be treated more or less in isolation from each other. First, the *specification problem* describes the difficulty to specify the program in sufficient detail to the computer. In more complex software systems that are composed of multiple interacting programs, this can be more naturally thought of as a *design problem*. Usually, a computer will require a higher level of detail than would be required for a human being to do the same task. This formalization of the intention, either into a formal specification or into a formal program, is the first big challenge when programming a computer.

It is important to note that the design process for complex software systems goes beyond designing algorithms that are implemented. It also includes the design of the software architecture, infrastructure, network, databases, and more. The process is much more akin to designing, say, a building, than to communicating a particular task or behavior. The design is often an iterative process, perhaps even a self-organized evolutionary process.

Once the program is specified in sufficient detail, the *communication problem* concerns the communication of the program to the computer in some language. Much of computer science research is focused on finding efficient programming languages to communicate the expected behavior from humans to computers. Most programming languages are formal languages which can be compiled into machine code and therefore directly interpreted by the computer - this translates into significant work for the human programmer who has to learn a new language in order to write programs. The burden of the communication therefore lies with the human programmer - the machine simply executes what it is being told, and cannot be at fault for misbehaving. An alternative approach that has only recently gained traction is that of natural language programming. The idea is to program the computer in our own native language, rather than in the computer's native language. Of course, the burden of translating natural language into a formal language that is interpretable by the computer is once again on the human side, but with current machine learning techniques, this can be done without manually programming the whole procedure.

The final step in programming a computer is the *control problem*, i.e. to make sure that the computer actually follows the program that has been communicated to it. This task can be very simple, if the hardware is sufficiently reliable and the program does not contain any bugs, but it can also be quite challenging, for example when working with unreliable analog hardware.

A common naive perception of conventional programming, often entertained by students who have finished their first semester of programming, is that programming is only a communication problem. This may hold true for programming simple functions in a classroom environment, where the programs are simple enough to be specified directly and the control problem is abstracted away. In contrast, when programming real computing systems (especially unconventional ones), all three problems become important.

3.5.3 A taxonomy of computer programs

Before continuing with the concept(s) of programming, it is necessary to first define what it is that we call a computer program in the first place. We begin by the original definition of a computer program in terms of the Turing machine and gradually add different ingredients in order to reach the complex software systems that we work with today.

A Turing machine [Turing, 1936] M can be seen as computing a function on the integers $M : N \rightarrow N$. The set of all functions that can be computed by a Turing machine is called the set of

all computable functions. Such a machine can be seen as a "program", where the underlying transition function defines how the "program" operates and thereby determines its input-output behavior. However, the machine can also be seen as a general-purpose computer which can execute any other (computable) function. In this view, the machine (often denoted by U for *universal machine*) will take as input an encoding $< M >$ of the machine M which it should simulate, as well as an encoding $< x >$ of the input x to that simulated machine.

This definition of a machine corresponds to what Turing [1936] calls the a-machine (for "automatic machine"), whose motion is completely determined by its configuration. We will call the program that such a machine can implement an *algorithm*. An algorithm is therefore a computable function on the integers (or, a function on an isomorphism on the set of integers). This corresponds to what are called *pure functions* in functional programming. Such functions are stateless, and do not depend on the environment - their behavior only depends on the input given to them. For example, the function $gcd : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is a pure function, and Euclid's algorithm is an algorithm to compute this function. In contrast, the function $\mathrm{Date.now()}$ is not pure - if we call it at two different times, it will return different values as output, despite getting the same input (which is, no input). For the same reason, random number generators are not pure. As a further example, the function $\mathrm{readline()}$ is not pure because it takes input from the environment (in this case, the user).

Obviously, this does not capture all kinds of programs that we are interested in. Interaction with the environment is essential and without it, we are left with little more than a calculator. Roots of this can be found already in Turing [1936], where he described not only the above-mentioned a-machine, but also the c-machine (for choice machine). In Turing's words:

> When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator.

This is exactly the kind of interaction that we want to model. A simple way to include this environment into our models is to include the environment as an additional input and output parameter. Such an *interactive algorithm* would then be modeled as a function $f : \mathbb{N} \times E \to \mathbb{N} \times E$ where E is the set of all possible states in which the environment can be. This is also the way that interaction is dealt with in functional programming languages, where such functions are called *impure*. Unfortunately, this approach is quite limited and is not enough to model all aspects of interaction in programs.

Evidently, interaction is needed to accurately describe today's computers and programs. What is described by pure functions, or a-machines, could be called sequential computing. However, real-world computing is not sequential, instead we have programs communicating with humans and with each other - in different threads on the same computer and across the world through the internet. Moreover, computation in unconventional substrates will certainly not be without interactions with the environment - it may not even be possible to clearly separate the environment from the computing system in the future (see the argument by Brooks [1991] below). In his Turing Award lecture, Milner [1993] presented models of interaction as complementary to the closed-box computation of Turing machines, and argues that a new conceptual framework is needed to move from sequential computing to interactive and concurrent computing. While working on concurrent computing, Milner developed such a framework for interaction in the form of a calculus for communicating systems (CCS) and, later, the π-calculus. Milner [2006] later argued that computer science has now grown into informatics, which primarily deals with interactive systems, rather than with sequential computation, and requires a new logic of interaction. In similar spirit, Wegner [1997] argues that interaction is strictly more powerful than non-interactive algorithms, because Turing machines cannot handle the passage of time nor

can they model the arrival of events during a computation. Even in the Interactive Turing Machine [van Leeuwen and Wiedermann, 2001], the machine can either process or receive input, but not both at the same time. This is entirely different from event-based, asynchronous, massively parallel computation as we find it in neuromorphic hardware. Wegner and Goldin [2003] argue for a paradigm shift in computer science to move from sequential algorithms to interactive systems that are firmly embedded in their environment. Brooks [1991] makes the argument that algorithms alone are not sufficient to build intelligent system - interaction is also needed:

> Real computational systems are not rational agents that take inputs, compute logically, and produce outputs. . . It is hard to draw the line at what is intelligence and what is environmental interaction. In a sense, it does not really matter which is which, as all intelligent systems must be situated in some world or other if they are to be useful entities.

Similar to the interactiveness that is needed to expand pure algorithms to interactive algorithms, Valiant [2013] coins the term *ecorithm* to describe algorithms that interact with the environment and adapt to it in a way that modifies their behavior in ways that may not be anticipated by the human programmer. Such ecorithms are already all around us in the form of learning algorithms or evolutionary algorithms. Such ecorithms get to the core of this blurred line between intelligence and environmental interaction that Brooks [1991] argues for. These are also the kind of algorithms which Karpathy [2017] collectively refers to as "Software 2.0", where much of the programming work shifts to the collecting, cleaning and maintaining of a good dataset (or environment, in a non-supervised learning setting).

Until now, we have only discussed single algorithms; pure algorithms, interactive algorithms, and ecorithms which adapt to the environment. But in reality, all these algorithms work together in a complex system of interacting software (where software refers to the collection of all kinds of algorithms that are implemented and run). Such a software system can consist of millions of lines of code - but that is still not enough to fully characterize the system and its complexity since it can also interact with other software systems, as well as with human and non-human users. Booch [2011] asserts that such complex software systems are among the most complex artefacts ever created. Furthermore, Booch [2011] uses the term "software-intensive systems" to highlight that such systems are not composed solely of software but also include the underlying hardware on which the software is run.

It shall be noted that this discussion about programs is deeply rooted in the paradigm of digital computing, which manifests itself in various ways. Most notably, we have disregarded the physical aspects of the computing system entirely, i.e. what a computer scientist would refer to as "hardware". This separation of a computer into hardware and software is a fiction that is immensely useful in practice - it allowed the separate development of the two disciplines of "hardware engineering", which builds ever more powerful computing devices, and "software engineering", which builds ever more expressive software systems "on top of" the hardware. However, it is important to view this hardware-software as what it is - a pragmatic distinction - without giving it more ontological significance than it has: there is no clear separation between the two domains. A systems programmer who works with machine language will consider much of the circuitry as hardware, whereas an application programmer who works with a high-level programming language will consider machine language and a significant part of the operating system (e.g. memory management, scheduling) itself as hardware. In an unconventional computing setting, where the fiction of a computer that is made up of separate hardware and software cannot be maintained, we will need to come up with new concepts for programming. This will be discussed later on in this section.

To summarize the terminology introduced in this section, we will use *algorithms* interchangeably with pure functions to refer to those functions that are efficiently computable on a Turing machine, which corresponds with pure functions as defined in functional programming. We will use *programs* interchangeably with interactive algorithms or impure functions to refer to functions that allow for side effects through the environment and/or interaction with other programs. We will also use the term *ecorithms* or *learning algorithm* to highlight those interactive algorithms that learn from, or adapt to, their environment. Finally, we will use the term *software system* to refer to collections of programs that work together in a system.

3.5.4 Programming concepts

From Figure 1, it appears that programming is the act of coming up with a program that satisfies some intention *with no regard for how that program is found*. This is a rather general notion of programming which goes beyond the notion that is typically associated with the term.

In common parlance, programming refers to the process in which a *human programmer* designs and implements a program or software system. Job titles have emerged for programmers that work on different types of programs (as introduced in the previous section): algorithm designers typically work with pure functions, programmers typically work on programs (i.e. interactive algorithms), while software engineers work on software systems. More specialized job titles have also emerged, e.g. machine learning engineers that are focused on working with learning algorithms.

However, as shown in Figure 1, programming need not be done by a human programmer (or designer, or engineer). As shown by the blue arrows in the diagram, programming can also be automated in various different ways. In such a setting, it is a computer program that generates the desired program. This has been called automated programming or meta-programming and even machine learning or optimization methods can be considered in this setting.

While there is a clear delineation between computable and uncomputable functions, and a (slightly less) clear delineation between tractable and intractable algorithms, there is no clear delineation between what can be programmed and what cannot. However, it is natural to start our discussion by asking for the ultimate limits of what can and what cannot be programmed.

We begin by restricting the conversation to algorithms, i.e. pure functions, as defined in the previous section. The question is: what algorithms can be programmed? Of course, every computable algorithm is programmable in the trivial sense in that it exists, and therefore can be found, *in principle*. However, the discovery of algorithms is a tedious and difficult process. Books and journals from theoretical computer science have been filled with discoveries of new algorithms that outperform existing ones. Important questions about the very foundation of computer science and mathematics, such as the famous $P \stackrel{?}{=} NP$ problem, would be solved by the discovery of an algorithm that can solve an NP-complete problem in polynomial time. However, despite decades of research, no such algorithm has been found - and, worse yet, nobody has been able to proof the existence (or non-existence) of such an algorithm.

A straightforward procedure would be to simply search the space of all possible algorithms (using some enumeration scheme, similar to the one proposed by Turing [1936]), simulate each one on a universal machine and determine whether or not it solves the given problem, e.g. the NP-complete traveling salesman problem. The problem with this simple procedure, and any other such meta-programming procedure based on exhaustive search, is that it is

not only intractable but formally undecidable because it involves determining whether or not a given algorithm terminates - and this is the famous halting problem which Turing proved to be undecidable. If we restrict the computational universe to only those programs that provably terminate, the search procedure does become computable. The problem is, of course, that we cannot construct the set of all programs that terminate without, again, solving the halting problem.

Accepting the hopelessness of such an exhaustive search over all algorithms, we may turn to the more practical question of what algorithms a human programmer can program. A formal discussion is not possible without a full understanding of the cognitive abilities (and limits) of humans - and while philosophers and cognitive scientists are in the process of working this out, we may not want to hold our breath while waiting for their results. It is clear that a human programmer can only navigate a subset of all possible algorithms. While not a formal proof, this becomes quite evident to whoever has tried to compile and decompile a computer program in some high-level programming language - this results in obfuscated code that is difficult or simply impossible for a human programmer to fully understand. Similarly, attempts to *fully* understand the inner workings of an artificial neural network have failed (though it must be said that they have yielded some very interesting and useful insights). This should serve as an empirical justification for the claim that not every algorithm is effectively programmable by a human programmer. Avižienis [1983] shows that as programs become large and complex, it is inevitable that they be in some measure incorrect. Conrad [1988] further argues that non-programmable systems are more efficient than those that are programmable (by humans), leading to a tradeoff between efficiency and understandability/programmability of algorithms (or systems).

Machine learning has found a different solution to this problem. By giving up the exhaustiveness of the search, we can restrict the search space to only those programs of some fixed structure of which we know that they terminate (after some maximum number of computational steps), and we can attempt to find the algorithm in this search space which best *approximates* the desired algorithm. By using some universal approximator, such as an artificial neural network (ANN), we can define a set of algorithms, i.e. the set of all ANNs of a certain size, and search the space of all algorithms in this set, i.e. by tuning the network weights of the ANNs.

Many nature-inspired algorithms, including those that take inspiration from the brain or human cognition, have been found by nature through evolution, and we have tried to program such algorithms ourselves in the traditional, manual, way. The effort it takes to program such systems has been repeatedly underestimated, and recently the methods of machine learning have surpassed the programming abilities of human programmers in tasks like playing chess, Go, or video games [Silver et al., 2018]).

It may simply be that our current (manual) programming methods and languages are not well-suited to the tasks to which we ascribe some level of intelligence. We shall continue this discussion in Section 3.5.7.

To summarize, we can conclude that there are many different ways to come up with a computer program, and while there may not be a correct way to program, there are certainly more suitable ways to program certain classes of programs. While a low-level programming language is very effective for writing numerical computation routines, it is decidedly less effective for writing a procedure to recognize faces in images. A facial recognition program can certainly be written in assembly language, but it will be more efficient to use Tensorflow to train a neural network to recognize faces using a labeled dataset. It is possible to think about how to setup the rule of a 2D cellular automaton to solve the one-dimensional density classification task with $\rho_c = 0.5$,

but it may be better or easier to find that rule with a genetic algorithm, as shown by Mitchell et al. [1994].

It is worth pointing out that programming need not be purely a software matter. Programming is never done in isolation from the hardware. It is true that conventional computers can be programmed without thinking about the exact hardware that the program will be executed on, but this only shows that conventional computers are programmed with this hardware model implicitly assumed. The hardware/software divide has no ontological significance other than being useful in practice [Moor, 1978]. The Turing-equivalence between hardware and programming language is a luxury that does not necessarily extend to unconventional computers, so we will need to consider programming as co-developing hardware and software. While this does add complexity from the hardware into the programming process, it also enables us to come up with novel programming paradigms and solutions that are not possible or tractable on conventional computers [Hooker, 2020].

In the following section, we will discuss programming concepts in a situation when the separation between hardware and software is less clear.

3.5.5 Blurring the hardware/software separation

Instead of directly rushing into the full diversity of unconventional computing, we may wish to gradually ease into this wilderness. The field-programmable gate array (FPGA) provides a very useful case study for our purposes. As a reconfigurable hardware system, it blurs the line between hardware and software, leading to a re-interpretation of existing concepts as well as entirely new concepts. However, it is still a fully digital device implemented in standard CMOS technology that works with a global clock for synchronous operation (mixed signal or even analog integrations of FPGAs exist, but we will ignore these in the current section). As such, it "shields" us from the complexity of asynchronous, multiple-timescale dynamics, as well as from analog behavior and other physical effects that make it difficult (or impossible) to use the symbolic description that digital circuit blocks allow.

FPGAs contain programmable logic blocks and a hierarchy of interconnects that allow blocks to be wired together. These interconnects are reconfigurable, allowing the FPGA to change its architecture and dataflow. This makes FPGAs a natural choice for parallel computing applications. Usually, memory elements are included in the logic blocks which enable them to perform not only logic gates but also sequential logic (i.e. finite-state machines).

We begin by taking a look at some of the concepts that are involved in the "programming" of an FPGA. Note that this is not meant to be a comprehensive introduction or explanation of FPGAs but instead serves only as a rough outline of the steps involved in working with reconfigurable hardware:

1) **Circuit/hardware design** Of course, the FPGA chip must itself first be designed and manufactured. This happens through the well-developed design process of integrated circuits. At this stage, the scope and limits of the FPGA are fixed, such as the number of logic gates and memory elements, as well as the clock speed and other characteristics.

2) **Configuration design** The FPGA is configured through a design written in a hardware description language (HDL) or as a schematic design. The schematic design is easier to visualize, whereas the HDL makes it easier to work with large designs through the nesting of functionality - similar to the functional abstraction in conventional programming languages. The design abstraction that is used in HDLs is called the register-transfer

level (RTL) and models synchronous digital circuits in terms of the data transfer between hardware registers and the logical operations performed on these signals. This is a level of abstraction higher than the gate-level description.

3) Synthesis The design/configuration of the FPGA is passed through a logic synthesis program (an electronic design automation tool) which yields a netlist (a description of the interconnects on the FPGA). Recently, the level of abstraction has been raised (cf. Section 3.5.1) by so-called high-level synthesis programs, which take a behavioral specification of the system (in a language like C), transcompile this code into a HDL and then synthesize this to a netlist, using a logic synthesis program. This allows designers to work at an algorithmic level, which is higher than the register-transfer level (RTL) used by HDLs.

4) Configuration The actual configuration of the FPGA happens through a so-called place-and-route process, which is usually performed by the FPGA manufacturer's proprietary software. After verifying and validating place and route results, a binary file is generated which is transferred to the FPGA and then used to reconfigure the FPGA.

Steps 2-4 are also often lumped together and called "FPGA programming". However, the difference between (software) programming and FPGA programming should be evident from this description. Much research on FPGAs has focused on figuring out how to make the programming easier and more efficient, inspired by the great simplicity with which digital CPUs can be programmed. The tools that work for conventional programming do not work in situations where the hardware cannot be cleanly abstracted away from the software. Naturally, it can also be argued that the field still needs more time to develop effective methods for programming FPGAs - digital general-purpose computing has had decades to develop such effective programming methods.

3.5.6 Unconventional programming

The present section aims to give a (necessarily incomplete) overview of methods for programming unconventional computers. There is no clear separation between "conventional" and "unconventional" computers. Instead, we perceive computers as existing on a spectrum of (un)conventionality, where on one end we find the CPU and move past the GPU, FPGAs towards more unconventional devices like digital neuromorphic accelerators (e.g. Loihi [Davies et al., 2018]), analog neuromorphic computers (e.g. Dynap-se [Moradi et al., 2018]), photonic reservoir computers [Duport et al., 2012] and even more exotic devices, like the Dopant Processing Unit [Ruiz-Euler et al., 2020] or the Physarum Machine [Adamatzky, 2007]. Naturally, the "more unconventional" computers will require "more unconventional" programming paradigms in order to harness their full computational potential.

The idea of exploiting this inherent computational potential of computers in unconventional substrates has been phrased as a "natural" way of programming [Stepney, 2012], the "programming" of non-programmable systems [Grünert, 2017], adaptive programming [Lawson and Wolpert, 2006], or simply: unconventional programming paradigms [Banâtre et al., 2005].

We identify a list of proposed methods and paradigms for programming:

Human programming As already mentioned, the "standard" approach for programming conventional CPUs is to have a human programmer design the software manually (cf. Section 3.5.4). The exact methods that are used by a human programmer to come up with a program are not clear, but it seems to require a certain level of modularity and composability of the program (in the sense of hierarchical systems, cf. [Simon, 1991]), in order for

a human programmer to understand and reason about the program - though this may be more a reflection of programming language design [Dijkstra, 1968] than of a human programmer's cognitive abilities.

Standard programming paradigms Three different programming paradigms have emerged as "standard" and are taught to every computer science student. The first and arguably most common and dominant paradigm is that of *imperative programming*. In imperative programming, a list of commands is executed sequentially to change the state of the computer in order to achieve some goal. As such, an imperative program can be compared to a recipe which describes *how* something is done, through a sequence of imperatives. In contrast, *declarative programming* describes *what* the program should do, without describing the control flow. Finally, *object-oriented programming* (OOP) has already been mentioned and describes a programming paradigm that is centered on the notion of objects, which contain data (called attributes of the object) and procedures (called methods of the object). This is a form of *structured programming*, which aims to increase the clarity and quality of code and is used as an essential tool to manage the complexity of large-scale software systems.

Parallel programming With the advent of multi-core microprocessors came the need to use these resources simultaneously. This led to the development of parallel programming techniques, in which multiple processes are carried out simultaneously. Some programs are naturally parallelizable, these are called *embarassingly parallel*, whereas others are harder or even impossible to parallelize because of dependencies due to their sequential nature. Amdahl's law [Amdahl, 1967] gives an equation for the possible speedup of a program through parallelization:

$$S_{\text{latency}}(s) = \frac{1}{1 - p + \frac{p}{s}}$$

where S_{latency} is the potential speedup of the latency of the entire task, s is the speedup in latency of the execution of the parallelizable part of the task, and p is the percentage of the execution time of the whole task concerning the parallelizable part of the task before parallelization. Most tasks do not achieve a linear speedup but only a near-linear speedup for small numbers of parallel processing elements, which flattens out for larger numbers of parallel processing elements.

Concurrent programming Concurrent systems describe any system in which the lifetime of multiple computing processes overlap - the computation itself must not necessarily happen at the same instant (as in parallel computing). Moreover, concurrent systems can (but need not) be spread out in space, either on the same chip (different cores), or across different chips, or even across different computers communicating over a network. Concurrent computing introduces many problems that do not exist in parallel computing, and process calculi have been developed to model and reason about the behavior of concurrent systems.

Meta-programming Meta-programming is a technique in which one program (the meta-program) treats other programs as data. In this way, the meta-program is able to "program" another program. An example is *program synthesis*, in which a program is generated from a specification (cf. Figure 1). When such a specification is complete, it is called "deductive" program synthesis, but methods for the synthesis of programs from incomplete specifications (e.g. constraints or input-output examples) have also been developed under the name *inductive programming* (or program induction, inductive program synthesis).

Lake et al. [2015] proposed an inductive programming approach coupled with probabilistic programs as a model for human-level concept learning. According to Gulwani et al. [2017], program synthesis has been considered the holy grail of AI since its inception in the 1950s. Recently, deep learning has also been applied to this task under the name neural program learning [Devlin et al., 2017] or synthesis [Kant, 2018, Sun et al., 2018], urging the development of large-scale dataset to train these data-hungry models [Alet et al., 2021].

Reflective programming Reflective programming is a form of meta-programming in which a program has access to its own source code, both for introspection and for self-modification.

Natural language programming In natural language programming, the computer is programmed through natural language, e.g. English. Such systems are usually assisted by an ontology in the form of a knowledge base.

This approach to move from formal programming languages towards natural language has been criticized by Dijkstra [1979], who argued that natural language introduces unwanted ambiguity while formal languages provide the luxury of making it easier to think clearly and avoid logical fallacies and concludes that "machines to be programmed in our native tongues [...] are as damned difficult to make as they would be to use".

Evidently, computer science has not replaced formal programming languages with natural language programming, but the methods of natural language programming have still found applications in some restricted domains. Wolfram Alpha, a computational engine, accepts natural language queries, converts these into formal programs in their Wolfram language, and then runs this code. Desai et al. [2016] combined natural language programming with program synthesis for a system that is able to produce programming expressions in a specific target language from natural language input.

Evolutionary programming Evolutionary algorithms are population-based optimization algorithms that can be used to generate computer programs. A standard way to set this up in order to find a program that solves some problem is to define a fitness function that is maximized by a program that solves this problem. Ideally, the fitness function should increase gradually for programs that are closer to the correct solution. Then, a population of (random) programs is generated and in each generation, the population is changed to keep only the k best programs and somehow generate new programs from these k best ones, e.g. through mutation (random variation) or cross-over (combination of existing programs).

Evolutionary algorithms have been used to generate rules for a cellular automaton to solve computational problems that are difficult to solve by manually designing a learning rule [Mitchell et al., 1994]. In an unconventional computing setting, evolutionary methods have been used to "program" nanoparticle clusters to perform Boolean functions [Bose et al., 2015].

Probabilistic programming Probabilistic programming reflects the merging of general-purpose programming with probabilistic modeling. A probabilistic program specifies a probabilistic model, e.g. a Bayesian network, and uses a general-purpose inference scheme to solve the problem. Probabilistic programs extend conventional imperative programs with 1) the ability to draw random values from a distribution and 2) the ability to condition values of variables in a program through observations [Gordon et al., 2014].

Kulkarni et al. [2015] designed a probabilistic programming language for vision that can express complex generative vision models. Some of these programs outperform the state-of-the-art in computer vision with programs that are less than 50 lines long.

Machine learning Machine learning has already briefly been described, and while a thorough treatment is out of scope for our purposes, we identify the key idea as learning a machine learning model which represents a program, in the form of an input-output mapping from given input-output examples (supervised learning), or a behavior that is learned from some reward signal in an environment (reinforcement learning).

Differentiable programming In differential programming, programs are written in a way that they are fully differentiable with respect to some loss function. Differentiable programming has been employed to merge deep learning with physics engines in robotics [Degrave et al., 2019], has been applied to scientific computing [Innes et al., 2019], and even towards a fully differentiable Neural Turing Machine model [Graves et al., 2014].

Deep learning Deep learning is essentially a subset of differential programming in which the programs are artificial neural networks (ANNs). The architecture (topology) of such an ANN is usually fixed and then the weights are modified through gradient descent, where the gradients are computed by the backpropagation algorithm. This has been an immensely powerful and successful approach, and much of the current momentum in artificial intelligence and neuromorphic computing is due to deep learning.

(Physical) reservoir computing Reservoir computing is a technique that has emerged out of work on Echo State Networks (ESNs) [Jaeger, 2001] and Liquid State Machines [Maass et al., 2002]. ESNs are recurrent artificial neural networks in which the hidden units, called the reservoir, are randomly connected and only the output weights are trained via a linear regression technique. Therefore the computation is happening in the reservoir, giving these methods the name of reservoir computing. Because of the random, potentially unknown connections within the reservoir, this model has been widely used in the unconventional computing community to model the computation in unconventional, yet high-dimensional, nonlinear and complex substrates [Tanaka et al., 2019]. It is also one of the leading computational paradigms in the Post-Digital project, and multiple ESRs are working on experimental setups - electronic, opto-electronic or fully optic - that are using this reservoir computing paradigm.

Dataflow programming In dataflow programming, a program is represented as a directed graph that models how the data flows between operations. As such, it is close to the engineering discipline of signal processing. This can be extended naturally to the dataflow computer architecture, an unclocked and non-deterministic computer architecture in which data is processed at each processing node as soon as it is available. Due to the unclocked, non-deterministic, event-driven (where events are the availability of data at a processing node) computation in this paradigm, we identify this as a *promising paradigm for unconventional computing*.

Moreover, dataflow programming is closely connected to stream processing where (potentially real-time) data can be processed in parallel by multiple processing units, e.g. in GPUs or FPGAs. Furthermore, it connects to the ideas of spatial programming [Becker et al., 2016] where the dataflow structure is fixed and then used in a spatial arrangement that is reflected in the computer's architecture, and space-time programming [Beal and Viroli, 2015] which is way of *aggregate programming* for the control of large networks of spatially embedded computing devices.

A classical formalism which may serve as a tool for the design and analysis of dataflow programs is *Petri nets* [Petri, 1962]. Petri nets are a general, graph-based formalism which can model many sorts event-driven, concurrent processes, not only parallel unclocked computer programs but also real-world systems like production processes in factories.

Control engineering In analog computing, the dynamics of the computer are usually described through differential equations. In order to program such a computer, it is more natural to think in terms of control theory, which has developed a rich repertoire of methods to drive a dynamical system into a mode of operation that is robust, stable, and implements the desired dynamics. As such, when working with analog computer systems, we may benefit greatly from tapping in to the knowledge on control theory that has been accumulated over the past century. A promising avenue is represented by data-driven control, in which a model of the system to be controlled is learned from experimental data using machine learning techniques.

Neural engineering Control theory has been used as the formalism of choice for the neural engineering framework by Eliasmith [2005]. In this framework, high-level specifications in the language of dynamical systems can be designed into networks of neurons and compiled down to mixed-signal spiking neuromorphic accelerators like Braindrop [Neckar et al., 2019], using the Nengo programming environment [Bekolay et al., 2014].

Neuromorphic engineering / signal processing and circuit design A standard approach in neuromorphic engineering, as envisioned by Mead [1990], is to design application-specific microchips using sub-threshold analog CMOS technology for specific tasks, for example for tactile perception [Mastella and Chicca, 2021]. This approach uses tools from signal processing and control engineering, as well as computational neuroscience to implement a desired behavior in networks of spiking neurons.

Neuromorphic synthesis Neftci et al. [2013] developed a new approach that they call *synthesizing cognition* in order to automatically map the function of a finite state machine onto an abstract layer which models the unreliable hardware system (in this case an analog neuromorphic chip). They demonstrate their approach on a real-time context-dependent visual classification task. They provide a systematic way of synthesizing simple high-level behavior into a noisy, imprecise neuromorphic chip. This approach tackles, and solves, many of the issues that have been mentioned in this report: noise (cf. Section 3.3), robustness (cf. Section 3.4), multiple-timescale dynamics (cf. Section 3.2), and mixed-signal processing (cf. Section 3.1).

Reservoir transfer A similar problem was solved in a different way by He et al. [2019]. In order to circumvent the limited observability of their analog spiking neuromorphic hardware which makes training very difficult (and impossible on-chip), they trained a network of spiking neurons in software and developed a method that they call *reservoir transfer* to transfer the properties of this well-functioning software reservoir onto their hardware. They solve issues of device mismatch, multiple timescales and noise in their analog hardware and demonstrate their approach with an ECG hearbeat abnormality detector using the reservoir transfer method.

Neuromorphic compilation hierarchy Zhang et al. [2020] propose a more general framework in order to raise the level of abstraction for neuromorphic computing. They propose a compilation hierarchy that includes some approximations in order to compile high-level descriptions of neural networks into a variety of different spiking neuromorphic systems.

Quantum programming A quantum computer can be programmed in various ways, but a dominant approach taken by leaders in the quantum computing industry is to work with the quantum circuit model. This has lead to the development of frameworks like Qiskit [Treinish et al., 2021] that enables developers to write quantum programs in conventional programming languages like Python. The program is written by designing a quantum circuit in which quantum operators are represented as gates acting on the input qubits,

after which a measurement collapses the quantum state into a single bit for each qubit. Commonly, this procedure is repeated multiple times to generate a histogram of output patterns, similar to classical randomized algorithms. Given the young age of quantum computers and the non-existence of large-scale quantum computers at this point, it makes sense that quantum computers are programmed on a circuit level. Raising the level of abstraction in quantum programming is an active area of research [Bichsel et al., 2020].

Holographic algorithms Valiant [2008] introduced the concept of holographic algorithms and defines them as algorithms that are derived through the concept of *holographic reduction* from one problem to another. The reduction of a problem A to another problem B is a widely-used concept in computer science: for example, a Boolean satisfiability problem can be reduced to a graph theoretical problem through a one-to-one mapping every way of satisfying the Boolean formula to some solution of the graph theoretical problem. In this way, the solution of problem B will directly yield a solution to problem A. In a holographic reduction, instead of a one-to-one mapping only a many-to-many mapping needs to be identifiable, i.e. the sum of the solution fragments to one problem maps to the sum of solution fragments to another problem. Valiant [2008] is able to derive such holographic algorithms that run in polynomial time to solve problems that were previously only solvable in exponential time.

Amorphous computing Amorphous computing [Abelson et al., 2000] is trying to fill the gap between the construction and the programming techniques required of systems which do not adhere to the strict precision requirements of digital computer chips.

Emergent programming Emergent programming is the automated assembling of instructions of a programming language using mechanisms which are not explicitly informed of the program to be created [Georgé et al., 2005], by using methods from adaptive multi-agent systems and relying on AMAS theory of cooperative self-organization [Gleizes et al., 1999].

Autonomic computing The goal of autonomic computing is to design systems that are able to adapt themselves in order to stay within a high-level description of desired behavior [Parashar and Hariri, 2005]. The field takes inspiration from the autonomic nervous system, which is able to stay within a stable "dynamic equilibrium" without global top-down control.

3.5.7 Critical gaps

In this section, we have investigated programming in a more general way than is usual in computer science. We have identified some programming paradigms that are particularly promising for unconventional computers, e.g. reservoir computing, program synthesis and induction, evolutionary computing, dataflow programming, neural engineering, and amorphous computing. In order to define a clear programming language for novel computing devices, we must first develop a formalism to describe computation that goes beyond the prevalent theory of digital computation. It is clear that everything that needs to be considered in a generalized theory of computing which includes unconventional computers must also be considered in such formalisms and methods for programming.

In addition to a formalism or language to describe unconventional computing, a number of critical gaps can be identified which necessitate further research before effective programming tools for unconventional computers can be developed. These gaps may only be possible to bridge with unifying formalisms, but practical solutions in specialized cases may precede such

a unification. We present three of these issues that we perceive as critical gaps in programming unconventional computers.

The physics/computing gap. Digital computers are programmable because we build them to be programmable, and we build them according to our digital theory of computation. As such, we have bi-stable circuit elements that can be switched at our will (and only at our will) and that are organized in complex architectures to enable us to specify complex behavior. In unconventional computers that exploit interesting physical effects for computation, the situation may well be reversed: instead of enforcing our theory of computation onto the material, the material will have a fixed physical behavior that we will attempt to exploit for our computation.

This presents the problem of aligning our models of physical behavior with our models of computation. Currently, the two are incompatible because they are using completely different formalisms: physics is usually described in the language of continuous-time dynamical systems whereas computer science is described in the language of logic, where no reference to real (physical) time exists (cf. Jaeger [2021] for a more detailed discussion). Analog computing (cf. Section 3.1) presents a promising opportunity here because analog computers are often described using the same formalisms that we use for physics.

Through a common language to talk about physics and computing, we may be able to match material effects to computational primitives or material structures to computational architectures - or more loosely, to match material properties to categories of computational problems that are well-suited to be computed in such a material.

The local/global gap. Programming a software system can be seen as a methodological bridge between the local behavior of individual components and the global behavior of the system as a whole. Digital computing has developed a powerful abstraction hierarchy to bridge the gap between local behavior, in the form of computational primitives that are implemented directly in the hardware through logic gates, and global behavior, i.e. the behavior of the software system as a whole.

Such a setup in which individual components are interacting with one another in a structured way to give rise to an organic whole with some behavior leads to what Weaver [1948] calls *organized complexity* and we are currently lacking the right tools or formalisms to analyze such complexity in general. This is one of the things that a general mathematical theory of computation, as outlined in Section 3.8, must provide. In digital computing, a number of techniques have been developed to manage the complexity that arises in software systems. Note that much of this complexity is (almost) entirely disconnected from the physical details of the computer. These techniques are briefly outlined below.

When working on a single program, programming languages provide a powerful hierarchy of abstractions that allows a programmer to work on a high-level, close to the behavioral description of the program, neglecting many practical details that are essential for the implementation. *Programming language abstraction* and a compilation hierarchy to increasingly low-level languages removed the need to specify details about memory management, process management, scheduling, and other "low-level" details. *Functional abstraction* makes it possible to raise the level of abstraction by re-using existing procedures within new procedures. With just one line of code, it is possible to import a library that provides an entire collection of well-optimized code to do anything, from numerical operations all the way to training deep neural networks. *Data abstraction* makes it possible for a programmer to use abstract data types to represent numbers, or complex composite data types without worrying about the technical details about how this data type actually translates into the memory allocation on the register-level.

Moving on to more complex software systems, powerful programming paradigms and design patterns have been developed to manage the system's complexity. As an example, the *object-oriented programming paradigm* has been developed to structure code in objects, which interact with each other only in pre-defined ways, thereby effectively managing the complexity of interactions that can take place in such a software system. This is done through *information hiding*, whereby the internal details of an object is hidden to the outside and only "public" methods and attributes of the object are visible (where the visibility can also be further categorized depending on "who" is "looking"). An example for a *design pattern* is the *Model, View, Controller* (MVC) design pattern for programs with a user interface. In this pattern, the program is divided into three interconnected parts: the Model contains the application's data structure which is read by the View to display this data to the user. The interaction of the user with the View is handled by the Controller, which then manipulates the data in the Model according to the user interaction. This design pattern increases the modularity of the code and thereby decreases the complexity of the overall system in practice.

Moving towards interacting software systems, we can take the example of concurrent systems. Concurrent systems describe any system in which the lifetime of multiple computing processes overlap - the computation itself must not necessarily happen at the same instant (as in parallel computing). Moreover, concurrent systems can (but need not) be spread out in space, either on the same chip (different cores), or across different chips, or even across different computers communicating over a network. Concurrent computing introduces many problems that do not exist in parallel computing, and process calculi have been developed to model and reason about the behavior of concurrent systems.

All of the above-mentioned methods have been developed to manage the complexity of software systems in order to enable reasoning about the system's behavior as a whole, including all of its parts. This enables the *top-down design* of software systems by specifying the global behavior, then incrementally moving towards local programs. A standard exercise in a job interview for a software engineering role concerns the design of a software system, e.g. how to design a real-time messaging platform. The design may start with the desired user-facing behavior, to provide a user interface through which users can send and receive messages. This requires a program running on the user's computer where the *View* (assuming the MVC design pattern mentioned above) allows to type and read messages. Upon pressing the send button, the program's *Controller* will then establish a networked connection the server and send the typed message, as read from the *Model*, to the server. The server receives this information, saves it into a database, determines the address to which this information needs to be sent. The receiver, whose *Controller* is continually pinging the server to see if new relevant information has been uploaded to the database, will see the new message and display it to the user. From this high-level design sketch, a software engineer will be able to implement the system.

In unconventional computing, it is currently not possible in general to design complex systems in the same top-down manner. However, some approaches do exist and the problem of controlling or designing global behavior from local interactions and programs is investigated in different forms in various disciplines: in robotics as emergent cognition, in logic and computer science as multi-agent systems, in various disciplines as agent-based modeling, in cognitive science and artificial intelligence as swarm intelligence, in machine learning as distributed or federated learning, and in complex systems research as the broad topic of emergence. More concretely, the Neural Engineering Framework by Eliasmith [2005] has been used with a software tool called Nengo to automatically compile high-level descriptions of control systems [Bekolay et al., 2014] to build large-scale brain models [Eliasmith et al., 2012b].

Leaky abstractions The concept of abstraction has already been introduced above. In short, an abstraction is a mechanism by which implementation details of a component are hidden away and replaced by a simpler interface of this component. This is the key mechanism by which complexity is built and managed in complex computer systems. It enables designers to easily jump between different levels of description with varying degrees of details by creating a hierarchy in which the bottom-most layer is a full description of the entire system and further layers iteratively create modules, or sub-systems, which interact with one another through pre-determined interfaces.

However, in many complex systems, such abstractions can begin to fail and leak details about its implementations that were supposed to be abstracted (hidden) away. This is then said to be a *leaky abstraction* [Spolsky, 2004]. In the case of a leaky abstraction, the designer must still know the underlying implementation of the component in order to guarantee the reliability of any system using this component. Leaky abstractions are a common source of unexpected behavior in complex software projects.

This is reminiscent of the difference between *decomposable* systems and *nearly decomposable* systems as proposed by Simon [1991]. In hierarchical systems, the interactions between sub-systems can be clearly distinguished from the interactions within individual sub-systems. In a decomposable system, we can treat the individual sub-systems as independent from another. In a nearly decomposable system, we can identify sub-systems but the interactions between these sub-systems are not negligible - information is leaking between sub-systems.

In many unconventional hardware systems, we will have components that can only be abstracted in a leaky way. This may be because the component offers only limited observability (as in the case of the Dynap-SE2 chip that ESR1 is working with), or because the components are of a stochastic nature. It is necessary to develop programming methods that can deal with such leaky abstractions.

Zhang et al. [2020] incorporate a form of this in their neuromorphic compilation hierarchy. Whereas the compilation of a digital computer program is deterministic and yields a well-defined lower-level program which is exactly equivalent to the higher-level program, their "neuromorphic compiler" turns a high-level description of the neuromorphic program (a neural network) into a control-flow graph that is only approximately equal to the high-level description.

A key difference between digital computing and unconventional computing may be seen in the "cleanliness" of the abstractions that are possible. Whereas the interactions of digital software systems can be controlled precisely, as illustrated above, the same cannot be said in unconventional computing systems. In the words of Simon [1991], the digital software system presents itself as a cleanly decomposable hierarchy because all interactions between the different submodules of the systems can be fully characterized and controlled. In contrast, a similar unconventional system would present a nearly decomposable hierarchy (if it is decomposable at all!), because the interactions between its different submodules cannot always be fully controlled.

3.6 Information and representation in cognitive systems

To build cognitive agents, relevant information about the world needs to be represented in the system for it to behave appropriately. The meaning of “information” here is broader, more complex, and less quantifiable than the fundamental measures such as Shannon information and algorithmic information theory [Cover and Thomas, 2006]. There is no unifying picture already at hand, different frameworks exist to deal with different aspects of cognition, and their

views on information differ. To illustrate this claim, two research areas are presented in this section: Knowledge Representation and Representation Learning. Each of them illustrates challenges faced when dealing with information inside a cognitive agent. Knowledge Representation tackles the concept of information when reasoning and Representation Learning when learning.

Knowledge Representation In Artificial Intelligence, Knowledge Representation refers to the way in which a logical agent interacting with the world accumulates and organizes knowledge while perceiving and acting. Here information is incorporated into the agent into pieces that can be combined and recombined to solve new problems. The emphasis is on the task of reasoning after the extraction of perceived information into the representation language used. Once a representation language is selected (such as first-order logic), it has a formal definition of how its representational expressions (syntax) relate to the outside world (semantics). The complex nature of information in Knowledge Representation can be illustrated by identifying its main roles [Davis et al., 1993]:

- Surrogate: the information inside the agent is a substitute for the thing present in the outside world. The constraints set by the representation language allow the agent to reason over previously acquired information. Thus, information can help to determine consequences of actions by reasoning instead of acting.
- Selection of the relevant information (often called "Ontological commitments"): all representations are imperfect approximations of the outside world. It is necessary to focus on some aspects of the world, simplify some and neglect others. Selecting the best representation needs to be carefully done according to the domain at hand. Multiple strands of research investigate what information to put in their system to deal with different domains, from the physical world [Forbus, 1988] to the manipulation of scientific diagrams [Forbus et al., 2011]. One central and still open question is then to what extent can these different areas of knowledge be unified to solve problems which involve several areas simultaneously.
- Fragmentary theory of cognition: the way the information is expressed is necessarily influenced by the cognitive task it is used for. For a logical agent, the design is centered on logical deduction. Alternatively, for a Reinforcement Learning agent, the design focuses on how to propagate information from a reward signal. Once the "what for" of information is decided, many constraints need to be incorporated. In Logic, a long history of investigations of valid reasoning converged to the constraint of expressing facts about the world into a rigorous syntactic language for which a calculus can be applied to derive new information. As a side effect, information is constrained to be context-independent, unambiguous, and cannot capture phenomena that create new types of "things" not reducible to the old ones (such as evolutionary phenomena where new species emerges).
- Medium for efficient computation: building a cognitive system not only requires specifying a set of valid rules of inference but also ensuring that they are used efficiently. Making the reasoning process more efficient is however often not sufficient, the representation itself needs to be adapted. In this respect, a large effort in Knowledge Representation concerns the development of different techniques (such as semantic nets and frames) to organize information in ways that facilitate reasoning.

Representation learning Deep Learning research focuses on extracting information from raw data (such as pictures) potentially with the help of supervision (such as labeled categories

that abstractly characterize the picture). Extracting information means learning an abstract representation of the data. While it can be straightforwardly quantified in a supervised task by monitoring the input/output behavior of a neural network, some research directions (sometimes grouped around the label "Representation Learning") investigate what makes a good representation and how to translate it into a learning algorithm [Bengio et al., 2012, Goodfellow et al., 2016].

First, we can analyze representations through the lens of transfer learning. In this setup, a good representation is one that makes subsequent learning on another task easier. For instance, unsupervised pre-training can be analyzed in how it impacts the initialization of a subsequent supervised learning task by accessing an otherwise inaccessible region of the cost landscape [Erhan et al., 2010]. It can also enrich the input representation when it is initially poor. One example is in natural language processing, when one-hot encodings are replaced by word embeddings [Mikolov et al., 2013]. More generally, one looks for features that are shared between tasks, they can be found at the level of input information (e.g low-level features are shared across vision tasks), or at the level of the output (e.g speech recognition of valid sentences from very different versions of the same phonemes for different persons).

Another complementary view comes from noting that neural networks distinguish themselves from other Machine Learning algorithms by using distributed representations. One important feature is that distributed representations are *expressive*: their parameters (i.e the weights for a neural network) are used to characterize many inputs even if they are not directly neighbors in the input space. This is not the case for non-distributed techniques such as k-means or decision trees (detail in [Goodfellow et al., 2016, Ch. 15.4]). This overlap between distributed representation is also critical for transfer and is used to design architectures that share the transferable features. Experimentally, it is found that this overlap induces a rich similarity space in which concepts that are close semantically have a close distance [Mikolov et al., 2013], a property that is absent from purely symbolic representation.

Finally, because of the statistical nature of Representation Learning, learning to do predictions based on observed data requires to make assumptions about the data-generating process (this point is formalized by the no-free lunch theorem [Wolpert and Macready, 1997]). These prior assumptions range from generic to specifically designed for the domain or task at hand. One of the main challenges is to translate them into a relevant learning algorithm. Some examples of assumptions are the smoothness of the function to learn, hierarchical organization of features, sparsity, efficient representation of causal relations, temporal and spatial coherence (an extensive list is available in [Bengio et al., 2012], a more speculative and recent list is in [Goyal and Bengio, 2020]).

3.6.1 Critical gaps

The identified critical gaps are:

Beyond quantitative measures of information The meaning of information in this section goes beyond a simple quantitative measure. It gets its meaning within a larger theoretical framework that captures relations between the information in the system and the outside world (i.e. the semantics). This aspect seems necessary to build cognitive systems. It also sets the bar to create a similar framework for cognitive systems based on dynamical unconventional hardware. [Jaeger, 2021] contains an extensive discussion on the necessity of semantic accounts of "information" in general theories of computing.

Knowledge transfer Bridging different domains of knowledge is a general challenge across

the two presented frameworks. For Representation Learning, it is about learning a representation that will help to learn another. For Knowledge Representation, it is about creating compatible knowledge bases. Interestingly the construction of powerful abstraction where the two meet is an ongoing subject of research in itself [Mitchell, 2021].

Learning vs Reasoning The two presented frameworks are very different and highlight a fundamental difficulty in bridging reasoning and learning together.

3.7 Inspiration from the brain

Brain inspiration is a major argument for the transition from digital to more unconventional hardware [Jaeger, 2021, Indiveri, 2021]. Many different physical phenomena are under scrutiny to reproduce brain-like phenomena [Marković et al., 2020]. This section discusses how cognitive science and neuroscience can respectively inform the design of computing devices at the level of hardware and software.

In the absence of a general theory of unconventional computing, it is not (yet) known how different phenomena at the substrate level relate to more complex information processing. This entails that relevant inspiration can be taken from both levels.

We will start by illustrating how we can take inspiration from the way biological brains exploit and integrate a vast number of biochemical, electrophysiological, and anatomical phenomena. Some of these effects are already well analyzed and will be presented as basic substrate opportunities. They often can be translated into already worked-out algorithms to achieve complex information processing. After this first exposition, we will allude to the daunting number of brain phenomena that are not well understood and formalized to the point of being useful for engineering purposes. They can be a source of inspiration both for a general theory of unconventional computing and to build complex algorithms.

Then we will present how cognitive science can provide landmarks to reach more complex information processing that goes beyond our engineering abilities.

3.7.1 Neuroscience (hardware inspiration)

Basic substrate opportunities Computational neuroscience offers many ways of modeling a single neuron, and thus many options for electronic microchip circuit design. There are 3 major classes of neuron models:

- Discrete-time, binary values models emphasize all-or-none activity of neurons (presence or absence of spike) and the delay between two spikes (the time discretisation reflects the refractory period). McCulloch and Pitts [1943] showed that it is possible to configure these neurons to realize logical functions AND, OR, and NOT, and beyond that, general Boolean circuits and recurrent Boolean systems.
- Discrete models can be made more precise (in terms of biological accuracy) in continuous time and values using ODEs. They either model the average firing rate of neurons as in leaky integrator models or the spike itself by either the timing of the spike (e.g. Leaky Integrator and Firing models) or the specific shape of the membrane potential during the spike (e.g. the Hodgkin–Huxley model [Hodgkin and Huxley, 1952]).
- Neural fields models have been motivated both by the huge number of neurons in the brain [Coombes, 2006] and the fact that information is encoded in populations of neurons

[Schöner et al., 2015]. In neural fields space is continuous (typically 2-dimensional, corresponding to cortical surfaces) and the activation landscape over this space corresponds to local average firing activities.

Each of these models can capture a very large number of phenomena. For instance the Hodgkin-Huxley model is able to reproduce many different kinds of temporal processing in cortical neurons including tonic spiking, phasic spiking, spike bursting, spike latency, subthreshold oscillations, rebound, bistability, and spike frequency adaptation [Izhikevich, 2004]. Also McCulloch-Pitts neurons are rich enough to simulate any finite-state machine [Minsky, 1967].

For unconventional hardware, one ideally wants to take inspiration from a neural model that exhibits relevant computational properties without detailing their implementation in actual biological brains. The Hodgkin-Huxley model, for example, is a conductance based model. As such it gives an equation for the membrane potential in terms of conductance for intra- and extracellular concentrations of sodium and potassium. While it is already a simplified model — the remaining ion currents are lumped together into a single *leak current* variable — it can be simplified even further by removing either potassium (the so called *transient-sodium* model [Izhikevich, 2007]) or leak currents (a model equivalent to that of Morris and Lecar [1981]). The resulting models are minimal for spiking: they can still reproduce this central aspect of neuronal behaviour while disregarding others such as spike frequency adaptation, certain kinds of bursting behaviour and chaos.

In general, the spiking behaviour observed in biological neurons rests on the combination of a variable whose dynamics are described by a fast positive feedback loop (e.g. sodium) and a recovery variable that evolves in a slower negative feedback loop (potassium or leak current in the example above) producing the up- and downstroke of a spike, respectively. In more abstract models, like those of FitzHugh [1961], Nagumo et al. [1962] or Izhikevich [2003], these variables do not need to correspond to biophysically meaningful quantities, as long as they capture the phenomenon of interest. Such models are also much more efficient in terms of numerical simulation.

As we go from conductance based towards simplified models or even to rate-based or binary descriptions, progressively stronger assumptions are made about the mechanisms underlying neural information processing. Similarly to what has been discussed in Section 3.5.5, the line between what is a relevant mechanism and what is its substrate-specific implementation becomes blurred.

The wide range of phenomena between biophysical and abstract models reflect the variety of candidate mechanisms upon which sensory, motor and other mental functions are based in the brain. An important topic of research concerns the variety of candidate mechanisms for *neural coding*. We highlight the main ones:

- Frequency coding is the most classical coding strategy. Here the information, for instance the intensity of a stimulus or signal, is encoded in the mean firing rate of the neuron.
- In Temporal Coding, the precise timing of the spike carries information. For instance, given a time-varying signal as input, a neuron can have a thresholding behavior and fires at the precise moment at which the input signal crosses some threshold [Mainen and Sejnowski, 1995].
- In Population Coding, a stimulus is encoded in joint activities of several neurons. Each neuron in the population has a different distribution of responses over some set of inputs that may be combined to determine the identity of the input [Georgopoulos et al., 1986].

- Sparse coding is a general strategy used by various brain areas where only a few neurons are activated at the same time [Olshausen and Field, 2004].

These basic considerations guide designers in the way they want to use effects at the substrate level. However, the main challenge to use neural networks in an engineering domain is to find design principles that bridge the local description of one neuron and the global network level to implement a function. For instance, the learning principle behind the Boltzmann Machine [Ackley et al., 1985] standardly requires neurons to be discrete-time, discrete value, and stochastic.

Worked-out algorithms Already a significant number of worked-out brain-inspired algorithms exist to bridge these local/global gaps. They drive designs of unconventional hardware.

Feedforward neural networks , the bread-and-butter kind of artificial neural networks for pattern recognition, have their original and by now classical motivation as essential models of visual processing in the human brain [Rosenblatt, 1958]. For visual pattern recognition today one uses (almost exclusively) a refined variant of feedforward networks, *Convolutional Neural Networks* (CNNs). The successes of CNNs in Deep Learning [Krizhevsky et al., 2012] are originally [Fukushima, 1980] inspired by the structure of feature detectors in the ventral stream of the visual cortex and the spatial invariance of neurons responding to objects in the inferotemporal cortex [Kobatake and Tanaka, 1994]. However, CNN's handcrafting of invariance to the position of features loses information about their spatial relationships. Alternative brain-inspired methods that specifically tackle this issue might lead to more robust models [Hinton, 2021, Olshausen et al., 1993].

Recurrent neural networks Hopfield Network models of brain memory storage and recall [Hopfield, 1982] and the Boltzmann Machine [Ackley et al., 1985] are at the origins of the current Deep Learning revolution [Sejnowski, 2018]. They both are recurrent neural networks and provided the first proofs of concepts of the principled exploitability of non-linear neural dynamics. Nowadays, Hopfield models of fixed point attractors to model hippocampus memory are being challenged by the constantly moving and rich dynamics observed experimentally (especially in place cells); this is a subject of current research [Buzsáki, 2019]. Moving from the hippocampus to the cortex, an alternative approach motivated by the messy arrangement of cortical neurons is reservoir computing whereby one can harness the dynamics of randomized networks to do computations [Maass et al., 2002]. This method is now a strong driver of unconventional computing applications [Tanaka et al., 2019].

Bottom-up/Top-down neural networks Additionally to time, one of the most common critiques from neuroscientists about the plausibility of deep learning models is their lack of "top-down" projections. Aside from lateral connections and non-hierarchical connections through the thalamus [Sherman and Guillery, 2011], the visual processing circuit of the brain includes a large amount of "top-down" connectivity between regions. These are often interpreted in terms of high-level priors in the framework of Bayesian inference. Though more difficult to train, some models achieve reasonable performance [Lotter et al., 2016].

Attention mechanisms Given limitations on the availability of computational resources, such as memory, it is natural for biological brains to exert some control over where these resources are directed at. The idea has been picked up in the context of encoder-decoder architectures, where an encoder RNN takes in a sequence and outputs an encoding for each sequence element. A separate decoder RNN then produces an output sequence

from the encodings, for example in translation tasks. As the output sequence is generated element-wise, an attention mechanism can be used to dynamically reweigh encodings depending on their importance for the output to be produced. A different, less biologically inspired approach has been proposed as *self-attention* [Vaswani et al., 2017] and has since become the state of the art in sequence modeling. Interestingly, Ramsauer et al. [2020] show the equivalence of self-attention with a modern form of Hopfield networks and Cordonnier et al. [2019] claim a close relation to CNNs.

Higher-level cognition This is the most speculative area. Far from the sensory and motor cortices, it is hard to study experimentally. Some proposals revolve around building more structured working memory in recurrent neural networks. It can be done by taking inspiration from the strength of digital computers by separating memory and computation with a trainable read/write operation [Graves et al., 2016]. Alternatively, meta-learning procedures can be applied to trained recurrent neural networks to learn to quickly adapt to new tasks [Wang et al., 2018]. Other proposals tackle symbolic thought directly by replacing classical symbols with randomly sampled high dimensional vectors (as a model of populations of neurons) and using different properties of their algebra to implement symbolic capabilities [Kanerva, 2009].

Untamed brain phenomena The 3 major classes of neuron models capture only a part of the physical-dynamical phenomena that are used by brains. Many biological effects and mechanisms arise at higher levels of complexity than single neurons. They are sources of inspiration for both a general theory of unconventional computing and for physical microchip / circuit design. An illustrative choice of such effects is presented here:

- A time scale hierarchy of plasticity mechanisms (from [Jaeger et al., 2021])
  - short-term plasticity (1ms-10ms), STDP, SDSP
  - long-term plasticity (10ms-500ms for weight change, 1h-years for weight preservation), LTP, LTD, plasticity of spike traveling delay [Bucher and Goaillard, 2011].
  - intrinsic plasticity (0.5s-10s), threshold adaptation
  - homeostatic plasticity (1s-1h), synaptic scaling
  - structural plasticity (1h-lifetime), architecture reorganization
- Oscillations are ubiquitous in the brain but they also appear to have an active role in computation. For instance, there is evidence in the retina [Koepsell et al., 2009] and in the hippocampus [Agarwal et al., 2014] of information coded by aligning the timing of spikes with the ongoing phase of the oscillations.
- The structural and dynamical complexity of biological dendrites vastly surpasses that of a classical artificial neuron. Their computational capacities are estimated to be close to the one of an entire artificial neural network [Jones and Kording, 2020].
- Astrocytes are non-neural cells in the brain that seem to play an important role in the regulation of neuronal activity [Jones and Kording, 2020].
- A multitude of cells in the hippocampus involved in navigation encode different aspects of the environment (such as specific locations or corners). These aspects are extracted and abstracted from different sensory modalities and integrated into coordinate frames attached to the environment [Behrens et al., 2018].

- The anatomical network architecture of perceptual processing systems goes beyond cleanly stratified hierarchies, even within area V1 of the visual cortex [Felleman and Van Essen, 1991, Sherman and Guillery, 2011, Niell and Stryker, 2010].
- Complex patterns of neuromodulation allow small networks to switch dynamical modes to compensate for temperature fluctuations of the environment [Alonso and Marder, 2020].
- The brain is known to form topographic maps where neurons are arranged such that nearby neurons code for similar features. For instance in V1, nearby neurons are activated for a similar edge orientation of the input image. Some of these topographic properties can be captured by self-organizing maps [Kohonen and Honkela, 2007].
- Chaotic dynamics may provide mechanisms for novelty detection and pattern recognition in the brain. While some single neuron models are capable of displaying chaotic activity, these need to be discerned from the case where chaotic attractors emerge as a collective phenomenon on a network level [Freeman et al., 2001]. Skarda and Freeman [1987] extensively studied odor recognition in the olfactory bulb of rabbits. They found that chaotic neural activity forms a base state upon expectance of an olfactory stimulus. Once such a stimulus is applied, the system bifurcates to a limit cycle corresponding to the recognized odor. This process effectively stabilizes one of the infinitely many unstable periodic orbits that constitute the skeleton of chaotic attractors [Ott, 2002]. Chaotic neural activity hence allows fast access to any of the stored limit cycle attractors and in addition provides a *trap state* in case an unfamiliar odor is presented.

3.7.2 Cognitive science (behavior inspiration)

Computing is hardly only mechanical, inasmuch as it is always about accomplishing a specific function. Whether it is about identifying convincing arguments for Logic (Aristotle), understanding brain predictive models of the world for statistical learning (Helmholtz) or taking inspiration from human clerks doing mental calculation for the formalization of digital computing (Turing), the elaboration of the concept of computing has always been intertwined with different aspects of human cognition. Of more immediate concern, the cognitive sciences offer requirements and ways to describe elaborate behavior that can drive and inspire design of more complex computing systems.

Cognitive processing landmarks We present here a list of requirements inspired by Eliasmith [2013] that synthesized decades of analysis in different areas of cognitive sciences and different mathematical perspectives (statistical, dynamical, and logical).

- Logical languages provide a set of target requirements that humans are able to exploit such as compositionality, productivity and systematicity. For instance compositionality refers to the capacity to capture the meaning of a complex representation by adding together the meaning of basic representations and is still a challenge for neural computations [Lake et al., 2015]. Additionally humans can manipulate a variety of expressive languages beyond the classical first-order logic [Bringsjord, 2008] and flexibly use the context of one situation to disambiguate and enrich the meaning of a word [McClelland et al., 2019].
- Reasoning with various formats of representation. While existing reasoning algorithms are in the vast majority language-like (i.e. the information is encoded in a discrete sequence of symbols), it is striking that they cannot alone account for human reasoning

which relies heavily on image-like representations [Kunda, 2018, Sloman, 2002].

- The general binding problem [Feldman, 2013] is a group of problems that arises in the manipulation of compositional symbol structure like human language and perception when spatially separated computations need to be bound together to create a coherent representation. This applies to many of the distributed in-memory computing systems.
- The human's capacity to manipulate abstract concepts is highly dependent and grounded in perception and interaction with the world [Hofstadter, 1996, Fauconnier, 2001]. Bridging the abstract and the concrete in cognitive systems is known as the symbol grounding problem [Harnad, 2007]. A concrete and ongoing sub-challenge consists in integrating language and perceptual models [Jaegle et al., 2021, Ramesh et al., 2021].
- Robustness requires that a cognitive system be able to deal with different sources of variability both at the hardware level (such as noise, heat fluctuations, resistance degradation of its components) and at the software level (such as the unpredictability of the environment, imprecisions in inputs). Notably, hardware robustness needs to be traded with energy consumption.
- Adaptability is a general requirement for a system to be able to update its future behavior based on past experiences. It can be done through statistical learning, various type of reasoning (such as logical or analogical) and many other ways. One starting point is to analyse how they can be arranged into a hierarchy of different timescales (see Section 3.2).
- For a system to adapt to its environment it needs to be able to use different memory systems. Cognitive sciences identified different memory systems: working memory, long-term memory, content-addressable memory, episodic memory (memory of specific events after one exposure), and procedural memory (automatically triggered memory). Notably, these concepts are only pointers to different behavioral capacities that highlight the need to coordinate different memory systems. The realization of these various memory systems in dynamical substrates is riddled with many open questions (see Section 3.2 for some examples related to time scales).

Lastly, when building engineering systems multiple scaling issues arise and some of them are becoming identifiable. The most perceived one involves the difficulty to predict what will happen when the system gets larger and the environment more complex ("scaling-up"). Comparing current models and tasks to brains and their environment shows they all are toy models. The difficulty to predict these scaling behaviors forces us to rely on experimentation [Brown et al., 2020, Eliasmith, 2013] while taking care that the pursued models do not intrinsically prevent scaling. This aspect of scaling can be contrasted with "scaling-out", different cognitive modules should be able to interact with other subsystems within the same overall architecture, in a fruitful way [Sloman, 2008]. Theoretical and experimental analyses of these interactions are very scarce. Investigations of child development reveal a curriculum where different modules and domains of knowledge interact and are progressively integrated [Karmiloff-Smith, 1995]. Children learn to draw different objects separately before being able to merge them to create for instance a bird with human-looking arms. Another striking example is the evidence for a deep intertwinement between action and perception at the level of the brain [Niell and Stryker, 2010] and of the behavior [Sloman, 2008, Gibson, 1979] suggesting a necessary co-design of perception and action modules.

The list of requirements presented in this section is by no means complete, but it offers some landmarks of minimal complexity for the design of complex enough unconventional systems.

Notably, very few models from cognitive science and neuroscience even try to reach all these criteria because of the sheer complexity of the task. One main reason is that this list implicitly assumes the need to integrate different functions and different modules (such as motor control, perceptual statistical learning, and symbol manipulations).

Some research in the field of cognitive architectures [Kotseruba et al., 2016] which study “The fixed (or slowly varying) structure that forms the framework for the immediate processes of cognitive performance and learning” [Newell, 1990] intends to face this challenge by building integrated systems. However, few of them are adapted to unconventional hardware mainly because of the lack of theoretical tools to control them. Unsurprisingly, many of these cognitive architectures rely on the flexibility of digital computing. Sloman [2002] presents an exhaustive list of how digital computers allowed to build cognitive systems.

Inspiration for new info-processing Finally, we note that there exist numerous examples of complex, gradually morphable mental representations of dynamical phenomena investigated by cognitive scientists that are still not yet well formalized to be of use for building cognitive systems. For instance, the framework of fluid concepts [Hofstadter, 1996, French, 2006] investigates the way perception and high-level reasoning interact dynamically in a bottom-up, top-down fashion. Other examples are radial categories [Lakoff, 1987], complex motion pattern representations [Bläsing et al., 2009, Tervo et al., 2016], and blending of mental representation [Fauconnier, 2001]. These are only a few pointers that illustrate the kind of inspiration that one could use to develop cognitive systems compatible with dynamical substrates.

3.7.3 Critical gaps

A summary of the identified critical gaps is presented here:

Gaps in current "worked-out" neural networks There are well-identified gaps in the current neural networks models: integrating fast bottom-up with "reflective" top-down processing of information; limiting the reliance on labeled data and tackling symbolic computations.

Untamed multiplicities of phenomena There is a large number of phenomena that the brain exploits in an integrated way. A major challenge is to develop formalisms that allow to opportunistically exploit many different phenomena for achieving a given function.

Engineering complex dynamical systems Our current systems are too simple. Given the difficulty to formalize them, scaling issues will necessarily arise.

Dynamical mental phenomena Cognitive science already identified a handful of relevant dynamical, graded mental phenomena that are still hard to formalize but that might guide the transition to more flexible unconventional computers.

3.8 Mathematical modeling: requirements and deficits

Digital computing – the unescapable role model for any alternative, “unconventional” approach to computing – is both a practical craftmanship as well as a rigorously scientific discipline. Approaches to unconventional computing still are in an early bootstrapping phase (quantum computing excepted), characterized by a wide-spanning experimental trial-and-error search for the right angles of attack. A unifying theoretical basis is missing.

3.8.1 The requirements

In this subsection we mostly give a condensed rehearsal of an extensive analytic survey on theory-building for unconventional computing by PI Jaeger [Jaeger, 2021].

The awe-inspiring forces of digital technologies to unleash productive powers for almost every sort of human activity are made possible and lasting through the firm rooting of digital computing practice in a body of mathematical theory. This body of mathematical theory is richly structured and comprises numerous subtheories, including among others

- the formalization of algorithms (as Turing machine, lambda calculus, random access machines, cellular automata, general grammars or in many other equivalent formalizations),
- the Chomsky hierarchy of formal languages and automata models,
- the theory of computability,
- the theory of computational complexity,
- first-order logic models to specify the semantics of algorithms,
- a multitude of specialized formal logics for calculi of knowledge representation formats and user modeling.

In Jaeger [2021] we have described the theory components of the digital computing universe in some detail. Figure 2 gives an impression of the structuring of this theory cosmos.

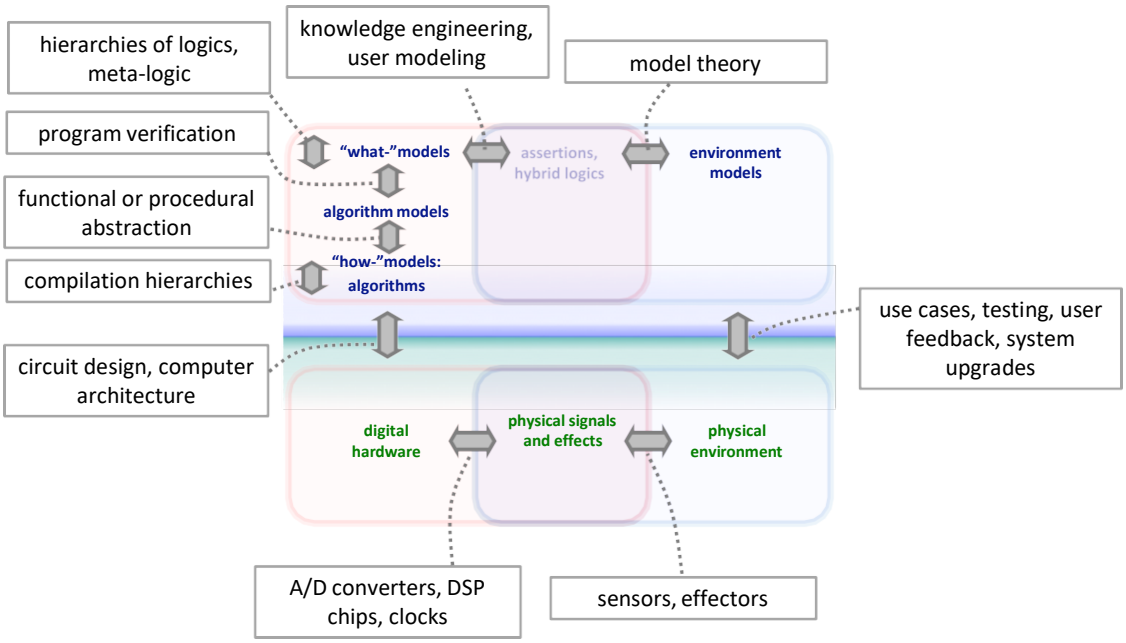

Figure 2: The structure of the digital computing theory universe (top: models, formal world descriptors, formal environment models) and how it relates to the real world (bottom: hardware, signals, environment). Three main categories of formal theories are "how"-models which specify the symbolic mechanics of computing processes (automata, machine code, Boolean circuits,), algorithm models (eg. programming languages, Turing machines, cellular automata), and "what"-models (logic-based specifications of computing semantics). Shaded arrows and text boxes indicate systematic transformations and interrelations between subtheories.

This body of formal theories

- is highly standardized and taught to computer science students worldwide in essentially the same way, based on canonical textbooks,
- is internally connected by well-understood mappings and translations between the different subtheories,
- has grown from ancient roots in two millennia of an intellectual quest for understanding the essence of rational reasoning, from Aristotle's syllogisms to the 20th century of modern mathematical logic, culminating in Turing's formalization of reasoning as a replicable mechanistic process,
- and since antiquity committed to seeing rational reasoning (and hence, computing) as inherently discrete and symbolic.

We emphasize the need for a richly structured compendium of subtheories, as opposed to the idea that one could or should aim for a single, grandly unified foundational theory for unconventional computing. In the reality of a mature computing discipline, a wide spectrum of experts must collaborate. They are each specialists for the hierarchical layers of computing systems, from microchip design and manufacturing to computer architectures, strata of cross-compilable programming languages, program specification and verification, and use case and user modeling. Each of these specialists needs his/her own formal tools and languages which is adequate for the respective segment of computing systems.

A similar anchoring in a rigoros formal coordinate system is missing, and we find that even the necessity to develop a compendium of specialized sub-theories is rarely perceived. Almost all of the existing approaches to theory-building (compare listing in Section 2.1) either address only a limited section of the "computing" landscape, or aim for a singular and comprehensive theory. Stepney and Hickinbotham, leading theorists for unconventional computing, list a number of existing mathematical formalisms that are tailored to specific subsets of material phenomena or computational functionalities, and otherwise remark that an *"over-reaching formalism ... may be desirable"* [Stepney and Hickinbotham, 2018].

In Jaeger [2021] we investigate in some detail two further worked-out conceptualizations of "computing" which are complementary to the digital/symbolic paradigm. The aim of that study is to identify fundamental conceptual invariants shared by all views on "computing", in order to distil necessary conditions for our ultimate goal of establishing a rigorous theory for unconventional computing.

The first of these two non-digital views on "computing" are sampling-based methods for inferences about probability distributions (SIP methods for short). The central commitment here is to regard cognitive processes as probabilistic inference, and "computing" as realizing such inferences through sampling procedures like *Monte Carlo Markov Chain (MCMC)* [Neal, 1993] or *particle swarm* methods [Dellaert et al., 1999]. The requisite random sampling processes are still mostly simulated on digital computers, but also have become realized in non-digital computing systems, namely in DNA computing [van Noort et al., 2002] for solving optimization and search problems, and more recently and with a wider application range in analog spiking neuromorphic hardware [Indiveri et al., 2011, Haessig et al., 2018, Moradi et al., 2018, Neckar et al., 2019, He et al., 2019]. According to the *neural sampling* view forwarded in theoretical neuroscience [Buesing et al., 2011, Pecevski and Maass, 2011], temporal or spatial collectives of neuronal spike events can be interpreted (or used by the brain) as samples. Due to this inviting analogy to biological brains and the low-power characteristics of analog spiking neurochips, research in such hardware and corresponding "algorithms" is expanding. But more importantly in the context of this deliverable report, SIP methods open views on "computing" that differ from

the digital/symbolic paradigm in interesting ways.

The second alternative conceptualization of "computing" are models of information processing that are cast in the format of ordinary differential equations (ODE methods for short). Since almost a century, biological systems — neural and others — have been studied in *general systems theory* [von Bertalanffy, 1968] or *cybernetics* [Wiener, 1948]. This tradition co-evolved with the engineering science of signal processing and control [Wunsch, 1985]. A classical landmark in interpreting the human brain as a dynamical (self-)control system is the *Design for a Brain* of Ashby [1952]. In another co-evolving strand, neural dynamics became modeled in a theoretical physics spirit, by isolating and abstracting dynamical neural phenomena into systems of differential equations, exemplified in the neuron model of Hodgkin and Huxley [1952]. Later, when the mathematical theory of qualitative behavior of dynamical systems [Abraham and Shaw, 1992] had matured and in particular after *chaos* and *self-organization* in dynamical systems became broadly studied, dynamical systems modeling rose to a commonly accepted perspective in cognitive psychology and cognitive science [Smith and Thelen, 1993, van Gelder and Port, 1995]. Today the separations between these historical traditions have almost dissolved. Mathematical tools from dynamical systems theory are ubiquitously employed in modeling neural and cognitive phenomena on all scales and abstraction levels, in a diversity that defies a survey. Even when seen only from within mathematics, dynamical systems theory is a highly diversified field. Here we will only consider models expressed with ordinary differential equations, since these ODE models are the most wide-spread ones. Again, ODE methods open our eyes to aspects of information processing which are hardly accessible in the digital/symbolic paradigm, in particular aspects of continuous time and state spaces, real-valued representations of "information" or "knowledge", and self-organization, dynamically controlled modulation of "data", stability and robustness.

We have thus three alternative, worked-out conceptualizations of "computing": the digital /symbolic computing one (DC methods for short), SIP models and ODE models. Across the diversity of perspectives and mathematical formalisms, in Jaeger [2021] a number of common organizational principles was worked out, which allow us to relate these three sorts of models to each other and distil coordinates in which, we believe, *every* fully fledged theory framework for "computing" should become localized — including a future general theory (or rather, system of interrelated subtheories) for "computing" based on nonlinear phenomena in unconventional materials. In the remainder of this subsection we review the four common coordinates that were explored in detail in Jaeger [2021].

Coordinate 1: Subtheory interrelations. All three paradigms become formalized in interrelated subtheories and models, with principled mappings between them:

In DC: We distinguish two sorts of formalisms and models in the world of DC: how-formalisms / models and what-formalisms / models. The former describe digital hardware systems in a double hierarchy. In the first dimension, the *Chomsky hierarchy*, automata (and grammar) models capture computing systems in a hierarchy of computational power, from finite-state automata up to the most general and powerful class, the Turing machine and its equivalents. In the second dimension, formalisms/models are ordered with respect to how close they are to the underlying hardware, from microchip-specific instruction sets and assembler programming languages at the "low" end, up to graphical programming interfaces at the "high" end. Different abstraction layers of such how-models are linked through compilation cascades. – What-models are logic-based descriptions of how-models. There exist numerous different formal logics each of which is designed to best model specific aspects of what computing processes "mean" or "effect". The seman-

tics that is declared for every logic connects the procedural "how"-mechanisms to what they "mean" or "effect" in the external environment.

In SIP: The analogue of DC how-models are specifications of *sampling procedures* (or just "samplers"), with MCMC samplers being the arguably most powerful class. However, differing from the situation in DC, the topic of inter-relating different samplers according to suitably defined measures of expressiveness has not yet been systematically explored. The analogue of DC what-models are probabilistic models which connect procedurally to samplers and semantically to real-world environments. Noteworthy examples are Bayesian networks when inferences in them are executed by sampling, or models of biological neural processing based in the neural sampling paradigm.

In ODE: Here we find only a single how-formalism: the very mathematical formalism of ordinary differential equations. These models directly capture the real-time, metric dynamics of continuous variables in computing systems — voltages and currents in analog microchips and neuronal circuits, activations of neural assemblies or concepts in models of neural cognitive processes. While there are inclusion hierarchies of classes of such ODEs — foremostly, ordering them by the highest order of derivatives, speaking of first-order, second-order etc. ODEs — the ODE formalism per se is always the same. What-models capture *qualitative* phenomena in dynamical systems, giving rise to a zoo of geometrically defined objects like point attractors or repellors or saddle-nodes, periodic orbits, chaotic attractors, stable and unstable manifolds, basins of attraction etc.

Coordinate 2: Semantics. Computing systems interact with their real-world environment by user input/output, signal input, and/or effector output. Semantics are defined for the what-models in each of our three considered paradigms, but not for the how-models. A mathematical formalization of the "real world" environment, plus a formal semantic mapping between the information structures inside the computing system and its outside embedding, is always available:

In DC: The logics which are used for DC what-models each come with a formalization of external environments, typically expressed in set-theoretic *S-interpretation*, where the "S" stands for the set of symbols used in the respective logic. These symbols (in particular symbols of objects, relations and functions) are semantically *interpreted* by sets inside the S-structures. Formally, the semantic relationship is established between symbolic expressions ϕ written in the symbolism of the respective logic, and an S-interpretation $\mathfrak{I}$. One says that "ϕ holds in $\mathfrak{I}$", or that "$\mathfrak{I}$ is a model of ϕ" (written: $\mathfrak{I} \models \phi$).

Viewing "computation" through the eyes of formal logic has a history that dates back to Aristotle's syllogisms. When Turing [1936] devised his machine model (now called the Turing machine) in his epochal paper he did not aim for a practical model of computing hardware but for a treatment of the formal decision problem in logic, which in turn is the abstraction of conceptual/symbolic human reasoning that marks the culmination of more than 2000 years of first philosophical, then mathematical inquiry.

In SIP: In probability theory, the formal models of segments of reality are *probability spaces*. A probability space is a three-component mathematical structure standardly written as $(\Omega, \mathfrak{F}, P)$, where Ω (often called "universe" or "population") is a set of *elementary events* which can be intuitively understood as locations in spacetime where measurements could possibly be made; $\mathfrak{F}$ is a certain subset structuring imposed on Ω; and P is a *probability measure* that assigns probability values to the subsets (called *events*) of this structuring. The formal connection between such world models $(\Omega, \mathfrak{F}, P)$ and the descriptive what-

formalism of probability theory is established by *random variables*. I remark that the naming “random variable” is entirely misleading. Mathematically, random variables are functions, not variables; and they are not random, but deterministic. They are the formal model of observation or measurement procedures. The randomness of random variables results not from that they somehow return random values, but that random arguments are given to them, following the probabilities prescribed by the item P in the world model $(\Omega, \mathbf{F}P)$. The basic statements that can be made in the formal language of probability theory are expressions $P(X(\omega) \in A) = p$, which states that the probability of obtaining a measurement value in a value range A when the measurement procedure X (a random variable) is applied to some “random” elementary event $\omega \in \Omega$, is p.

In ODE: In the DC and SIP paradigms, there are dedicated formalisms for modeling the “real” environment in which a computing system is situated, namely the formalisms of S-interpretations and probability spaces. Formal expressions in DC or SIP what-models are written down in a formalism different from the how-formalisms. These formal expressions of what-models express claims *about* something being the case in the respective environment model. A semantic theory relates a formalism in which claims about (or observations in) some environment can be expressed, to another formalism needed to represent the “real-world” structures in the environment.

In the ODE modeling world, a rigorous semantic theory is not available. In Jaeger [2021] we speculate that the semantic blindness of ODE modeling is historically explainable by the circumstance that the evolution of ODE modeling methods for a long time was mostly driven by the needs of physics, and physicists want to *isolate* their objects of investigation from the environment as perfectly as possible. As a consequence, the mathematical theory of isolated dynamical systems that have neither input nor output (called *autonomous* dynamical systems) is much further developed than the theory of input-driven, output-generating (*“non-autonomous”*) systems. Theory-building for non-autonomous systems has only recently started and is still a niche field whose challenges and results are not widely perceived outside a small circle of specialists. When today mathematical neuroscientists analyse computational processes in brains with ODE methods, they are virtually forced to treat the brain as an isolated system due to the unavailability of formal tools for analysing complex non-autonomous systems.

In the ODE paradigm there is an option which is not available for DC and SIP, which allows one to formally connect a computing system to its environment on the procedural level of how-formalisms. Both the computing system and the environment can be modeled with differential equations, and the coupling between these two systems is effected through shared variables. Mathematically speaking, the coupled system is again an autonomous system. Coupling (sub)systems together by shared variables in systems of differential equations, and analysing the compound system as an autonomous system, is constitutive for control theory. But, although this coupling of a subsystem into a compound system is obviously of great importance for complex system modeling in general, this coupling of two systems is not of a semantical character: the two systems are described in the same formalism, they stand side by side on an equal footing and in their symmetric relationship there is no “aboutness”.

A properly semantical theory for the ODE paradigm would have to connect qualitative phenomena (attractors, bifurcations etc), which occur in a computing system and which can be interpreted as carriers of information, with givens of some sort in the environment (objects, actions, ...), which would be formalized in a formalism of “qualitative physics” that still needs to be invented (the symbolic logic formalisms that are called “qualitative

physics" [Forbus, 1988] in artificial intelligence research are not suited as world modeling substrate for the ODE paradigm).

Coordinate 3: Formal time. In whatever way "computing" is conceived, it must be cast as a *process*, not as a static object. The temporal evolution of a computing process is modeled in interestingly different ways:

In DC: Formal time surfaces in distinctively different ways in how- versus what-formalisms. For how-formalisms the story is quickly told. Automata models and (imperative) programming languages specify formal, discrete update rules which transform symbolic configurations (e.g. bit vectors of a memory cell array, or a data structure in a programming language) in discrete "update" steps. At the most hardware-close level, these update steps become ultimately mirrored in the physical clock cycles of digital microchips. The how-formalisms do not assign "real" time durations to state update operations – the only aspect of temporality which is present in DC how-formalisms is the ordering of updates from a "previous" to a "next" symbolic configuration.

In logic-based what-formalisms, the analogue of state updates are logical *derivation* steps. A derivation step leads from a set of previously derived logical expressions to a new one which is *entailed by*, or *inferred from* the already given ones. Again, like in the state update steps of how-formalisms, there is no specification of temporal duration for these derivation steps: logical inference is logical in nature, not temporal or causal.

In SIP: Formal time is knitted into SIP formalisms and models in intricate ways, often involving further theoretical elements reflecting space and temporal hierarchies. In how-formalisms (that is, specifications of sampling procedures), two widely separated timescales are always present: the fast timescale of generating a single sample point, and the slow timescale of accumulating large numbers of them to obtain increasingly precise representations of probability distributions.

The condition that the sampling from a probability distribution takes time to yield a high-precision representation of the distribution, leads to involved constraints when sampling methods are put to practice. In static information processing tasks like image classification, the input interface of the classifying system is *clamped* to input image and transforms it to a sample (e.g. through sending spikes from a retina), allowing the sampling to grow the sample large enough to decode from it the result with the desired degree of accuracy. In online processing tasks (for instance autonomous robot action selection and control), the computing system's input and output are data streams. This is the generic situation in adaptive online signal processing and control [Farhang-Boroujeny, 1998] and in the study of *situated agents* [Steels and Brooks, 1993], which comprise humans, animals, robots, software avatars or computer game characters. Solving such tasks, where input patterns or output patterns are themselves temporally evolving, requires sampling mechanisms where the next generated sample point depends on the history of previously generated ones. Important classes of formalisms of this kind include spiking recurrent neural networks [He et al., 2019], temporal restricted Boltzmann machines [Sutskever et al., 2009] and sampling-based instantiations of dynamical Bayesian networks [Murphy, 2002].

In ODE: In the how-models (that is, ordinary differential equations), the formal time model is hidden in the dot in $\mathbf{x}$, the shorthand for the derivative $d\mathbf{x}/dt$ with respect to time t. The symbol t denotes the "real" time, measured in physical units like seconds, when ODE models are written down to describe physical computing systems. This is categorically different from the abstract state updates in DC models: One will never find the word

"second" mentioned in a textbook of theoretical computer science! The fact that a formal "1 sec" segment of t really means one physical second in ODE analog circuit models has important consequences for analog computing practice. First, the designer of analog circuits must match time constants in his/her formal ODE models to the physical time on board of the microchip. Second, analog computing systems must be timescale-matched to their physical task environment because both operate in the same physical time.

As to ODE what-models, the quantification of time lapses in physical units is lost, and only the direction of the arrow of time is preserved. This loss of numerical measurability of time is a consequence of the very definition of "qualitative phenomena" in dynamical systems, which rely on topological homeomorphisms between dynamical systems to achieve equivalence classes of qualitatively identical – but quantitatively differing – phenomena and systems.

Coordinate 4: Hierarchical structuring of formal constructs. Human cognitive processing admits — or in some views [Newell and Simon, 1976], is even constituted by — the compounding ("chunking") of representational or procedural mental states or mechanisms into larger compositional items, giving rise to compositional hierarchies of representations, mechanisms or processes. Any practically useful account of "computing" should be *scalable* to increasingly complex tasks. This requirement has led in each of our three considered paradigms to the specification of ways how one can hierachically *compose* the respective formal constructs into increasingly compounded information structures:

In DC: All how- and what formalisms in DC admit to compose symbolic configurations into more compounded ones, which then can be used as building blocks in yet higher levels of compositionality. In how-models one finds compositional hierarchies that progress, for example, from bits to ASCII symbols to words to lists to arrays ... to objects to scripts to programs. Likewise, the syntax of logic what-formalisms admit arbitrarily deeply nested functional expressions, connected into arbitrarily large and modular expressions by logical connectives like AND, NOT, OR and many others.

All these structuring principles and operators are cognitively interpretable. Writing computer programs or adding new logic formulas to an AI knowledge base is easy for a halfway trained human. This should not come as a surprise because, as we mentioned before, the theory and practice of symbolic computing originated as an attempt to reflect and automate human reasoning.

In SIP: We start our discussion with the what-models of SIP, that is, specifications of probability distributions. Probability distributions can be hierarchically organized in at least three ways, all of which are widely used.

First, when one considers joint distributions of many random variables, the latter can be arranged in hierarchical *Bayesian networks* which typically capture observables and hypothetical explanatory factors in a layered arrangement with observable random variables at the bottom ("symptoms", "evidence", "sensor data"), and in the top layer explanatory variables that are intended to represent hidden causes (like diseases or engine faults) of the observations in the bottom layer.

Second, hierarchies of distributions canonically arise in probabilistic models as conceived in Bayesian probability [Jaynes, 2003, Jaeger, 2019], where distributions become themselves distributed in *hyperdistributions*. In the original motivation of Bayesian probability, these higher-level hyperdistributions reflect the subjective prior beliefs of an intelli-

gent agent about which lower-level distributions are more or less plausible. Applying this principle to modeling probabilistic cognitive systems one obtains formalisms that are hierarchical in a substantial sense. The relationship between a distribution and a hyperdistribution is asymmetric. A hyperdistribution could be said to control, modulate or *bias* its lower-level children distributions. This gives rise to cognitive processing models whose dynamics unfolds in an interplay of bottom-up pathways (from sensor input to their cognitive interpretations) and top-down pathways (cognitive expectations modulating the perceptive filtering below). Prominent representatives of such bidirectional cognitive processing systems are Grossberg's Adaptive Resonance Theory models [Grossberg, 2013], Friston's free-energy models of processing hierarchies in brains [Friston, 2005] and Tenenbaum's models of human cognition [Tenenbaum et al., 2006].

Third, more elementary (lower-dimensional) distributions can always and precisely be mathematically combined into compound (higher-dimensional) distributions by *product* operations. Conversely, high-dimensional distributions can sometimes be approximately *factorized* into products of low-dimensional ones. Such factorizations are a major enabler to construct and evaluate high-dimensional probabilistic models of real-world systems in machine learning, computer graphics and physics [Huang and Darwiche, 1994].

In ODE: In ODE modeling, the hierarchical structuring of formal constructs is intimately connected with timescales. When researchers in cognitive neuroscience, robotics and autonomous agents or machine learning conceive of their respective intelligent agent architectures as dynamical systems, they almost by reflex assign *fast timescales* to subprocesses that operate close to the sensory-motor interface boundary at "low", elementary levels of the cognitive processing hierarchy, and the assign *slow timescales* to subprocesses operating at "higher" levels of the hierarchy. In our opinion, timescale hierarchies are key for finding hierarchicity in dynamical systems.

In dynamical systems that compute (brains, digital and non-digital computers and microchips), timescales of information processing (sub)processes are directly connected to a temporal hierarchy of *memory* mechanisms, which range from a few milliseconds (giving the actual duration of the experiental "now") up to lifelong persisting memory traces that define a personality.

Mathematical and applied research on multiple timescale dynamics in ODE models is so rich that an overview cannot be attempted here. Jaeger's research group at the University of Groningen is a partner in a EU Horizon 2020 project "Memory technologies with multi-scale time constants for neuromorphic architectures" (*MemScales*) which is entirely devoted to multiple timescales in neuromorphic computing systems. A recent technical report surveys some important branches of timescales modeling in neuroscience, mathematics and computer science / machine learning [Jaeger et al., 2021]. ESRs 1 and 2 have familiarized themselves with this line of research and their PhD thesis projects will accomodate timescale hierarchies in different ways.

We mention that the distinction between how- and what-formalisms, which for us is a strong guide, is analog to the distinction between the algorithmic and the implementation layers of brain modeling according to the classical methodological framework for neuroscience of Marr [1980]. An extensive survey by Guo et al. [2021] reviews and structures research in neuromorphic computing along Marr's modeling layers.

Summarizing the yield of this excursion into different worked-out conceptualizations of "computing", we obtain the following "big picture" of what is missing with regards to a rigorous formal foundation for computation in unconventional substrates:

1. A foundational theory for unconventional computing is still missing at present.
2. This lack is widely perceived and deplored, but we doubt that the complexity of the task to develop such a foundation is properly acknowledged. Specifically, one should not attempt to find a single unified theory framework, but instead one will have to develop a whole network or interconnected subtheories, formulated in different formalisms, in order to serve the requirements of practical work along all the layers of computing engineering, from hardware design to system configuration / programming / training up to user interfacing and use-case modeling.
3. We identified (among others, not mentioned here but described in Jaeger [2021]) four dimensions of modeling computing systems which we believe every theory system for "computing" must cover: (i) a hierarchical organization of interconnected subtheories with how- and what-theories; (ii) formal semantic accounts of what an executing computation "means"; (iii) a model or at least a clear understanding of how the time of computing processes relates to the semantic "meaning" of the computation; (iv) formal mechanisms to build arbitrarily complex hierarchical information structures or processes in the formalisms, in order to solve arbitrarily complex tasks.

Given these demanding constraints, the task to develop a theory foundation for unconventional computing is daunting. In the next subsection we propose a possible starting point for this endeavour.

3.8.2 From bistable switching to modes: generalizing digital computing toward a theory of unconventional computing

On the present day there are no clearly visible guides how one should proceed to build a theory system which can capture a kind of "computing" which can exploit a wide range of nonlinear, nanoscale physical phenomena in unconventional material substrates. The problem is underdetermined and there are many degrees of freedom to search for a solution. Here we sketch the approach that is pursued in Jaeger's MINDS group at the University of Groningen. While the research of ESRs 1 and 2 is not defined in a concrete relation to this foundational programme — that would be impossible because that foundational programme is not well-defined yet — the results obtained by these ESRs will provide instructive examples of "computing" processes which lead beyond the conceptual confines of digital computing.

We start from the insight that a theory system of "computing" should root in a formal construct which represents the elementary physical carriers of "information". In our three role models these formal constructs and the corresponding elementary physical realizations are

in DC the "bit" symbols 0 and 1 (or False and True) realized by the two switchable states of bistable electronic (or other) circuits;

in SIP probability distributions which are approximately realized in samples of physical events;

and in ODE vectors $\mathbf{x} \in \mathrm{R}^n$ as mathematical models of n-dimensional (and in principle measurable) state projections of a continuous-time physical computing system.

In the how-formalisms of all three paradigms, these elementary formal constructs each come with a specific way of how the exist and change in *time*:

the bits in DC are "set" ("written") in specific points in the ordered sequence of update steps of an automaton or program, and once they are set they remain at their value for an arbitrary

number of further system update steps until they are flipped,

the probability distributions in SIP gradually take shape and precision as the sampling procedure progresses,

and the state vectors in ODE exist only at single time points t on the real-valued timeline, changing smoothly with t;

and they can be composed into hierarchical compounds

in DC by organizing bits in nested bitstrings,

in SIP by conditional probability cascades, product operations and hyperdistribution relations,

and in ODE by coupling systems of ordinary differential equations through shared variables.

For a general theory of "computing" based on nonlinear effects in unconventional substrates, we wish to define a sort of elementary formal construct which corresponds to (in principle observable) real physical phenomena, and which similarly comes with a formal representation of temporal evolution and hierarchical compositionality. In our internal parlance, we refer to these — still elusive — elementary formal constructs as *modes*.

In addition to the temporality and compositionality requirements, we find that the mode construct also needs to encorporate some formalization of spatial extension (*"spatiality"*). Many nanoscale nonlinear phenomena with an apparent potential for computational exploits are intrinsically spatially organized, for example (a small selection) moving solitons, potential walls, waves, boundaries between different crystallization orientations, particle clusters or nanotube meshes. We note that even the construct of bits has an implicit spatial component: 1-bit memory devices must be spatially separated from each other and connected by signal propagation lines, leading to the spatial organization of connection graph topologies.

At present we can express our developing ideas about modes only in intuitive terms, supported by suggestive graphics (Figure 3).

A theory system for "computing" in general physical systems should include digital computing systems as a special case. We thus desire that bits can be seen as a special case of modes. Figure 3 conveys our basic intuitions.

Modes generalize from bits in several ways:

- While a bit is either on or off, modes can be present with different *intensities* (e.g., energies), fading in and fading out.
- While a bit is localized in a "point" (at least conceptually and in good physical approximation) at the gate of a binary device, modes can be *extended* over a spatial region, and possibly *move* across the physical substrate.
- While in DC there are only exactly two modes, there may be an unlimited number of modes in generalized computing systems (Figure 3**C**) – as many as one can realize different "observers" (filters, feature detectors).

Working out these initial ideas into a rigorous mathematical mode construct is ongoing in MINDS, in collaboration with materials scientists, mathematicians and theoretical computer scientists. ESRs 1 and 2 are immersed in this interdisciplinary research, and ESRs 3, 5, 9 and 10 who will spend their first secondments in the MINDS group will likewise be exposed to these investigations, which are more abstract and formal than the experimental research in their original host environments — a challenging exercise and experience in interdisciplinarity.

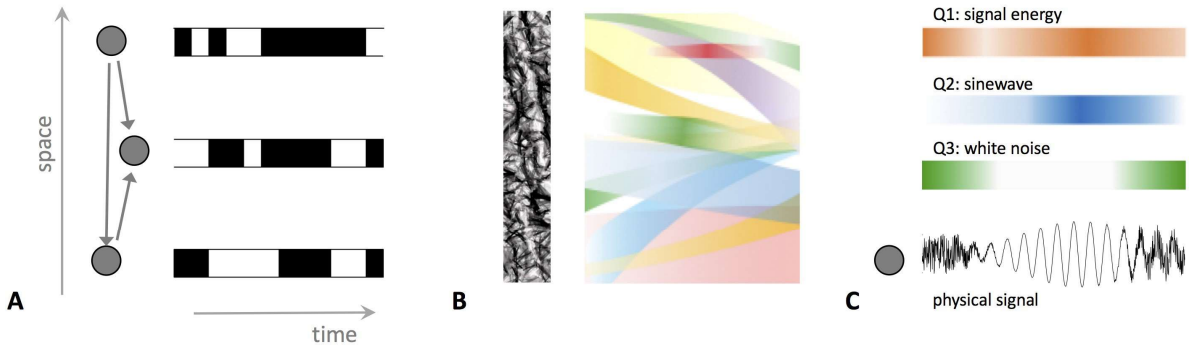

*Figure 3: Digital (**A**) and generalized (**B**,**C**) modes (schematic). **A:** the two digital modes (switching states of a bistable physical device, indicated by black and white) are defined in fixed nodes in a connection graph which represents the "space" of a digital computing system. **B:** in continuously extended computing media, modes may gradually appear and disappear, move, and overlap. — The vertical image bar on the left side is borrowed from materials scientist Beatriz Noheda at the University of Groningen and shows a microscopic image of a self-organized structuring of a ferromagnetic film. Jaeger's MINDS group collaborates with Noheda's team in a joint project where electromagnetic dynamics in such materials are modeled for computational exploits. **C** visualizes that a given physical phenomenon (here: a 1-dimensional temporal signal, e.g. a voltage signal) can instantiate different modes, of which three are indicated. For instance, the "white noise" mode is present with a high intensity only at the beginning and end of the shown episode.*

4 Conclusion

This document is an attempt to highlight the critical gaps in our theoretical and formal understanding and modeling of "computing" in non-digital physical systems. It cannot identify just a few "critical gaps" because there is not yet a global picture of unconventional computing. The views presented are necessarily constrained by the biases of the main authors (ESRs 1 and 2) that reflect their respective backgrounds, their current projects and their still limited exposure to the fields of unconventional computing.

We presented the workable theoretical directions that we can see from the current scattered landscape of unconventional computing. We are inspired by the theoretical underpinnings of digital computing where an interrelation of formal systems allowed humans to collaborate to incrementally build one of the most complex systems with a strong foundation, which is now called "Turing's Cathedral" [Dyson, 2012]. However, we identify sparkling opportunities that call for an investigation of alternative foundations, ones that exploit not only binary switching but any physical phenomena, that allow controlling "wild" systems [Jaeger, 1998] that cannot be entirely controlled and observed but that can be steered toward the right direction. We elaborated one way of unifying complementary approaches structured in bottom-up and top-down interactions. The bottom is physics. We argue that physical phenomena need to be exploited beyond binary switching. The brain is living proof that it is possible to exploit a large repertoire of physical phenomena within the same system. The concept of dynamical "modes" presented in Section 3.8 is a first step in this direction. The top is the cognitive aspects where behavioral analysis of humans and other animals provide landmarks toward which the global behavior of our engineered systems needs to move. In between there is programming: every complex well-functioning artifact needs to be designed, configured, and interacted with. The concept of programming is progressively slipping and needs to be enriched to take into account the substrate phenomena, the cognitive operations it affords, and their progressive automation. Programming takes a biological flavor: it is less about the full control of an isolated system but the engineering of interactions between a cognitive system and its environment.

This approach can help to bring a shared perspective, a common thread by which a productive and strategical research endeavor could start and bring together local research strands in Material Sciences, Photonics, Artificial Intelligence, Computer Science, Mathematics and Cognitive Science.

References

S. Aaronson. *Quantum Computing since Democritus*. Cambridge University Press, 2013. 10.1017/cbo9780511979309.

H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. F. Knight Jr, R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss. Amorphous computing. *Communications of the ACM*, 43(5): 74–82, 2000.

R. H. Abraham and C. D. Shaw. *Dynamics: The Geometry of Behavior*. Addison-Wesley, Redwood City, 1992. e-book at http://www.aerialpress.com/dyn4cd.html.

D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9(1):147–169, jan 1985. 10.1207/s15516709cog0901_7.

A. Adamatzky. Physarum Machine: Implementation of a Kolmogorov-Uspensky Machine on a biological substrate. *Parallel Processing Letters*, 17(04):455–467, dec 2007. 10.1142/s0129626407003150.

G. Agarwal, I. H. Stevenson, A. Berenyi, K. Mizuseki, G. Buzsaki, and F. T. Sommer. Spatially distributed local fields in the hippocampus encode rat position. *Science*, 344(6184):626–630, May 2014. 10.1126/science.1250444. URL https://doi.org/10.1126/science.1250444.

A. Alaghi and J. P. Hayes. Survey of stochastic computing. *ACM Transactions on Embedded Computing Systems*, 12(2s):1–19, may 2013. 10.1145/2465787.2465794.

F. Alet, J. Lopez-Contreras, J. Koppel, M. Nye, A. Solar-Lezama, T. Lozano-Perez, L. Kaelbling, and J. Tenenbaum. A large-scale benchmark for few-shot program induction and synthesis. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 175–186. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/alet21a.html.

L. Alonso and E. Marder. Temperature compensation in a small rhythmic circuit. *Elife*, (9): e55470, 06 2020. 10.7554/elife.55470.sa1.

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the 1967 Spring Joint Computer Conference - AFIPS*. ACM Press, 1967. 10.1145/1465482.1465560.

M. Andraud and M. Verhelst. From on-chip self-healing to self-adaptivity in analog/rf ics: challenges and opportunities. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pages 131–134. IEEE, 2018.

W. R. Ashby. *Design for a Brain*. John Wiley and Sons, New York, 1952.

A. Avižienis. Framework for a taxonomy of fault-tolerance attributes in computer systems. In

Proceedings of the 10th annual international symposium on Computer architecture - ISCA '83. ACM Press, 1983. 10.1145/800046.801633.

J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, editors. *Unconventional Programming Paradigms*. Springer Berlin Heidelberg, 2005. 10.1007/11527800.

J. Beal and M. Viroli. Space–time programming. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 373(2046):20140220, jul 2015. 10.1098/rsta.2014.0220.

T. Becker, O. Mencer, and G. Gaydadjiev. Spatial programming with OpenSPL. In *FPGAs for Software Programmers*, pages 81–95. Springer International Publishing, 2016. 10.1007/978- 3-319-26408-0_5.

T. Behrens, T. Muller, J. Whittington, S. Mark, A. Baram, K. Stachenfeld, and Z. Kurth-Nelson. What is a cognitive map? organizing knowledge for flexible behavior. *Neuron*, 100:490–509, 10 2018. 10.1016/j.neuron.2018.10.002.

T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. R. Voelker, and C. Eliasmith. Nengo: a python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7, 2014. 10.3389/fninf.2013.00048.

Y. Bengio, A. C. Courville, and P. Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, abs/1206.5538, 2012. URL http://arxiv.org/abs/1206.5538.

T. Besold, A. d'Avila Garcez, S. Bader, H. Bowman, P. Domingos, P. Hitzler, K.-U. Kühnberger, L. C. Lamb, D. Lowd, P. Machado Vieira Lima, L. de Penning, G. Pinkas, H. Poon, and G. Zaverucha. Neural-symbolic learning and reasoning: A survey and interpretation. arxiv manuscript, 2017. URL https://arxiv.org/pdf/1711.03902.

B. Bichsel, M. Baader, T. Gehr, and M. Vechev. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, jun 2020. 10.1145/3385412.3386007.

L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin (New Series) of the American Mathematical Society*, 21(1):1–46, 1989.

B. Bläsing, G. Tenenbaum, and T. Schack. The cognitive structure of movements in classical dance. *Psychology of Sport and Exercise*, 10:350–360, 05 2009. 10.1016/j.psychsport.2008.10.001.

K. Boahen. A neuromorph's prospectus. *Computing in Science & Engineering*, 19(2):14–28, mar 2017. 10.1109/mcse.2017.33.

G. Booch. The large-scale structure of software-intensive systems. *Interface Focus*, 2(1):91–100, nov 2011. 10.1098/rsfs.2011.0066.

S. K. Bose, C. P. Lawrence, Z. Liu, K. S. Makarenko, R. M. J. van Damme, H. J. Broersma, and W. G. van der Wiel. Evolution of a designless nanoparticle network into reconfigurable boolean logic. *Nature Nanotechnology*, 10(12):1048–1052, sep 2015. 10.1038/nnano.2015.207.

O. Bournez and A. Pouly. A survey on analog models of computation. May 2018.

S. Bringsjord. *Declarative/Logic-Based Cognitive Modeling*, pages 127–169. Cambridge Handbooks in Psychology. Cambridge University Press, 2008. 10.1017/CBO9780511816772.008.

R. A. Brooks. Intelligence without reason. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'91, page 569–595, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. ISBN 1558601600.

T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL https://arxiv.org/abs/2005.14165.

D. Bucher and J.-M. Goaillard. Beyond faithful conduction: Short-term dynamics, neuromodulation, and long-term regulation of spike propagation in the axon. *Progress in neurobiology*, 94:307–46, 06 2011. 10.1016/j.pneurobio.2011.06.001.

L. Buesing, J. Bill, B. Nessler, and W. Maass. Neural dynamics as sampling: A model for stochastic computation in recurrent networks of spiking neurons. *PLoS Comp. Biol.*, 7(11): e1002211, 2011.

G. Buzsáki. *The Brain from Inside Out*. Oxford University Press, 2019. ISBN 9780190905385. 10.1093/oso/9780190905385.001.0001.

J. Cabessa and H. T. Siegelmann. Evolving recurrent neural networks are super-turing. In *The 2011 International Joint Conference on Neural Networks*. IEEE, jul 2011. 10.1109/ijcnn.2011.6033645.

T. Chen, J. van Gelder, B. van de Ven, S. V. Amitonov, B. de Wilde, H.-C. R. Euler, H. Broersma, P. A. Bobbert, F. A. Zwanenburg, and W. G. van der Wiel. Classification with a disordered dopant-atom network in silicon. *Nature*, 577(7790):341–345, jan 2020. 10.1038/s41586-019-1901-0.

A. Clark. Whatever next? predictive brains, situated agents, and the future of cognitive science. *Behavioral and Brain Sciences*, 36(3):181–204, may 2013. 10.1017/s0140525x12000477.

M. Conrad. The price of programmability. In *A Half-Century Survey on The Universal Turing Machine*, page 285–307, USA, 1988. Oxford University Press, Inc. ISBN 0198537417.

C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, jul 2003. 10.1109/mm.2003.1225959.

S. Coombes. Neural fields. *Scholarpedia*, 1(6):1373, 2006. 10.4249/scholarpedia.1373. revision #138631.

J. Copeland, E. Dresner, D. Proudfoot, and O. Shagrir. Time to reinspect the foundations? *Communications of the ACM*, 59(11):34–38, oct 2016. 10.1145/2908733.

J.-B. Cordonnier, A. Loukas, and M. Jaggi. On the relationship between self-attention and convolutional layers. *arXiv preprint arXiv:1911.03584*, 2019.

T. M. Cover and J. A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, USA, 2006. ISBN 0471241954.

M. Cucchi, C. Gruener, L. Petrauskas, P. Steiner, H. Tseng, A. Fischer, B. Penkovsky, C. Matthus, P. Birkholz, H. Kleemann, and K. Leo. Reservoir computing with biocompati-

ble organic electrochemical networks for brain-inspired biosignal classification. *Science Advances*, 7(34):eabh0693, aug 2021. 10.1126/sciadv.abh0693.

A. d'Avella, P. Saltiel, and E. Bizzi. Combinations of muscle synergies in the construction of a natural motor behavior. *Nature Neuroscience*, 6(3):300–308, 2003.

M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, jan 2018. 10.1109/mm.2018.112130359.

R. Davis, H. E. Shrobe, and P. Szolovits. What is a knowledge representation? *AI Magazine*, 14(1):17–33, 1993. URL http://dblp.uni-trier.de/db/journals/aim/aim14.html#DavisSS93.

L. N. de Castro. Fundamentals of natural computing: an overview. *Physics of Life Reviews*, 4 (1):1–36, mar 2007. 10.1016/j.plrev.2006.10.002.

J. Degrave, M. Hermans, J. Dambre, and F. wyffels. A differentiable physics engine for deep learning in robotics. *Frontiers in Neurorobotics*, 13, mar 2019. 10.3389/fnbot.2019.00006.

F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte Carlo localization for mobile robots. In *Proc. IEEE Int. Conf. on Robotics and Automation*, 1999.

S. Deneve. Bayesian spiking neurons I: Inference. *Neural Computation*, 20(1):91–117, jan 2008. 10.1162/neco.2008.20.1.91.

A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, may 2016. 10.1145/2884781.2884786.

D. Deutsch and C. Marletto. Constructor theory of information. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 471(2174):20140540, 2015.

J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. rahman Mohamed, and P. Kohli. RobustFill: Neural program learning under noisy I/O. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/devlin17a.html.

E. W. Dijkstra. Complexity controlled by hierarchical ordering of function and variability. In *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, pages 181–185, 1968.

E.W. Dijkstra. On the foolishness of "natural language programming". In *Lecture Notes in Computer Science*, pages 51–53. Springer-Verlag, 1979. 10.1007/bfb0014656.

F. Duport, B. Schneider, A. Smerieri, M. Haelterman, and S. Massar. All-optical reservoir computing. *Optics Express*, 20(20):22783, sep 2012. 10.1364/oe.20.022783.

G. Dyson. *Turing's Cathedral: The Origins of the Digital Universe (Vintage)*. Vintage Books, 2012. ISBN 1400075998.

C. Edwards. Moore's law. *Communications of the ACM*, 64(2):12–14, jan 2021. 10.1145/3440992.

C. Eliasmith. A unified approach to building and controlling spiking attractor networks. *Neural Computation*, 17:1276–1314, 2005.

C. Eliasmith. How to build a brain: A neural architecture for biological cognition. Oxford University Press, 2013.

C. Eliasmith, S. T. C., C. X., B. T., T. Y. DeWolf T., and D. Rasmussen. A large-scale model of the functioning brain. *Science*, 338(6111):1202–1205, 2012a.

C. Eliasmith, T. C. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, and D. Rasmussen. A large-scale model of the functioning brain. *Science*, 338(6111):1202–1205, 2012b.

J. Endrullis, J. W. Klop, and R. Bakhshi. Transducer degrees: atoms, infima and suprema. *Acta Informatica*, 57(3-5):727–758, 2019.

D. Erhan, A. Courville, Y. Bengio, and P. Vincent. Why does unsupervised pre-training help deep learning? In Y. W. Teh and M. Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 201–208, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL https://proceedings.mlr.press/v9/erhan10a.html.

B. Farhang-Boroujeny. *Adaptive Filters: Theory and Applications*. Wiley, 1998.

G. Fauconnier. Conceptual blending and analogy. *The Analogical Mind: Perspectives from Cognitive Science*, pages 255–285, 01 2001.

J. Feldman. The neural binding problem(s). *Cognitive Neurodynamics*, 7, 02 2013. 10.1007/s11571-012-9219-8.

D. J. Felleman and D. C. Van Essen. Distributed hierarchical processing in the primate cerebral cortex. In *Cereb cortex*. Citeseer, 1991.

R. FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical journal*, 1(6):445–466, 1961.

K. Forbus, J. Usher, A. Lovett, K. Lockwood, and J. Wetzel. Cogsketch: Sketch understanding for cognitive science research and for education. *Topics in Cognitive Science*, 3(4):648–666, 2011. https://doi.org/10.1111/j.1756-8765.2011.01149.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1756-8765.2011.01149.x.

K. D. Forbus. Qualitative physics: past, present and future. In *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, pages 239–296. Morgan Kaufmann, 1988.

S. Forrest. Emergent computation: self-organizing, collective, and cooperative phenomena in natural and artificial computing networks. *Physica D*, 42:1–11, 1990.

W. J. Freeman, R. Kozma, and P. J. Werbos. Biocomplexity: adaptive behavior in complex stochastic dynamical systems. *Biosystems*, 59(2):109–123, 2001.

R. French. The dynamics of the computational modeling of analogy-making. In P. A. Fishwick, editor, *Handbook of Dynamic System Modeling*, chapter 2. Chapman & Hall, 2006.

K. Friston. A theory of cortical response. *Phil. Trans. R. Soc. B*, 360:815–836, 2005.

K. J. Friston, J. Daunizeau, J. Kilner, and S. J. Kiebel. Action and behavior: a free-energy formulation. *Biological Cybernetics*, 102(3):227–260, 2010.

K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 02 1980. 10.1007/BF00344251.

J. Gama, v. Indrė, A. Bifet, M. Pechennizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys*, 1(1):1–35, 2013.

J.-P. Georgé, M.-P. Gleizes, and P. Glize. Experiments in neo-computation based on emergent programming. In *Multiagent System Technologies*, pages 237–239. Springer Berlin Heidelberg, 2005. 10.1007/11550648_24.

S. George, S. Kim, S. Shah, J. Hasler, M. Collins, F. Adil, R. Wunderlich, S. Nease, and S. Ramakrishnan. A programmable and configurable mixed-mode FPAA SoC. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–9, 2016. 10.1109/tvlsi.2015.2504119.

A. P. Georgopoulos, A. B. Schwartz, and R. E. Kettner. Neuronal population coding of movement direction. *Science*, 233(4771):1416–1419, 1986. 10.1126/science.3749885. URL https://www.science.org/doi/abs/10.1126/science.3749885.

J. J. Gibson. *The Ecological Approach to Visual Perception*. Houghton Mifflin, Boston, 1979.

M.-P. Gleizes, V. Camps, and P. Glize. A theory of emergent computation based on cooperative self-organization for adaptive artificial systems. In *Fourth European Congress of Systems Science*, pages 20–24, 1999.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Future of Software Engineering Proceedings*. ACM, may 2014. 10.1145/2593882.2593900.

A. Goyal and Y. Bengio. Inductive biases for deep learning of higher-level cognition. *CoRR*, abs/2011.15091, 2020. URL https://arxiv.org/abs/2011.15091.

A. Graves, G. Wayne, and I. Danihelka. Neural Turing machines. 2014. arXiv 1410.5401.

A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. P. Badia, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu, and D. Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538 (7626):471–476, Oct. 2016. ISSN 00280836. URL http://dx.doi.org/10.1038/nature20101.

S. Grossberg. Adaptive resonance theory. *Scholarpedia*, 8(5):1569, 2013.

G. Grünert. *Unconventional programming: non-programmable systems*. PhD thesis, Friedrich-Schiller-Universität Jena, 2017. URL https://www.db-thueringen.de/receive/dbt_mods_00031925.

S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017. 10.1561/2500000010.

Y. Guo, X. Zou, Y. Hu, Y. Yang, X. Wang, Y. He, R. Kong, Y. Guo, G. Li, W. Zhang, S. Wu, and H. Li. A Marr's three-level analytical framework for neuromorphic electronic systems. *Advanced Intelligent Systems*, page 2100054, 2021.

G. Haessig, A. Cassidy, R. Alvarez, R. Benosman, and G. Orchard. Spiking optical flow for event-based sensors using IBM's TrueNorth neurosynaptic system. *IEEE Trans. on Biomedical Circuits and Systems*, 12(4):860–870, 2018.

M. S. Hammoodi, F. Stahl, and A. Badii. Real-time feature selection technique with concept drift detection using adaptive micro-clusters for data stream mining. *Knowledge-Based Systems*, 161:205–239, 2018.

S. Harnad. Symbol grounding problem. *Scholarpedia*, 2(7):2373, 2007. 10.4249/scholarpedia.2373. revision #73220.

J. Hasler. Large-scale field-programmable analog arrays. *Proceedings of the IEEE*, 108(8): 1283–1302, aug 2020a. 10.1109/jproc.2019.2950173.

J. Hasler. Defining analog standard cell libraries for mixed-signal computing enabled through educational directions. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, oct 2020b. 10.1109/iscas45731.2020.9181124.

J. Hasler and E. Black. Physical computing: Unifying real number computation to enable energy efficient computing. *Journal of Low Power Electronics and Applications*, 11(2):14, mar 2021. 10.3390/jlpea11020014.

J. Hasler and B. Marr. Finding a roadmap to achieve large neuromorphic hardware systems. *Frontiers in Neuroscience*, 7, 2013. 10.3389/fnins.2013.00118.

X. He, T. Liu, F. Hadaeghi, and H. Jaeger. Reservoir transfer on analog neuromorphic hardware. In *Proc. 9th Int. IEEE/EMBS Conf. on Neural Engineering*, pages 1234–1238, 2019.

J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, jan 2019. 10.1145/3282307.

G. E. Hinton. How to represent part-whole hierarchies in a neural network. *CoRR*, abs/2102.12627, 2021. URL https://arxiv.org/abs/2102.12627.

A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500, 1952.

D. R. Hofstadter. *Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought*. Basic Books, Inc., New York, NY, USA, 1996. ISBN 0465024750.

N. Hogan and T. Flash. Moving gracefully: quantitative theories of motor coordination. *Trends in Neuroscience*, 10(4):170–174, 1987.

S. Hooker. The hardware lottery. *arXiv preprint arXiv:2009.06489*, 2020.

J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79:2554–8, 05 1982. 10.1073/pnas.79.8.2554.

C. Horsman, S. Stepney, R. C. Wagner, and V. Kendon. When does a physical system compute? *Proc. of the Royal Society A*, 470:20140182., 2014.

C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *Int. J. of Approximate Reasoning*, 11(1):158, 1994.

G. Indiveri. Introducing 'neuromorphic computing and engineering'. *Neuromorphic Computing and Engineering*, 1(1):010401, jul 2021. 10.1088/2634-4386/ac0a5b.

G. Indiveri, B. Linares-Barranco, T. J. Hamilton, A. van Schaik, R. Etienne-Cummings, T. Delbruck, S.-C. Liu, P. Dudek, P. Häflinger, S. Renaud, J. Schemmel, C. Cauwenberghs, J. Arthur, K. Hynna, F. Folowosele, S. Saighi, T. Serrano-Gotarredona, J. Wijekoon, Y. Wang,

and K. Boahen. Neuromorphic silicon neuron circuits. *Frontiers in Neuroscience*, 5:article 73, 2011.

M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt. A differentiable programming system to bridge machine learning and scientific computing, 2019. arXiv 1907.07587.

E. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, 2004. 10.1109/TNN.2004.832719.

E. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*. MIT Press, Cambridge MA, 2007.

E. M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14 (6):1569–1572, 2003.

E. M. Izhikevich. Polychronization: Computation with spikes. *Neural Computation*, 18(2):245–282, feb 2006. 10.1162/089976606775093882.

H. Jaeger. Today's dynamical systems are too simple. Commentary to Tim van Gelder's "The dynamical hypothesis in cognitive science". *Behavioral and Brain Sciences*, 21(5):643, 1998.

H. Jaeger. Observable operator models for discrete stochastic time series. *Neural Computation*, 12(6):1371–1398, jun 2000. 10.1162/089976600300015411.

H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148(34):13, 2001.

H. Jaeger. Controlling recurrent neural networks by conceptors. Technical Report 31, Jacobs University Bremen, 2014. arXiv:1403.3369.

H. Jaeger. Principles of statistical modeling. Lecture notes, Jacobs University Bremen, 2019. https://www.ai.rug.nl/minds/uploads/LN_PSM.pdf.

H. Jaeger. Toward a generalized theory comprising digital, neuromorphic, and unconventional computing. *Neuromorphic Computing and Engineering*, 2021. 10.1088/2634-4386/abf151.

H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304:78–80, 2004.

H. Jaeger, D. Doorakkers, C. Lawrence, and G. Indiveri. Dimensions of "timescales" in neuromorphic computing systems (deliverable report for the European Union project MeM-Scales). https://arxiv.org/abs/2102.10648, 2021.

A. Jaegle, F. Gimeno, A. Brock, A. Zisserman, O. Vinyals, and J. Carreira. Perceiver: General perception with iterative attention. *CoRR*, abs/2103.03206, 2021. URL https://arxiv.org/abs/2103.03206.

E. T. Jaynes. *Probability Theory: the Logic of Science*. Cambridge University Press, 2003. First partial online editions in the late 1990ies. First three chapters online at http://bayes.wustl.edu/etj/prob/book.pdf.

I. S. Jones and K. P. Kording. Can single neurons solve MNIST? the computational power of biological dendritic trees, 2020. arXiv 2009.01269.

P. Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, 1(2):139–159, 2009.

N. Kant. Recent advances in neural program synthesis, 2018. URL :http\protect\leavevmode@ifvmode\kern+.2222em\relax//arxiv.org/pdf/1802.02353v1:PDF. arXiv 1802.02353.

A. Karmiloff-Smith. *Beyond Modularity: A Developmental Perspective on Cognitive Science*. 01 1995. ISBN 9780262276740. 10.7551/mitpress/1579.001.0001.

A. Karpathy. Software 2.0. 2017. URL https://karpathy.medium.com.

S. Kiebel, J. Daunizeau, and K. Friston. A hierarchy of time-scales and the brain. *PLoS Computational Biology*, **4**(11):e1000209, 2008.

L. B. Kish. Quantum computing with analog circuits: Hilbert space computing. In V. K. Varadan and L. B. Kish, editors, *Smart Structures and Materials 2003: Smart Electronics, MEMS, BioMEMS, and Nanotechnology*. SPIE, jul 2003. 10.1117/12.497438.

K. Kitajo, D. Nozaki, L. M. Ward, and Y. Yamamoto. Behavioral stochastic resonance within the human brain. *Physical Review Letters*, 90(21):218103, 2003.

H. Kitano. Biological robustness. *Nature Reviews Genetics*, 5(11):826–837, nov 2004. 10.1038/nrg1471.

E. Kobatake and K. Tanaka. Neuronal selectivities to complex object features in the ventral visual pathway of the macaque cerebral cortex. *Journal of neurophysiology*, 71:856–67, 04 1994. 10.1152/jn.1994.71.3.856.

K. Koepsell, X. Wang, V. Vaingankar, Y. Wei, Q. Wang, D. Rathbun, M. Usrey, J. Hirsch, and F. Sommer. Retinal oscillations carry visual information to cortex. *Frontiers in Systems Neuroscience*, 3:4, 2009. ISSN 1662-5137. 10.3389/neuro.06.004.2009. URL https://www.frontiersin.org/article/10.3389/neuro.06.004.2009.

T. Kohonen and T. Honkela. Kohonen network. *Scholarpedia*, 2(1):1568, 2007. 10.4249/scholarpedia.1568. revision #127841.

I. Kotseruba, O. J. A. Gonzalez, and J. K. Tsotsos. A review of 40 years of cognitive architecture research: Focus on perception, attention, learning and applications. *CoRR*, abs/1610.08602, 2016. URL http://dblp.uni-trier.de/db/journals/corr/corr1610.html#KotserubaGT16.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

T. D. Kulkarni, P. Kohli, J. B. Tenenbaum, and V. Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

M. Kunda. Visual mental imagery: A view from artificial intelligence. *Cortex*, 105, 02 2018. 10.1016/j.cortex.2018.01.022.

B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, dec 2015. 10.1126/science.aab3050.

G. Lakoff. *Women, Fire and Dangerous Things: What Categories Reveal About the Mind*. University of Chicago Press, Chicago, 1987. ISBN 978-0-226-46803-7.

J. Larus. Spending moore's dividend. *Communications of the ACM*, 52(5):62–69, may 2009. 10.1145/1506409.1506425.

J. W. Lawson and D. H. Wolpert. Adaptive programming of unconventional nano-architectures. *Journal of Computational and Theoretical Nanoscience*, 3(2):272–279, apr 2006. 10.1166/jctn.2006.3009.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, may 2015. 10.1038/nature14539.

D. B. Lenat. CYC - a large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, nov 1995. 10.1145/219717.219745.

Y. Li, Z. Wang, R. Midya, Q. Xia, and J. J. Yang. Review of memristor devices in neuromorphic computing: materials sciences and device challenges. *Journal of Physics D: Applied Physics*, 51(50):503002, sep 2018. 10.1088/1361-6463/aade3f.

M. L. Littman, R. S. Sutton, and S. Singh. Predictive representations of state. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, page 1555–1561, Cambridge, MA, USA, 2001. MIT Press.

W. Lotter, G. Kreiman, and D. D. Cox. Deep predictive coding networks for video prediction and unsupervised learning. *CoRR*, abs/1605.08104, 2016. URL http://arxiv.org/abs/1605.08104.

W. Maass. Noise as a resource for computation and learning in networks of spiking neurons. *Proceedings of the IEEE*, 102(5):860–880, 2014.

W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11): 2531–2560, 2002. http://www.cis.tugraz.at/igi/maass/psfiles/LSM-v106.pdf.

Z. Mainen and T. Sejnowski. Reliability of spike timing in neocortical neurons. *Science*, 268: 1503–1505, 07 1995. 10.1126/science.7770778.

D. Marković, A. Mizrahi, D. Querlioz, and J. Grollier. Physics for neuromorphic computing. *Nature Reviews Physics*, 2(9):499–510, 2020.

D. Marr. *Vision. A Computational Investigation into the Human Representation and Processing of Visual Information*. Freeman, 1980.

M. Mastella and E. Chicca. A hardware-friendly neuromorphic spiking neural network for frequency detection and fine texture decoding. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, may 2021. 10.1109/iscas51556.2021.9401377.

J. L. McClelland, F. Hill, M. Rudolph, J. Baldridge, and H. Schütze. Extending machine language models toward human-level language understanding. *CoRR*, abs/1912.05877, 2019. URL http://arxiv.org/abs/1912.05877.

W. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

C. Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636, 1990.

T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013. URL http://arxiv.org/abs/1310.4546.

R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):78–89, jan 1993. 10.1145/151233.151240.

R. Milner. Turing, computing and communication. In *Interactive Computation*, pages 1–8. Springer Berlin Heidelberg, 2006. 10.1007/3-540-34874-3_1.

M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., USA, 1967. ISBN 0131655639.

M. Mitchell. Abstraction and analogy-making in artificial intelligence. *CoRR*, abs/2102.10717, 2021. URL https://arxiv.org/abs/2102.10717.

M. Mitchell, J. P. Crutchfield, and P. T. Hraber. Evolving cellular automata to perform computations: mechanisms and impediments. *Physica D: Nonlinear Phenomena*, 75(1-3):361–391, aug 1994. 10.1016/0167-2789(94)90293-3.

J. H. Moor. Three myths of computer science. *The British Journal for the Philosophy of Science*, 29(3):213–222, sep 1978. 10.1093/bjps/29.3.213.

C. Moore. Recursion theory on the reals and continuous-time computation. *Theoretical Computer Science*, 162(1):23–44, aug 1996. 10.1016/0304-3975(95)00248-0.

S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps). *IEEE Transactions on Biomedical Circuits and Systems*, 12(1):106–122, 2018.

C. Morris and H. Lecar. Voltage oscillations in the barnacle giant muscle fiber. *Biophysical journal*, 35(1):193–213, 1981.

D. Mumford. Pattern theory: a unifying perspective. In A. Joseph, F. Mignot, F. Murat, B. Prum, and R. Rentschler, editors, *Proc. of First European Congress of Mathematics, Vol. I, Invited Lectures Part 1*, number 3 in Progress in Mathematics, pages 187–224. Birkhäuser, Basel, 1994.

D. Mumford. Pattern theory: The mathematics of perception. In *Proc. ICM 2002, Vol. 1*, pages 401–422, 2002. https://arxiv.org/abs/math/0212400.

K. P. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, Univ. of California, Berkeley, 2002.

M. Musisi-Nkambwe, S. Afshari, H. Barnaby, M. Kozicki, and I. S. Esqueda. The viability of analog-based accelerators for neuromorphic computing: a survey. *Neuromorphic Computing and Engineering*, 1(1):012001, jul 2021. 10.1088/2634-4386/ac0242.

J. Nagumo, S. Arimoto, and S. Yoshizawa. An active pulse transmission line simulating nerve axon. *Proceedings of the IRE*, 50(10):2061–2070, 1962.

R. Neal. Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Dpt. of Computer Science, University of Toronto, 1993.

A. Neckar, S. Fok, B. V. Benjamin, T. C. Stewart, N. N. Oza, A. R. Voelker, C. Eliasmith, R. Manohar, and K. Boahen. Braindrop: A mixed-signal neuromorphic architecture with a dynamical systems-based programming model. *Proc of the IEEE*, 107(1):144–164, 2019.

E. Neftci, J. Binas, U. Rutishauser, E. Chicca, G. Indiveri, and R. J. Douglas. Synthesizing cognition in neuromorphic electronic systems. *Proceedings of the National Academy of Sciences*, 110(37):E3468–E3476, jul 2013. 10.1073/pnas.1212083110.

A. Newell. *Unified theories of cognition*. Harvard University Press, 1990.

A. Newell and H. A. Simon. Computer science as empirical inquiry: symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

C. Niell and M. Stryker. Modulation of visual responses by behavioral state in mouse visual cortex. *Neuron*, 65:472–9, 02 2010. 10.1016/j.neuron.2010.01.033.

W. Olin-Ammentorp, K. Beckmann, C. D. Schuman, J. S. Plank, and N. C. Cady. Stochasticity and robustness in spiking neural networks. *Neurocomputing*, 419:23–36, 2021.

B. Olshausen, C. Anderson, and D. Van Essen. A neurobiological model of visual attention and invariant pattern recognition based on dynamic routing of information. *Journal of Neuroscience*, 13(11):4700–4719, 1993. ISSN 0270-6474. 10.1523/JNEUROSCI.13-11-04700.1993. URL https://www.jneurosci.org/content/13/11/4700.

B. A. Olshausen and D. J. Field. Sparse coding of sensory inputs. *Current Opinion in Neurobiology*, 14(4):481–487, 2004. 10.1016/j.conb.2004.07.007.

E. Ott. *Chaos in dynamical systems*. Cambridge university press, 2002.

T. Oya, T. Asai, and Y. Amemiya. Stochastic resonance in an ensemble of single-electron neuromorphic devices and its application to competitive neural networks. *Chaos, Solitons & Fractals*, 32(2):855–861, 2007.

Y. Paquot, F. Duport, A. Smerieri, J. Dambre, B. Schrauwen, M. Haelterman, and S. Massar. Optoelectronic reservoir computing. *Scientific Reports*, 2(1), feb 2012. 10.1038/srep00287.

M. Parashar and S. Hariri. Autonomic computing: An overview. In *International workshop on unconventional programming paradigms*, Lecture Notes in Computer Science, pages 257–269. Springer Berlin Heidelberg, 2005. 10.1007/11527800_20.

D. L. Parnas. Software aspects of strategic defense systems. *Communications of the ACM*, 28 (12):1326–1335, dec 1985. 10.1145/214956.214961.

L. Pecevski, D.and Büsing and W. Maass. Probabilistic inference in general graphical models through sampling in stochastic networks of spiking neurons. *PLoS Comp. Biol.*, 7(12): e1002294, 2011.

C. Petri. *Kommunikation mit Automaten*. Dissertation thesis, University of Bonn, Institute of Mathematics, 1962.

M. A. Petrovici, A. Schroeder, O. Breitwieser, A. Grübl, J. Schemmel, and K. Meier. Robustness from structure: inference with hierarchical spiking networks on analog neuromorphic hardware. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2209–2216. IEEE, 2017.

G. Primiero. Information in the philosophy of computer science. In *The Routledge Handbook of Philosophy of Information*, pages 108–125. Routledge, 2016. ISBN 9780367370466. https://doi.org/10.4324/9781315757544.

M. I. Rabinovich, R. Huerta, P. Varona, and V. S. Afraimovich. Transient cognitive dynamics, metastability, and decision making. *PLOS Computational Biology*, 4(5):e1000072, 2008.

A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. Zero-shot text-to-image generation. *CoRR*, abs/2102.12092, 2021. URL https://arxiv.org/abs/2102.12092.

H. Ramsauer, B. Schäfl, J. Lehner, P. Seidl, M. Widrich, T. Adler, L. Gruber, M. Holzleitner, M. Pavlović, G. K. Sandve, et al. Hopfield networks is all you need. *arXiv preprint arXiv:2008.02217*, 2020.

A. S. Rekhi, B. Zimmer, N. Nedovic, N. Liu, R. Venkatesan, M. Wang, B. Khailany, W. J. Dally, and C. T. Gray. Analog/mixed-signal hardware error modeling for deep learning inference. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

E. T. Rolls and G. Deco. *The Noisy Brain - Stochastic Dynamics as a Principle of Brain Function*. Oxford University Press, jan 2010. 10.1093/acprof:oso/9780199587865.001.0001.

F. Rosenblatt. The Perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

L. A. Rubel. The extended analog computer. *Advances in Applied Mathematics*, 14(1):39–50, 1993.

H.-C. Ruiz-Euler, U. Alegre-Ibarra, B. van de Ven, H. Broersma, P. A. Bobbert, and W. G. van der Wiel. Dopant network processing units: Towards efficient neural-network emulators with high-capacity nanoelectronic nodes, 2020. arXiv 2007.12371.

R. Sarpeshkar. Analog versus digital: Extrapolating from electronics to neurobiology. *Neural Computation*, 10(7):1601–1638, oct 1998. 10.1162/089976698300017052.

G. Schöner. The dynamics of neural populations capture the laws of the mind. *Topics in Cognitive Science*, pages 1–15, 2019.

G. Schöner, J. Spencer, and D. Group. *Dynamic Thinking: A Primer on Dynamic Field Theory*. 11 2015. ISBN 9780199300563. 10.1093/acprof:oso/9780199300563.001.0001.

T. J. Sejnowski. *The Deep Learning Revolution*. The MIT Press, 2018. ISBN 9780262038034.

N. Semenova, X. Porte, L. Andreoli, M. Jacquot, L. Larger, and D. Brunner. Fundamental aspects of noise in analog-hardware neural networks. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 29(10):103128, 2019.

N. Semenova, L. Larger, and D. Brunner. The general aspects of noise in analogue hardware deep neural networks. *arXiv preprint arXiv:2103.07413*, 2021.

C. E. Shannon. Mathematical theory of the differential analyzer. *Journal of Mathematics and Physics*, 20(1-4):337–354, apr 1941. 10.1002/sapm1941201337.

S. M. Sherman and R. W. Guillery. Distinct functions for direct and transthalamic corticocortical connections. *Journal of Neurophysiology*, 106(3):1068–1077, 2011. 10.1152/jn.00429.2011. URL https://doi.org/10.1152/jn.00429.2011. PMID: 21676936.

P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc. Press, 1994. 10.1109/sfcs.1994.365700.

H. Siegelmann and E. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, feb 1995. 10.1006/jcss.1995.1013.

H. T. Siegelmann and E. D. Sontag. Analog computation via neural networks. *Theoretical Computer Science*, 131(2):331–360, sep 1994. ISSN 0304-3975. 10.1016/0304-3975(94)90178-3.

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419): 1140–1144, dec 2018. 10.1126/science.aar6404.

H. A. Simon. The architecture of complexity. In *Facets of Systems Science*, pages 457–476. Springer US, 1991. 10.1007/978-1-4899-0718-9_31.

H. A. Simon and J. Laird. *Sciences of the Artificial*. The MIT Press, Aug. 2019. ISBN 0262537532. URL https://www.ebook.de/de/product/36034287/herbert_a_simon_john_e_laird_sciences_of_the_artificial.html.

C. A. Skarda and W. J. Freeman. How brains make chaos in order to make sense of the world. *Behavioral and brain sciences*, 10(2):161–173, 1987.

A. Sloman. The irrelevance of Turing machines to artificial intelligence. In M. Scheutz, editor, *Computationalism: New Directions*. MIT Press, 2002.

A. Sloman. Architectural and representational requirements for seeing processes, proto-affordances and affordances. In *Logic and Probability for Scene Interpretation*, 2008.

L. Smith and E. Thelen, editors. *A Dynamic Systems Approach to Development: Applications*. Bradford/MIT Press, Cambridge, Mass., 1993.

J. Spolsky. The law of leaky abstractions. In *Joel on Software*, pages 197–202. Apress, 2004. 10.1007/978-1-4302-0753-5_26.

L. Steels and R. Brooks, editors. *Building Agents out of Autonomous Behavior Systems*. Lawrence Erlbaum, 1993.

S. Stepney. Unconventional computer programming. In *AISB/IACAP World Congress 2012 on Natural Computing / Unconventional Computing and its Philosophical Significance*, page 12, 2012.

S. Stepney and S. Hickinbotham. UCOMP roadmap: Survey, challenges, recommendations. In S. Stepney, S. Rasmussen, and M. Amos, editors, *Computational Matter*, chapter 2, pages 9–32. Springer Verlag, 2018.

E. Stromatias, D. Neil, M. Pfeiffer, F. Galluppi, S. B. Furber, and S.-C. Liu. Robustness of spiking deep belief networks to noise and reduced bit precision of neuro-inspired hardware platforms. *Frontiers in neuroscience*, 9:222, 2015.

S.-H. Sun, H. Noh, S. Somasundaram, and J. Lim. Neural program synthesis from diverse demonstration videos. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4790–4799. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/sun18a.html.

I. Sutskever, G. E. Hinton, and G. W. Taylor. The recurrent temporal restricted Boltzmann machine. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21 (NIPS 08)*, pages 1601–1608, 2009.

T. Tanaka, G. abnd Yamane, J. Héroux, R. Nakane, N. Kanazawa, S. Takeda, H. Numata, D. Nakano, and A. Hirose. Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100–123, 2019. preprint in https://arxiv.org/abs/1808.04962.

J. Tenenbaum, T. L. Griffiths, and C. Kemp. Theory-based Bayesian models of inductive learning and reasoning. *Trends in Cognitive Science*, 10(7):309–318, 2006.

D. Tervo, J. Tenenbaum, and S. Gershman. Toward the neural implementation of structure learning. *Current Opinion in Neurobiology*, 37:99–105, 04 2016. 10.1016/j.conb.2016.01.014.

K. A. Thoroughman and R. Shadmehr. Learning of action through adaptive combination of motor primitives. *Nature*, 407(Oct. 12):742–747, 2000.

S. Thorpe, A. Delorme, and R. V. Rullen. Spike-based strategies for rapid processing. *Neural Networks*, 14(6-7):715–725, jul 2001. 10.1016/s0893-6080(01)00083-1.

J. Torrejon, M. Riou, F. A. Araujo, S. Tsunegi, G. Khalsa, D. Querlioz, P. Bortolotti, V. Cros, K. Yakushiji, A. Fukushima, H. Kubota, S. Yuasa, M. D. Stiles, and J. Grollier. Neuromorphic computing with nanoscale spintronic oscillators. *Nature*, 547(7664):428–431, jul 2017. 10.1038/nature23011.

M. Treinish, J. Gambetta, P. Nation, P. Kassebaum, Qiskit-Bot, D. M. Rodríguez, S. De La Puente González, Shaohan Hu, K. Krsulich, L. Zdanski, J. Yu, J. Gacon, D. McKay, J. Gomez, L. Capelluto, Travis-S-IBM, A. Panigrahi, Lerongil, Rafey Iqbal Rahman, S. Wood, L. Bello, Divyanshu Singh, , Drew, J. Schwarm, MELVIN GEORGE, M. Marques, O. C. Hamido, RohitMidha23, S. Dague, and S. Garion. Qiskit: An open-source framework for quantum computing, 2021.

A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc.*, 42(2):230–265, 1936.

A. M. Turing. Computing machinery and intelligence. *Mind*, LIX(236):433–460, Oct. 1950. 10.1093/mind/lix.236.433.

L. Valiant. *Probably Approximately Correct: Nature's Algorithms for Learning and Prospering in a Complex World*. BASIC BOOKS, June 2013. ISBN 0465032710.

L. G. Valiant. Robust logics. *Artificial Intelligence*, 117(2):231–253, mar 2000. 10.1016/s0004-3702(00)00002-3.

L. G. Valiant. Holographic algorithms. *SIAM Journal on Computing*, 37(5):1565–1594, jan 2008. 10.1137/070682575.

T. van Gelder and R. Port, editors. *Mind as Motion: Explorations in the Dynamics of Cognition*. Bradford/MIT Press, 1995.

J. van Leeuwen and J. Wiedermann. Beyond the turing limit: Evolving interactive systems. In *International Conference on Current Trends in Theory and Practice of Computer Science*, number 2234 in LNCS, pages 90–109. Springer, 2001.

D. van Noort, F.-U. Gast, and J. S. McCaskill. DNA computing in microreactors. In N. Jonoska and N. C. Seeman, editors, *DNA7*, volume 2340 of *LNCS*, pages 33–45. Springer Verlag, 2002.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

L. von Bertalanffy. *General System Theory*. Braziller, N.Y., 1968.

J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, 34:43–98, 1956.

M. M. Waldrop. The chips are down for Moore's law. *Nature*, 530(7589):144–147, feb 2016. 10.1038/530144a.

J. X. Wang, Z. Kurth-Nelson, D. Kumaran, D. Tirumala, H. Soyer, J. Z. Leibo, D. Hassabis, and M. M. Botvinick. Prefrontal cortex as a meta-reinforcement learning system. *Nature Neuroscience*, 21:860–868, 2018.

W. Weaver. Science and complexity. *American Scientist*, 36(4):536–544, 1948. ISSN 00030996. URL http://www.jstor.org/stable/27826254.

P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40 (5):80–91, may 1997. 10.1145/253769.253801.

P. Wegner and D. Goldin. Computation beyond turing machines. *Communications of the ACM*, 46(4):100–102, apr 2003. 10.1145/641205.641235.

J. Weis, P. Spilger, S. Billaudelle, Y. Stradmann, A. Emmel, E. Müller, O. Breitwieser, A. Grübl, J. Ilmberger, V. Karasenko, et al. Inference with artificial neural networks on analog neuromorphic hardware. In *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*, pages 201–212. Springer, 2020.

N. Wiener. *Cybernetics, or control and communication in the animal and the machine*. MIT Press, 1948.

S. Wolfram. *A Project to Find the Fundamental Theory of Physics*. Wolfram Media, Inc., 2020. Core contents online at https://www.wolframphysics.org/technical-introduction/.

D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. 10.1109/4235.585893.

R. Woods and G. Lightbody. Robustness in digital hardware. In *Robust Intelligent Systems*, pages 3–21. Springer London, 2008. 10.1007/978-1-84800-261-6_1.

G. Wunsch. *Geschichte der Systemtheorie (in German)*. Oldenbourg Verlag München, 1985.

Z. Ying, C. Feng, Z. Zhao, S. Dhar, H. Dalir, J. Gu, Y. Cheng, R. Soref, D. Z. Pan, and R. T. Chen. Electronic-photonic arithmetic logic unit for high-speed computing. *Nature Communications*, 11(1), may 2020. 10.1038/s41467-020-16057-3.

L. A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Communications of the ACM*, 37(3):77–84, mar 1994. 10.1145/175247.175255.

Y. Zhang, P. Qu, Y. Ji, W. Zhang, G. Gao, G. Wang, S. Song, G. Li, W. Chen, W. Zheng, F. Chen, J. Pei, R. Zhao, M. Zhao, and L. Shi. A system hierarchy for brain-inspired computing. *Nature*, 586(15 Oct):378–384, 2020.

C. Zhou, P. Kadambi, M. Mattina, and P. N. Whatmough. Noisy machines: Understanding noisy neural networks and enhancing robustness to analog hardware errors using distillation. *arXiv preprint arXiv:2001.04974*, 2020.