Università Politecnica delle Marche
Scuola di Dottorato di Ricerca in Scienze dell'Ingegneria
Curriculum in Ingegneria Informatica, Gestionale e dell'Automazione

---

From Symbolic Artificial Intelligence to Neural Networks Universality with Event-based Modeling

Ph.D. Dissertation of:
Nicola Falcionelli

Advisor:
Prof. Aldo Franco Dragoni

Curriculum Supervisor:
Prof. Francesco Piazza

XVIII edition - new series

Università Politecnica delle Marche
Scuola di Dottorato di Ricerca in Scienze dell'Ingegneria
Curriculum in Ingegneria Informatica, Gestionale e dell'Automazione

---

From Symbolic Artificial Intelligence to Neural Networks Universality with Event-based Modeling

Ph.D. Dissertation of:
Nicola Falcionelli

Advisor:
Prof. Aldo Franco Dragoni

Curriculum Supervisor:
Prof. Francesco Piazza

XVIII edition - new series

Università Politecnica delle Marche
Scuola di Dottorato di Ricerca in Scienze dell'Ingegneria
Facoltà di Ingegneria
Via Brecce Bianche – 60131 Ancona (AN), Italy

Acknowledgments

To my supervisor prof. Aldo Franco Dragoni, for his passionate teaching, his ability to inspire people, and for his guidance throughout my academic journey;

to Stefano Bromuri and Jérémie Cabessa, for the helpful comments, for giving me the chance to mature my ideas and to improve my work;

to my high school professor Francesco Discepoli, for being much more than a teacher, and for igniting my passion in computer science;

to Paolo Sernani, for his selflessness, authenticity, technical knowledge, wise advices, and for being the mentor I needed, but not the one I deserved;

to Dagmawi Neway Mekuria, for the exciting discussions, amazing cultural exchanges, and for sharing with me the path through this Ph.D.;

to my office mates Marina Paolanti, Selene Tomassini, Luca Romeo, Riccardo Rosati and the other members of InterviewLab, for sharing and supporting each other in everyday's life joys and pains;

to Michela Paolucci, for always being there, for the unconditional support, for always believing in me, and for giving me the courage of throwing my heart over the hurdle;

to my parents, my sister, my grandparents, and all my family, for making me the person I am today, for never forcing me in any of my choices, for their patience and sacrifices;

to Mirco Zingaretti, Filippo Bedoni, Jimmie Santoni, and to all my friends, for making my life more colorful and enjoyable ever since I can remember;

to all the Scouts in the "Jesi 1" Group, for making me understand the importance of serving, and for always giving new opportunities to push myself beyond my limits;

to all of you I express my most sincere gratitude, for constantly proving that no man is an island, and that "happiness is only real when shared".

Ancona, February 15, 2020

Nicola Falcionelli

Abstract

Representing knowledge, modeling human reasoning, and understanding thought processes have always been central parts of intellectual activities, since the first attempts by greek philosophers. It is not just by chance that, as soon as computers started to spread, remarkable scientists and mathematicians such as John McCarthy, Marvin Minsky and Claude Shannon started creating Artificially Intelligent systems with a symbolic oriented perspective. Even though this has been a partially forced path due to the very limited computing capabilities at the time, it marked the beginning of what is now known as Classical (or Symbolic) Artificial Intelligence, or essentially, a set of techniques for implementing "intelligent" behaviours by means of logic formalisms and theorem proving. Classical AI techniques are indeed very direct and human-centered processes, which find their strenghts on straightforward human interpretability and knowledge reusability. On the contrary, they suffer of computability problems when applied to real world tasks, mostly due to search space combinatorial explosion (especially when reasoning with time), and undecidability.

However, the ever-increasing capabilites of computer hardware opened new possibilities for other more statistical-oriented methods to grow, such as Neural Networks. Even if the theory behind these methods was long known, it was only in recent years that they managed to achieve significant breakthroughs, and to surpass Classical AI techniques on many tasks. At the moment, the main hurdles of such statistical AI techniques are represented by the high energy consumption and the lack of easy ways for humans to understand the process that led to a particular result.

Summing up, Classical and Statistical AI techniques can be seen as two faces of the same coin: if a domain presents structured information, little uncertainty, and clear decision processes, then Classical AI might be the right tool, or otherwise, when the information is less structured, has more uncertainty, ambiguity and clear decision processes cannot be identified, then Statistical AI should be chosen.

The main purpose of this thesis is thus (i) to show capabilities and limits of current (Classical and Statistical) Artificial Intelligence techniques in both structured and unstructured domains, and (ii) to demostrate how event-based modeling can tackle some of their critical issues, providing new potential connections and novel perspectives.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

While broadly exploited in programming as a common paradygm, event-based modeling, also commonly referred to as Event-Driven Architecture (EDA), can play a big role in many Artificial Intelligence areas. This thesis will focus on improving existing event-based architectures and on proposing new techniques ranging from symbolic-AI to neural networks, also proving how such a paradygm is not only more efficient in terms of computation resources, but that it allows to extend neural systems beyond the current machine learning conceptual framework.

1.1 Event-based Modeling

From data-flow to reactive logic programming paradygms, the purpose of event-based systems is to propagate the concept of change and compute the results accordingly. At the cost of having to deal with event queues, listeners, message passing strategies and theoretical caveats, this usually leads to an increase in performance, as computation is only performed when strictly needed.

Reactive languages' runtimes leverage on the notion of graphs, where edges and nodes respectively represent dependencies among values and the computation steps. This representation helps keeping track of changes and how the "flow of data" gets processed. Standard strategies to implement propagation of change through the graph are known as "push-pull" algorithms, that are effectively responsible for event reaction mechanisms. As a general technique, the reactive phyolosophy can be combined either to imperative, object oriented, functional and rule-based programming approaches. Similarly, it can be successfully exploited in Artificial Intelligence, starting from rule-based symbolic techniques to the more graph-oriented learning tools like neural networks.

1.2 Domains

Among the years, Artificial Intelligence found its way across a great variety of domains, and it is only until recent times that a distinction became particularly evident. On one side, while Classical Artificial Intelligence started to be increasingly known as "Good-Old-Fashioned-AI" (GOFAI), it became more clear how, thanks to its rigorous nature, such techniques are better suited on domains with a clear "structure", on which rules, theorems, and reasoning strategies could be straightforward modeling tools. More practically, they are usually widespread in application fields such as databases, commonsense reasoning, story understanding and ambient assisted living. On the other side, the quick growth of Deep Learning has proven how in less constrained domains and with more ambiguous data representations, statistical learning approaches are to be preferred over the classical AI techniques. Within these circumstances, it is much more convenient to abandon the idea of having a precise domain description, accepting instead the fact that the only reasonable way to go is to have approximate models that can be made more accurate over easily scalable resources like data quantity and parallel computing. Numerous examples can be found in computer vision, speech processing, reinforcement learning, and many other fields.

1.3 Main Contributions

This thesis mainly focuses on understanding how the event-based paradygm can improve the current state of the art in two opposite sides of Artificial Intelligence: Symbolic AI and Neural Networks. It will be organized into three chapters, as a progression from rule-based symbolic AI, gradually moving into statistical learning and interpretability domains, finally reaching the so called "third-generation" neural networks.

1.3.1 Reactive Symbolic AI

The first chapter will introduce the Event Calculus formalism, an event-based reactive First Order Logic language for explicitly perform temporal reasoning. As powerful as it is, like in many other logic-based systems, the main drawback is reasoning scalability and performance. Thus, the main contribution of this chapter will be an indexing technique to tackle this problem, using a medical monitoring scenario as a testbed for both reasoning performance and language expressiveness. The modeling techniques proposed for modeling domain rules are further extended providing an Event-Calculus representation of Timed Automata.

1.3.2 Interpretability in Statistical AI

While the first chapter focused on rule-based medical knowledge, the second chapter will tackle instead the medical imaging domain, identifying the limits of symbolic techniques in tasks such as radiological image analysis. After proposing and discussing preprocessing pipelines and neural networks architectures, the main contribution of this chapter is to try to overcome the "black-box" nature of Deep Learning systems, which is particularly relevant in this application. In general, statistical learning techniques are usually based on the complex behaviour that emerges from the interactions of simple components, such as neurons. Thus, interpreting and explaining how and why such complex systems, composed of simple interacting agents, behave in a particular way is also what led to another marginal contribution within this chapter: a multi-agent simulator for analyzing heuristical solutions to a particular optimization problem known as the dynamic Vehicle Routing Problem.

1.3.3 Event-based Neural Networks

Apart from the black-box nature and interpretability aspects related to neural network techniques, other concerns regard the quite heavy hardware resources required to run Deep Learning software still persist. This chapter will thus focus on discussing how the event-based paradygm can help in drastically reducing computational overhead while at the same time enhancing their representation capabilities (together with the introduction of recurrent connections). The most notable contribution is the creation of a novel event-based neural network designer and simulator, for creating and analysing the dynamics of spiking neural circuits, which could be particularly useful when designing neuromorphic systems (e.g. for embedded applications). With the use of such simulator, several notable spiking neural circuits implementing key features for general purpose computation have been already identified, and will be shown in the last part of the chapter.

Chapter 2

Reasoning in Structured Time-dependent Domains

Classical Artificial Intelligence has always tried to model human reasoning processes, such as deduction or induction, and to bring them inside computer systems, with the goal of creating "artificially" smart devices and intelligent agents.

Even with the rise of statistical Machine Learning techniques, Classical or Symbolic Artificial Intelligence finds its way on those domains in which information is strongly structured, and where there is a need for understanding machine decisions.

However, even in these latter conditions, being able to reason with time-changing domains has always represented a challenge for symbolic-AI approaches, leading to the creation of many different formalisms, including the Event Calculus.

The Event Calculus will be the main protagonist of this chapter, as it will not only be used as the main language for modeling knowledge across two separate domains, but also some of its critical aspects will be discussed and improved.

2.1 Introduction

Logic programming and declarative languages have always been particularly suited to represent data structures suchs as graphs, lists, and trees, as well as human knowledge in the form of rules, clauses and ontologies. In addition, it allows to just focus on the data representation itself, leaving to the reasoner's machinery the burden of computing a result given a query. Thus, most of the effort by a human designer has to be put on finding a good representation for a particular domain, which should be compact, intuitive and efficient.

Efficiency though has not only to do with a system is modeled, on the contrary, it mostly depends on the reasoning machinery, and precisely on the mechanisms in which clauses, facts and predicates are retrieved from the knowledge base. This is why, in this chapter, the Event Calculus's introduction is immediately

followed by a discussion on such reasoning can be made faster and more efficient by using indexers inside the reasoning engine.

After that, during the central part of the chapter, it will be shown how, thanks the formalisms' features, "human-readable" domain knowledge can be translated into actual logic formulas that can run on a real inference engine. These domains have been chosen as two good candidates in which logic programming would be particularly suitable (for the strongly-structured domain knowledge) and useful (as both domains require a transparent and human-understandable decision process).

Then, these models are loaded into different Event Calculus engines (with different optimizations), and used to understand how well they behave in terms of computation time and scalability under different conditions.

2.2 Related Work

The Event Calculus (EC) has been particularly useful as a knowledge representation and reasoning tool in several domains. In the context of Personal Health Systems, EC and Multi-Agent-Systems (MAS) have been successfully applied to the self-management of diabetes [4, 5], and more generally to patient's monitoring [6, 7, 8]. Still in the medical domain, [9] proposes a symbolic representation of ECG waves and uses it for cardiac arrythmia recognition by means of Prolog rules (the ECG waves dataset in EC form can be found at [10]).

In the field of MAS, [11] shows how the EC can be exploited to build intelligent agents, within an underlying Mobile Multi-Agent platform such as MAGPIE. In [12] a way to model workflows and business processes in EC is proposed. Concepts such as AND/OR splits/joins and reasoning strategies are thoroughly explained, and some potential applications are discussed as well. Notably, the use of EC as a programming model for AI in games is discussed in [13], opening for possible connections with Finite State Machines.

Execution time and complexity have always been a major constraint in logic programming and in EC specifically. Even though the Cached-Event Calculus and the Reactive-Event Calculus already improve the performance of plain EC a lot, this is still not enough for most practical purposes. In this direction, [14, 15] propose several EC-machinery integrated indexing strategies that provide a more efficient Knowledge-Base management and query execution. A different approach is the one took in [16], in which, thanks to logic-theory-compiling, part of the EC flexibility is traded off for performance.

Regarding model checking, EC can be extended to include Modal Logic operators, as shown in [17]. [18] instead presents four several EC reasoning strategies, which can be potentially exploited for model checking purposes also with the models presented in this chapter. Instead, Timed Automata have

been widely used to model Real-Time systems and protocols, with the goal of formally verifying their properties [19]. Notable examples, such as model-based schedulability analysis or mutual exclusion protocols verification, can be found on the UPPAAL website [20].

2.3 The Event Calculus

Among a set of possible formalisms, the Event Calculus has been chosen as the reference one, as it combines great expressiveness with feasibility of reasoning, intuitiveness, readability and resource availability (implementation and literature) [18, 6, 16]. More precisely, being a logic formalism for reasoning about actions and their effects in time [21], it is a suitable tool for modeling expert systems representing the evolution in time of an entity by means of the production of events.

From a technical point of view, EC is based on many-sorted first-order predicate calculus, known as domain-independent axioms, which are represented as normal logic programs that are executable in Prolog. The underlying time model of EC is linear. EC manipulates fluents, where a fluent represents a property that can have different values over time. The term F=V denotes that a fluent F has value V as a consequence of an action that took place at some earlier time-point and not terminated by another action in the meantime. Table 2.1 summarizes the main EC predicates. Predicates, functions, symbols and constants start with lowercase letter, while variables start with uppercase letter. Predicates in the text are referenced as predicate/N, where predicate is the name of the predicate and N its arity (e.g. number of arguments).

The domain independent axioms of EC are the following:

$$\begin{aligned} &\text{holdsAt}(F = V, 0) \leftarrow \\ &\qquad \text{initially}(F = V). \end{aligned} \tag{2.1}$$

$$\begin{aligned} &\text{holdsAt}(F = V, T) \leftarrow \\ &\qquad \text{initiatesAt}(F = V, T_s), \\ &\qquad T_s < T, \\ &\qquad \text{not broken}(F = V, [T_s, T]). \end{aligned} \tag{2.2}$$

Predicate (2.1) states that a fluent F holds value V at time 0, if it has been initially set to this value. For any other time T > 0, the predicate (2.2) states that the fluent holds at time T if it has been initiated to value V at some earlier

Table 2.1: Main Event Calculus predicates

Predicate	**Meaning**
initially(F=V)	The value of fluent F is V at time 0
holdsAt(F=V,T)	The value of fluent F is V at time T
holdsFor(F=V,[T_{min},T_{max}])	The value of fluent F is V between T_{min} and T_{max}
initiatesAt(F=V,T)	At time T the fluent F is initiated to have value V
terminatesAt(F=V,T)	At time T the fluent F is terminated from having value V
broken(F=V,[T_{min},T_{max}])	The value of fluent F is either terminated at T_{max}, or initiated to a different value than V between T_{min} and T_{max}
happensAt(E,T)	An event E takes place at time T updating the state of the fluents

time point T_s, and it has not been broken on the meanwhile.

$$\begin{aligned} &\text{broken}(F = V, [T_{min}, T_{max}]) \leftarrow \\ &\qquad \text{terminatesAt}(F = V, T), \\ &\qquad T_{min} < T, \\ &\qquad T_{max} > T. \end{aligned} \tag{2.3}$$

$$\begin{aligned} &\text{broken}(F = V_1, [T_{min}, T_{max}]) \leftarrow \\ &\qquad \text{initiatesAt}(F = V_2, T), V_1 \neq V_2, \\ &\qquad T_{min} < T, \\ &\qquad T_{max} > T. \end{aligned} \tag{2.4}$$

Predicates (2.3) and (2.4) specify the conditions that break a fluent. Predicate (2.3) states that a fluent is broken between two time points Tmin and Tmax if within this interval it has been terminated to have value V. Alternatively, predicate (2.4) states that a fluent is broken within a time interval if it has been

initiated to hold a different value.

$$\begin{aligned} \text{holdsFor}(F = V, [T_{min}, T_{max}]) \leftarrow \\ \text{initiatesAt}(F = V, T_{min}), \\ \text{terminiatesAt}(F = V, T_{max}), \\ \text{not broken}(F = V, [T_{min}, T_{max}]). \end{aligned} \tag{2.5}$$

$$\begin{aligned} \text{holdsFor}(F = V, [T_{min}, +\infty]) \leftarrow \\ \text{initiatesAt}(F = V, T_{min}), \\ \text{not broken}(F = V, [T_{min}, +\infty]). \end{aligned} \tag{2.6}$$

$$\begin{aligned} \text{holdsFor}(F = V, [-\infty, T_{max}]) \leftarrow \\ \text{terminatesAt}(F = V, T_{max}), \\ \text{not broken}(F = V, [-\infty, T_{max}]). \end{aligned} \tag{2.7}$$

Predicates (2.5), (2.6) and (2.7) deal with the validity intervals of fluents. In particular, predicate (2.5) specifies that a fluent F keeps value V for a time interval going from Tmin to Tmax if nothing happens in the middle that breaks such an interval. Predicates (2.6) and (2.7) behave in the same way, but deal with open intervals.

The domain dependent predicates in EC are typically expressed in terms of the initiatesAt/2 and terminatesAt/2 predicates. One example of a common rule for initiatesAt/2 is

$$\begin{aligned} \text{initatesAt}(F = V, T) \leftarrow \\ \text{happensAt}(Ev, T), \\ Conditions[T]. \end{aligned} \tag{2.8}$$

The above definition states that a fluent is initiated to value V at time T if an event Ev happens at this time point, and some optional conditions depending on the domain are satisfied.

2.3.1 Cached Event Calculus and jREC

Straightforward implementations of EC [21] have time and memory complexity which are not practical for developing real applications. This is due to the fact that every time the EC engine is queried, the computation starts from scratch, and all fluents validity intervals are calculated again. Cached Event Calculus (CEC), proposed by Chittaro and Montanari [22], tries instead to overcome this

inefficiency by giving EC a memory mechanism, and moving computation from query time to update time.

CEC formalizes the concept of Maximal Validity Interval (MVI), that represents a time interval in which a particular fluent holds without being terminated by any event. A fluent is also associated to a list of MVIs, in order to express all the time intervals in which that fluent holds continuously.

Whenever the rule engine is updated (e.g. by inserting a new event occurrence), the fluents' MVIs are calculated, and then stored for further use, allowing incremental computation for following updates. Also, every time a new event is added to the database, CEC manages to compute MVIs only for the fluents that can vary with that event, and does not check the MVIs of those fluents that cannot possibly change, thus avoiding unnecessary computation.

jREC is a reasoning tool implemented in Java and tuProlog that is based on a lightweight version of CEC known as Reactive Event Calculus (REC) [6].

jREC consists of three main components:

- The Prolog theory, which represents the actual CEC axiomatization that is loaded into tuProlog;
- The Java engine, which allows to query and update the database without having to interact directly with tuProlog, as well as adding specific domain-dependent theories;
- The Tester, which is a GUI based stand-alone tool for editing theories, visualizing fluents' MVIs and event occurrences, mainly used for prototyping and developing domain-dependent theories.

2.4 Indexing Data Structures

With the goal of improving the EC reasoning performance, three different indexing data structures are used. Section 2.5 will take care of detailing how these achieve this goal by indexing events and/or fluents.

2.4.1 Red-black trees

A red-black tree (RBT) is a well known data structure proposed by Rudolf Bayer in 1972 [23]. It is a binary search tree which provides $O(log(n))$ Worst Case time complexity for operations such as node searching, insertion and deletion, as well as $O(n)$ Worst Case space complexity [23]. This is made possible thanks to node coloring: every node of the tree is augmented with an extra bit, and based on the value of such bit, the node is considered to be red or black.

The aforementioned operations rely on such coloring feature to achieve Worst Case logarithmic time complexity and linear space complexity. In fact, every

operation that modifies the RBT has to comply with very precise policies which constrain how the nodes should be moved or re-painted. These policies guarantees that the nodes in an RBT are always balanced after every operation, giving such data structure the property of self-balancing. Even though the obtained balance is not perfect, it is proven to be good enough to provide the declared performances [23].

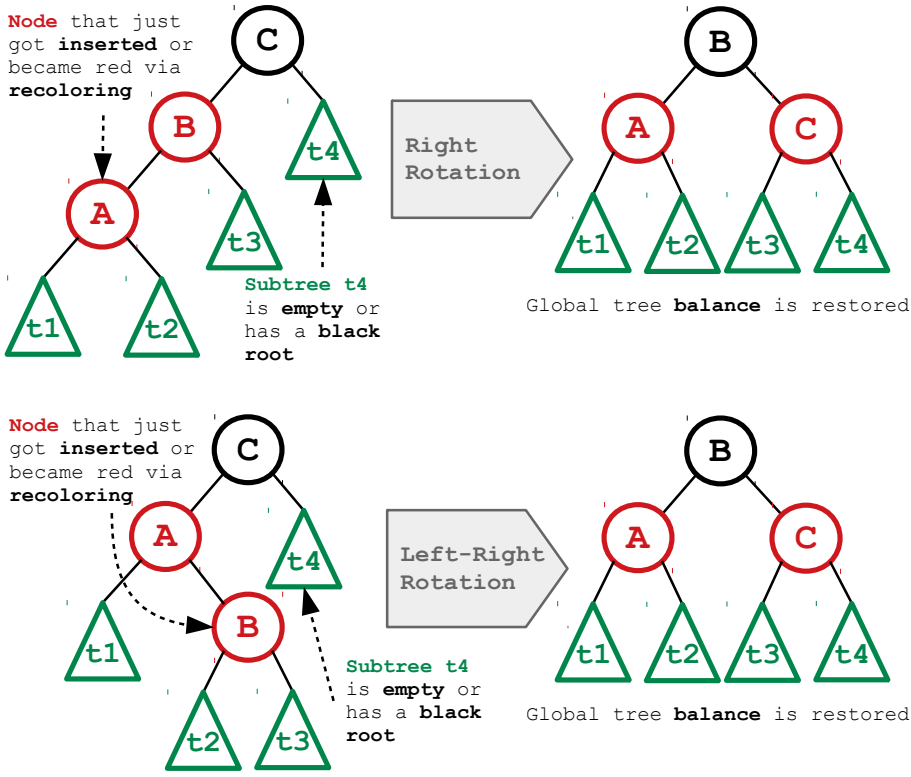


Figure 2.1: This picture illustrates how a Red-Black tree keeps its structure balanced by means of right/left rotations and node recoloring. These are triggered when nodes are inserted (or deleted) and the balancing/coloring rules of the data structure are violated

2.4.2 K-d trees

K-d trees [24] are binary trees optimized to deal with k-dimensional points. As reported in [1], given a set of k-dimensional points, a k-d tree can be generated by splitting recursively the hyperplane containing the points at every level of the tree, alternating the coordinate that is split according to the depth of the tree. Fig. 2.2 shows how splits are performed on a 2-dimensional tree of depth

3, where at each level the value of the splitting coordinate is the median value, deciding if a new point should go to the left or to the right of an existing tree node.

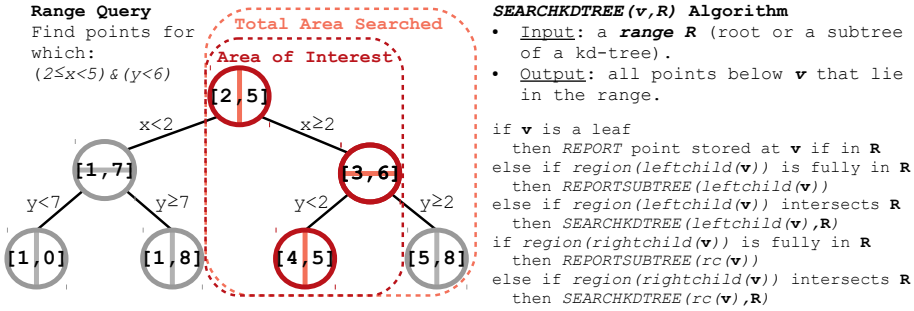


Figure 2.2: Range Query on a k-d tree [1]

Figure 2.2 shows also the effect of searching a k-d tree via a range query performed on it. The range query algorithm recursively searches for regions contained or intersected by the region specified in the range query. If the region found is contained in the region specified in the query, then the whole region is returned. If the region of the tree intersects the region specified by the query, then the points reported are only those ones included in the region of the query.

The k-d tree data structure has a set of important properties when dealing with searches of multi-dimensional points: (a) a k-d tree for a set P of n points uses $O(n)$ storage and can be constructed in $O(nlog(n))$ time; (b) the operations of adding or deleting a point have a complexity of $O(log(n))$; (c) a rectangular range query on the k-d tree takes $O(\sqrt{n}+k)$ time, where k is the number of reported points residing the rectangular area identified by the query.

These properties are fundamental to create a version of the EC that can scale up to be used in dynamic applications with a large number of events (narratives).

2.4.3 Interval trees

An interval tree is a data structure that provides efficient means for querying time intervals, and find out what events occurred during a given time interval. In terms of computational complexity, an interval tree composed of n points requires $O(n)$ space for storage; its construction requires $O(nlog(n))$ time; and a query requires $O(log(n)+m)$ time, where n is the total number of intervals and m the number of reported results. Because of these properties, an interval tree is a suitable structure for scaling up EC.

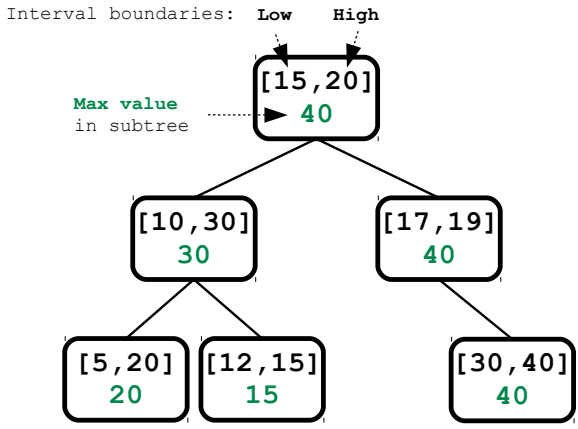


Figure 2.3: Depiction of how intervals are stored inside an Interval Tree. Notice that intervals in the same subtree do overlap, while intervals in different subtrees do not. This feature is particularly useful to store and retrieve fluents' MVIs that might be affected by an upcoming event

2.5 Reasoning Optimization

The main hurdle that hampers the diffusion of logic-based systems in the practical world is efficiency. Even though imperative paradygms usually allow better performance, imperative programs are also usually compiled, thus less flexible and readable than logic theories. For this reason, the design of the proposed optimized EC system have to:

- be Scalable. The computation times for queries must not explode as the Knowledge Base grows.
- provide Human Readable knowledge representation, easing domain-knowledge translation and the presence of humans-in-the-loop.
- be Flexible, allowing for runtime customization without having to stop the execution or either recompiling the code.

2.5.1 Knowledge-Base Indexing Strategies

Efficient handling of massive event streams, while preserving the philosophy of Event Calculus, and in broader terms, of Logic Programming, is a non-trivial task. Techniques such as (i) event windowing/forgetting [16], (ii) theory pre-compilation [16] and (iii) a priori assumptions on event temporal ordering, can help to ease the burden of this process, but at the same time their adoption will cause the reasoning approach to be less general and less flexible.

The idea presented in this chapter is to separate the Knowledge Base (KB) management from the reasoning itself, as already proposed by [15]. The KB is implemented by an indexing data structure, which will take care of storing and accessing Prolog facts for later and more efficient retrieval. For the reasoning part on the other hand, the EC Prolog engine has to query the indexer every time it wants to read/write some data from/to the KB. The indexing can be performed on many criteria, but since Event Calculus rules heavily depend on temporal constraints expressed as events and fluents, the most reasonable and general choice would be to index events and MVIs on a temporal basis.

2.5.2 Standard EC Indexing

To create a scaled up version of standard EC, tree indexing structures such as k-d trees [15] and interval trees are used to substantially reduce the Prolog computation overhead that arises from handling massive event streams. Even though the introduction of these two different techniques leads to the creation of two distinct EC engines, they are provided with a generic interface that abstracts the communication to those tree structures.

These engines are based on a k-d Tree Indexer and an Interval Tree Indexer, both used to store ground terms, but with different internal representations:

- For the k-d Tree Indexer, events and fluent are considered as k-dimensional points. Assuming the events to be instantaneous, they are indexed with respect to one time dimension. Since MVIs are instead intervals, the time dimensions used for indexing are two.

- For the Interval Tree Indexer, events are stored as intervals whose start and end points are the same, while MVIs have different start and end times. In addition, the intervals contain metadata for their identification such as the name of the event, or the name and value of the MVI's fluent.

By the way, one common point among the two indexers is that they keep the event terms and the MVI terms in two separate trees.

A standard EC implementation is enriched with additional predicates that interact with the indexers to insert, delete or range query points. It should be noticed that, since the interface towards the two indexers is the same, the following implementation will refer to a generic tree data structure. The

insertion of an event is done with the following predicate,

$$
\begin{aligned}
&\text{insert}(Ev, T) \leftarrow \\
&\qquad \text{index}(Ev, T), \text{cache}(Ev, T). \\
&\text{index}(Ev, T) \leftarrow \\
&\qquad \text{functor}(Ev, Name, Arity), \\
&\qquad Ev = ..[Name|Args], \\
&\qquad Args = [Head|Tail], \\
&\qquad \text{index\_tree}(Ev, Name, Arity, Head, T). \\
&\text{cache}(Ev, T) \leftarrow \\
&\qquad \text{findall}(\text{mvi}(F = V, T_s), \\
&\qquad\qquad \text{terminatesAt}(F = V, T), ListTerm), \\
&\qquad \text{close\_interval}(ListTerm, T), \\
&\qquad \text{findall}(\text{mvi}(F = V, T), \\
&\qquad\qquad \text{initiatesAt}(F = V, T), ListInit), \\
&\qquad \text{open\_interval}(ListInit, T).
\end{aligned}
\tag{2.9}
$$

The insert/2 predicate consists of two steps. First, the index/2 predicate stores the event in the tree with the index_tree/5 predicate. Second, the cache/2 predicate checks whether the just indexed event is closing or opening the MVI of any fluent F, which is then cached in a separate tree. A MVI can be open to infinity or closed within two time boundaries. The close_interval/2 predicates do the operation of closing open MVIs when needed, such predicates are specified as follows,

$$
\begin{aligned}
&\text{close\_interval}([], T). \\
&\text{close\_interval}([Head|Tail], T) \leftarrow \\
&\qquad\qquad Head = \text{mvi}(F = V, \_), \\
&\qquad\qquad \text{range\_query}(F = V, [T_s, +\infty]), \\
&\qquad\qquad \text{delete\_tree}(F, V, T_s, +\infty), \\
&\qquad\qquad \text{index\_tree}(F, V, T_s, T). \\
&\text{close\_interval}([Head|Tail], T) \leftarrow \\
&\qquad\qquad \text{close\_interval}(Tail, T).
\end{aligned}
\tag{2.10}
$$

First, the range_query/2 predicate retrieves from the tree all the open MVIs that must be closed due to the happening of the just indexed event. Second, the delete_tree/4 event predicate deletes the MVI from the tree. Finally, the indexed_tree/4 indexes the MVI as closed in the tree. The process for indexing

open periods is similar with the difference that no open periods are retracted from the tree. Once the procedure of adding the new event to the rule engine has finished, the next step is to check whether the event is triggering any of the domain dependent rules. In EC notation this means to query the holds_at/2 predicate, which is defined as follows,

$$\begin{aligned} &\text{holdsAt}(F = V, T) \leftarrow \\ &\qquad \text{not var}(T), \text{number}(T) \\ &\qquad \text{intersect\_query}(F = V, [T_{start}, T_{end}], T), \\ &\qquad T > T_{start}, T < T_{end}. \end{aligned} \tag{2.11}$$

The intersect_query/3 predicate queries the tree only those MVIs that intersect T. Thus, improving the computation time of the original holdsAt/2 predicate.

Finally, the domain dependent rules that model high level properties are expressed in terms of the EC initiatesAt/2 predicate, which is reformulated as follows,

$$\begin{aligned} &\text{initiatesAt}(F = V, T) \leftarrow \\ &\qquad \text{query\_tree}(\text{happensAt}(Ev, T), [WT_S, WT_E]), \\ &\qquad Conditions[T]. \end{aligned} \tag{2.12}$$

The query_tree/2 predicate queries, to the tree storing the events, only those events that happened within a given time window whose boundaries are WT_S and WT_E. Thus, pruning the number of alternatives that a standard EC implementation would search, which translates to a better performance. In addition, the query_tree/2 predicate decomposes the different parts of an event in order to perform such queries,

$$\begin{aligned} &\text{query\_tree}(\text{happensAt}(Ev, T), [WT_S, WT_E]) \leftarrow \\ &\qquad \text{var}(T), \text{not var}(Ev), \\ &\qquad \text{functor}(Ev, Name, Arity), \\ &\qquad Ev = ..[Name|Args], \\ &\qquad \text{retrieve\_range}(Ev, Name, Arity, Args, T, \\ &\qquad\qquad [WT_S, WT_E]). \end{aligned} \tag{2.13}$$

Given the extensive use of tree structures, the proposed EC reasoner can be wrapped under the name of Tree Event Calculus, as they are both based on tree indexing techniques (i.e. k-d trees and interval trees, thus TEC-KD and TEC-IT).

2.5.3 JREC Indexing

Instead of chaching standard Event Calculus with trees as in the previous section, indexing data structures can be applied to an already more efficient version of EC such as the Reactive Event Calculus, implemented in jREC. In its standard version, jREC does not apply any simplifying assumption or technique to the event streams: this forces the reasoner to spend a very high amount of resources every time the engine's knowledge base (KB) is updated with new events. Whenever a list of new events has to be asserted into the KB, jREC must perform the following steps:

- Sort the list of new events chronologically;
- Read all the events already present in the KB and put them in a list;
- Retract all the events from the KB;
- Sort the list of KB's events chronologically;
- Merge the list of new events with the list of events read from the KB;
- Sort the newly obtained list chronologically and remove duplicates;
- Assert the events from the newly obtained list back into the KB;
- Calculate the effects on fluents' MVIs.

This procedure indeed maintains the reasoning as general and flexible as possible, but it is also the main source of jREC inefficiency, since every new event(s) insertion causes the engine to sort the event lists multiple times.

To tackle such issue, an indexing data structure (i.e. the previously mentioned red-black trees) is sintegrated in jREC's machinery. RBTs will take the duty of maintaining the events temporal ordering by avoiding unnecessary sorting operations, and ensuring fast execution times.

It should be noticed that, since (i) an event normally contains multi-dimensional data (i.e. timestamp and some quantities' value), (ii) an RBT only allows single-dimensional indexing, and (iii) jREC needs the events to be ordered chronologically, the only choice is to consider the events timestamp as the key on which the indexing will be performed.

2.6 Modeling Medical Domain-Knowledge

One of the contributions of this chapter is to use the EC language to model different medical aspects that might be of interest in a Personal Health System/self-monitoring scenario.

In this context, the possibility to extend and modify the Knowledge Base at runtime, without the need of rebuilding any binary file, gives logic-based systems an advantage over procedural/imperative approaches when it comes to extensibility, flexibility and customization capabilities. Especially when considering the latest European GDPR regulations that enforce algorithmic fairness and AI explainability [25], logic-based systems that are able to provide result interpretability and code readibility are favorable over traditional Machine Learning reasoning techniques.

Diabetes has been chosen as the main use case, as a sustainable and personalized treatment of this disease is considered to be one of the major challenges for the health sector in the next future [26, 27, 28]. More precisely known as Diabetes Mellitus (DM), it is actually a group of metabolic disorders causing high blood glycemic levels for prolonged periods of time. The disease is usually divided into classes, with the most common two being the following:

- Type 1 Diabetes Mellitus (T1DM), in which a regular external source of insuline is needed, which is normally supplied by manual injections or insulin pumps coupled with Continuous Glucose Monitoring (CGM) devices.

- Type 2 Diabetes Mellitus (T2DM), which instead is associated with elderly people, and usually treated with diet, physical exercise and Self-monitoring of Blood Glucose (SMBG) eventually followed by metforming intake and manual insulin injections.

As the whole field of diabetes is very broad, this chapter focuses on a particularly hard to control T1DM called Brittle Diabetes, in which the patients experience very big swings in glucose levels, alternating periods of hypoglycemia and hyperglycemia [29].

Another critical task is to monitor physiological parameters that are considered to be risk factors especially for Diabetes chronic patients, such as body weight, cholesterol and blood pressure [30, 28]. Blood pressure control, and more precisely hypertension monitoring, has been chosen as another medical domain's use case, since it combines high relevance in terms of prevention with low invasiveness of measurements.

The last use case employed to stress the modeling capabilities of EC is cardiac arrythmia. Such as in the case of Diabetes, cardiac arrythmia is a chronic condition that offers a very broad spectrum of diagnoses depending on the shape and frequency of hearth waves. By analyzing the most common type of waves (the P wave and the QRS complex), it is for example possible to distinguish between the two most straightforward types of arrhythmia: sopraventricular tachycardia and sinusal bradycardia. Sopraventricular Tachycardia is usually defined as a cardiac frequency over 100 beats/minute (or a time less than

100ms between two QRS complexes), while, on the contrary, sinusal bradycardia corresponds to a cardiac frequency less than 60 beats/minute (or a time more than 1s between two QRS complexes) [31].

2.6.1 Monitoring Rules Definition

In order to detect alert conditions, a sequential and a complex rule patterns are proposed. They check physiological values collected from the patient's Body Area Network, and are based on the literature available for glucose, blood pressure and heartbeat monitoring [32, 33]. Glucose and blood pressure targets for hypoglycemia, hyperglycemia and hypertension have been extracted from diabetes guidelines [34, 35, 36], while thresholds for arrhythmias are taken from [31]. The patterns identify alert conditions in the patient's health status by modeling the sensor inputs as events that are evaluated in the body of the rules. The two patterns are:

Pattern 1: Brittle diabetes, defined as a glucose rebound going from less than 3.8 mmol/l (hypoglycemia) to more than 8.0 mmol/l (hyperglycemia) in a period of six hours. This pattern can be expressed by a sequential rule.

Pattern 2: Pre-hypertension, defined as two events of high blood pressure in a period of one week. This pattern can be expressed by a complex rule.

Pattern 3: Tachycardia, defined as nine or more QRS complexes in a period of five seconds. This pattern can be expressed by a complex rule.

Pattern 4: Bradycardia, defined as four or less events of QRS complexes in a period of five seconds. This pattern can be expressed by a complex rule.

2.6.2 Monitoring Rules implementation in JREC

Pattern 1 is implemented as follows:

$$\begin{aligned} \text{initiatesAt}(A = \text{true}, T) :- \\ &\text{happensAt}(\text{ev}(2, A, W), T), \\ &\text{happensAt}(\text{ev}(1, A, \_), T_1), \\ &T_s \text{ is } (T - W), \\ &T > T_1, \\ &T_1 >= T_s, \\ &\text{no\_alert}(A, T_s). \end{aligned} \tag{2.14a}$$

$$\begin{aligned} \text{terminatesAt}(A = \text{true}, T) :- \\ &\text{happensAt}(\text{ev}(1, A, \_), T). \end{aligned} \tag{2.14b}$$

$$\begin{aligned}\text{happensAt}(\text{ev}(1, \text{`brittle diabetes'}, W), T) :- \\ \text{hours\_to\_epoch}(6, W), \\ \text{happensAt}(\text{glucose}(G), T), \\ G =< 3.8.\end{aligned} \tag{2.14c}$$

$$\begin{aligned}\text{happensAt}(\text{ev}(2, \text{`brittle diabetes'}, W), T) :- \\ \text{hours\_to\_epoch}(6, W), \\ \text{happensAt}(\text{glucose}(G), T), \\ G >= 8.\end{aligned} \tag{2.14d}$$

Rules (2.14a) and (2.14b) represent a generic sequential rule template with two events. In particular, the fluent F (i.e. the alert) is initiated with value A when: (i) two temporal ordered events occur inside a certain time window and (ii) when the fluent does not hold anywhere else inside the time window (no_alert/2). The fluent F is instead terminated when the first event of the ordering happens.

Rules (2.14c) and (2.14d) customize the template for the glucose monitoring use case. They instantiate the variables in the ev/3 term, specifying the time window width (W), the alert name (A) and the threshold values for G.

Pattern 2 is expressed in the following way:

$$\begin{aligned}\text{initiatesAt}(A = \text{true}, T) :- \\ \text{happensAt}(\text{alertcheck}(A, W, NMax_1), T), \\ T_s \text{ is } (T - W), \\ \text{count\_events\_tw}(N_1, \text{evc}(1, A), T_s, T), \\ N_1 >= NMax_1, \\ \text{no\_alert}(A, T_s).\end{aligned} \tag{2.15a}$$

$$\begin{aligned}\text{terminatesAt}(A = \text{true}, T) :- \\ \text{happensAt}(\text{alertcheck}(A, W, \_), T), \\ \text{holdsAt}(A = \text{true}, T).\end{aligned} \tag{2.15b}$$

$$\begin{aligned}\text{happensAt}(\text{evc}(1, \text{`pre-hypertension'}), T) :- \\ \text{happensAt}(\text{blood\_pressure}(S, D), T), \\ S >= 130, \\ D >= 80.\end{aligned} \tag{2.15c}$$

$$\begin{aligned}\text{happensAt}(\text{alertcheck}(\text{`pre-hypertension'}, W, 2), T) :- \\ \text{weeks\_to\_epoch}(1, W), \\ \text{happensAt}(\text{evc}(1, \text{`pre-hypertension'}), T).\end{aligned} \tag{2.15d}$$

Rules (2.15a) and (2.15b) represent a generic complex rule template with one event type. In particular, the fluent F (i.e. the alert) is initiated with value A when: (i) there are least $NMax_1$ occurrences of the alertcheck/3 event inside the time window and (ii) when the fluent does not hold anywhere else inside the time window (no_alert/2). Also, the count_events_tw/4 predicate is necessary to handle different event temporal orderings without having to duplicate the rule body for every permutation. Rules (2.15c) and (2.15d) customize the template for the hypertension monitoring use case. They instantiate the variables of the evc/2 and the alertcheck/3 terms specifying the time window width (W), the alert name (A) and the threshold values for S and D.

Patterns 3 and 4's have been respectively coded as follows:

$$\begin{aligned}\text{initiatesAt}(\text{tachycardia} = \text{true}, T) :- \\ \text{seconds\_to\_epoch}(5, W), \\ T_0 \text{ is } (T - W), \\ \text{count\_events\_tw}(N, \text{qrs}(\text{basic}), T_0, T), \\ N >= 8, \\ \text{no\_alert}(\text{tachycardia}, T_0).\end{aligned} \tag{2.16a}$$

$$\begin{aligned}\text{terminatesAt}(\text{tachycardia} = \text{true}, T) :- \\ \text{holds\_at}(\text{tachycardia} = \text{true}, T).\end{aligned} \tag{2.16b}$$

$$\begin{aligned}\text{initiatesAt}(\text{bradycardia} = \text{true}, T) :- \\ &\text{seconds\_to\_epoch}(5, W), \\ &T_0 \text{ is } (T - W), \\ &\text{count\_events\_tw}(N, \text{qrs}(\text{basic}), T_0, T), \\ &N <= 3, \\ &\text{no\_alert}(\text{bradycardia}, T_0).\end{aligned} \tag{2.17a}$$

$$\begin{aligned}\text{terminatesAt}(\text{bradycardia} = \text{true}, T) :- \\ &\text{holds\_at}(\text{bradycardia} = \text{true}, T).\end{aligned} \tag{2.17b}$$

Unlike the previous jREC implementations of Patterns 1 and 2, in this case rule templates have not been used (thus leading to a less general but more straightforward coding). The "artificial" events (evc) that wrap real ones in rule (2.15) are not needed anymore, since no condition has to be put on the argument(s) of the qrs/1 event. Apart from that, the mechanism for counting event occurrences (count_events_tw/4) and alert flooding avoidance (no_alert/2) are the same as in (2.15).

2.6.3 Monitoring Rules Implementation in TEC

Pattern 1 is expressed in the following way:

$$\begin{aligned}\text{initiatesAt}(\text{alert}(\text{one}) = \text{'brittle diabetes'}), T) :- \\ &\text{hours\_ago}(6, T_s, T), \\ &\text{query\_tree}(\text{happensAt}(Ev_1, T_1), [T_s, T]), \\ &\text{query\_tree}(\text{happensAt}(Ev_2, T_2), [T_s, T]), \\ &Ev_1 = \text{glucose}(V_1), Ev_2 = \text{glucose}(V_2), \\ &V_1 =< 3.8, V_2 >= 8.0, \\ &T_2 > T_1, \\ &\text{not query\_tree}(\text{happensAt}(\text{alert}(\text{one}, \\ &\qquad \text{'brittle diabetes'}), T_{alert}), [T_s, T]).\end{aligned} \tag{2.18}$$

Rule (2.18) follows the same structure as rule (2.12). In particular, queries the interval tree for two glucose events, within a time window of six hours, that satisfy the threshold conditions of V_1 and V_2. In addition, it specifies three different temporal conditions, which are: (i) the hours_ago/3 predicate that

defines the lower boundary of the time window, (ii) the $T_2 > T_1$ inequality, that ensures the correct temporal ordering between the glucose measurement events and (iii) the last Interval Tree query, which ensures that in the rule's time window there is not the same alert already, to avoid the generation of multiple alerts associated with the same events.

Pattern 2 is implemented as follows:

$$\begin{aligned}
&\text{initiatesAt}(\text{alert}(\text{two}) = \text{`pre-hypertension'}, T) :- \\
&\quad \text{weeks_ago}(1, T_s, T), \\
&\quad \text{not query_tree}(\text{happensAt}(\text{alert}(\text{two}, \\
&\qquad \text{`pre-hypertension'}), T_{alert}), [T_s, T]), \\
&\quad \text{more_or_equals_to}(2, \\
&\qquad (\text{query_tree}(\text{happensAt}(Ev_1, T_1), [T_s, T]), \\
&\qquad Ev_1 = \text{blood_pressure}(Sys_1, Dias_1), \\
&\qquad Sys_1 >= 130, Dias_1 >= 80, \\
&\qquad \text{within_weeks}(1, T_1, T))).
\end{aligned} \tag{2.19}$$

Rule (2.19) also follows the structure of rule (2.12). In particular, queries the interval tree for two blood pressure events, within a time window of one week, that satisfy the threshold conditions of Sys_1, $Dias_1$, Sys_2 and $Dias_2$. The predicate more_or_equals_to/2 checks if at least two over-threshold blood pressure events happened inside the time window. The use of this predicate is necessary to handle different event temporal orderings without having to duplicate the rule body for every permutation. In addition, the rule specifies two different temporal conditions, which are: (i) the weeks_ago/3 predicate that defines the lower boundary of the time window and (ii) the first Interval Tree query, which ensures that in the rule's time window there are no other alerts (i.e. avoiding the generation of multiple alerts associated with the same events).

Patterns 3 and 4's have been respectively coded as follows:

$$\begin{aligned}
&\text{initiatesAt}(\text{alert}(\text{three}) = \text{`tachycardia'}, T) :- \\
&\quad \text{seconds_ago}(5, T_s, T), \\
&\quad \text{not query_tree}(\text{happensAt}(\text{alert}(\text{three}, \\
&\qquad \text{`tachycardia'}), T_{alert}), [T_s, T]), \\
&\quad \text{more_or_equals_to}(9, \\
&\qquad (\text{query_tree}(\text{happensAt}(Ev_1, T_1), [T_s, T]), \\
&\qquad Ev_1 = \text{qrs}(\text{basic}), \\
&\qquad \text{within_seconds}(5, T_1, T))).
\end{aligned} \tag{2.20a}$$

$$
\begin{aligned}
&\text{initiatesAt}(\text{alert}(\text{four}) = \text{'bradycardia'}, T) :- \\
&\quad \text{seconds\_ago}(5, T_s, T), \\
&\quad \text{not query\_tree}(\text{happensAt}(\text{alert}(\text{four}, \\
&\qquad \text{'bradycardia'}), T_{alert}), [T_s, T]), \\
&\quad \text{less\_or\_equals\_to}(4, \\
&\qquad (\text{query\_tree}(\text{happensAt}(Ev_1, T_1), [T_s, T]), \\
&\qquad Ev_1 = \text{qrs}(\text{basic}), \\
&\qquad \text{within\_seconds}(5, T_1, T))).
\end{aligned}
\tag{2.20b}
$$

Indeed, these implementations follow the structure of rule (2.19). It should be noticed anyway that (2.20b) makes use of the less_or_equals_to predicate, in contraposition to the other rules implemented in TEC. This predicate allows the rule to check if any five seconds interval contains less than five QRS complexes events, and if so, to generate a bradycardia alert. The regular more_or_equals_to predicate instead appears in (2.20a), since a tachycardia alert is generated when more than eight QRS complexes happen inside the five seconds time window.

2.7 Modeling Real-Time Systems Domain-Knowledge

Having computer systems with stable and predictable behaviour is of vital importance in mission critical applications. In those applications, CPU processes and tasks must guarantee compliance to strict time constraints, which are usually expressed as deadlines for their execution times. To ensure that a set of tasks can be scheduled without having any deadline miss, a whole set of techniques have been developed [37]. Since it would be too hard to ensure compliance to these constraints if the programmer would have to take care of it, Real-Time techniques have been included at an Operating System level. In this way, the Scheduler will be the OS's component responsible to assess schedulability within certain constraints, while the programmer can focus on implementing the task's features. In order for the Real-Time techniques to work, the programmer must provide the Worst-Case Execution Time (WCET) for each task, which is usually a pessimistic estimate obtained statistically by running it many times.

2.7.1 Timed Automata

Finite State Machines (FSM) are a very common tool to represent systems with relatively simple behaviours, such as vending machines, turnstiles and traffic

lights. They are also useful for a variety of applications like regular expression checking, videogame AI and software engineering. In this latter field, they are expecially suited for formal verification of programs and network protocols, but they are very limited in terms of expressiveness (i.e. being unable to explicitly model time dependencies). For this reason, there is often the need to rely on more powerful techniques, such as Timed Automata (TA). They enrich the FSM semantics and synthax by providing additional constructs and mechanisms that allow to effectively model timed systems, such as:

- a finite set of Clocks. Although all clocks increase with the same speed, they can be set (or reset) individually upon state transitions.
- a finite set of Guards. They are conditions that check clock values put on state transitions, that, if not satisfied, prevent the system from going to a state from another.
- a finite set of State Invariants. Mainly used for Model Checking, their purpose is to ensure progress, by preventing an Automata to be indefinetely stuck in a certain state (reachability analysis).

Reachability analysis is one of TA's main applications. Roughly, it works by finding an Automata's the possible runs (a run is a list of $\langle StateName, ClockValue \rangle$ 2-tuples), and checking if these behave according to certain requirements. For example, they could never (or always) reach a particular state, or be locked in certain sequences of states. Finding all the possible runs is not a trivial task: since the values clocks can hold are continuous, tools such as the Regions of Equivalence are needed to manage the infinite number of possible runs. This and other techniques allow TA's verification problems to be decidable [19].

In this context, TAs have been extensively used to model and verify Real-Time systems, Network Protocols and concurrent algorithms successfully, ensuring important properties such as safety and progress [20]. Such improvements have been made possible also thanks to already existing Model Checkers such as UPPAAL [38]. A relevant example can be found in [39], which shows how Schedulability Analysis can be carried out within UPPAAL, modeling Real-Time concepts such as tasks' deadlines, dependencies, periods, and WCETs.

2.7.2 Real-Time Systems and Logic-based Agents

The goal of translating Timed Automata into EC formulas is to establish a connection between the Real-Time part and the Knowledge Representation (KR)/Reasoning part of a logic-based agent, as shown in Figure 2.6. This connection would allow to have a common underlying language which is useful for both building the agent's knowledge base as well as to model, check and verify its Real-Time properties.

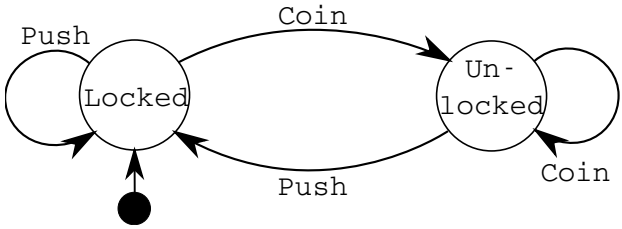


Figure 2.4: A simple FSM representing a classical turnstyle mechanism. As long as a coin is provided, the turning bars will remain unlocked. When they are pushed (a person transits), the turnstyle will stay locked until a new coin is inserted.

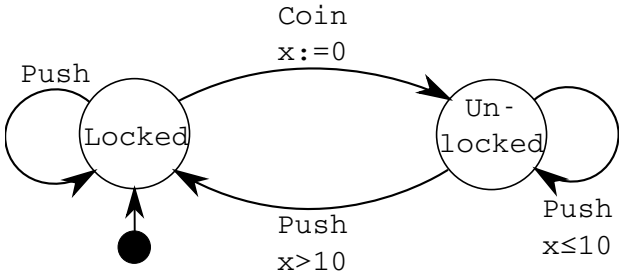


Figure 2.5: A simple TA representing a variation of a classical turnstyle. When a coin is loaded, the turning bars will stay unlocked for 10 time units. If they are pushed after such time, the turnstyle will lock again, until a new coin is delivered.

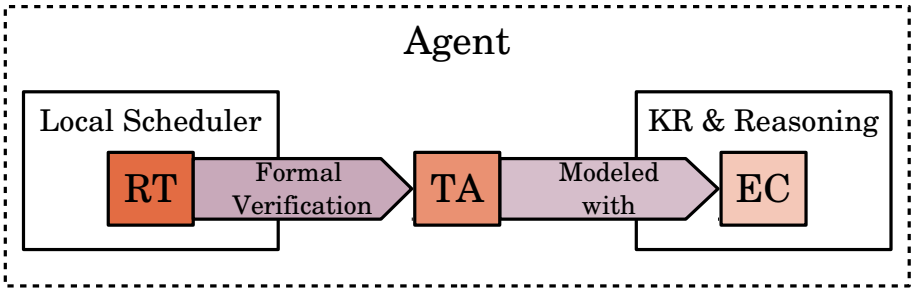


Figure 2.6: Agent's components logical mapping

In other words, if an agent (or an agent network) is represented as a Timed Automaton (or a Timed Automata network), it would be possible to implement it in Event Calculus formulas, without having the need of third parties model checkers or ad-hoc languages. Then, once a comprehensive EC theory of such agent(s) is built, different EC reasoning techniques might be used depending on the context [18]. For TA formal verification, the most suitable reasoning technique would be Model Finding, instead, as needed in the previous domain, when a rule-base must be continuously checked at runtime, Deduction shall be preferred choice.

2.7.3 Modeling Technique

Another contribution of this chapter is to propose a technique to model Timed Automata execution semantics and structure by using Event Calculus elements, such as facts, events and fluents. It is done by means of a logic theory articulated into two components:

- An automata-independent theory that contains the general machinery for the TA semantics;
- An automata-dependent theory that instantiate an actual TA, representing its graph structure, states, transitions, guard constraints and clock assignments.

This two-part design allows modularity and incremental programming: in fact, to create a new Automaton, it will be enough to write the corresponding Automata-dependent theory, without the need to modify the machinery.

2.7.4 Automata-Independent theory

The Automata-independent theory contains the general machinery for clocks, guards, and state transition to work.

Clocks have been modeled by using integer-valued fluents named clk/1. Since real-valued fluents are not yet ready for practical use in the Cached/Reactive Event Calculus, only Discrete Timed Automata can be modeled. However, this should not be a limitation, since in Real-Time applications, time is usually measured by the system clock, and from a formal point of view, [40] has proven that a theory expressed in continuous time EC can be translated in discrete time EC without losing in expressiveness. As shown in listing 2.1, clock fluents are initiated and terminated by set/2 events, which implement the mechanism of clock setting and resetting.

Listing 2.1: Generic TA implementation: clocks set/reset

```
initiates(set(C,V),status(clk(C),V),T).

terminates(set(C,_),status(clk(C),Vold),T):-
        holds_at(status(clk(C),Vold),T).
```

When a set(C,V) event happens, the clock specified in the variable C gets set to the value V . All clocks are also incremented simultaneously by tick/0 events, which simulate the flowing of time in the system (listing 2.2).

Listing 2.2: Generic TA implementation: flow of time

```
initiates(tick,status(clk(C),Vnew),T):-
        holds_at(status(clk(C),Vold),T),
        Vnew is Vold + 1,
        not (happens(set(C,_),T)).

terminates(tick,status(clk(C),Vold),T):-
        holds_at(status(clk(C),Vold),T).
```

It should be noticed that in order to avoid ambiguity as a double fluent initialisation, a tick/0 event increments a fluent only if there is not any other set/2 event happening simultaneously. TA's states are modeled as simple boolean fluents. Since the system can only be in one state at a time, only one state fluent (for each TA instance) can hold at a certain timepoint. State transitions are instead represented as events, that can lead to the termination or initialisation of state fluents. Such events, identified with the variable Lab, shall be the labels of the transitions that goes from old states Sold to new states Snew. If a Lab event happens at timestamp T, clauses in listing 2.3 show that state fluents Sold and Snew are terminated/initiated only (i) if there is an arc going from the old state to the new one (with label Lab), (ii) if the system is currently in the correct state to perform the transition, (iii) and the guard relative to such transition is satisfied at that timestamp.

Listing 2.3: Generic TA implementation: state transitions and guards

```
initiates(Lab,Snew,T):-
```

```
        arc(Lab, Sold, Snew),
        holds_at(Sold, T),
        guard(Lab, Sold, Snew, T).

terminates(Lab, Sold, T):-
        arc(Lab, Sold, Snew),
        holds_at(Sold, T),
        guard(Lab, Sold, Snew, T).
```

2.7.5 Automata-dependent theory

The purpose of the Automata-dependent theory is to instantiate the actual automata by defining its graph structure, labels (of states and transitions), guard conditions, clocks and clock resets. Such instantiation is traduced into writing facts and mostly ground clauses that shall unify the variables defined in the Automata-independent theory. The translation from a TA in a graphical form to an Event Calculus theory will be shown by means of an example, which will consist in modeling the TA in figure 2.7. It models a simple timed system, that can be thought as a lightbulb and a button. If the button is pushed while the lightbulb is switched off, it will turn on; then if the button is pressed again within a certain time window, the lightbulb will shine even brighter, otherwise it will turn off again.

The Event Calculus modeling of this TA is performed as follows:

- The initial state of the system is specified by the initially/2 facts. These establish the value of the clock fluent clk(x) and which one of the state fluents holds at timestamp -1 (i.e. from the beginning). From the code in listing 2.4 it can be seen that the clock is initally set to 0, and the initial state of the system is the off state.

Listing 2.4: Light control system's TA: initial state

```
initially(status(clk(x),0)).

initially(off).
```

- The arc/3 facts in listing 2.5 model the graph structure of the automata. The first argument is the transition's label, the second one is the transition's old state and the third is the transition's arrival state. For each transition in the TA's graph, one of these facts must be present in the Automata-dependent theory. States that are not connected to others by any transition cannot be represented within the current technique; this is actually an advantage, since isolated states do not make much sense in both FSMs and TAs.

Listing 2.5: Light control system's TA: graph structure

```
arc(press,off,light).
arc(press,light,off).
arc(press,bright,off).
arc(press,light,bright).
```

- Guards are implemented as shown in listing 2.6. If a transition does not have any guard, it will be enough to put a guard/4 term with the first three (ground) arguments being the transition's label, the transition's leaving and arrival state, and the last one being an emtpy variable (to unify any possible timestamp). If instead some constraint has to be imposed on clock values, it will be enough to put a condition on the clock fluent's value in the body of the guard/4 clause, using the variable T instead of the empty variable.

Listing 2.6: Light control system's TA: guards

```
guard(press,off,light,_).

guard(press,light,off,T):-
        holds_at(status(clk(x),X),T),
        X > 3.

guard(press,light,bright,T):-
        holds_at(status(clk(x),X),T),
        X =< 3.

guard(press,bright,off,_).
```

- Clock resets are instantiated by event chaining. Since clocks can be reset by launching set/2 events (see section 2.7.4), they have to be generated automatically when the appropriate transition is taken in the TA. For this particular automata (fig. 2.7), code in listing 2.7 shows how the set(x,0) event is launched every time the TA goes from the off state to the on state, wrapping the press event that triggers the transition (effectively setting the clock x to 0).

Listing 2.7: Light control system's TA: clock reset

```
happens(set(x,0),T):-
        holds_at(off,T),
        happens(press,T).
```

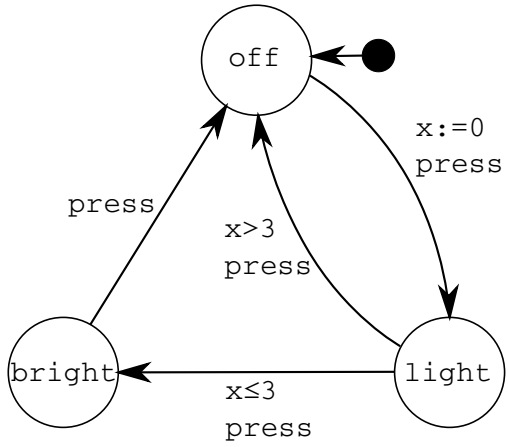


Figure 2.7: The TA describing a simple light control system.

2.8 Performance and Scalability Tests

One of the main limitations of logic-based approaches, and specifically of the Event Calculus, is reasoning performance. Thus, the reasoning engine variants (jREC-RBT,TEC-KD,TEC-IT) have been evaluated on the Medical and the Real-Time domains, with both synthetic and real event data, on a on a standard desktop computer running Java 8 on top of Ubuntu 16.04, with 16 GBs of RAM and a i7-6700k CPU.

2.8.1 Medical Rules Checking

To appreciate how the performances of the EC engines evolve when the number of events increases, a series of random datasets has been created, each one containing a different number of events.

The events of each dataset are fed into the reasoners one by one, and the time needed by each agent to trigger the alert is recorded. Every experiment is repeated one-hundred times to obtain the mean and standard deviation values.

In order to explore every test combination, two different event dataset conditions have been set up:

- The events in the dataset are concentrated in a short period of time, so that all of them happen inside the rule's time window. This will be called the "dense" events condition.

- The events in the dataset are spread across the time axis, so that the rule's timewindow contains only a fraction of them. This will be called the "sparse" events condition.

In this way, 4 test scenarios have been defined to study all the combinations of Rule-Type/Event-Condition:

- Complex Rule/Dense Events
- Complex Rule/Sparse Events
- Sequential Rule/Dense Events
- Sequential Rule/Sparse Events

For the Sequential Rule/Dense Events scenario, the tests had to be stopped at 500 events, due to too high computation times from the TEC-IT and TEC-KD engines. Instead, for all the other scenarios, tests could have been only run up to 3000 events due to the jREC-RBT engine consuming too much memory.

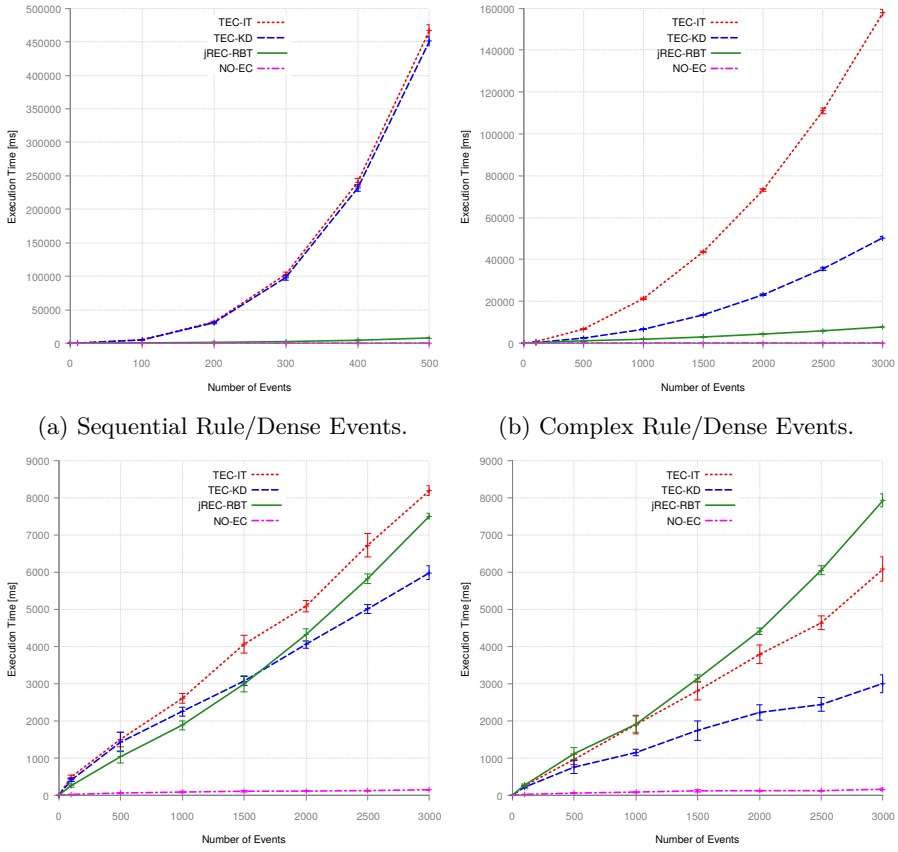
For the purpose of these tests, the use of synthetic datasets did not represent a threat to the experiment validity. It instead turned out to be a useful feature, since it allowed to stress the reasoners on critical tasks. It should be also clear that such datasets have been created to be as realistic as possible, with particular attention to event inter-arrival times and physiological values.

Absolute values of the execution times needed by a reasoner to trigger an alert is the most direct feature to evaluate the reasoning feasibility in a specific scenario. However, since they behave very differently depending on the Rule Type/Events Condition combination, the most efficient engine to adopt changes from case to case. As execution time absolute values also directly depend on the computer hardware, execution time trends represent a more stable feature across different machines. Such trends can be in fact used as a good insight to evaluate their scalability when the number of events to be handled increases significantly.

In the Sequential Rule/Dense Events scenario (fig. 2.8a), the execution time trends for the TEC-IT and the TEC-KD engines are shown to be growing very fast, in a (at least) polynomial fashion. The jREC-RBT engine is as well showing a polynomial-like trend (fig. 2.10a), but with a lower growing rate. It is also shown to be faster than the other two by almost two orders of magnitude (fig. 2.9a). For the Complex Rule/Dense Events scenario (fig. 2.8b), the situation is similar, even though less extreme. TEC-IT and TEC-KD engines are shown to have polynomial-like trends, with TEC-IT's trend growing significantly faster than TEC-KD's. JREC-RBT is also shown to have a slowly increasing polynomial-like trend (fig. 2.10b), and to be around one order of magnitude faster than the other two engines (fig. 2.9b).

Plots in figures 2.8c and 2.8d highlight a more balanced situation, with TEC-KD being the fastest engine among both Sequential and Complex Rule type. In the Sequential Rule/Sparse Events scenario (fig. 2.8c) TEC-IT turns out to be the slowest engine, with jREC-RBT showing just slightly faster execution

times. In the Complex Rule/Sparse Events (fig. 2.8d), the situation for the said two engines is the opposite, with jREC-RBT being the slowest one. Also the execution time trends are quite similar. In both figures 2.8c and 2.8d TEC-IT and TEC-KD exhibit a linear-like trend, with TEC-KD's trend slope being consistently lower than TEC-IT's. On the other hand, the jREC-RBT trend is less encouraging, as it shows a slowly growing polynomial behavior.



(a) Sequential Rule/Dense Events.

(b) Complex Rule/Dense Events.

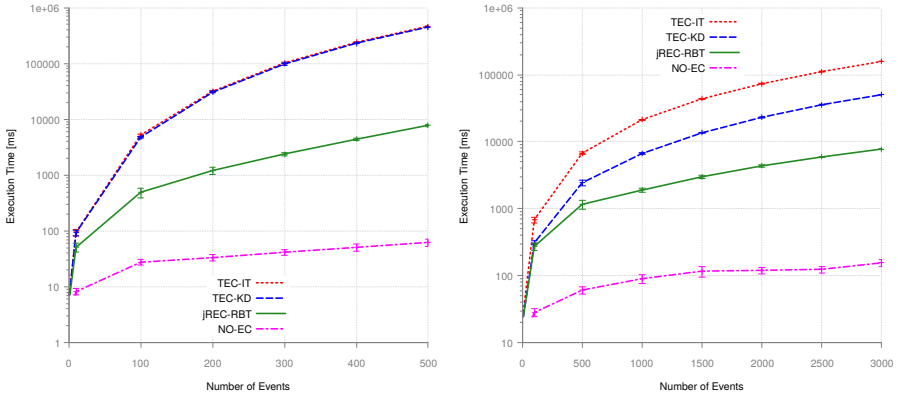
(c) Sequential Rule/Sparse Events.

(d) Complex Rule/Sparse Events.

Figure 2.8: Milliseconds needed by the three EC engines to compute an alert, for all the Rule-Type/Event Condition combinations.

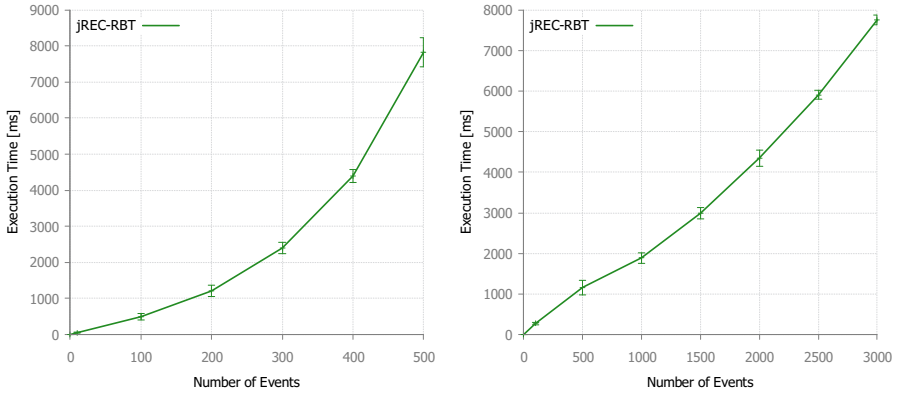
2.8.2 Discussion

A crucial parameter that affect the performance is the number of events that happen inside a rule time-window. The more this number increases, the longer it will take for the reasoning engine to check the rule (the events outside the rule's time-window are not checked). This is the reason for which the engines



(a) Sequential Rule/Dense Events (log). (b) Complex Rule/Dense Events (log).

Figure 2.9: Logarithmic scale plots of fig. 2.8a and fig. 2.8b



(a) Sequential Rule/Dense Events. (b) Complex Rule/Dense Events.

Figure 2.10: Detail of fig. 2.8a and fig. 2.8b showing only the jREC-RBT engine.

are usually slower in the Dense Event condition rather than in the Sparse Event condition.

From the perspective of Rule Types, the explanation of why the Sequential Rule is more difficult to check than the Complex, can be found on event ordering. For the Sequential Rule in fact, the engines not only have to check if some events did happen, but also whether or not they happen in a particular order.

Thus, by combining these consideration, it can be explained why the Sequential Rule/Dense Events represents the worst case scenario for the proposed engines, while the Complex Rule/Sparse Events is the easiest one. Moreover, it can be also asserted that the jREC-RBT engine is the most suitable one for the

Dense Events condition, while the TEC-KD is the one to choose for the Sparse Events condition. This is true for both rule types.

The reason for the engines in the Dense Event condition being usually slower than in the Sparse Event condition lies in how many events occurr inside the rule's time-window. The more this number increases, the longer it will take for the engine to check the rule (the events outside the rule's time-window are not checked). In other words, a crucial parameter that affects engines' performance can be defined as the ratio bewteen the average event inter-arrival time and the duration of a rule time-window.

From the perspective of Rule Types, the explanation for why the Sequential Rule is more difficult to check than the Complex, can be found on event ordering. For the Sequential Rule in fact, the engines not only have to check if some events did happen, but also whether or not they happened in a particular order.

Thus, by combining these consideration, it is clear why the Sequential Rule/-Dense Events represents the worst case scenario for all the proposed reasoning engines, while the Complex Rule/Sparse Events is the easiest one. Moreover, it can be also asserted that the jREC-RBT engine is the most suitable one for the Dense Events condition, while the TEC-KD is the one to choose for the Sparse Events condition.

Unsurprisingly, plots in figure 2.8 and figure 2.9 show that the NO-EC implementation outperforms all the logic engines by two orders of magnitude and shows a linear-like trend. This is not only due to the more efficient nature of hardwired solutions, as it also depends on different hypothesis that are made on events timestamps. The proposed logic-based approaches do not apply any simplifying assumption, and can correctly reason on not chronologically ordered data; on the other hand, to put the NO-EC engine as the lowest possible baseline, it has been implemented in such a way that the events have to be fed in chronological order to obtain correct results. In most monitoring scenarios it should be technically possible to keep this latter assumption and modify the logic-based reasoners accordingly, thus trading reasoning generality with performance. Anyway, at least in this chapter's context, in order to keep the PHS as flexible and as expressive as possible, the use of hardwired or Complex Event Processing solutions should be avoided, and the assumption on chronologically ordered data not applied.

2.8.3 Proof of Correctness

Now that the overall engine performance and scalability properties over the medical domain have been explored, a first step towards a system validation consists on observing how the proposed monitoring rule patterns behave on real data. Hence, for this purpose, the Brittle Diabetes rule in section 2.6.1 has

been tested on a publicly available Continuous Glucose Monitoring dataset [41], which contains hundreds of thousand of blood glucose recordings[1] from hundreds of patients. In the format of csv files, this dataset is essentially a set blood glucose logs, providing the measurements recorded by digital CGM devices for several patients, and in multiple time slots. As expected, each patient's blood glucose has been measured and recorded every 5 minutes, for periods that range from a minimum of about 2 to a maximum of 20 hours. After getting rid of few errors present in the data, the longest recording batches for each patient have been extracted, and the Brittle Diabetes rule has been tested on top of them.

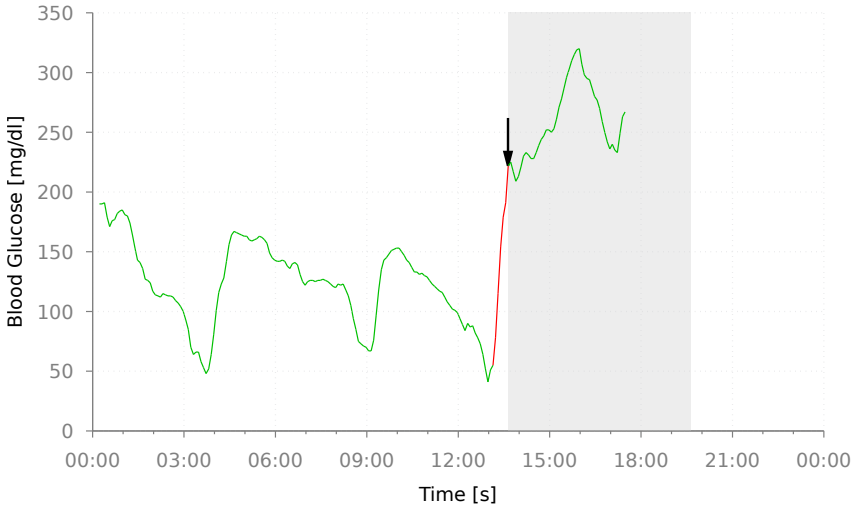


Figure 2.11: Single Brittle Diabetes Alert

Among all the plots extracted from the dataset, just two of the most relevant ones have been reported.

The red sections of the plots lines highlight the sequences of blood glucose events that triggered the Brittle Diabetes rule, and the arrows indicate the moment in which the alerts are generated. The grey regions represent instead the periods of time in which the alert generation is disabled, by means of the no-alert condition. This condition is necessary in order to avoid alert flooding, or in other words, to prevent that each new blood glucose event generates an additional alert (see implementation in sections 2.6.2 and 2.6.3).

Plots 2.11 shows a situation in which the rule is triggered only once. As expected, an alert is only generated when the blood glucose swings below and above the thresholds, inside the time window defined in section 2.6.1. Plot 2.12

[1] In this dataset, Blood Glucose measurements are found in a different unit of measure. Thus, for this test, target values in the Brittle Diabetes rule (see section 2.6.1) have been converted from $mmol/L$ to mg/dL.

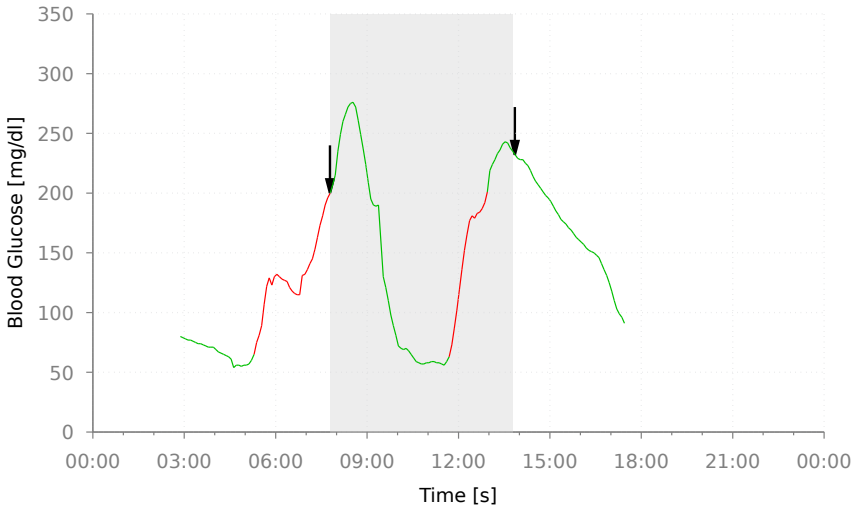


Figure 2.12: Double Brittle Diabetes Alert

shows instead how the engines behave when a double swing occurs. The first swing generates an alert normally, but, what should be noticed is that the second one is triggered slightly after the above-threshold blood glucose event happens. This latter event falls inside the no-alert window, so this swing's alert is triggered by the first blood glucose event that happens right after the no-alert window termination.

To sum up, these tests have proven that the Brittle Diabetes rule running on the proposed reasoning engines is suitable for analyzing real data coming from CGM devices. In fact, not only they have been useful to assess that the Brittle Diabetes rule behave well in a realistic scenario in terms on alert generation and alert flooding avoidance, but they've been also helpful to ensure that the number of events is appropriate to have a computationally feasible rule evaluation.

2.8.4 Timed Automata Simulation

As a very broad test landscape has been set up for the medical domain, this one will be much more compact, only considering two reasoning engines: (RBT) indexed jREC and standard jREC.

The number of state transitions occurrences (implemented in EC as events) and the number of TA instances (bigger TA-dependent theory) have been selected as the two main criterias to evaluate reasoning performance on the second domain. They can be considered as two orthogonal dimensions, and have led to two separate tests:

1. The first fixes the number of TA instances to one, while the number of

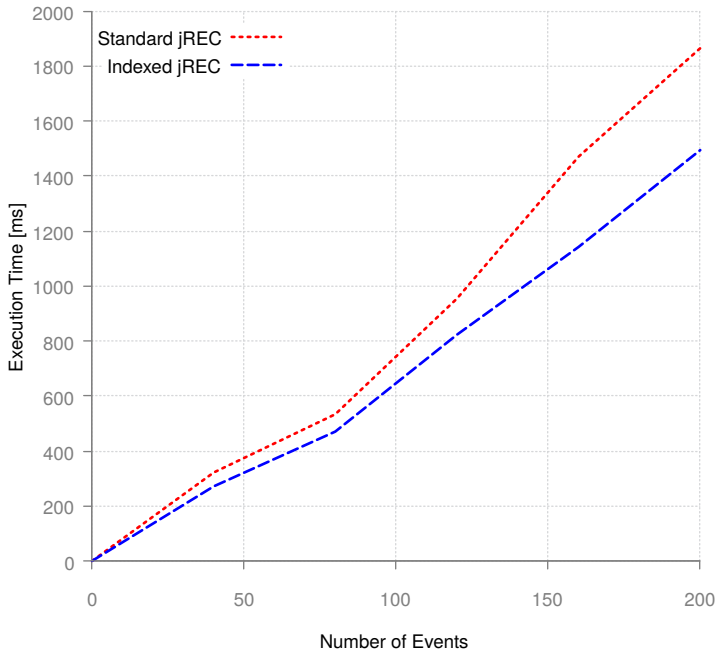


Figure 2.13: Execution time for a single TA instance run, with events spanning from 0 to 200.

events (state transition occurrences and clock ticks) spans from 0 to 200, with a step of 40;

2. The second test fixes the number of events to 200, while the number of TA instances spans from 0 to 5, with step 1.

Within this setup, performance has been evaluated by measuring the time needed by the jREC reasoner to execute TAs runs. The reasoner is fed with a list of events, which represent the state transitions occurrences, and as the output, a list of states (with relative time references) is returned as a list of MVIs. The TA's implementation chosen for the tests is the one shown in section 2.7.4, and the input events have been selected in such a way that every TA visits all of its three state cyclically. In addition to the state transition occurrence events, tick events had to be included to allow the clock fluents to work, thus modeling the flow of time in the system.

2.8.5 Discussion

Plots in Fig. 2.13 and Fig. 2.14 highlight that the indexed jREC performs generally better that the standard version. This was indeed expected, given

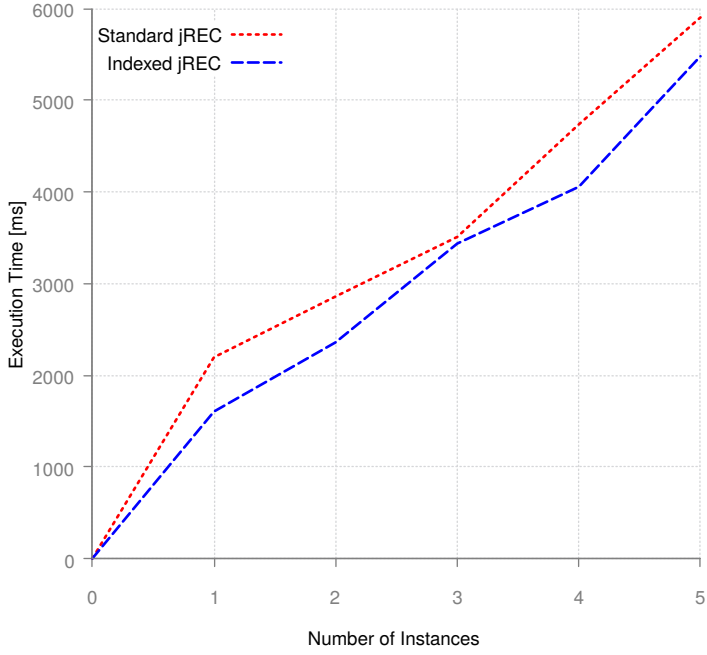


Figure 2.14: Execution time for multiple TA instances runs, with number of events fixed to 200.

the more efficent event management accomplished by the indexing, but this difference is more noticeable in Fig. 2.13's plot. It can be explained by the fact that for the multiple TA instances tests (Fig. 2.14), the number of events is kept constant, and the indexing mechanism does not help managing the bigger TA-dependent theory. The overhead caused by more TA instances prevails over the gain obtained by the event indexing.

More importantly, the execution time trends for both tests seem to follow a linear pattern. Even though this is a desirable behaviour in terms of scalability, other factors such as TA's complexity (number of nodes, number of transitions, etc) and inter-arrival time of transition occurrences over average guards conditions' time-windows might affect this trend considerably.

2.9 Conclusion

This chapter has explored the opportunities offered by the Event Calculus formalism by exploiting it as the main modeling and reasoning tool in two different structured domains.

Modeling-wise, two functional, compact and modular methodologies for representing medical monitoring rules and (discrete) Timed Automata have been presented. The logic-programming paradygm allowed to translate human-domain-knowledge directly into formulas, both from natural language (monitoring rules) and from formal definitions (Timed Automata).

Instead, focusing on the reasoning itself, the problem of efficiency has been discussed and tackled by integrating the deduction process with indexing data structures, for speeding up fact checking and retrieval. The proposed optimization techniques have then been tested on both domains, analyzing their behaviour depending on different factors, such as Rule Types, or event conditions.

Such tests have shown that the indexing techniques do improve EC efficiency while preserving the logic programming phylosophy, and are necessary for practical applications. Among the tested techniques, the one that showed the overall better performance was the jREC with the Red-Black Tree indexing. Tests on the real glucose monitoring dataset showed that the rules implementation behaved as expected, while those on Timed Automata highlighted promising execution time trends, suggesting that their simulation within the EC is indeed feasible.

Chapter 3

Statistical AI and Interpretability

When data, information, and processes of a certain domain do not appear to be in a "well-defined" structure, classical (symbolic) Artificial Intelligence techniques do not fit particularly well. In same cases, it is still possible to go on this path by employing approaches such as probabilistic logics or fuzzy logic, but their are often quite crumbersome, over-engineered, or too specific.

A good example is the biomedical domain, in which, even though tasks often rely on structured procedures, Deep Learning approaches are quickly becoming very useful across several medicine subdomains, especially in imaging. Diagnostic techniques adopted for medical tasks are usually based on a set of rules, but finding clear discriminatory features is not always possible. For this kinds of problems, Deep Learning is a well suited solution, because it automatically identifies patterns and correlations in large amounts of data, that even a human operator often struggles to identify. Moreover, they ensure significant performances, already higher than those of a human doctor in a lot of applications [42].

Other times, a system that shows complex behavior cannot be simply represented as whole, but rather by modeling its elementary components and their interactions. These elementary components are often called agents, as they are usually seen as entities that can perceive the environment, take some decision, and perform actions changing the said environment. For example, in optimization, closed models and optimal solutions can only be obtained mostly under simplifying assumptions over the real world. This is when heuristics and approximate methods kick in, with some of them actually getting inspiration from "natural Multi-Agent-Systems" such as ant colonies or bees.

Even if Statistical AI techniques are more effective in these contexts, they are not particularly helpful for explaining the problem that they are modeling, neither for justifying why a certain decision has been taken (e.g. by a neural network classifier). So, this chapter will not only show two Statistical AI techniques for medical imaging classification and agent-based optimization, but will also give some insights on how human-friendly interpretations can be extracted.

3.1 Introduction

In recent years, biomedical image analysis has been largely influenced by Deep Learning techniques. Due to the reliability and the maturity that such techniques are reaching, they are becoming the weapon of choice also for experimental automatic diagnosis systems. As lung cancers are a family of pathologies with one of the highest mortality rates, having an automatic system for speeding up the diagnosis would positively impact both patients and physicians sides. For these reasons, a novel Deep Learning methodology for analyzing Computed Tomography (CT) images of malignant lung cancers is presented, with the goals of predicting their histological conditions, and obtaining visual interpretations for such predictions. The histology prediction system is organized in a pipeline of four blocks. In the first module the lungs and extract the nodules are segmented from raw CT scans, in the second one, transfer learning is applied on a 3D convolutional neural network (C3D) for extracting features from filtered lung data; in the third one, the dimensionality is reduced with Principal Component Analysis, and in the last one, data is finally classified using an SVM. In terms of data, a dataset containing 904 samples (514 adenocarcinomas and 390 squamous cell lung cancers) has been assembled by selecting data from Cancer Imaging Archive and then adding other CT scans provided by the "Casa Sollievo della Sofferenza" research hospital. During the experimental phase, several configurations and setups have been tested, with the best one recording an accuracy on test data of 83%. On the interpretability side, usage of saliency maps is proposed for understanding how lung features lead to neurons' activations. Among the several analysis techniques available for the 3D-CNN in the proposed architecture, it is shown how they apply to CT images, and find out which ones are the most useful for explanation purposes.

In the last part of the chapter, an empirical analysis of the Dynamic Vehicle Routing Problem based on a Multi-Agent model will be shown. With the purpose of statistically evaluating how solutions to this problem evolve when varying global system parameters, an agent-swarm simulator has been implemented within the Netlogo framework, and used to extract experimental data. The experimental data allowed to discover surprising emergent behaviors of the system, which would have been impossible to guess in advance during the agents' design phase.

3.2 Related Work

Regarding the CT analysis scenario, images are usually segmented to extract only the relevant information (i.e. a desired organ) [43], while the rest is considered as noise. A possible approach for CT image pre-processing is the

one described in [44], in which a clean 3D image of the lung is obtained by segmenting each CT slice with a sequence of morphological operations. Initially, they applied rules from the DICOM standard [45] to filter out chest regions, then created a binary mask, performed consecutive dilations and erosions for removing blood vessels, and finally applied the binary mask for extracting the lung region. With this one being the most notable example, most of the works in this field employ some kind of preprocessing, highlighting it as a fundamental point for analysis of biomedical images, regardless of different frameworks developed for different approaches in the analysis.

Computer tools for helping or speeding up the diagnosis process usually fall under the name of CADe/CADx (Computer Aided Detection and Diagnosis). One of the most relevant CADe/CADx is DeepLung [46], a Deep Learning system designed for the classification of benign and malignant nodules. It is composed of two modules, one detecting suspect regions inside the lung by means of the combination of 3D Faster R-CNN [47] and a U-net-like network [48], while the second analyses malignancy condition of the 3D region of interest with a Gradient Boosting Machine [49]. Even if Deep Lung is a good solution for discriminating nodules' condition (malignant/benign), it was not suited for the purpose of classifying the histology of lung cancers. Limited to the cancer detection phase, the solution in [50] shows how the pre-trained Neural Network U-net [48] can be exploited for lung segmentation. Although U-net guarantees excellent results in terms of segmentation, many researchers still choose to work with more classic techniques related to computer vision, such as [44].

Instead of directly expoiting only the 3D information contained in CT images, this chapter follows a different approach, treating CT scans as videos. Deep Learning solutions for automatic video-analysis, which at first glance might seem unrelated, become instead of great interest for this particular task. For example, [51] investigates how different models are able to learn spatio-temporal features, and show how C3D, their proposed solution, improves state-of-the-art accuracy in video classification. Also, [52] has further proved the flexibility of 3D-CNNs, successfully applying them to another branch of 3D imaging, such as brain fMRIs. Regarding the network's training, a transfer learning approach has been chosen, as [53] shows that scratch training is usually the worst solution. In the medical context, transfer learning is even more advisable, due to the scarce availability of data, which often denies the possibility of an effective scratch training, and limiting the undesirable condition of overfitting.

To this day, a common problem of Deep Learning solutions in many fields is their "black-box" nature. This is even more critical in medicine, where nobody would really accept a diagnosis if the reasons that led to it are completely unintelligible to humans, without even considering regulations explicitly prohibiting this to happen (e.g. GDPR's "Right to explanation", articles 12-15 and 21-22).

In the last couple of years, several studies have started to provide answers to this problem, and expecially focusing of Deep Learning models [54]. In the medical imaging domain, techniques that extract visual explanations such as [55] are of particular interest, as they are able to assess in which measure each pixel contributes to the network output, essentially producing a pixel relevance or saliency map. Up to now, there is very little work done on applying such explanation techniques to 3D-CNN models for video or 3D image processing, as they have been mostly employed for "regular" 2D-CNN: to the author's knowledge, the one presented in this chapter is the first attempt to add interpretatability features to a diagnostic system based on 3D images and 3D-CNNs.

Deep Learning has also found its way into modeling agent behaviours. In particular, the Deep-Q Reinforcement Learning technique has quickly became the state of the art for many tasks which were previously considered untractable with classical logic-based methods [56]. Additionally, in some cases, emergent behaviours have been observed, with the agents managing to learn tricks or to exploit glitches in the mathematical models to continue improving their performance [57]. Even without the ability to learn, multi-agent systems can be used to model a variety of domains, from optimization to social phenomenons [58]. The fact that, many optimization problems, when taken under real life circumstance, quickly become intractable [59], opened the field to a whole research field of sub-optimal search strategies and heuristics that rely on constraint relaxation, simulated annealing, gradient descent variants and swarm intelligence. Swarm intelligence can be exploited for example for solving classical optimization tasks, such as with the Ant Colony Optimization strategy [60], a nature-inspired meta-heuristic for finding optimal paths in graphs.

3.3 Explainability and Interpretability

Until the advent of Deep Learning, Artificial Intelligence has always been explainable by definition, as it was usually intended as a way to formalize and translate human reasoning into computers and algorithms. By the way, in its most recent meaning, explainable Artificial Intelligence (XAI) is still a relatively young topic, and as many try to propose common taxonomies and standards, convergence is still far to be reached. Since the words "interpretability" and "explainability" have been already used, it is now the time to provide more details about what is intended with such concepts. [54] defines an explanation as a good answer to a "why" question (for classification tasks) or a "why should" question (for decision problems). So, in this chapter's scenario, this would adapt as finding an answer to the question "Why does this CT image contain an Adenocarcinoma/Squamous Cell Carcinoma?".

Talking about the "quality" of an explanation leads to the introduction of

interpretability and completeness. This are seen as two faces of the same medal: "interpretability" is how much the internal of a system is understandable by humans, while "completeness" focuses on describing an operation in an accurate way. It is immediately evident that a tradeoff arises, as the more complete an explanation is, the less interpretable it would be, and viceversa. The challenge of XAI here is then to produce both interpretable and complete explanations.

On a more technical side, there is already a quite broad landscape of techniques for obtaining visual explanation from Neural Networks models, or in other words, methods for calculating how an input image affects networks'dynamics according to different criteria, and computing saliency or relevance of pixel regions accordingly. [61] provides a Keras-based toolbox that wraps the implementations of many of the existing methods, organized in a three categories taxonomy:

- Function-based methods, which compute gradients of the output neurons with respect to input data;
- Signal-based methods, that alters the order of ReLU units during the computation of gradients;
- Attribution methods, for directly estimating how much each pixel is responsible for a certain output.

3.4 Deep Learning for CT imaging

The diagnostic procedure for lung cancer diagnosis is usually composed by a sequence of two different exams [62]: Computed Axial Tomography (CAT or CT) scan, and then a lung biopsy. With the first one, suspicious regions are identified, observing CT images, most of the time being small, roundish growths on the lung (nodules). In the second one, tissue samples from suspicious regions (nodules) are examined mainly by percutaneous agobiopsy or by bronchial or transbronchial fine needle biopsy, and their microscopic structure is carefully analyzed, in order to suggest a diagnosis. However, the presence of a nodule is very common, and it does not necessarily lead to a deadly condition; its nature can be either benign (non-cancerous) or malignant (cancerous). In the first case, the patient needs to be examined periodically, but its health is not particularly at risk. Instead, the malignant cancer's nature forces the patient to start being treated as soon as possible. In the unfortunate latter case, the cytologic or tissue sample helps to identify its peculiar nature. According to the different structures in the tissues, a malignant lung cancer can be classified in two families: small cell lung cancer (about 15% of lung cancers) and non-small cell lung cancer (about 85% of lung cancers) [63]. In the last one, [64] identifies another two

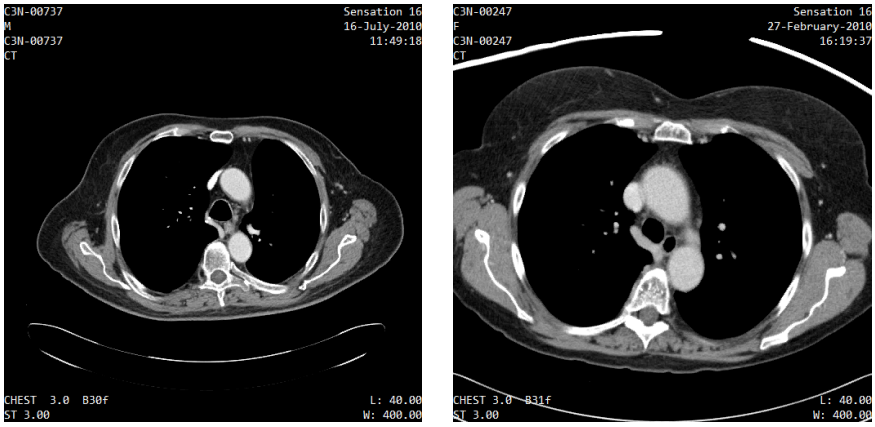


Figure 3.1: Sample chest slices extracted from two patients' CT images, showing an adenocarcinoma (left) and a squamous cell lung cancer (right). They highlight sources of heterogeneity (e.g. orientation, scale and bed position), non-relevant information (e.g. other organs), and give an idea of the classification task's difficulty.

main subfamilies, such as adenocarcinoma and squamous cell carcinoma (see fig. 3.1). Even though different lung cancer histotypes are more commonly found in certain lung regions [65], a radiologist physician can only assess the cancer's nature with a certain degree of confidence based on experience, until a sample is collected by invasive procedures. Considering the importance of an early diagnosis, these medical examinations should be performed as quickly as possible, trying to minimize the time interval between the two; despite that, delays of weeks or in some extreme cases of months are often recorded [66].

The DICOM standard [45] was created for identifying a common set of rules for collecting data and metadata, visualizing it and exchanging medical information about patients. The goal of this standard is to avoid heterogeneity among hospital databases known as Picture Archiving and Communication System (PACS). The product of the Computed Axial Tomography exam, a CT image, consists on a set of images that correspond to the sections of the anatomical structures inside the human body. Images can be generated thanks to the different absorption of X-rays through the different body structures. During the exam, a single section is usually not enough, so a sequence of them is produced. A CT image can be therefore considered as a full-fledged 3D image.

The atomic unit for representing this type of data structure is the voxel, which describes the volume unit. The DICOM standard regulates this structure, spatially describing the voxel with the triplet (z, x, y), which indicates, in millimetres, the size of every side along each of the three directions. As it will presented later, this is very important, since it represents one of the elements

of heterogeneity introduced by the different machines. Taking care of this was one of the challenges during the design phase of the system. The scale used to describe values contained in each voxel is called the Hounsfield Unit Scale [67], which represents the amount of electromagnetic waves absorbed (radiodensity) by the different tissues/materials inside the patient's body. Usually, working within the human body, the values range between -1,000 and 3,000. Although this scale of values has a totally different meaning than classical grayscale, it is possible to switch from one to the other, thanks to the following linear model:

$$HU = GrayValue * Slope + Intercept \tag{3.1}$$

The *Slope* and *Intercept* parameters are related to particular CT scanners, and described with more detail in the DICOM standard.

3.4.1 C3D and Transfer Learning

C3D is a neural network created for video analysis, trained on a dataset with over a million videos. It can be described with two macroblocks: a convolutional one, which has the task of extracting the features and the classifier, composed of fully connected layers. Following the transfer learning approach, the latter classifier block is discarded, and replaced with an ad-hoc Support Vector Machine. C3D is designed to accept an input tensor of dimension $(112, 112, 16, 3)$; more precisely, an input is composed by 16 frames (in this case slices), which are images with a $(112, 112)$ resolution. The last component of the tensor means that network is fed with 3-channels (RGB) images. Since CT images only have one channel, it had to be replicated it three times, obtaining a tensor of the right dimension. As mentioned above, the neural network has been split in two parts, dropping the fully connected one. Finally, the output of the the convolutional component of the network is a tensor with dimension $(4, 4, 512)$.

In terms of training, the usual strategies are: from scratch training, transfer learning or fine-tuning. On one hand, in pre-trained models, the network is seen as a composition of two macro-blocks: the convolutional one, which extracts features from data, and the fully connected one, which acts as a classifier. Such models can be adapted to substantially different domains, by only using the first macro-block as a feature extractor and replacing the last macro-block with another classifier (e.g. new fully connected layers or an SVM). Such technique falls under the name of transfer learning. On the other hand, a completely opposite solution is to train the model from scratch. A compromise between these two training modalities is to fine-tune a network, or in other words, taking a pre-trained model and slightly adapting the original parameters (i.e. weights and biases) with partial re-training on additional data (from the new application domain). In this regards, [53] shows that scratch training is usually the worst

Table 3.1: LUAD and LUSC samples from Cancer Imaging Archive datasets

Dataset Name	LUAD	LUSC	Total
NSCLC-Radiogenomics [71]	197	56	253
NSCLC-Radiomics [72]	51	152	203
NSCLC-Radiomics-Genomics [73]	42	36	78
TCGA-LUAD [74]	151	0	151
TCGA-LUSC [75]	0	72	72
LIDC-IDRI [76]	2	7	9
GiveAScan [77]	0	9	9
	443	332	772

solution. The choice of using a pre-trained network, therefore following a transfer learning approach, demonstrated to reduce the problem of overfitting, an otherwise very likely situation with small datasets [68].

3.4.2 Datasets

Apart from the limited data availability that often arises from privacy issues, also the scope of this study needed a new dataset to be assembled from different data sources. For example, the aforementioned LIDC-IDRI and LUNA16 datasets [69], which have been created for the classification of benign/malignant nodules from CT images, could not be used entirely, as for the most part, it does contain any information about cancer histological typing.

For this study, two histotypes of malignant lung's cancer have been considered: adenocarcinoma and squamous cell carcinoma. Choosing such two classes was both due to data availability, and the consequence of the diseases' incidence: they are in fact the two most common histological lung cancer conditions, and according to [64], 40% of all malignant lung cancers are adenocarcinomas, while about 30% are squamous cells ones. This study therefore embraces almost 70% of malignant lung cancer cases.

Thanks to the Cancer Imaging Archive [70] and its contributors, enough data from different datasets originally built for other (and often broader) purposes has been collected, by manually rearranging, filtering and selecting their contents. Finally, a uniform dataset with 775 samples in the form of CT images annotated with their lung cancer histotype has been obtained, specifically 443 samples of lung adenocarcinoma (LUAD) and 332 of squamos cell carcinoma (LUSC). Table 3.1 shows how the selected CT images are distributed across the several Cancer Imaging datasets.

Assembling a dataset in such a way leads to a very dis-homogeneous situation, since CT images have been mostly acquired with different imaging devices, and over three decades (from 1980s to 2010s). Even though this might represent a

disadvantage in terms of raw classification accuracy, it is actually a desirable property towards a more realistic system. An automatic diagnostic tool should indeed not be strictly tied to a particular CT scanner model or equipment configuration, but instead it should be robust enough to correctly process data from a broad range of devices and processes.

Apart from these public datasets, the "Casa Sollievo della Sofferenza" research hospital [78] has supported this study providing access to additional data in anonymized way. Pathology reports of patients with a diagnosis of lung cancer were extracted from the Unit of Pathology information system and CT scans of the chest of the same patients executed prior the pathologist diagnosis (3 months earlier at most) were extracted if available. Considering the already unbalanced condition of the assembled dataset, an additional 71 samples of lung adenocarcinomas and 58 of squamous cell carcinoma (129 in total) have been selected. Adding such data to the previous assembly was crucial for testing the system's generalization capabilities. Even if within this context some forms of heterogeneity are desirable (such as differences in colors and shapes), other forms have to be eliminated. For example, a significant difference has been discovered between public and private data relies in CT images' number of slices, as they approximately range from 2000 (private dataset) to 600 (Cancer Imaging Archive datasets). This depends on different CT devices used for the acquisition, and the relative DICOM parameters, which in this case, set CT images spatial information like the physical surface of a pixel and the physical distance between consecutive slices (a smaller distance leads to a better spatial resolution and so to an higher slices count). To tackle this problem, some of the slices from the private dataset's CT images have been filtered out during the first pre-processing phase, in order to have uniform data structures.

3.4.3 System Architecture

As mentioned above, the main goals of this system are to automatically discriminate CT scans into their relative cancer histotype (lung adenocarcinoma and squamous cell carcinoma) and generating both interpretable and complete explanations to let human operators understand the correlations identified by the learning system. Even if the classification task reduces to a binary problem, the proposed system could potentially be extended with relative ease to multi-class problems (in this case other lung tumor histotype), once having enough data to enrich the current dataset in a balanced way.

When designing the proposed CT image analysy system (shown in fig. 3.2), a modular approach has been followed, so that each block should be as independent as possible from the others. From an experimental point of view, this choice allowed to test a great number of setups, replacing only the content of the

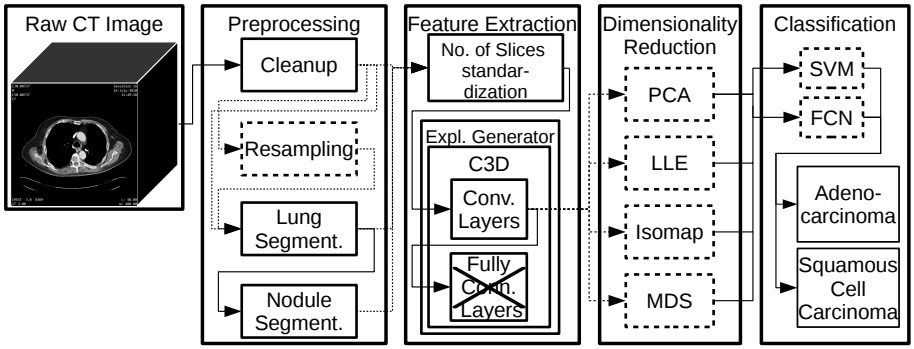


Figure 3.2: The proposed system's modular architecture. Dashed lines represent "alternative" data paths or components that can either be enabled or disabled in the pipeline. Regular lines instead represent "mandatory" paths and components.

blocks, without any modification of the global system. This approach allowed to quickly insert additional blocks during the experiments, such as for example, a dimensionality reduction block. There's a slight interdependence between the first two blocks, since the output of the pre-processing must be dimensionally coherent with the neural network input layer, but this is not a serious issue, as it can be seen as a global system parameter.

3.4.4 Pre-Processing

The preprocessing phase essentially consist in two different aspects: the elaboration of images to maximize their information content, and then the transformations to obtain dimensionally homogeneous data structures, according to standard DICOM triplet. As far as the first aspect is concerned, two different methods for segmenting CT images to extract regions of interest (ROI) have been tested, while for the second aspect a spatial transformation that modifies an image's voxel triplets (z, y, x) has been applied.

Such pre-processing techniques are performed in a sequential manner: the first (eventual) operation in the pipeline is the spatial transformation, in which the original CT image can be resampled to have a homogeneous voxel dimensions according to the DICOM standard (fig. 3.3); then, the ROIs are found by segmenting the CT image to isolate the lungs (fig. 3.4), and finally extracting nodules from the already segmented lung regions (fig. 3.5). All these solutions are chosen from the Kaggle's Science Data Bowl 2017 [79].

Regarding segmentation and ROIs extraction, they are performed one slice at a time, which is usually $(512, 512)$ greyscale image. Each one of them not only contains a section of the chest, but many unwanted components as well (i.e.

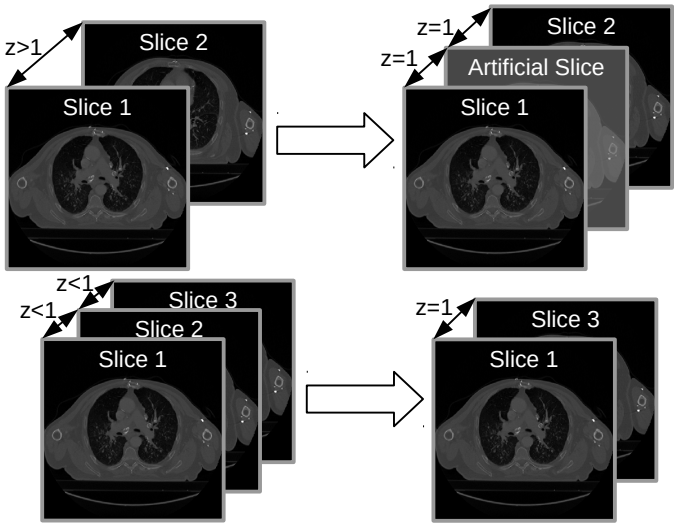


Figure 3.3: Effect of the resampling pre-processing on CT images' inter-slice distance. The artificial slices are generated by interpolation, often resulting in distortions or artifacts.

other organs, bones, artifacts, noise, CT devices' bed, etc.). For these to be fed in the chosen neural network model, they needed to be reduced to a $(112, 112)$ resolution, so first the slices have been cleaned up by cropping the borders, and then scaled donw to the desired resolution. Performing the cropping before the scaling allowed to lose less information, as usually CT images have lateral bands of approximately 100 pixels without any useful content.

3.4.5 Feature Extraction

After the pre-processing phase(s), the data is now ready for the C3D (section 3.4.1) pre-trained model (section 3.4.1), to perform the feature extraction.

Convolutional neural networks generally have a very rigid structure: the inputs must be dimensionally homogeneous. For this reason, given the heterogeneity in the number of slices for each CT image (see section 3.1), a way to dimensionally standardize the data (after being pre-processed) was needed, in order to feed it to the network. Thus, a maximum number of 512 slices per sample has been set, as a compromise between losing too much information discarding slices (from bigger CT images) and creating too much distortion adding empty slices (for smaller CT images). Another reason of choosing 512 is because it is a multiple of 16, the number of slices accepted as input by neural network, so that the CT image could have been split into exactly 32 blocks of 16 slices. As depicted in figure 3.6, slices at the beginning and at the end of a CT image have

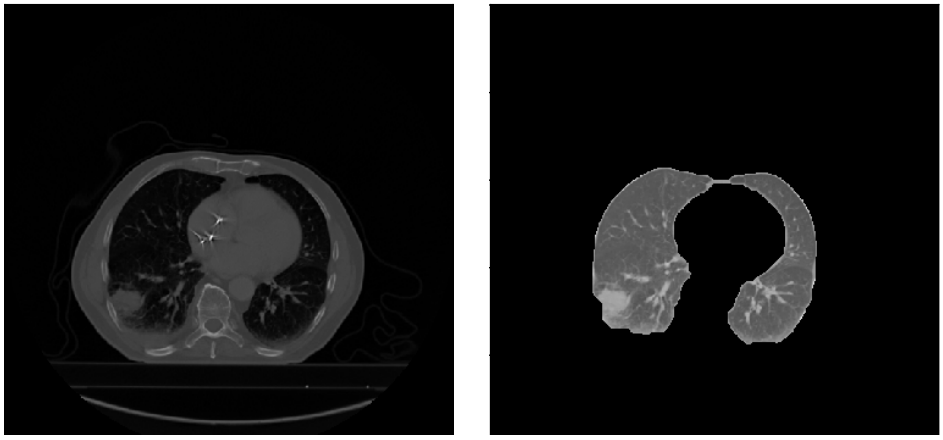


Figure 3.4: Effect of the watershed [2, 3] pre-processing (right) applied to a raw CT image's slice (left).

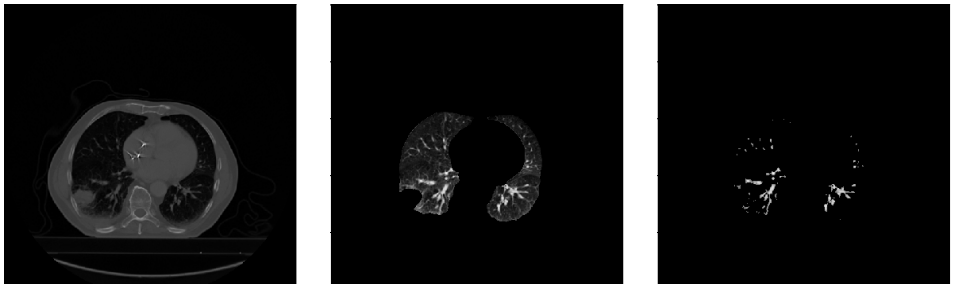


Figure 3.5: Effect of the nodule extraction pre-processing. Starting from the raw CT image (left), a watershed-like segmentation is applied (center), and finally the nodule are extracted (right).

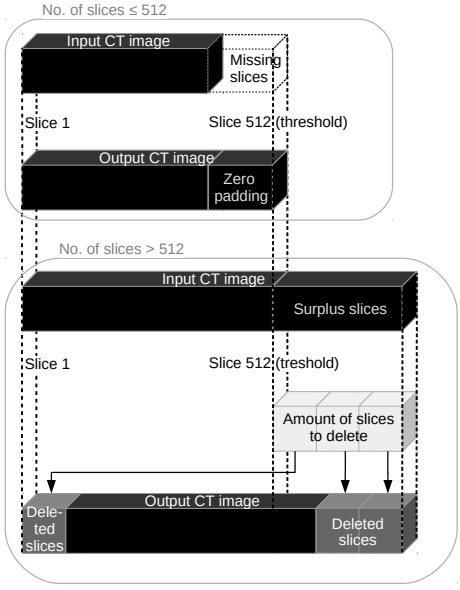


Figure 3.6: CT images' number of slices standardization, for both under-threshold and above-threshold situations. This has to be done before feeding the CT image to the C3D feature extractor, after the pre-processing phase.

been discarded, since they usually contain section outside the lungs. When CT image's number of slices exceeds the threshold, the difference between slices' number and the threshold is evaluated and divided in three parts: the first identifies the interval of slices to discard at the beginning of CT image and then last two the discarded interval at the end of a CT image. This one-to-two ratio empirically has been fixed empirically, since it usually happens to have roughly twice as much useless slices at the end of the CT image rather than at the beginning. In the case of short CT images, the difference between the number of slices and the threshold is filled with tensors of the same size of the slices, containing only zeros (zero padding). In this way it is possible to provide the network with a homogeneous structure of size $(112, 112, 512)$.

Finally, the network extracts a feature vector of 8192 elements (flattening C3D's $(4, 4, 512)$ output tensor) for each 16 slices block, and overall, for each "standardized" CT image sample, creates a $(8192, 32)$ matrix (512 slices per CT image, splitted into 32 blocks of 16 slices).

3.4.6 Classification

After extracting the features for every element in the dataset, classification has been carried out by means of an SVM or a Fully Connected Network.

Anyhow, having a $(8192, 32)$ matrix, or a 262144 elements feature vector for each sample, such a considerably high data dimensionality could potentially affect the classification accuracy negatively. Preliminary experiments showed that classification algorithms, and SVM more specifically, become inefficient when the number of features in a vector is greater than the number of samples. For this reason, additional solutions had to be applied, such as linear and non-linear dimensionality reduction techniques. For linear one, a regular Principal Component Analysis was chosen, reducing the feature vector to 775 elements (equal to the number of samples in the assembled dataset). This dimensionality reduction of 99.7% has significant effects on the algorithm, both in terms of training speed and accuracy. For the non-linear one, some manifolds learning techniques have been instead considered, such as: Isomap, Locally Linear Embedding (LLE), Multi-dimensional Scaling (MDS). In this latter tests, the original the feature vector is reduced to a representation of 400 elements.

For the classification task, two common SVM versions [80] (SVC and NuSVC) and a Fully-Connected Network (FCN) have been compared. There are two different implementations for SVM: they are characterized by different tuning parameters for regularization ("C" for the first one, "Nu" for the other one), that have been respectively set to 1.0 and 0.4. Such values have been set after preliminary experiments on the said classifiers, in which all possible combinations of "C" ranging from 0.6 to 1 and "Nu" ranging from 0.2 to 0.6 have been tested. Given the slight unbalancedness of the datasets, also adjustments to the "classweight" parameter according to classes frequency were needed. For both the implementations, radial basis function (RBF) and linear kernels have been tested, therefore coming up with a total of four different classifier configurations. For Fully-Connected Network instead, different solutions have been tried out, such as changing the number of layers and the number of the nodes within the layers. The best configuration is composed by two FCN, with 512 nodes in each one and the output layer, with two nodes (each for every class).

3.4.7 Experimental Results

Having to deal with a binary classification problem, in addition to the classical accuracy (on training and test sets) and AUC, also recall, precision and f1-score, have been evaluated, to get an idea of the behaviour of the model for the individual classes. Usually, binary problems are described as positive-class/negative-class (e.g. cancers vs non-cancer), but in this case it does not make much sense to apply such a distinction, given that both classes are two different cancers histotypes. These metrics have been then evaluated on both cancer classes.

The experiments were carried out using the 10-fold cross validation technique

Table 3.2: Metrics on Public Dataset (4 best configurations)

Class	Pre-processing	Classifier	Acc. Train	Acc. Test	AUC	Recall	Precision	F1-Score
LUAD	**Nodule Extr.**	**NuSVC (RBF)**	0.92 ± 0.00	$\mathbf{0.83 \pm 0.05}$	$\mathbf{0.87 \pm 0.04}$	0.83 ± 0.05	0.87 ± 0.04	0.84 ± 0.04
LUSC						0.82 ± 0.08	0.78 ± 0.07	0.80 ± 0.07
LUAD	Nodule Extr.	NuSVC (linear)	0.88 ± 0.00	0.81 ± 0.04	0.87 ± 0.04	0.81 ± 0.06	0.85 ± 0.05	0.83 ± 0.03
LUSC						0.81 ± 0.08	0.76 ± 0.07	0.78 ± 0.08
LUAD	Resampling, Nodule Extr.	NuSVC (RBF)	1.00 ± 0.00	0.77 ± 0.05	0.87 ± 0.04	0.84 ± 0.07	0.78 ± 0.09	0.80 ± 0.05
LUSC						0.69 ± 0.10	0.76 ± 0.11	0.71 ± 0.08
LUAD	Nodule Extr.	FCN	1.00 ± 0.00	0.81 ± 0.04	0.81 ± 0.04	0.82 ± 0.05	0.85 ± 0.03	0.84 ± 0.03
LUSC						0.80 ± 0.07	0.77 ± 0.06	0.78 ± 0.06

Table 3.3: Metrics on Private Dataset (4 best configurations)

Class	Pre-processing	Classifier	Acc. Train	Acc. Test	AUC	Recall	Precision	F1-Score
LUAD	**Nodule Extr.**	**NuSVC (RBF)**	0.94 ± 0.00	$\mathbf{0.79 \pm 0.02}$	$\mathbf{0.85 \pm 0.02}$	0.83 ± 0.04	0.81 ± 0.03	0.82 ± 0.02
LUSC						0.74 ± 0.04	0.77 ± 0.05	0.76 ± 0.04
LUAD	Nodule Extr.	NuSVC (linear)	0.91 ± 0.00	0.78 ± 0.02	0.84 ± 0.02	0.80 ± 0.04	0.82 ± 0.03	0.81 ± 0.03
LUSC						0.76 ± 0.05	0.75 ± 0.04	0.75 ± 0.03
LUAD	Resampling, Nodule Extr.	NuSVC (RBF)	1.00 ± 0.00	0.76 ± 0.06	0.85 ± 0.06	0.84 ± 0.06	0.77 ± 0.09	0.80 ± 0.06
LUSC						0.67 ± 0.11	0.75 ± 0.08	0.70 ± 0.08
LUAD	Resampling, Nodule Extr.	NuSVC (linear)	0.98 ± 0.00	0.73 ± 0.05	0.80 ± 0.05	0.76 ± 0.06	0.76 ± 0.08	0.76 ± 0.06
LUSC						0.69 ± 0.07	0.69 ± 0.05	0.80 ± 0.05

in order to prevent overfitting. Plots in figure 3.7 shows the best setups' behaviour among each fold, for both public and private datasets. Plots in figure 3.8 show instead the 10-folds average ROC curve for every system configuration, both on public and private datasets.

Table 3.2 shows the results obtained on the Cancer Imaging Archive: in the first line the metrics refer to adenocarcinoma (LUAD) as the positive case, while in the second line they refer to squamous cell carcinoma (LUSC) as positive case. It can be noticed that the metrics calculated considering squamous cell lung cancer as positive class have a greater variance, but this can be interpreted as an effect of dataset's unbalancedness. Specifically, the experiments with the nodule-extraction pre-processing method ensure the best performance, in terms of general accuracy and the various metrics (recall, precision, f1-score). It is interesting to point out that with the nodule-extraction pre-processing, using the NuSVC algorithm with the RBF kernel, the difference in performance comparing adenocarcinoma and squamous cell is very small. Replacing NuSVC's RBF with a linear kernel results in the second best experimental setup, obtaining just slightly lower performance. As shown in table 3.2, the NuSVC (RBF) classifier is always preferable, even with the other pre-processing methods.

Table 3.3 shows the results on the private datasets (Cancer Imaging Archive's and Casa Sollievo della Sofferenza's joined datasets). Compared to the previous case, they show a slight drop in performance, which can be related to the increased heterogeneity of the joined datasets (e.g. due to largely different CT scanners). This implies that it would probably be necessary to think about a homologation for this kind of data, prior to pre-processing, so that software systems could handle them more effectively. As in the previous case, the two

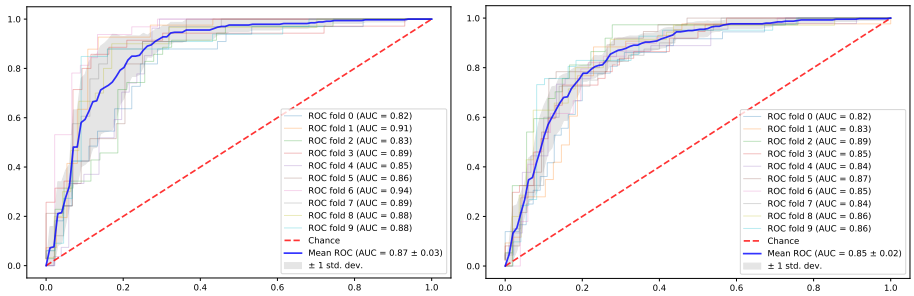


Figure 3.7: ROC curves of the best experimental setups for public dataset (left) and for private dataset (right), evaluated with 10-fold crossvalidation

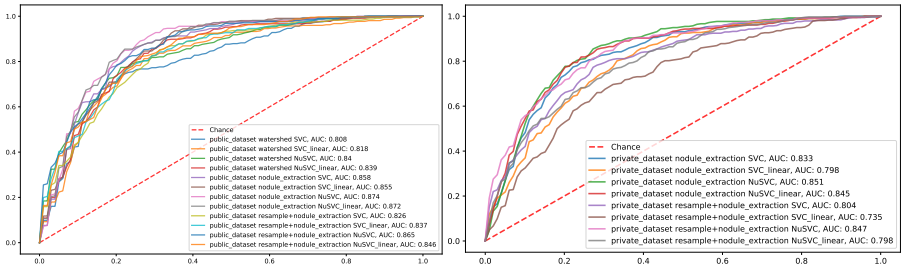


Figure 3.8: ROC curves of the experimental setups, evaluated on public dataset (left) and on private dataset (right). The AUCs are slightly higher than the ones in the tables, as they are automatically calculated with floating point (non-rounded) class values.

best experimental setups turn out to be composed of nodule-extraction pre-processing method combined with the NuSVC algorithm, respectively using the RBF and the linear kernel. In this case, almost similar results to previous dataset have been obtained, both generally speaking and for single classes.

Ranking the three pre-processing methods, it can be stated that nodule-extraction is unsurprisingly the best, as it allows the system to focus just on the most information-rich part of the lungs. The faster lung segmentation implemented within the nodule extraction pre-processing seems to be good enough for the task, and using the more accurate watershed transformation is simply not worth it in terms of computational resources. Even if the watershed-only pre-processing method segments the lungs more accurately, the extra non-relevant information eventually causes a slight drop in performance (with respect to the nodule-extraction method). Lastly, what is instead counter-intuitive is that resampling the CT images actually results in a performance loss: despite this procedure standardizes the CT images slices' distance, it causes the introduction of artifacts which ultimately leads to a distortion in the

information content.

3.4.8 Explanation Generation

In order to produce saliency maps of CT images that given as an input, the analyzer module has been attached "around" the feature extraction process performed by C3D. Since a transfer learning approach is being used, saliency maps cannot be directly computed from the output layer of the system, as the last layers of the network have been replaced with dimensionality reduction and classifier blocks. Anyway, with the proposed methodology it is still possible to highlight the regions that lead to the strongest activation of the network's last layer, and thus to produce a 3D saliency map where a pixel intensity corresponds to a measure of how much it influences the later classification. Thus, as will later shown, it can be safely assumed that the generated explanations contain useful insights to understand why a certain cancer histotype classification is obtained.

Picture 3.9 shows how the explanation generation module fits inside the system's CT processing workflow. The explanations are generated by the analyzer block wrapping C3D, which is implemented with "iNNvestigate" [61], a Neural Network prediction analysis toolboox based on Keras and Tensorflow. While C3D extracts the features of a CT image, the analyzer computes 16 saliency maps (one for each of the slices in the input block) with the desired method. The maps are then upscaled to (412×412) pixels and blended with full size blocks (both raw and segmented). This procedure is ran for all the slices in a CT image, classifying and generating explanations for a block of 16 slices at a time, and by doing so, giving a visual interpretation of how lung features are responsible for the classification.

The iNNvestigate toolbox's interface has been designed to transparently be applied to all networks model, as long as they are end-to-end and written in Keras. It provides a plethora of prediction analysis methodologies and a convenient programming interface for easily accessing them; nonetheless, even though the network model was entirely written in Keras, some of the methods where not available due to implementation issues or incompatibility. This is most probably due to the fact that some of the methodologies included in the toolbox have been specifically created for "regular" 2D-CNNs, thus running into architectural errors or code exceptions when trying to wrap them around 3D network models. A list of methods, shown in Table 3.4, which have been found to be working with the particular Keras-based C3D implementation included in the system:

Apart for the unavailable analysis methods, still good explanations have been obtained for all the others, and it was also possible to assess which ones are the

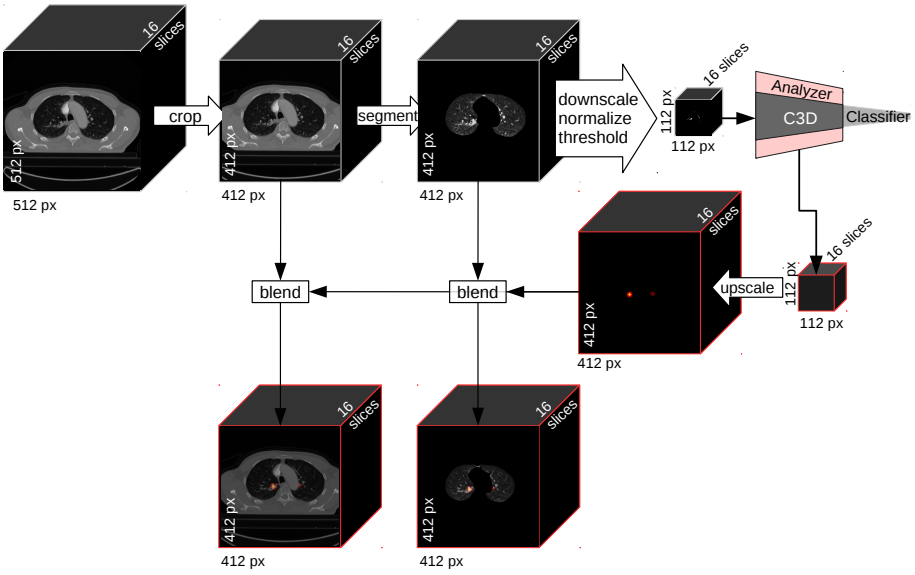


Figure 3.9: Explanation generation workflow. By blending the saliency maps with raw/preprocessed CT images, it was possible to highlight which parts of the CT image and the lung the most responsible for the network's activation.

best ones in terms of interpretability.

For demonstration purposes, the best six methods have been selected, and at least one for each class (Function, Signal and Attribution). As briefly shown in pictures 3.10 and 3.11, and more extensively in the links included in the captions, all of the six methods manage to highlight lung features, even though with different levels of noise, artifacts and dimension of regions. By looking at these aspect, it can be stated the best explanations for this particular network architecture and for CT images feature extraction, are the ones provided by LRP methods[1] (with "sequential a flat" and "sequential b flat" presets), as they show little to no noise, and clearly highlight variations in shape and position of lung details.

As expected, C3D is activated by parts that appear, disappear or "move" the CT scan, allowing for the network to detect them and generate features. In an further attempt to interpret the explanations by carefully looking at the highlighted regions of consecutive slices, it can be observed that adenocarcinomas and squamous cell carcinomas consistently show different nodules locations within the organ (peripheric vs central) and different shapes (ramified vs localized). These are the features that, among the others such as healthy

[1]LRP and Adenocarcinoma: `https://youtu.be/wZR0259TyAs` and Squamous Cell Carcinoma: `https://youtu.be/5hA0DBIqs3A`

Class	Name	Working
Function	gradient	Yes
	smoothgrad	No (incompatible)
Signal	deconvnet	Yes
	guided_backprop	Yes
	pattern_net	No (incompatible)
Attribution	input_t_gradient	Yes
	deep_taylor	Yes
	pattern_attribution	No (incompatible)
	lrp.* (layer-wise relevance propagation)	Yes
	integrated_gradients	No (exception)
	deep_lift_wrapper	No (exception)

Table 3.4: Compatibility of iNNvestigate's methods and C3D

bronchial tubes or loculi which are common to both classes, are the ones that ultimately allow the dimensionality reduction and classification algorithms to find a discrimination. Also, these differences in nodules' position and shape are confirmed by the medical literature [65].

3.5 Multi-Agent Interactions

Even though Deep Learning is undoubtely the most common statistical-AI approach, it still makes sense to give interpretations for other complex's systems behaviour, such as agent-swarm dynamics. Consequently, this section briefly presents an empirical analysis of the Dynamic Vehicle Routing Problem in multiple configurations, within a agent-swarm optimization scenario.

The Vehicle Routing Problem (VRP) is a class of problems in operations research, widely used in logistics for fleet management. In its most general configuration, it consists of minimizing the cost (e.g. time or distance) of serving one or more customers by one or more vehicles. Each vehicle has a limited capacity, and refers to one or more depot for replenishment. When the position of the customers nodes is not fixed, and not known a priori by the vehicles, the problem becomes known as the Dynamic Vehicle Routing Problem (DVRP) [81].

3.5.1 Multi-Agent Model of DVRP

In this setup of agent-based optimization, this section introduces a simulator developed within the Netlogo framework [82] for analyzing the behaviour of an agent-swarm designed for solving a DVRP. Given the practical impossibility of doing so in a closed or in a globally optimal form, the main use of this simulator is to empirically show how changes in the overall system's parameters (e.g. the number of agents in the swarm) affect global fitness metrics. Differently

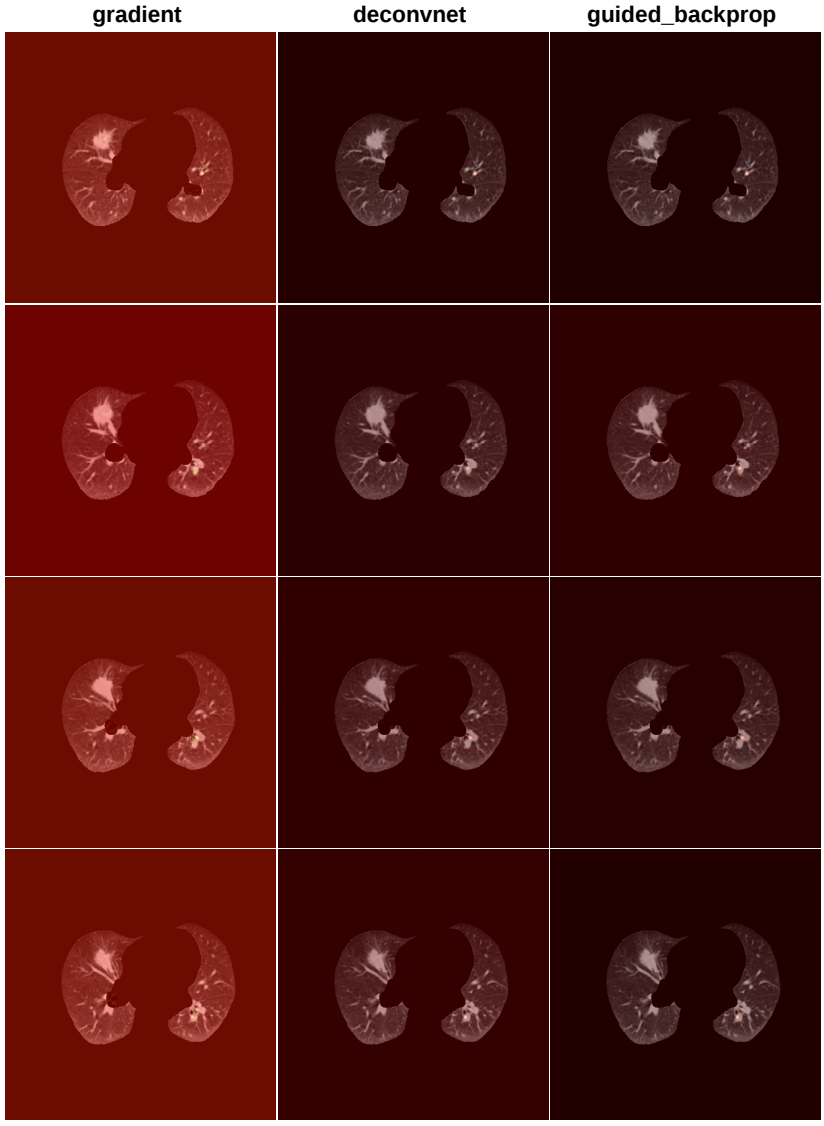


Figure 3.10: Five consecutive slices (from top to bottom) blended with saliency maps obtained by the "gradient" (youtu.be/MjU4y7avy0Y),"deconvnet" and "guided_backprop" (youtu.be/7aWjnLpJ-6c) analysis methods. For a better view check out the links.

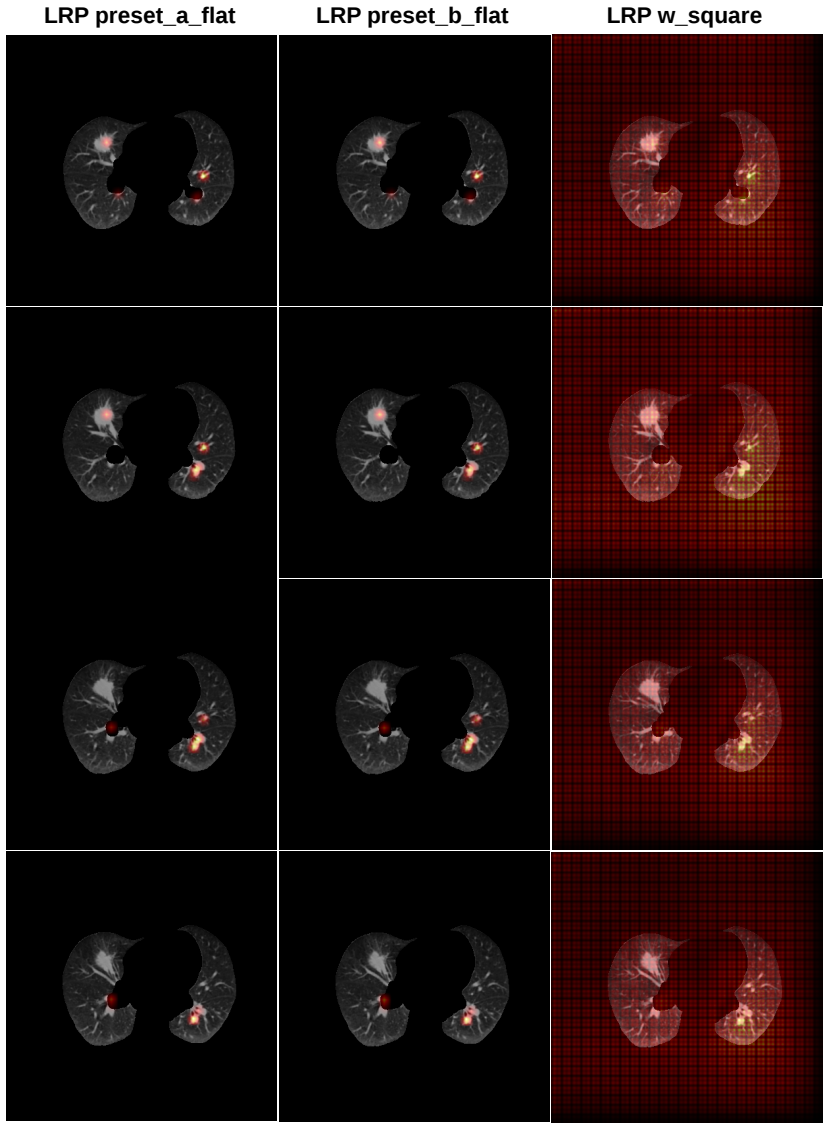


Figure 3.11: Five consecutive slices (from top to bottom) blended with saliency maps obtained by the "LRP a flat sequential preset" (youtu.be/wZR0259TyAs),"LRP b flat sequential preset" and "LRP w_square preset" analysis methods. For a better view check out the links.

from classical optimization problems, the simulator does not use graphs for data representation. Instead, it is based on a discrete 2-dimensional space, where traveling agents (TA) can freely move to serve as many customers as they can, until they run out of resources and have to go back to a depot for replenishment. Every TA can decide autonomously, in a completely distributed fashion, which customer it should target and where it should move, based on its own knowledge of the surroundings. It is important to highlight that the TAs do not interact with each other in any way, for example, not agreeing in advance who is going to serve which customer. Instead, the customers' positions are discovered dynamically as the TAs explore the 2-dimensional environment. If one or more customers happen to be inside a TA's field of view, and the TA has not yet run out of resources, it will follow the greedy policy of going towards the nearest customer. As soon as they run out of resources, the TA will point towards the depot, and start searching for new customers again. Then, when all the customers have been served, three overall fitness metrics can be computed:

- d, the distance covered by all TAs;
- t, the time needed for all the customers to be served;
- An overall cost $c = \alpha d + \beta t$, a linear combination of the previous metrics, where α and β are two normalization and weight constants.

By varying the system's parameters, the simulations allow to experimentally determine which are the most convienient configurations, and thereby, effectively enhance existing real-case fleets or help in making fleet management choices (i.e. how many vehicles, vehicles capacity, how many depots, etc.).

3.5.2 Experimental Setup

The entry point of the Multi-Agent System is the simulator's graphical user interface (see fig. 3.12), from which a certain configuration can be setup by manually specifying the values for the set of parameters, together with other general settings (visualization, simulation speed, etc.). Once done with the parameters initialization, a simulation can be launched, and the graphical components representing the 2-dimensional space, the physical entities (i.e. TAs, obstacles, depots) and abstract features (i.e. TA's remaining resources and field of view) will be displayed/updated accordingly. The customizable aspects of the system have been organized in two main groups:

- *Physical features*, that include the total number of TAs, the TA's loading capacity and the number of depots;

- *Smart features*, which comprehend TA's angle of vision, depth of vision, and the ability to keep memory of previously encountered customers (that they could not serve) over the 2-dimensional space.

The setup and running phases can also be performed automatically, by exploiting the Netlogo BehaviourSpace functionality. It allowed to effectively script the simulations, define parameter ranges, and to obtain an extensive configuration-space analysis. From an experimental point of view, Smart and Physical features have been treated as two independent sets of discrete variables. This led to the definition of two separate experiments, in which global metrics have been measured after separately varying Physical or Smart features values. This assumption of variable independence was indeed necessary in order to make the results both more graphically readable and to avoid combinatorial explosion of possible configurations (due to the cartesian product of the parameters' ranges). Results of the configuration-space analysis are shown in heatmaps (see fig. 3.13 and 3.14): every point represents the average of a fitness metric across 10 repetitions with a particular parameter's combination.

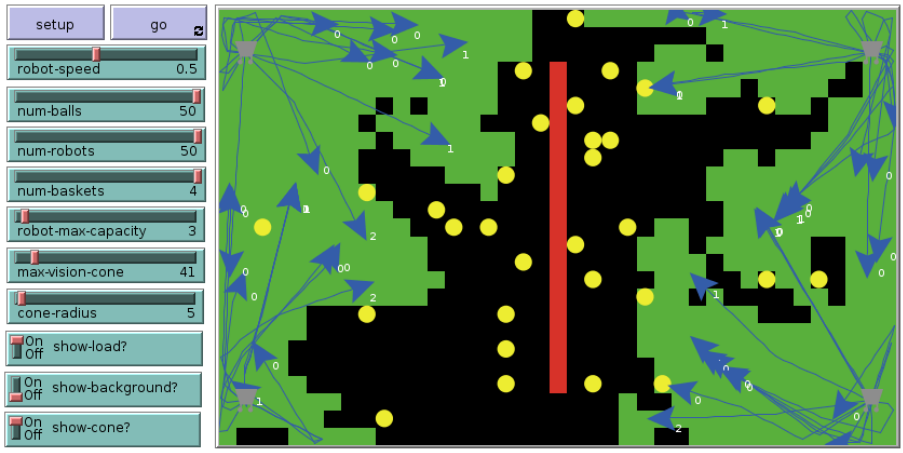


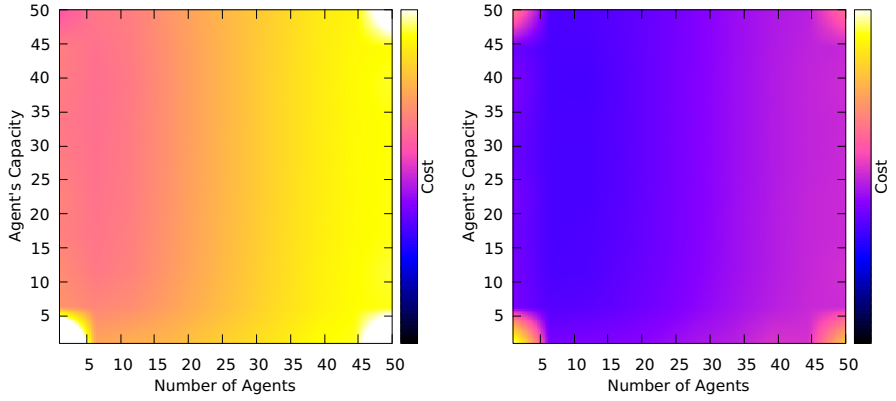
Figure 3.12: Simulator GUI showing tunable parameters and graphical counterparts of TAs (arrows), customers (circles), viewed cells (green), obstacles (red), load (white numbers) and depots (grey boxes).

3.5.3 Interpreting Agents' Collective Behaviour

Heathmaps in fig. 3.13 show how the cost metric c changes with respect to variations on the number of TAs, the TAs' capacity and the number of depots. As expected, the case with 4 depots show overall better performance than the other, and how TAs do not need a big capacity in order to obtain good solutions. In this case, the fact of having many TAs is more important than a large storage

capacity. Anyway, the number of TAs cannot be arbitrarily large, as the global distance d (which is part of the overall cost metric c) will grow as well. In the case of 1 depot, the best configuration turned out to be a single TA with the largest capacity.

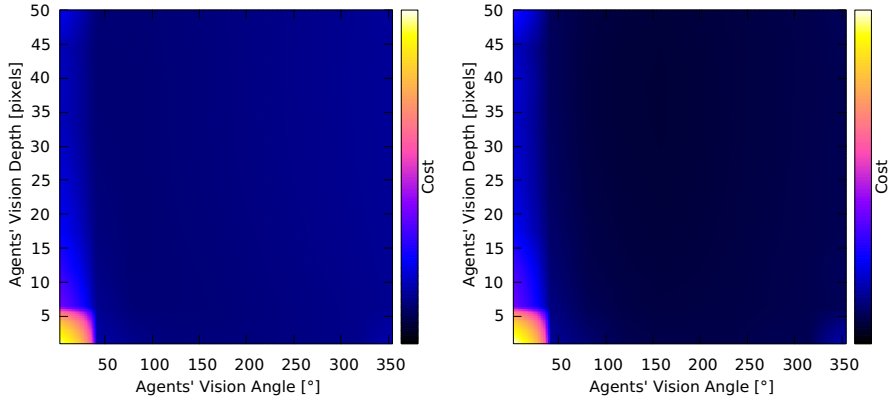
Heatmaps in fig. 3.14 show the effects of changing vision and memory features. They both highlight how the vision angle is predominant over the vision depth, and good performance can be obtained with relatively low vision capabilities. It is surprising to see that the memory feature actually has a slightly negative impact, but turns out to be more useful with limited vision capabilities (especially with low vision angles). This



(a) 1 Depot.

(b) 4 Depots.

Figure 3.13: Physical Features tests. The other parameters are kept constant.



(a) Memory

(b) No Memory.

Figure 3.14: Smart features tests. The other parameters are kept constant.

3.6 Conclusion

This chapter has first proposed a Deep Learning system for classifying the histotypes (adenocarcinoma and squamous cell) of patients' malignant lung cancers from the visual information provided by Computed Tomography images, and for generating explanations to better understand the reasons behind such classifications. It is composed as a modular pipeline of four blocks: the first pre-processes CT images to filter out non-relevant information, the second extracts CT images' features with the C3D's convolutional block, the third applies Principal Component Analysis to reduce features' vector dimensionality, and the fourth classifies the samples with two Support Vector Machine's variants. Explanations are generated by an ad-hoc Neural Network analyzing toolbox strictly coupled to the 3D-CNN model, which provides saliency maps from CT images given as input.

In order to test the system, a new dataset from several public sources gathered within the Cancer Imaging Archive has been assembled, and later enriched it with additional data from the "Casa Sollievo della Sofferenza" Research Hospital. The results are encouraging, showing an overall accuracy on the publicly available dataset of 0.83, f1-scores of 0.84 and 0.80 (respectively for adenocarcinoma and squamous cell lung cancer as positive classes), and an AUC of 0.87. The private dataset presents a small performance drop, which highlights one of the main challenges: CT images heterogeneity.

Also, within this particular implementation, the best explanations can be generated with the Layer-Wise Relevance Propagation method (a flat sequential preset), which successfully manages to highlight relevant lung features, and leads to a correct interpretation. Anyway, having an end-to-end Convolutional Neural Network would allow to produce much better explanations. Since such a network would lead to the reintroduction of the last fully connected layer(s), and thus to drop both the external dimensionality reduction block and classifier, more advanced segmentation and data homologation techniques would be needed to avoid a a significant drop in accuracy.

On the final part, this chapter also introduced an agent-swarm simulator for solving and analyzing the DVRP within different configurations, giving means for interpreting the system's collective behavior. After defining such configurations in terms of parameters, and proposing fitness metrics, experimental data have been extracted and used to have an understanding of how the agent-swarm behaves, for example discovering that agents with memory do the same (or even worse) than agents with no memory, or figuring out that agents with relatively low "vision" capabilities can do almost as good as those with good "vision".

Chapter 4

Computing with Event-based Neural Networks

Since the recent advancements in the Deep Learning field, research on neural networks has mostly shifted towards Machine Learning and pattern recognition. While groundbreaking results are still achieved everyday in classification, regression, segmentation, reinforcement learning and many other tasks, inherent architectural constraints limit the use of such technologies in a pervasive and more impactful way.

This chapter will focus on stressing the need for different neural models and architectures, as a way to enhance both efficiency, transparency and computational power of neural networks. As the primary tool for investigation, a new Event-based Neural Network Simulator is proposed and used to create novel neural circuits that implement specific functions or algorithms, further suggesting the event-based paradygm as a general purpose computational model.

4.1 Introduction

As shown in the previous chapter, Deep Learni-ng and Convolutional Neural Networks are great tools for extracting information from semistructured data, when human-centered feature engineering and extraction is not straightforward. Even though the recent rise of such techniques is mostly due to massively parallel computing architectures for speeding up matrix operations and to the possibility of harvesting big amounts of data, still the mathematical models and the conceptual framework are more or less the same as 30 years ago, when multi-layer perceptrons and backpropagation have been popularized first [83].

Spiking (or Event-based) Neural Networks, which are also commonly known as "Third Generation" Neural Networks, represent instead an effective way to push Deep Learning's boundaries: by more closely modeling the biological neuron's dynamics, they drastically increase computational efficiency and broaden the spectrum of potential applications.

This chapter will first focus on the more theoretical aspects of neural computation, on the differences between "representing" a function or "computing" a function; then it will move to the Event-based Neural Network Interactive Simulator, from the needs and requirements that led to its development, to the more practical aspects regarding the actual implementation. In the last part, several neural circuits, which have been studied and designed with the proposed simulator, will be presented and dissected.

4.2 Related Work

Neuromorphic Engineering is an umbrella term for a whole set of techniques and methodologies that aim to emulate the human nervous system in man made systems, and more specifically, in recent years, there has been an increasingly growing interest in doing so on both computer hardware and software [84].

From the hardware perspective, the research on Very Large Scale Integration (VLSI) has been producing a handful of reconfigurable hardware platforms on which (Spiking) Neural Networks can be physically implemented, such as [85, 86, 87]. The purpose of such dedicated hardware platforms arises from a neuroscience side, primarily from the need of having tools for studying and reproducing realistic brain dynamics, given that with complex biologically inspired neuron models, large scale simulations via software are unpractical and do not scale well on standard parallel architectures. With the mostly widespread ones being CUDA and OpenCL, they are instead widely used as the main hardware accelerators for Deep Learning and Machine Learning applications based on linear algebra and tensor operations, but still they fail in reaching low power computing devices and to accomplish real edge-computing. To fill this gap, most silicon manufactures are trying to come up with their own version of "AI- accelerators", or, in other words, Application Specific Integrated Circuits (ASICs) for performing tensor operation faster and with low energy consumption [88, 89, 90]. Even if the first tests are encouraging [91], these computing platforms are mostly unaccessible for third parties at the present day, or restricted to few selected research centers. Moreover, such "AI-accelerators" do not really mimic nervous system's dynamics, as what they do is to be very efficient at linear algebra operations and completely ignoring any event-based or spiking activity, which are instead fundamental blocks in how information is actually processed in biological systems [92]. An example on how this paradygm can, in practice, boost the performance of data processing, is found in event-based cameras: also known as Dynamic Vision Sensors [93], they have been designed with standard CMOS processes to emulate the biological retina, and thus leveraging the event-based architecture to obtain extremely high dynamic ranges, fast response times, data-transfer bandwith reduction, and

power consumption. DVSs are rare examples of a (commercially available) event-based architecture being there not just for simulating and studying biological processes, but for substantial practical reasons.

As mentioned earlier, while simulation of biologically plausible large neural networks can be carried out with relative ease from a programmer's perspective thanks to mature software tools such as [94, 95, 96, 97], the main common problem until now is the high computation times needed for complex spiking dynamics' calculations; in addition, being designed for creating and handling large scale networks, they do not easily provide direct control over individual neuron and connections. Even though this is partially possible in [98], a simple graphical front-end to the NEURON environment [97] that allows GUI-based design and analysis of spiking networks, it is still does not provide mature enough usability and features.

While to "handcraft" a neural network does not make much sense from a classical Machine Learning point of view, since Deep Learning's purpose is to automatically fit a model to some data without human intervention, with event-based or spiking architectures this actually becomes meaningful. "Third generation" Neural Networks are in fact inherently more powerful than non-dynamic feedforward networks, with both recurrent and spiking models being already known to be Turing Complete [99, 100, 101, 102], or in other words, capable of computing any arbitrary function. Such universal behaviour can be achieved with relatively small networks (from tens to hundreds of units, considering a finite tape), reasonably leading to the assumption that special purpose (neural) circuits for implementing particular functions can be designed from an even smaller number of neurons. To this day, there is still not much work done on specialized neural circuits, nor on software tools to ease the design process, which can be effectively compared to the writing of a program both conceptually and in terms of complexity.

4.3 Beyond the Deep-Learning Conceptual Framework

Neural Networks have proven themselves to be amazing tools for several machine learning tasks: starting from feedforward (convolutional) networks for (image) classification, different models and architectures have been developed for time-changing domains, such as reinforcement learning [103, 104], speech recognition and signal processing [105]. Since recurrent networks play a big role in these applications, allowing networks to exhibit dynamic behaviour, and can be actually exploited for general purpose computation [100, 106, 107], this section will first focus on their mathematical model. Then, on a parallel track, classical

spiking neuron models will be introduced, as they are known to be very energy efficient (at least in biological systems) while not compromising computational power [102].

4.3.1 Recurrent Models

Feedforward network models are organized in layers, forming an acyclic graph, and have a finite response after an input is provided. Recurrent connections instead introduce loops, creating a cyclic graph, and enabling temporal dynamics. Such cyclic networks exhibit what is known as an infinite response, or in other words, after an input is provided, it can technically continue effecting a network indefinitely. Even if this does not seem like a desirable feature, it is instead what makes recurrent networks able to have memory and to be exploited for signal processing. Equations 4.1 and 4.2 show the mathematical model of such networks, in which the output of a neuron at a certain time depends on the state of the network at past times, without any constraint on the connections (the w_{ij} values).

$$x_i(t+1) = f(\sum_{j=1}^{n} w_{ij}(t) \cdot x_j + \sum_{j=1}^{m} \tilde{w}_{ij}(t) \cdot \tilde{x}_j + b_i(t)) \quad \text{for } i = 1, 2, \ldots, n \tag{4.1}$$

$$\vec{x}(t+1) = f(\mathbf{W}(t) \cdot \vec{x}(t) + \tilde{\mathbf{W}}(t) \cdot \vec{\tilde{x}}(t) + \vec{b}(t)) \tag{4.2}$$

The i-th neuron activation for the next timestep depends on the sum of all connected neurons' activations scaled by their respective synaptic weights (index j, or weights' matrix columns) at the current timestep, plus its bias.

More precisely, as the equations show, each quantity has an explicit time dependency, highlighting not only the neurons' activation variability, but also the possibility to change, according to some rule (not discussed here), for weights and biases. x_i and $\vec{x}$ are respectively the i-th internal neuron's activation and the internal neuron activation vector at time t (there are n internal neurons). w_{ij} is the weight of the connection going from neuron j to neuron i, which is also represented by $\mathbf{W}$ $(n \times n)$ matrix. Since every network has at least one or more inputs, input neurons are distinguished from the internal ones, indicating with $\tilde{x_i}$ and $\vec{\tilde{x}}$ the i-th input neuron's activation and the input neuron activation vector (there are m input neurons). Analogously, input weights are denoted as $\tilde{w_{ij}}$ or alternatively with the $\tilde{\mathbf{W}}$ $(n \times m)$ matrix. b_i and $\vec{b}$ are the i-th neuron bias and the bias vector, respectively. All the vectors are considered to be column vectors, and finally, f, the activation function, is intended to be applied element-wise in the matrix notation. The quantities in the model are usually either real numbers or rational numbers, while the most common activation

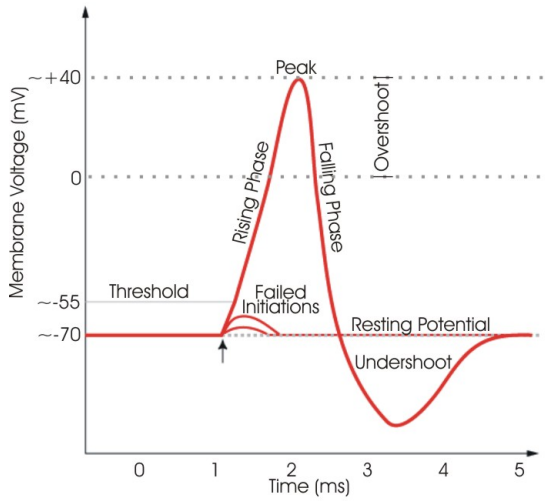


Figure 4.1: Behaviour of a typical spiking neuron.

function are shared with classical Deep Learning models, such as the threshold (or step) function, the sigmoid function, the hyperbolic tangent, and so on.

Talking about the weights, one fact is to state that they are dynamic, and another is to say how they should be changed. This is particularly true for recurrent models, in which backpropagation cannot be applied direcly [108]. Solving this problem by coming up with clever architectures that simplify training is what granted the success of models such as LSTMs and ESNs [105, 109], even though training strategies that work in the general case have yet to be identified.

4.3.2 Spiking Models

Spiking models more closely mimic biological neurons considering quantities such as (membrane) potential, capacitance, resistance, synaptic delay, plasticity, and many others, depending on the level of the desired biological accuracy level (the more accurate the model, the difficult the simulation will be). Picture 4.1 shows a typical pattern of a neuron's membrane potential evolution, together with the several stages the neuron goes through. The most important feature of spiking neurons is their efficiency: excluding the scenario of classical software simulations (which for particularly accurate biological models require intensive calculations), their physical realizations are one of the most efficient information processing systems known, being several orders of magnitude more efficient than classical CPU-based processing.

One of the most straightforward spiking models is the Integrate and Fire (IF),

in which the neuron is associated with a simple capacitor circuit subject to an input current. While the input current (if non-zero) charges the capacitor, the voltage across such capacitor increases ($V = V_0 + \frac{1}{C}\int Idt$): this voltage is thus taken as the model for the neuron's membrane potential.

Equation 4.3 shows the discrete-time dynamical model of the IF mechanism, considering not only the membrane potential charging, but also firing and refractory effects.

$$V_m(t+1) = \begin{cases} V_m(t) + \frac{I(t)}{c_m}, & \text{if } V_{rest} \leq V_m(t) < V_{thr} \\ V_{peak}, & \text{if } V_{thr} \leq V_m(t) < V_{peak} \\ V_{under}, & \text{if } V_{peak} \leq V_m(t) \\ V_m(t) + 1, & \text{if } V_{under} \leq V_m(t) < V_{rest} \end{cases} \tag{4.3}$$

$$V_{under} < V_{rest} < V_{thr} < V_{peak} \tag{4.4}$$

When the neuron is in the active state (first condition), it charges relatively to the input current $I(t)$ scaled by the membrane capacitance c_m, until the membrane potential $V_m(t)$ exceeds a certain threshold V_{thr}. Entering the firing state (second condition), $V_m(t)$ gets assigned to the voltage peak (spike) value, V_{peak}. Right after the spike, there is a sudden drop in membrane's potential (third condition), representing the beginning of the refractory state, in which the neuron is unreceptive to external inputs. Such state is needed in order to avoid excessive firing activity. While the membrane potential is below the rest threshold V_{rest}, it can only increase linearly with time, until it reaches the rest potential. This completes the state cycle, as the neuron is now active again. It should be noticed that for this mechanism to work, fixed voltage quantities must follow constraint 4.4.

The IF neuron can be slightly modified and made more realistic by adding a "leak" term to the membrane potential charging mechanism (first condition in equation 4.3). Maintaning the analogy with the aforementioned electrical circuit, there would be a resistor put in parallel to the capacitor, letting it not only charge thanks to the input current, but also discharge through such additional component. In this way, a neuron will not keep its membrane potential constant for an indefinite amount of time, as without any external input, it will slowly fade away. The first condition from equation 4.3 is thus modified as shown in 4.5.

$$V_m(t+1) = \begin{cases} V_m(t) + \frac{I(t)}{c_m} - \frac{V_m(t)}{R_m \cdot c_m}, & \text{if } V_{rest} \leq V_m(t) < V_{thr} \\ \dots \\ \dots \\ \dots \end{cases} \tag{4.5}$$

The only difference of this Leaky Integrate and Fire (LIF) neuron is the presence of the R_m resistance, which causes a spontaneus membrane potential drop during the neurons' active phase. For the membrane potential to be increased, the input current must satisfy $I(t) > \frac{V_m}{R_m}$, otherwise it will just get dissipated through the resistance.

As spiking neurons are only able to produce spikes, or in other words binary events, information cannot be directly encoded as rational numbers like in recurrent or feedforward models. Finding a mapping between such numbers and sequences of spikes is the process of defining a neural encoding. Among the many existing encodings that have been studied and proposed (both in biological systems and in computer science), the most common ones are:

- Inter-spike time encoding, in which information is stored in time intervals between two subsequent spikes;

- Spike-count encoding, that instead stores information in spike trains with different lengths and frequency (over a certain time window).

Last but not least, networks of spiking neurons require also synaptic models for translating an incoming spike going through a synapse into a change of the subsequent neuron's membrane potential. First off, spikes do not propagate "instantaneously" along axons and synapses: to avoid this, most neural simulators can in fact let the user specify custom synaptic delays for individual neurons or groups of neurons. Once a spike has went through the synapse, the neuron's potential is usually "instantaneously" changed by a certain amount, depending on the synaptic connection strength, or, in the case of more biologically accurate models, such increase is made to be less sudden, using, for example, decreasing exponential functions. Synapses can be either excitatory or inhibitory: an excitatory synapse leads to an increase of the neuron's membrane potential, which is also known as an Excitatory Post-Synaptic Potential (EPSP), while an inhibitory synapse will do the opposite, producing an Inhibitory Post-Synaptic Potential (IPSP). Synaptic plasticity regulates the behaviour of a synapse, by changing the strenght of its connections (depression and potentiation), based on previous spiking activity. This process goes under the name of Spike Timing Dependent Plasticity (STDP), and has been also exploited to train Artificial Spiking Neural Networks with some success [110]. By the way, there is no evidence of applications using STDP with fairly complex tasks and non-trivial network topologies. Thus, it is safe to say that training a Spiking Network is generally harder than a Recurrent Network.

4.4 Computational Power of Neural Networks

As temporal dynamic is added into neural networks thanks to recurrent and spiking models, not only they get more complicated and similar to their biological counterparts, but also their computational power increases, being able to learn a broader class of functions, and most interestingly, to perform universal computation.

One of the most peculiar attemps in having a neural network to actually perform "algorithmical" problems is found in [111], where a recurrent network controls the reading/writing head of an external memory element. By training such recurrent network as a controller for the memory's reading/writing head, the system is able to replicate simple tasks, such as element-wise copying from one memory location to another (up to a maximum length). However, results from [102, 100] have proven that there is no need for an external memory element for reaching theoretical universality, and that whole Turing Complete systems can be constructed solely of (spiking/recurrent) neural elements.

4.4.1 Turing Completeness

A Turing Machine (TM) is the most common theoretical tool for describing computation, as according to the Church-Turing thesis, they can compute any natural numbered function that can be calculated by an effective procedure.

As an abstract machine, it is characterized by three main components:

- An infinite tape that store symbols;
- A read/write head that can move to the left or to the right on the tape;
- A finite state machine (or a table) that, based on current state and read symbol, tells the head which symbol to write, where to move, and which state to enter next.

More formally, a single tape Deterministic Turing Machine is defined with a 6-tuple as follows:

$$T = \langle S, s_0, F, A, \beta, \delta \rangle \tag{4.6}$$

Where:

- S is the (finite and non-empty) set of machine states;
- $s_0 \in S$ is T's initial state;
- $F \subset S$ is the set of T's final states on which computation halts, also known as accepting states (an initial tape configuration is said to be accepted by T if it halts in one of F states);

- A is T's alphabet, a (finite and non-empty) set of tape symbols;
- $\beta \in S$ is the blank symbol, that denotes an empty cell of the tape;
- $\delta : (S \setminus F) \times A \nrightarrow S \times A \times \{left, right\}$ is the transition function, accepting a state S (excluding the final states F) and an alphabeth symbol as input, and associating a symbol, state and tape movement as output. The transition function can be partial, and if it is not defined for some input combinations, T halts. This function is commonly expressed as a table or a Finite State Automata.

By letting $\delta(s1, a_1) = \langle s_2, a_2, m \rangle$, an instruction of T can be expressed as a 5-tuple $\langle s_1, a_1, s_2, a_2, m \rangle$. As a sequence of "actions", an instruction can be seen as T being in state s_1, the moving head reading the symbol a_1 from the tape, causing a transition to state s_2. Then, the head writes the symbol s_2 on the tape, and finally moves to the left or to the right according to m.

While a particular Turing Machine can be seen as "program" that computes a specific function, systems that are instead able to simulate every possible Turing Machine are called Turing Complete, or (computationally) universal. In other words, for a system to be Turing Complete and thus to simulate a Universal Turing Machine (UTM), it should be able to accept as input both the program's "data" and a "description" of the program itself.

Among the many languages and systems that have been discovered to be Turing Complete, there is a relatively high interest in searching the "smallest" possible Turing Complete system [112], even though this highlighted a tradeoff between such systems' size (in terms of states and number of symbols) and the computational complexity needed to simulate a UTM. Other interesting examples of Turing Completeness that arise from quite simple or unexpected systems are: (i) the one-instruction-set computer [113], in which universality is reached with just one machine instruction known as the "Subtract and branch if less than or equal to zero" (or SUBLEQ), (ii) the "Rule 110" cellular automaton [114], a semi-chaotic cellular automata that shows remarkable (weakly) universal behaviour, (iii) and the "Magic: the Gathering" card game computer [115], a fully functional methodology for embedding any TM into a game of "Magic: the Gathering" by using standard cards and without breaking the game's rules.

4.4.2 Universality of Neural Networks

It is long known that neural networks are powerful learning tools, and part of their success throughout the years is also justified by the Universal Approximation Theorem [116]. It states that even a single hidden layer feedforward network containing a finite number of neurons is enough to approximate continuous functions on compact subsets of $\mathbb{R}_n$. More precisely, with the inequality

$|F(x) - f(x)| < \epsilon, \quad \forall x \in [0,1]^m$ it essentially states that any continuous function defined on the m-dimensional hypercube $[0,1]^m$ can be approximated up to a certain ϵ by the sum of all hidden layer neurons' activations $F(x)$.

Since feedforward models are basically a constrained graph topology (layers and no loops), in a sense, the Universal Approximation Theorem can be seen as a special case of a more general property related to recurrent and spiking networks (also feedforward networks' graphs are a special case of recurrent networks' graphs). Actually, when such limitations are removed, and recurrent connections between neurons are allowed (introducing temporal dynamics into the system), the "Universal Approximation" feature is not only maintained, but improved, enabling Turing Complete computation [102, 100, 107]. In other words, while a feedforward network can be used as a function approximator, recurrent networks do even better, being able to actually compute any possible function.

Moving towards spiking models, the fact of not directly representing information as rational numbers, but via some neural encoding, can be a threat for the Turing Completeness property. However, it turns out that also spiking networks are Turing Complete, thanks to Maas' construction using a continuous-time spiking model [101, 102]. Such construction aims at simulating particular instructions of a Turing Complete languages or systems with neural modules, encoding information on the phase difference between "spiking oscillators", and most notably requires inhibitory and excitatory synapses, as well as a "pacemaker" module. This latter component, defined as a spiking neural oscillator, provides a periodic source of spikes to several stages of the construction, and serves as a spike source synchronizer. Synchronization is indeed a necessary concept in Maas' construction, as it is based on continuous-time spiking model, in which infinitesimal time differences between spikes can lead to drastically different behaviour of the modules. This problem can be partially avoided by using a discrete-time spiking model, even though this would probably change the construction itself.

"Synchronicity" seems to be a constant part of Turing-Complete systems. From discrete-time models that have built-in synchronization, to physical systems, or continuous-time models, they all need synchronization to coherently process information. Apart from the previous example with continuous-time spiking neural networks, this is also confirmed in [117], which proposes a Turing Complete system based on water droplet fluid dynamics, that need synchronization from an external magnetic field to function properly.

After these considerations, the process of training neural networks gains a new perspective: while in the classical understanding it would mean to find the right weights that better approximate a function, with reccurrent or spiking models it can be seen as finding the right weights for computing a function (or

implementing an algorithm). Unfortunately, an automatic general procedure for fitting recurrent or spiking networks to any particular function (so that it can compute it) does not exist, as state of the art training techniques such as LSTMs' backpropagation-through-time, ESNs' min-squared, or spiking networks' STDP rule(s) focus more on the classical notion of training, such as, in this case, learning to approximate temporal patterns.

4.5 Event-based Neural Network Simulator

Even though recurrent and spiking networks are theoretically Universal Computers, to exploit them for actually computing a desired function is something that, at least for now, could not be done via automatic training. Encouragingly enough, complex behaviour can be obtained with relatively small networks (compared to the massive models used in classical Deep Learning), thus allowing for human-based design to be the central part of finding a network that implements specific functions or algorithms. However, state of the art neural network simulators and classical Deep Learning frameworks do not represent valid tools for this purpose, as they have been mainly developed either for analyzing the dynamics of biologically plausible spiking networks, or for statistical machine learning problems. This led to the development of a new tool, called the "eveNNt Designer" (from the acronym EVent-based Neural NeTwork), which will be presented in this section.

4.5.1 Mathematical Model

The proposed software includes its own simulator based on a mathematical model inspired from both recurrent and spiking neurons. First of all, it does not focus on biological similarity, as spiking dynamics are reduced to the skeleton, roughly following the IF model presented in section 4.3.2. With this model, it makes more sense to treat potential spikes as binary events, as all it is needed to know about them is their source (neuron) and timestamp. This actually suggests a connection with the Address-Event-Representiation (AER), a widely used data protocol in neuromorphic devices such as Dyamic Vision Sensors [93, 118]. All the computation is carried out in matrix form, thus also recalling recurrent networs' update mechanism.

Letting n be the number of neurons, other matrices and vectors needed for the simulation can be defined:

- $\mathbf{A}$ is the weights adjacency matrix, an (n, n) matrix describing the network's directed graph, with its elements $a_{i,j} \in \mathbb{Z}$ being the synaptic weights;

- $\vec{s}$ is the spikes vector, an $(n, 1)$ binary vector with elements $s_i \in \{0, 1\}$ that describe the presence of a spike incoming in neuron i;
- $\vec{p}$ is the potentials vector, an $(n, 1)$ vector with elements $p_i \in \mathbb{Z}$ storing the membrane potential of neuron i;
- $\mathbf{R}$ is the parameters matrix, an $(n, 3)$ matrix containing any i-th neuron's rest potential (r_{i1}), threshold potential (r_{i2}) and refractory time (r_{i3}).

The network update procedure will be described by using Octave/Matlab notation for matrix/vector multiplications, element-wise operations such as the product or inequalities, and referencing to single rows or columns of a matrix. A matrix-based methodology in this case is preferrable compared to iterating vector elements with loops, as apart from having a more compact notation, the operations can be handled much more efficiently by standard BLAS libraries. To perform a network update cycle, the following operations are executed:

1. $\vec{p_{inact}} = \vec{p} < \mathbf{R}(:, 1)$
   Gets all the neurons that have a membrane potential below the rest potential. This means that such neurons are in the refractive period, and thus, inactive. It produces an $(n, 1)$ binary vector, with ones and zeros corresponding to inactive and active neurons' positions respectively.
2. $\vec{p_1} = (\vec{p} + 1) .* \vec{p_{inact}}$
   Increases inactive neurons' membrane potential towards their rest.
3. $\vec{p_{act}} = 1 - \vec{p_{inact}}$
   Gets all the active neurons by inverting $\vec{p_{inact}}$.
4. $\vec{p_0} = \mathbf{A}^T \cdot \vec{s}$
   Calculates membrane potential increments for all neurons that are reached by a spike, according to their input synapses' weight. This is easily possible thanks to adjacency matrices' properties.
5. $\vec{p_2} = (\vec{p} + \vec{p_0}) .* p_{act}$
   Gets membrane potential updates only for active neurons, and stores them in a temporary vector.
6. $\vec{p_{lower}} = p_2 < \vec{\mathbf{R}}(:, 1)$
   Gets all neurons which membrane potential got below their rest after the update, due to their inhibitory (negative weight) presynaptic connections.
7. $\vec{p_2} = \vec{p_2} .* (1 - \vec{p_{lower}})$
   Sets to zero all active neurons' potentials that got below the rest.
8. $\vec{p_2} = \vec{p_2} + (\vec{p_{lower}} .* \mathbf{R}(:, 1))$
   Resets all lower-than-rest potentials to rest

9. $\vec{p} = \vec{p_1} + \vec{p_2}$
   Merges inactive and active neurons new membrane potentials back to a single vector. Notice that, with this procedure, if an element of $\vec{p_2}$ is zero, the element at the same row of $\vec{p_1}$ will be non-zero, and vice versa.

10. $\vec{s} = \vec{p} \geq \mathbf{R}(:, 2)$
    Generates the new spikes vector by checking which of the neurons' membrane potentials has reached its threshold.

11. $\vec{p} = (-\mathbf{R}(:, 3). * \vec{s}) + (\vec{p}. * (1 - \vec{s}))$
    Sets the membrane potentials of neurons that produced a spike to a negative value, according to their refractory time specified in $\mathbf{R}$, while leaving the non-spiking neurons' potential untouched.

Due to such matrix-based implementation, the simulation time depends solely on the number of neurons, as this is the only quantity that affects matrices' and vectors' dimensions. Therefore, the number of connections and the presence of spikes do not matter, as they are just values of such vectors/matrices.

4.5.2 Tools and Libraries

The eveNNt simulator is implemented in the C++ language, leveraging on the following tools and libraries:

- Qt Framework as the main development platform, providing good portability, maturity, robust and flexible gui development.

- the Armadillo library [119] for all matrix and vector operations carried out by the simulator backend. It essentially provides an Octave/MATLAB-like interface towards standard Basic Linear Algebra (BLAS) libraries such as Intel MKL, CUBLAS, and OpenBLAS. When Armadillo is compiled, a particular BLAS library can be chosen as its core; therefore, replacing one BLAS library with another does not require any change in the code, but only to rebuild the library linked with the new BLAS core.

- the QVGE graph editor [120] as the starting point for the graphical frontend of the application. It is an easy to use open-source graph editor written in C++/Qt that allows to handlily create and customize graphs both on a graphical and symbolic (add labels and attributes to nodes and edges) point of view. It also includes keyboard shortcuts, action undo/redo, graph export in different formats, and integration with the OGDF library for advanced graph visualization/manipulation.

4.5.3 GUI and Features

Summing up, at current state of development, the eveNNt Simulator offers the following features:

- Appearence editing (node/edge color and shape, customizable grid and canvas);
- Network structure editing (nodes and edges);
- Efficient visualization thanks to the Qt hardware accelerated rendering engine (it can easily visualize thousands of nodes and edges on regular desktop pc);
- Custom attributes for nodes and edges;
- Group selection, group-based edits;
- Ordering and alignment of nodes (OGDF layout algorithms and snap to grid);
- Run, pause, step neural network simulation;
- Fast simulation, manages to run networks with thousands of nodes on regular desktop hardware, without GPU (CUBLAS) parallelization;
- Visualize network dynamics in real time during the simulation;
- Active and responsive UI even during the simulation;
- Import/export networks status, including custom attributes and simulation status, to standard graph representation formats.

Pictures 4.2 and 4.3 show two screenshot of the eveNNt Simulator frontend, highlighting some of the aforementioned features [1].

Even though the included functionalities are already enough for performing experiments on spiking neural networks designs, other features should be added for a better usability and overall completeness of the software:

- Show and export temporal plots of time-changing information such as the membrane potential, or the spike history;
- Add visible labels to nodes and edges, containing both time-changing parameters, such as the membrane potential, as well as fixed ones, such as thresholds and synaptic weights;

[1] The interested reader can visit the following link for a better view of the eveNNt Simulator in action. `https://youtu.be/4_SMYHcARqA`

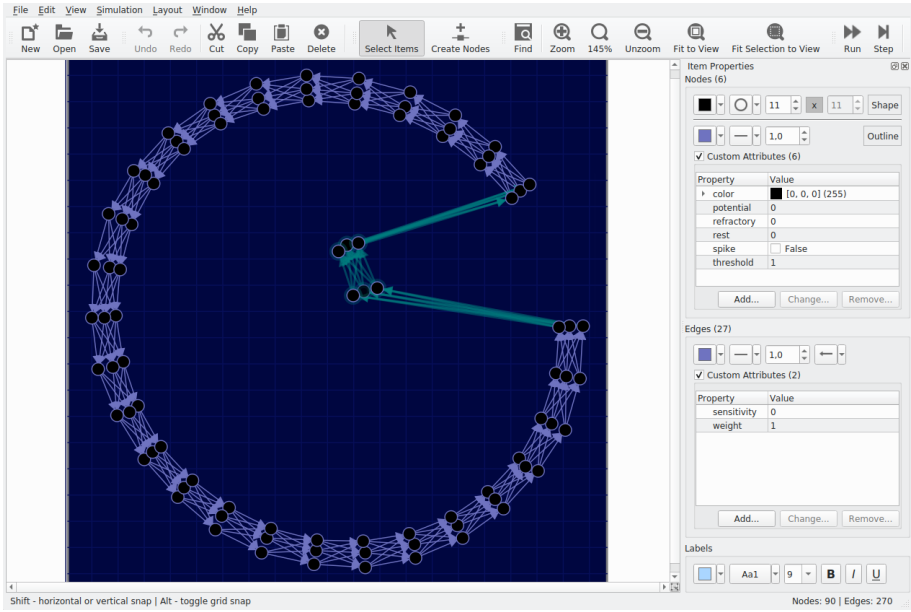


Figure 4.2: Group selection and multiple element moving.

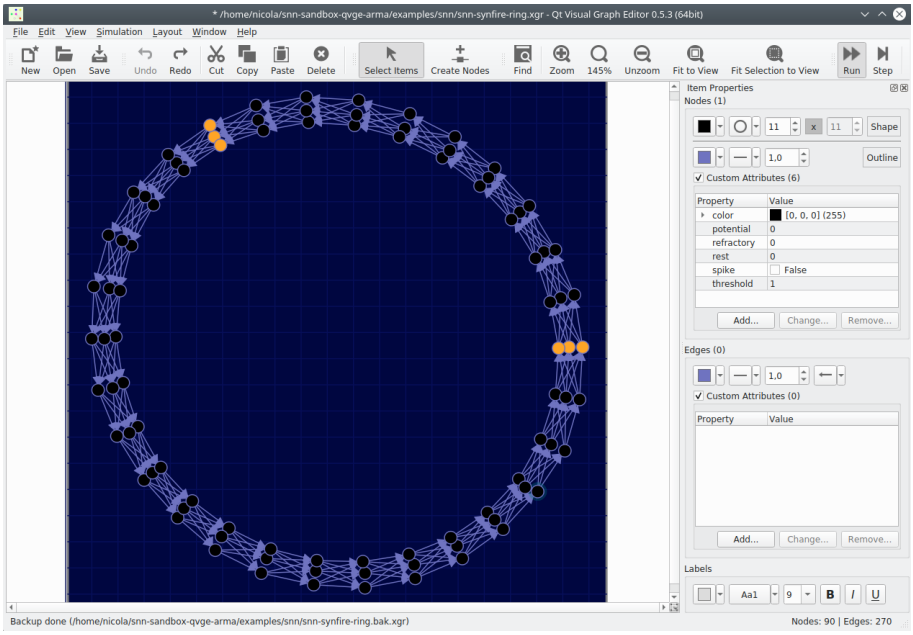


Figure 4.3: Running Simulation (the yellow dots represent the spikes).

- Change simulation's speed at runtime by introducing a user-regulated delay in the simulation loop;
- Add GPU support with CUBLAS as Armadillo's core;
- Add support for network structure/weight learning rules.

4.6 Notable Neural Circuits

This section will proceed to introduce some of the most peculiar networks, or neural circuits, that have been designed within the eveNNt Simulator. They all implement a specific function, each chosen either as a practical demonstration for the Turing-Completeness of spiking networks, or as a building block that would potentially be needed for a broad spectrum of tasks.

Such methodology can be seen also as a way to reintroduce explainability inside neural architectures, as the dynamics that lead the network to produce a certain output from some input have been finely tuned and comprehended by the human-designer. Even with an hypothetical rule or general procedure for finding out the network's structure/weights that compute a specific function, the generated networks would most probably be much smaller than Deep Learning's models, making them easier to understand.

Many of the proposed circuits use a spike-train-count encoding, mapping the information on the number of consecutive spikes, or on the number of spikes inside a certain time window.

4.6.1 Rule 110

Cellular Automata are very simple discrete dynamical systems defined on a 1-dimensional array of neighboring cells, in which every of them can be either in a "alive" or "dead" state at a certain time. A cell's state in a future timestep depends on the state of the cell itself and the two neighboring cells at the previous timestep, as shown in picture 4.4. By changing the future-cell rule configuration, the system's behaviour ranges from ordered and periodic to chaotic. Rule 110 is a particular Cellular Automata configuration know to fall in between these classification, being described as semi-chaotic. It has become particulary famous when it has been proven to be (weakly, as it required a modification to the standard definition of a Turing Machine) Universal [114].

In this spiking neural network implementation (figure 4.5), cells are represented by neurons, and their "alive" or "dead" state respectively by the presence or the absence of a spike at a certain time. Neurons 1,2 and 3 are present state cells, while neuron 8 is the central cell's future state. The other neurons have

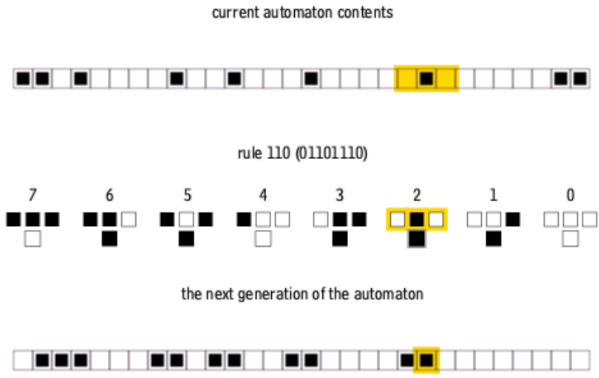


Figure 4.4: Rule 110's cells future state computation.

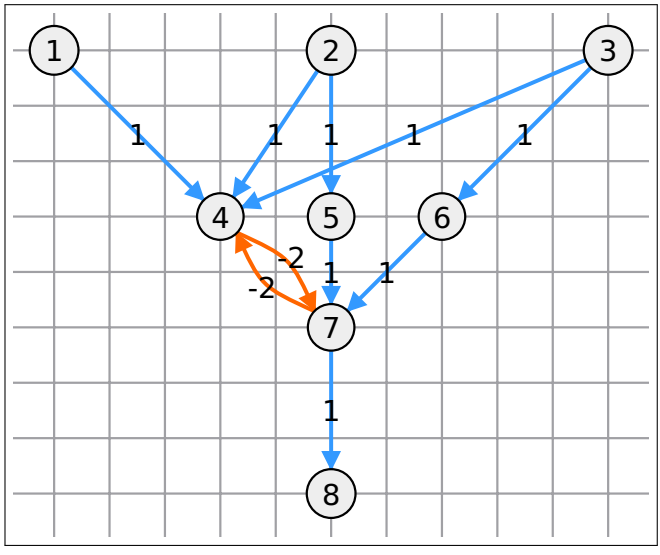


Figure 4.5: Module for computing cell's future state

Neuron	Threshold	Rest	Refractory
1,2,3,5,6,8	1	0	0
4	3	0	0
7	1	0	2

Table 4.1: Neurons' parameters for the Rule 110 module.

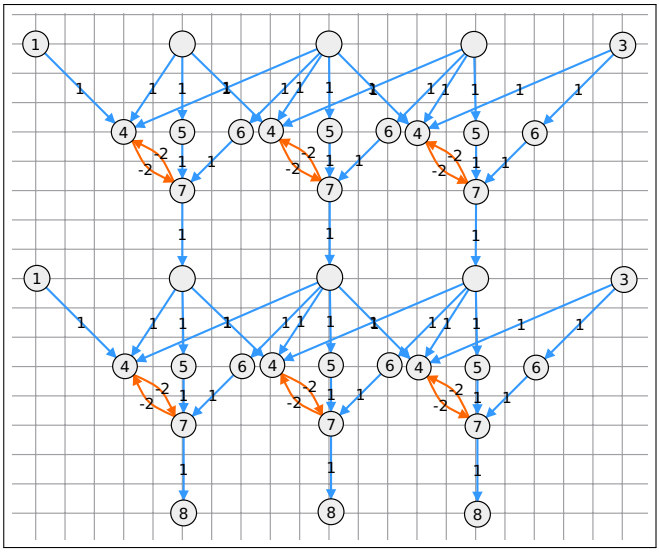


Figure 4.6: Six modules are linked together to compute more cell future states.

been connected in such a way that the pattern in figure 4.4 is followed. Finally, each neuron's parameters are reported in Table 4.1.

The module shown in figure 4.5 can be iteratively connected to other identical modules to form a grid as shown in picture 4.6, in which each row represents all the cells' states at a particular time [2].

4.6.2 Comparator and Sorting Network

A comparator is a theoretical device that computes the maximum and the minimum of two input numbers. Multiple comparators can be connected together through "wires" to form what is commonly known as a sorting network, a theoretical tool for studying sorting algorithms on a fixed number of values. As shown in figure 4.7, the values "travel" through the wires (horizontal lines), enter the comparators (vertical lines) and they are swapped to the opposite wire if they are in the wrong order, or kept one same wire if the order is correct (the smallest number always exits the comparator on top wire, and the biggest

[2]For a bigger simulation of this network, in which the typical Rule 110 pattern emerges, please refer to the following link: `https://youtu.be/DChC6_fdRQI`

Neuron	Threshold	Rest	Refractory
*	1	0	0

Table 4.2: Neurons' parameters for the comparator module.

on the bottom wire). Such tools have been extensively used in the past [121] to study sorting optimality, and to show that the most common sorting algorithms are equivalent if parallel comparisons are allowed.

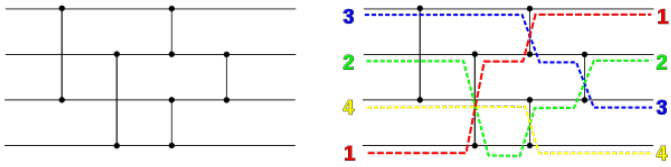


Figure 4.7: A simple sorting network, composed of four wires (horizontal lines) and five comparators (vertical lines).

On a mission to implement useful algorithms with spiking neural networks, sorting is one of the first examples that come to a programmers mind. For this purpose, sorting networks provide a useful modeling strategy to tackle this problem relatively easily, without having the need of setting up all the machinery for simulating an actual sorting algorithm.

To do this, the first step is to design the comparator module, shown in figure 4.8 and in table 4.2. This neural circuit uses the spike-count-encoding, and in this sense, it sorts spike trains based on their quantity. The first stage of the circuit copies the input spike trains: one copy per each input is propagated to the external stage, while the other is sent to the internal stage. The internal part of the circuit performs the difference of the incoming spike trains, and based on that, decides whether to enable the switching or the non-switching output of the circuit. In the meanwhile, the spike trains traveling on the outer stage will have reached the final stage as well, an based on the internal stage computation, they will follow the inverting or the non inverting path that has been previously enabled.

Finally, figure 4.9 shows how spiking comparator modules can be straightforwardly arranged to assemble a simple sorting network[3].

It is worth mentioning that the maximum number representable by this model is related to the maximum spike train length correctly processed by the network. If a spike train is so long that it reaches the final stage of the circuit before the

[3]To see this circuit in action within the eveNNt Simulator, please refer refer to the following link: `https://youtu.be/kwxQUNIt304`

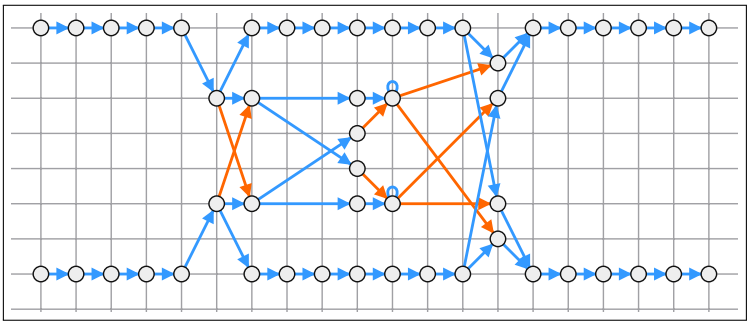


Figure 4.8: Spiking Comparator module. When omitted, the synaptic weight is $+1$ or -1, for excitatory (blue) or inhibitory (red) synapses respectively.

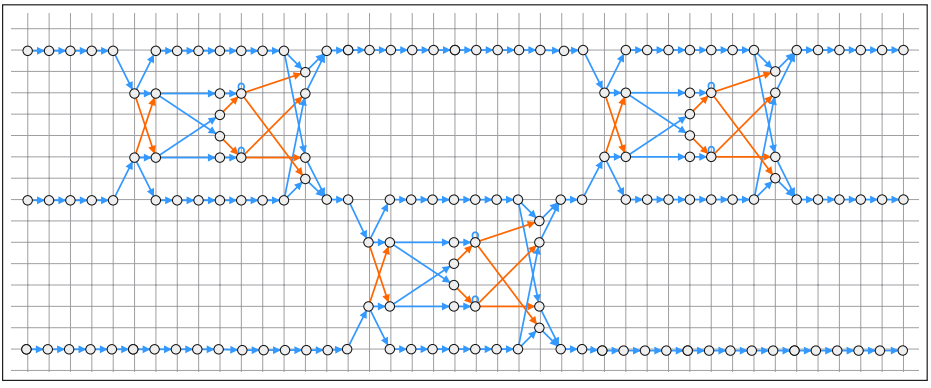


Figure 4.9: Comparator modules connected together to form a small spiking sorting network.

Neuron	Threshold	Rest	Refractory
in, d, osc, res, t1, b1, out	1	0	0
t2	v	0	$10 - v$
b2	$v + 1$	0	$9 - v$

Table 4.3: Neurons' parameters for the detector module.

internal stage finishes comparing the input trains, the whole mechanism loses its sense. The maximum number can be thus increased at will, by being sure that the external stage of the circuit is long enough to keep the spike trains away from the final stage until the internal stage has finished processing the inputs. In other words, the bigger the maximum number you want to represent, the (linearly) slower the comparator has to become (as more delay has to be put in the external stage of the circuit).

4.6.3 Detector

In order to perform computation and represent information based on spike trains' count, a component that fires if a spike train exactly contains a certain number of spikes is crucial.

The neural circuit represented in figure 4.10 implements this functionality, being able to fire only if an input spike train exactly contains v many spikes, in a window of 10 timesteps from the first spike ($v \in [1, 10]$). The input neuron *in* is the one receiving the spike train, which is forwarded to the top, to the bottom, and to the central stages of the circuit. Spike trains which contain at least v many spikes will manage to pass through the top stage, as the $t2$ neuron has its threshold set to v. $t2$ will fire only once for all the 10 timesteps window duration, as for the remaining time it will be inactive (the refractory time is set to $10 - v$). The bottom stage does the same, except with spike trains of length $v + 1$. These two opposite stages merge in the *out* neuron, and due to the array of inhibitory connections coming from the bottom stage, the *out* neuron will fire only if $b2$ didn't fire and if $t2$ did fire, or in other words, only if the spike train length is exactly v. The central stage manages to reset neurons potentials after the 10 timesteps window has passed[4].

By changing $t1$ and $t2$ thresholds and refractory times, any train length between 1 and 10 can be detected. This range can be technically improved by increasing top's and bottom's stages lengths (more delay) and by increasing the reset neuron threshold to the maximum desired train length. Similarly to the comparator module, the longer is the spike train you want to detect, the slower the detector becomes.

[4]To see this circuit in action within the eveNNt Simulator, please refer refer to the following link: `https://youtu.be/faQaSrGnLU0`

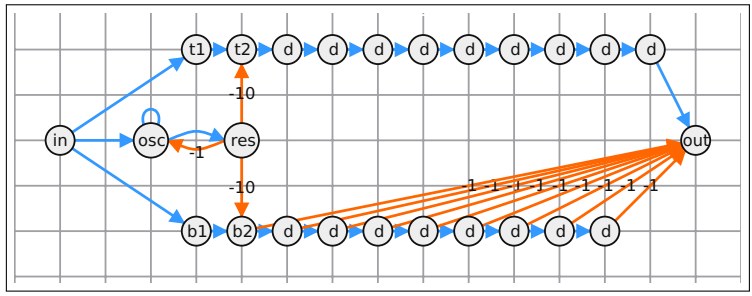


Figure 4.10: Spike train detector module. When omitted, the synaptic weight is $+1$ or -1, for excitatory (blue) or inhibitory (red) synapses respectively.

Neuron	Threshold	Rest	Refractory
on, off, osc, out	1	0	0
len	l	0	0
T	T	0	0

Table 4.4: Neurons' parameters for the threshold-controlled-generator module.

4.6.4 Threshold-controlled Generator

Continuing on the quest of providing the necessary building blocks for neural network-based computation, and having already added comparators and detectors to the landscape, it is now time to introduce spike train generators. In terms of requirements, such a circuit should be able to provide a spike trains source of a certain length l and a certain period T (with $l \leq T$), according to some neurons' parameters, that can be adjusted depending on the needs.

This is implemented by the simple circuit shown in figure 4.11 and in table 4.4. When a spike arrives to the *on* neuron, the circuit is activated and starts emitting spike trains, while sending a spike to the *off* neuron would inhibit all the circuit's neurons, effectively turning it off. The spike from the *on* neuron propagates to the upper stage of the circuit, enabling the "perpetual oscillator" neuron *osc* to fire continuosly, feeding the T neuron which regulates the generator's period. The T neuron emits one spike per each period, setting the *out* neuron on continuous firing. Every spike emitted by the *out* neuron is also propagated to the *len* neuron, which regulates the spike train length. After receiving a certain number of spikes, the *len* neuron fires, and disables *out*'s continuous firing (until it receives a new spike from T) [5].

Both spike trains's periods and lengths can be easily changed by modifying the thresholds of T and *len* respectively.

[5] To see this circuit in action within the eveNNt Simulator, please refer refer to the following link: `https://youtu.be/kzsKGIp9n1s`

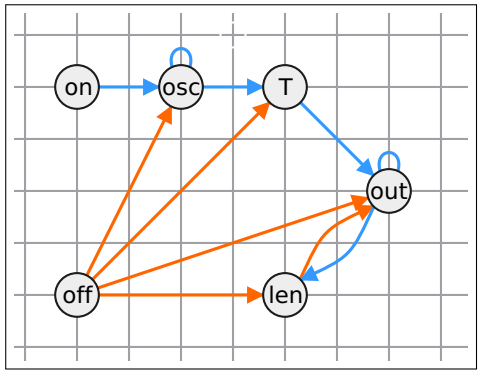


Figure 4.11: Spike train threshold-controller-generator module. When omitted, the synaptic weight is +1 or −1, for excitatory (blue) or inhibitory (red) synapses respectively.

4.6.5 Potential-controlled Generator

A different and possibly more versatile version of spike train generator is the potential-regulated-generator. Instead of having the spike train's length fixed a priori, it would be handful to have a circuit that generates spike trains of different lengths depending on runtime conditions. This can be done by using a neuron's potential as the main parameter driving generated spike trains' length.

The circuit shown in figure 4.12 and in table 4.5 implements a non-periodic potential controlled generator. The input spike train is used by the bottom stage of the circuit to decrement the *pot* neuron's potential by 1 at each spike, starting from a value of 9 (thanks to the connection with the reset neuron *res*). Even though the generator is non-periodic, *win* neuron's threshold and the *res* synapse towards the *pot* neuron are used to set a maximum time window for the whole spike train to arrive, starting from the first spike in the sequence. If after such time a new spike arrives, it is considered as the beginning of a new spike train. The *osc* and *win* neurons take trace of the time that passed after the first spike, with *win* emitting a spike after the time window is expired. Such spike fires up the second oscillator in the upper stage, that takes care of filling up the "potential gap" of *pot* caused by the inhibitory connection with the input. These spikes are also sent to the output neuron *out*, until *pot* reaches its thresholds and fires, disabling the oscillator, and causing its potential to be brought back to 9 (from the *res* neuron) [6].

Such circuit can be also used as a "spike train compactor", as it takes all the spikes in a 10 timesteps window and outputs them as a compact spike burst, removing "spaces" between consecutive spikes.

[6] To see this circuit in action within the eveNNt Simulator, please refer refer to the following link: `https://youtu.be/hwK1yxg-62g`

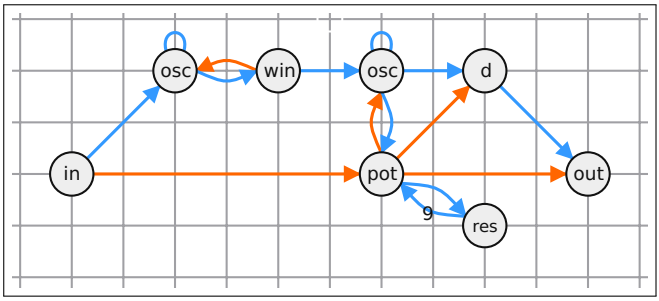


Figure 4.12: Spike train potential-controlled-generator module. When omitted, the synaptic weight is +1 or −1, for excitatory (blue) or inhibitory (red) synapses respectively.

Neuron	Threshold	Rest	Refractory
in, out, osc, d, res	1	0	0
win	9	0	0
pot	10	0	0

Table 4.5: Neurons' parameters for the potential-controlled-generator module.

4.6.6 Memory Cell

Another fundamental element for any computational model is the ability to store and retreive information, and in this case, spike trains. As opposed to LSTMs or other recurrent models, where long time dependencies are hard to learn and handle due to "fading", the idea here is to have past information, encoded in the form of spike trains, to last indefinetely. This is indeed possible thanks to the non-leaking crisp nature of the spiking neurons defined in the proposed simulator, and to the ring-like shape that allows spike trains to be stored indefinetely. Such ring shape can be also made more reduntant and robust to eventual damage (when thinking about an hypothetical physical neural network) connecting multiple fully-connected-layers into a ring, obtaining what is known as a Synfire ring topology.

The circuit shown in picture 4.13 and in table 4.6 implements this concept, assuming a maximum spike train length of ten, and with three-part design: an input (or write) line, a storing element (a looped chain of neurons), and an output (or read) line. As soon as a spike train transits on the *wrt* neuron, the first thing that happens is the current state of the memory being erased by the reset *res* neuron placed in the middle of the storing element (it inhibits all the neurons in the chain). *res* has also a refractory period of ten, so that it cannot fire again within the next ten time steps (avoids the loop storing element to be reset for each spike transiting on the input line). After the writing phase, the spike train continues to loop indefinetely on the storing element, and it can be

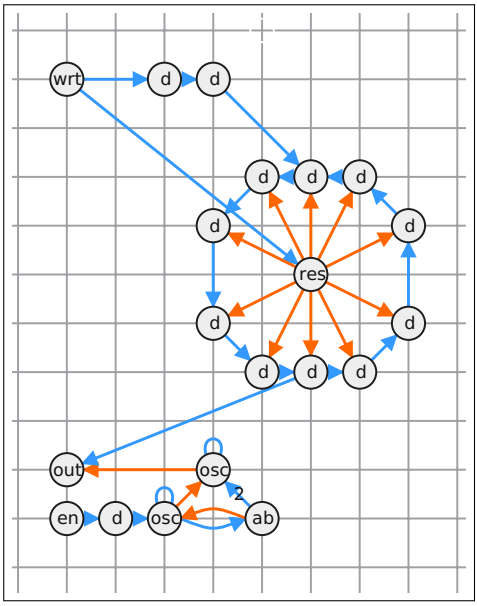


Figure 4.13: Memory module for storing spike trains. When omitted, the synaptic weight is +1 or −1, for excitatory (blue) or inhibitory (red) synapses respectively.

Neuron	Threshold	Rest	Refractory
wrt, d, out, en, osc	1	0	0
res	1	0	10
ab (anti-burst)	11	0	0

Table 4.6: Neurons' parameters for the memory module.

read with the bottom stage of the circuit. This stage is essentially designed to produce an anti-burst of spikes, or, in other terms, an absence of spikes (of a specific length) on an otherwise always active neuron. When the anti-burst is generated by the transit of a spike through the read-enable *en* neuron, the *out* neuron will not inhibited for a period of ten timesteps, which allow the spikes incoming from the storing element to not be cancelled out by the otherwise always active oscillator.

By connecting a memory module's write channel to another module's read channel in the correct way, a Turing Machine-like tape can be created, in which the stored spike trains can move between adjacent memory modules (or in this case, cells)[7].

[7]To see this circuits in action within the eveNNt Simulator, please refer refer to the following link:`https://youtu.be/HHMka0liMeY`

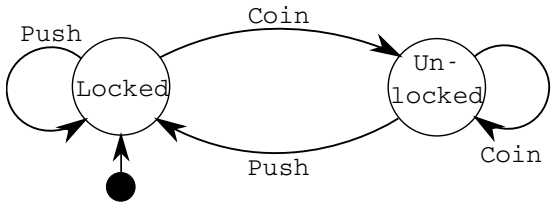


Figure 4.14: A simple 2 state, 4 transitions, Finite State Machine.

4.6.7 Finite State Machine

Finite State Machines (FSM) can be implemented relatively easily with spiking neural networks, thanks to their directed graph structure. Also, having all the other components ready, modeling a FSM is the last necessary step towards the construction of a Turing Machine network.

It is worth mentioning that FSMs and closely related automatas are particularly useful in many other domains outside computability theory, as shown in previous sections 2.6.2 and 2.7.3. A neural representation of such logical formulas and formal models is thus not only indeed possible, but relatively straightforward.

The neural circuit shown in figure 4.15 and table 4.7 implement the simple FSM in picture 4.14, and give an intuiton on how any FSM could be converted and simulated by a spiking network. The methodology is quite straightforward, needing essentially two types of neurons: state neurons s_i and transition enabling neurons e_{ij}. State neurons are "continuous oscillators", or self-excited neurons, that keep firing at each time step once they receive an input spike, until they are switched off through an inhibitory connection. An active state neuron s_i means that the FSM is currently in the state s_i (the circuit imposes that only one state can be active at each time). State neurons are connected through transition neurons t_{ij}, that take care of state changing. Normally, when no state transition is triggered, all the t_{ij} are inhibited by the transition-block neuron blt. Even if a spike goes through an e_{ij} neuron, the net balance of t_{ij} potential will still be zero, preventing spontaneous transitions. The only way to have a positive balance, thus reaching the threshold and actually performing the transition, is that the state neuron s_i that is the precodintion for such transition is active. When a transition is performed, the old state is disabled, and the new one is enabled[8].

Summing up, this methodology can be applied to any FSM, by opportunely creating state neurons for each state, transition and transition enabling neurons for each transition, and remembering to include a "transition-blocking" neuron

[8]To see this circuit in action within the eveNNt Simulator, please refer refer to the following link:`https://youtu.be/kZ-V_fOsk8Q`

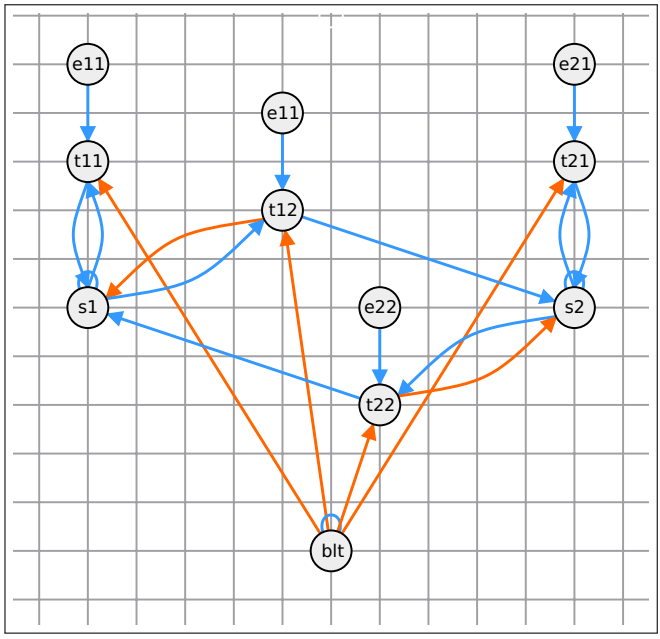


Figure 4.15: A simple Spiking FSM network. When omitted, the synaptic weight is $+1$ or -1, for excitatory (blue) or inhibitory (red) synapses respectively.

Neuron	Threshold	Rest	Refractory
*	1	0	0

Table 4.7: Neurons' parameters for the Spiking FSM Network.

for preventing spikes from state neurons to propagate all over the network.

4.6.8 Bit Inverting Turing Machine

Now that all the components are ready, a full spiking network Turing Machine can be finally implemented. Even if this section focuses on the construction of a particular "bit-inverting" Turing Machine as a proof of concept, it is theoretically possible to build every possible Turing Machine, thanks to the previous section's methodology for implementing FSMs using spiking networks. This example does not only experimentally show how a functional Turing Machine can be simulated within a spiking/recurrent neural network as a practical proof of their Universality, but wants also to highlight that such simulation can be done directly, without addional simulation overhead. It is in fact a common practice to prove computational Universality by trying to simulate simpler systems that are known to be Turing Complete, such as the ones shown in [112]. By the way,

such simulations usually suffer of some overhead, which is polynomial in the best case scenario, or exponential in the worst. Instead, by choosing a direct simulation of a particular TM, not only the overhead problem is substantially reduced, but also the chosen TM can be more easily programmed for actually performing a given task (such as inverting symbols).

The construction uses four of the modules introduced so far:

- Memory Cell circuits, for implementing the TM's tape;
- A FSM circuit, for handling the actual TM logic;
- Spike Train Detector circuits for reading the spike train from the memory cell, recognizing the symbol, and communicating the result to the FSM circuit;
- Threshold-Controlled Spike Train Generator circuits for generating the symbols' spike trains communicated by the FSM, and writing them to the tape.

Given the space constraints of this page, which preclude any possibility of showing the fully assembled TM's neural circuit reasonably[9], a more schematic, but yet explanatory representation is displayed in figure 4.16.

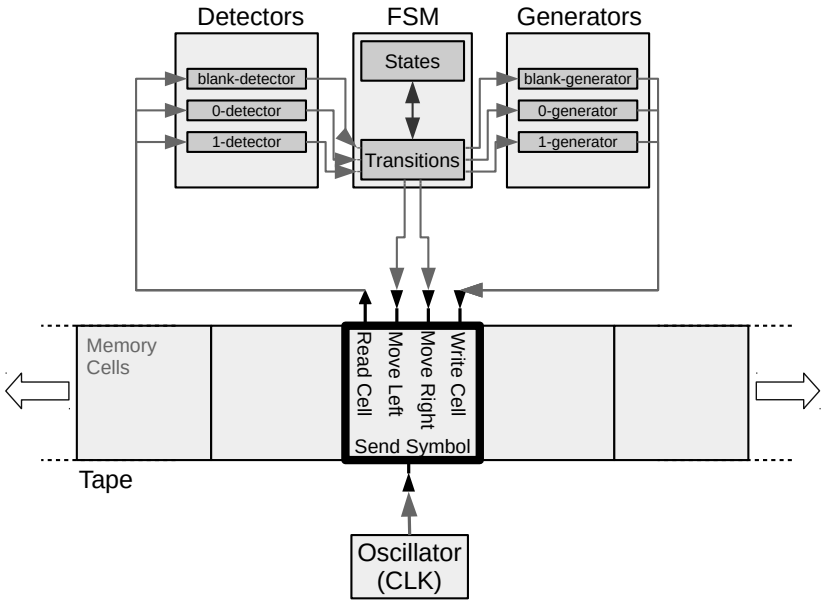


Figure 4.16: Conceptual diagram of the spiking network TM.

[9]To see the full circuit in action within the eveNNt Simulator, please refer refer to the following link:`https://youtu.be/l9GLcMINB78`

On a more specifical side, the implemented TM uses a 2-symbol alphabet (plus the blank symbol), coded on different spike train lenghts: a length of 1 corresponds to the "blank"symbol, a length of 5 to the "0", and a length of 8 to the "1". The TM will start its computation with four symbols stored on the tape, all positioned to the right side of the read/write head. Each TM computation step is triggered by a neuron oscillator, that fires every 100 time steps, giving enough time for the all the machinery of the previous TM computation step to complete their tasks. When this neuron fires, the symbol contained in read/write cell is sent to the detectors, so that the FSM circuit can decide the next state and actions. The actions can either be to convert a "0" into a "1" (or vice-versa) and then move the tape to the left, or to just move one last time the tape to the right before halting, if a blank symbol is read.

4.7 Practical Aspects

On a practical side, this chapter's contributions go in the direction of developing efficient and distributed artificially intelligent systems. In particular, a first practical implication is that spiking (or even just recurrent) neural networks cannot only be considered as learning tools, but should be seen as general purpose computers. This would potentially lead to a paradygm shift, where computation would be actually performed on systems which are more similar to the biological counterparts that natural selection and evolution have already proven to be effective. The second aspect is instead knowledge representation, as spiking neural networks have a built-in parallel reactive architecture that also enable human centered design. Apart from learning capabalities, an hypothetical developer of such a neuromorphic computing device could follow methodologies and tools similar to the ones presented in this chapter to implement the desired functions and algorithms.

Nonetheless, there are still a couple of factors that limit the diffusion of this technology:

- High cost of dedicated parallel hardware;
- Spiking models are computationally expensive to simulate;
- Lack of general learning rules for (spiking) networks with recurrent topologies.

In this sense, either from research in VLSI or in nanotechnologies, the development (or discovery) of methods to syntethize spiking neurons in large scales would be a significant breakthrough, as it would allow for the creation of real pervasive intelligent solutions in embedded devices, IoT appliances, smartphones, and many other application fields. At the same time, the discovery of a

general learning rule that works with nonconstrained network topologies would be an equally important step, as it would enable neural systems to analogously learn patterns and algorithms, fading the boundaries between the two worlds.

4.8 Conclusion

This chapter has shed a light on third-generation neural networks as a complete computational paradygm.

Inspired by both recurrent networks spiking neuron dynamics, an event-based neural network model, simulator and CAD-like software tool is proposed, and used to design neural circuits that implement specific functions.

These designed circuits can be used as building blocks for a general computation on event-based networks. As a proof of this, a practical demonstration of a event-based neural network Turing Machine is provided.

To the author's knowledge, this is the first practical attempt, that can be also observed within the proposed software tool, in simulating neural circuits that directly compute specific functions or algorithms.

Final Remarks

The first part of this thesis has focused on showing how the event-based logic formalism known as the Event Calculus can be used to easily model and perform efficient temporal reasoning on structured domains. Also, several optimization strategies are proposed and compared in order to bring such event-based temporal reasoning closer to the real world.

In the second part, Statistical AI techniques such as 3D Convolutional Neural Networks and Multi-agent Systems have been adopted as novel approaches for problems that would have been otherwise particularly hard to solve in an "exact" way. Then, tools and strategies for "interpreting" such black-box machine processes and behaviors are proposed.

As a way to overcome critical issues of Statistical AI techniques (specifically of Neural Networks), and to narrow the gap with Symbolic AI techniques, the third part focuses on event-based neural architectures. Apart from gains in terms of efficiency over the traditional neural models, due to the more intrinsicly energy efficient dynamics of spiking neurons, it shows how event-based Neural Networks are much more than classifiers or function approximators, being in fact able to perform Universal computation, and blurring the lines between what is now intended by a "learning" process in a neural network and the concept of "writing a program". With the purpose of practically demostrating this fact by designing neural computing modules, a novel software tool, the eveNNt Simulator, is introduced.

In conclusion, event-based modeling can be seen as a fundamentally valid paradygm in both Statistical and Classical AI techniques, providing efficient and effective ways to model temporal dynamics, while at the same time not having to give up on flexibility and generality.

Bibliography

[1] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, third edition, 2008.

[2] Ankasor, "Improved lung segmentation using watershed," 2017, [Online; accessed 17-February-2019].

[3] Albert Chon, Niranjan Balachandar, and Peter Lu, "Deep convolutional neural networks for lung cancer detection," *Standford University*, 2017.

[4] Özgür Kafalı, Stefano Bromuri, Michal Sindlar, Tom van der Weide, Eduardo Aguilar Pelaez, Ulrich Schaechtle, Bruno Alves, Damien Zufferey, Esther Rodriguez-Villegas, Michael Ignaz Schumacher, et al., "Commodity12: A smart e-health environment for diabetes management," *Journal of Ambient Intelligence and Smart Environments*, vol. 5, no. 5, pp. 479–502, 2013.

[5] Stefano Bromuri, Serban Puricel, Rene Schumann, Johannes Krampf, Juan Ruiz, and Michael Schumacher, "An expert personal health system to monitor patients affected by gestational diabetes mellitus: A feasibility study," *Journal of Ambient Intelligence and Smart Environments*, vol. 8, no. 2, pp. 219–237, 2016.

[6] Stefano Bragaglia, Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni, "Reactive event calculus for monitoring global computing applications," in *Logic Programs, Norms and Action: Essays in Honor of Marek J. Sergot on the Occasion of His 60th Birthday*, Alexander Artikis, Robert Craven, Nihan Kesim Çiçekli, Babak Sadighi, and Kostas Stathis, Eds., pp. 123–146. Springer Berlin Heidelberg, 2012.

[7] L. Chittaro, A. Montanari, M. Dojat, and C. Gasparini, "The event calculus at work: a case study in the medical domain," in *Second International Conference on Intelligent Systems Engineering, 1994*, Sep 1994, pp. 195–200.

[8] Luca Chittaro, Marco Del Rosso, and Michel Dojat, "Modeling medical reasoning with the event calculus: an application to the management

of mechanical ventilation," in *Artificial Intelligence in Medicine*, Pedro Barahona, Mario Stefanelli, and Jeremy Wyatt, Eds., Berlin, Heidelberg, 1995, pp. 79–90, Springer Berlin Heidelberg.

[9] François Portet, *Algorithms piloting for cardiac arrhythmias recognition*, Theses, Université Rennes 1, Dec. 2005.

[10] Alexander Artikis, "Datasets for symbolic event recognition," Retrieved from `http://users.iit.demokritos.gr/~a.artikis/ER-datasets.html`.

[11] Albert Brugués, Stefano Bromuri, Josep Pegueroles-Valles, and Michael Ignaz Schumacher, "MAGPIE: An agent platform for the development of mobile applications for pervasive healthcare," in *Proceedings of the 3rd International Workshop on Artificial Intelligence and Assistive Medicine (AI-AM/NetMed)*, 2014, pp. 6–10.

[12] Nihan Kesim Cicekli and Yakup Yildirim, "Formalizing workflows using the event calculus," in *Database and Expert Systems Applications*, Mohamed Ibrahim, Josef Küng, and Norman Revell, Eds., Berlin, Heidelberg, 2000, pp. 222–231, Springer Berlin Heidelberg.

[13] Matthew Fuchs Phd, "The event calculus as a programming model for game ai," .

[14] Nicola Falcionelli, Paolo Sernani, Albert Brugués, Dagmawi Neway Mekuria, Davide Calvaresi, Michael Schumacher, Aldo Franco Dragoni, and Stefano Bromuri, "Event calculus agent minds applied to diabetes monitoring," in *Autonomous Agents and Multiagent Systems*, Gita Sukthankar and Juan A. Rodriguez-Aguilar, Eds., Cham, 2017, pp. 258–274, Springer International Publishing.

[15] S. Bromuri, A. Brugues de la Torre, F. Duboisson, and M. Schumacher, "Indexing the Event Calculus with Kd-trees to Monitor Diabetes," *ArXiv e-prints*, Oct. 2017.

[16] A. Artikis, M. Sergot, and G. Paliouras, "An event calculus for event recognition," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 4, pp. 895–908, 2015.

[17] Iliano Cervesato and Angelo Montanari, "A general modal framework for the event calculus and its skeptical and credulous variants," *The Journal of Logic Programming*, vol. 38, no. 2, pp. 111 – 164, 1999.

[18] Erik T. Mueller, *Commonsense Reasoning: An Event Calculus Based Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2015.

[19] Md Tawhid Bin Waez, Juergen Dingel, and Karen Rudie, "A survey of timed automata for the development of real-time systems," *Computer Science Review*, vol. 9, pp. 1 – 26, 2013.

[20] "List of works that used UPPAAL," `http://www.it.uu.se/research/group/darts/uppaal/examples.shtml`.

[21] Robert Kowalski and Marek Sergot, "A logic-based calculus of events," *New Generation Computing*, vol. 4, no. 1, pp. 67–95, 1986.

[22] L. Chittaro and A. Montanari, "Efficient temporal reasoning in the cached event calculus," *Computational Intelligence*, vol. 12, no. 3, pp. 359–382, 1996.

[23] Rudolf Bayer, "Symmetric binary b-trees: Data structure and maintenance algorithms," *Acta Informatica*, vol. 1, no. 4, pp. 290–306, Dec 1972.

[24] Jon Louis Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[25] B. Goodman and S. Flaxman, "European Union regulations on algorithmic decision-making and a "right to explanation"," *ArXiv e-prints*, June 2016.

[26] World Health Organization, *Global Report on Diabetes*, World Health Organization, Geneve, 2016.

[27] Paul Zimmet, K George Alberti, Dianna J Magliano, and Peter H Bennett, "Diabetes mellitus statistics on prevalence and mortality: facts and fallacies," *Nature Reviews Endocrinology*, vol. 12, no. 10, pp. 616–622, 2016.

[28] Centers for Disease Control and Prevention, "Diabetes risk factors (2018, 02)," Retrieved from `https://www.cdc.gov/diabetes/pdfs/data/statistics/national-diabetes-statistics-report.pdf`.

[29] Diabetes.co.uk, the global diabetes community, "Brittle diabetes (labile diabetes)," Retrieved from `https://www.diabetes.co.uk/brittle-diabetes.html`.

[30] Diabetes.co.uk, the global diabetes community, "National diabetes statistics report," Retrieved from `https://www.diabetes.co.uk/Diabetes-Risk-factors.html`.

[31] D. H. Spodick, "Normal sinus heart rate: appropriate rate thresholds for sinus tachycardia and bradycardia," *South. Med. J.*, vol. 89, no. 7, pp. 666–667, Jul 1996.

[32] K. Dungan, "Monitoring technologies – continuous glucose monitoring, mobile technology, biomarkers of glycemic control," in *Endotext [Internet]*, L. J. De Groot, P. Beck-Peccoz, G. Chrousos, K. Dungan, A. Grossman, J. M. Hershman, and F. Singer, Eds. 2014.

[33] National Institute for Health and Care Excellence, "Hypertension in adults: diagnosis and management (2016, November)," Retrieved from `https://www.nice.org.uk/guidance/cg127/chapter/1-Guidance#measuring-blood-pressure`.

[34] Diabetes.co.uk, the global diabetes community, "Type 1 diabetes in adults: diagnosis and management," Retrieved from `https://www.diabetes.co.uk/diabetes_care/blood-sugar-level-ranges.html`.

[35] Diabetes.co.uk, the global diabetes community, "Diabetes and hyperglycemia," Retrieved from `https://www.diabetes.co.uk/Diabetes-and-Hyperglycaemia.html`.

[36] Diabetes.co.uk, the global diabetes community, "Diabetes and hypoglycemia," Retrieved from `https://www.diabetes.co.uk/Diabetes-and-Hypoglycaemia.html`.

[37] Giorgio C Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*, vol. 24, Springer Science & Business Media, 2011.

[38] "UPPAAL reference website," `http://www.uppaal.org/`.

[39] Alexandre David, Jacob Illum, Kim G Larsen, and Arne Skou, "Model-based framework for schedulability analysis using uppaal 4.1," *Model-based design for embedded systems*, vol. 1, no. 1, pp. 93–119, 2009.

[40] Erik T. Mueller, "Event calculus reasoning through satisfiability," *J. Log. and Comput.*, vol. 14, no. 5, pp. 703–730, Oct. 2004.

[41] T1D Exchange, "A randomized trial comparing continuous glucose monitoring with and without routine blood glucose monitoring in adults with type 1 diabetes (2016, September)," Retrieved from `https://t1dexchange.org/pages/resources/our-data/studies-with-data/`.

[42] J. Ker, L. Wang, J. Rao, and T. Lim, "Deep learning applications in medical image analysis," *IEEE Access*, vol. 6, pp. 9375–9389, 2018.

[43] Paola Campadelli, Elena Casiraghi, and Stella Pratissoli, "A segmentation framework for abdominal organs from ct scans," *Artificial Intelligence in Medicine*, vol. 50, no. 1, pp. 3 – 11, 2010, Knowledge Discovery and Computer-Based Decision Support in Biomedicine.

[44] T. Jin, H. Cui, S. Zeng, and X. Wang, "Learning deep spatial lung features by 3d convolutional neural network for early cancer detection," in *2017 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, Nov 2017, pp. 1–6.

[45] DICOM, "Dicom standard," 2019, [Online; accessed 14-February-2019].

[46] W. Zhu, C. Liu, W. Fan, and X. Xie, "Deeplung: Deep 3d dual path nets for automated pulmonary nodule detection and classification," in *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, March 2018, pp. 673–681.

[47] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, June 2017.

[48] Olaf Ronneberger, Philipp Fischer, and Thomas Brox, "U-net: Convolutional networks for biomedical image segmentation," in *MICCAI*, 2015.

[49] Jerome H. Friedman, "Greedy function approximation: A gradient boosting machine," *The Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001.

[50] Wafaa Alakwaa, Mohammad Nassef, and Amr Badr, "Lung cancer detection and classification with 3d convolutional neural network (3d-cnn)," *Lung Cancer*, vol. 8, no. 8, 2017.

[51] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri, "Learning spatiotemporal features with 3d convolutional networks," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 4489–4497.

[52] Muhammad Naveed Iqbal Qureshi, Jooyoung Oh, and Boreom Lee, "3d-cnn based discrimination of schizophrenia using resting-state fmri," *Artificial Intelligence in Medicine*, vol. 98, pp. 10 – 17, 2019.

[53] Nima Tajbakhsh, Jae Y Shin, Suryakanth R Gurudu, R Todd Hurst, Christopher B Kendall, Michael B Gotway, and Jianming Liang, "Convolutional neural networks for medical image analysis: Full training or

fine tuning?," *IEEE transactions on medical imaging*, vol. 35, no. 5, pp. 1299–1312, 2016.

[54] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal, "Explaining explanations: An approach to evaluating interpretability of machine learning," *CoRR*, vol. abs/1806.00069, 2018.

[55] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," *PLOS ONE*, vol. 10, pp. 1–46, 07 2015.

[56] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller, "Playing atari with deep reinforcement learning," *ArXiv*, vol. abs/1312.5602, 2013.

[57] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch, "Emergent tool use from multi-agent autocurricula," 2019.

[58] Jordi Sabater-Mir and Carles Sierra, "Reputation and social network analysis in multi-agent systems," in *AAMAS*, 2002.

[59] Paolo Toth and Daniele Vigo, *The vehicle routing problem*, SIAM, 2002.

[60] Marco Dorigo and Gianni Di Caro, "Ant colony optimization: a new meta-heuristic," in *Proc. of the 1999 congress on evolutionary computation*. IEEE, 1999, vol. 2, pp. 1470–1477.

[61] Maximilian Alber, Sebastian Lapuschkin, Philipp Seegerer, Miriam Hägele, Kristof T. Schütt, Grégoire Montavon, Wojciech Samek, Klaus-Robert Müller, Sven Dähne, and Pieter-Jan Kindermans, "innvestigate neural networks!," *CoRR*, vol. abs/1808.04260, 2018.

[62] American Cancer Society, "Exams and tests that look for lung cancer," 2016, [Online; accessed 18-February-2019].

[63] American Cancer Society, "What is small cell lung cancer?," 2016, [Online; accessed 14-February-2019].

[64] American Cancer Society, "What is non-small cell lung cancer?," 2016, [Online; accessed 14-February-2019].

[65] Cecilia Zappa and Shaker Mousa, "Non-small cell lung cancer: Current treatment and future advances," *Translational Lung Cancer Research*, vol. 5, pp. 288–300, 06 2016.

[66] J K Olsson, E M Schultz, and M K Gould, "Timeliness of care in patients with lung cancer: a systematic review," *Thorax*, vol. 64, no. 9, pp. 749–756, 2009.

[67] Dr Ian Bickle and Kyle Greenway et al., "Hounsfield unit," 2019, [Online; accessed 14-February-2019].

[68] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson, "How transferable are features in deep neural networks?," in *Advances in neural information processing systems*, 2014, pp. 3320–3328.

[69] Luna16, "Lung nodule analysis," 2016, [Online; accessed 17-February-2019].

[70] The Cancer Imaging Archive, "Tcia collections," 2014, [Online; accessed 17-February-2019].

[71] ksmith01, "Nsclc radiogenomics," 2014, [Online; accessed 17-February-2019].

[72] kirbyju, "Nsclc radiomics," 2014, [Online; accessed 17-February-2019].

[73] kirbyju, "Nsclc radiomics genomics," 2014, [Online; accessed 17-February-2019].

[74] kirbyju, "Tcga luad," 2014, [Online; accessed 17-February-2019].

[75] kclark01, "Tcga lusc," 2014, [Online; accessed 17-February-2019].

[76] bvendt01, "Lidc idri," 2014, [Online; accessed 17-February-2019].

[77] Give a Scan, "Give a scan," 2014, [Offline].

[78] Casa del Sollievo e della Sofferenza, "Casa del sollievo e della sofferenza," [Online; accessed 17-February-2019].

[79] Kaggle, "Data science bowl 2017," 2017, [Online; accessed 17-February-2019].

[80] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[81] Victor Pillac, Michel Gendreau, Christelle Guéret, and Andrés L. Medaglia, "A review of dynamic vehicle routing problems," *European Journal of Operational Research*, vol. 225, no. 1, pp. 1 – 11, 2013.

[82] Seth Tisue and Uri Wilensky, "Netlogo: A simple environment for modeling complexity," in *International conference on complex systems*. Boston, MA, 2004, vol. 21, pp. 16–21.

[83] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[84] Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, J. Douglas Birdwell, Mark E. Dean, Garrett S. Rose, and James S. Plank, "A survey of neuromorphic computing and neural networks in hardware," 2017.

[85] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, May 2014.

[86] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, Oct 2015.

[87] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, January 2018.

[88] "Intel nervana processor," `https://www.intel.ai/nervana-nnp/#gs.jbii1o`, Accessed: 2019-10-30.

[89] "Qualcomm zeroth processor," `https://www.qualcomm.com/news/onq/2013/10/10/introducing-qualcomm-zeroth-processors-brain-inspired-computing`, Accessed: 2019-10-30.

[90] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung, "Accelerating deep convolutional neural networks using specialized hardware," February 2015.

[91] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang, "A survey of fpga-based neural network accelerator," 2017.

[92] Edmund T. Rolls and Alessandro Treves, "The neuronal encoding of information in the brain," *Progress in Neurobiology*, vol. 95, no. 3, pp. 448 – 490, 2011.

[93] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128× 128 120 db 15 μs latency asynchronous temporal contrast vision sensor," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 2, pp. 566–576, Feb 2008.

[94] Andrew P. Davison, Daniel Brüderle, Jochen M. Eppler, Jens Kremkow, Eilif Müller, Dejan Pecevski, Laurent U. Perrinet, and Pierre Yger, "Pynn: A common interface for neuronal network simulators," *Frontiers in Neuroinformatics*, vol. 2, pp. 3637 – 3642, 2008.

[95] Dan F. M. Goodman and Romain Brette, "Brian: A simulator for spiking neural networks in python," *Frontiers in Neuroinformatics*, vol. 2, pp. 349 – 398, 2008.

[96] Marc-Oliver Gewaltig and Markus Diesmann, "Nest (neural simulation tool)," *Scholarpedia*, vol. 2, pp. 1430, 2007.

[97] Michael L. Hines and Nicholas T. Carnevale, "The neuron simulation environment," *Neural Computation*, vol. 9, pp. 1179–1209, 1997.

[98] "NeuronCAD," `https://bitbucket.org/azylbertal/neuroncad/src/master/`, Accessed: 2019-10-30.

[99] Jérémie Cabessa, "Turing complete neural computation based on synaptic plasticity," *PLOS ONE*, vol. 14, no. 10, pp. 1–34, 10 2019.

[100] H.T. Siegelmann and E.D. Sontag, "On the computational power of neural nets," *Journal of Computer and System Sciences*, vol. 50, no. 1, pp. 132 – 150, 1995.

[101] Wolfgang Maass, "On the computational complexity of networks of spiking neurons," in *NIPS*, 1994.

[102] W. Maass, "Lower bounds for the computational power of networks of spiking neurons," *Neural Computation*, vol. 8, no. 1, pp. 1–40, Jan 1996.

[103] W. Maass, "Lower bounds for the computational power of networks of spiking neurons," *Neural Computation*, vol. 8, no. 1, pp. 1–40, Jan 1996.

[104] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba, "Openai gym," *ArXiv*, vol. abs/1606.01540, 2016.

[105] Felix A. Gers, Nicol N. Schraudolph, and Jürgen Schmidhuber, "Learning precise timing with lstm recurrent networks," *J. Mach. Learn. Res.*, vol. 3, pp. 115–143, 2002.

[106] G. . Sun, H. . Chen, and Y. . Lee, "Turing equivalence of neural networks with second order connection weights," in *IJCNN-91-Seattle International Joint Conference on Neural Networks*, July 1991, vol. ii, pp. 357–362 vol.2.

[107] Lyudmila Grigoryeva and Juan-Pablo Ortega, "Echo state networks are universal," *Neural networks : the official journal of the International Neural Network Society*, vol. 108, pp. 495–508, 2018.

[108] Sepp Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, pp. 107–116, 1998.

[109] Herbert Jaeger, "The"echo state"approach to analysing and training recurrent neural networks," 2001.

[110] Timothée Masquelier, Rudy Guyonneau, and Simon J. Thorpe, "Competitive stdp-based spike pattern learning," *Neural Computation*, vol. 21, pp. 1259–1276, 2009.

[111] Alex Graves, Greg Wayne, and Ivo Danihelka, "Neural turing machines," *ArXiv*, vol. abs/1410.5401, 2014.

[112] Turlough Neary and Damien Woods, "The complexity of small universal turing machines: a survey," 2011.

[113] F. Mavaddat and Behrooz Parhami, "Urisc: The ultimate reduced instruction set computer," *International Journal of Electrical Engineering Education*, vol. 25, pp. 327 – 334, 1988.

[114] Matthew Cook, "A concrete view of rule 110 computation," *Electronic Proceedings in Theoretical Computer Science*, vol. 1, pp. 31–55, Jun 2009.

[115] Alex Churchill, Stella Biderman, and Austin Herrick, "Magic: The gathering is turing complete," 2019.

[116] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366, 1989.

[117] Manu Prakash and Neil A. Gershenfeld, "Microfluidic bubble logic.," *Science*, vol. 315 5813, pp. 832–5, 2006.

[118] André van Schaik and Shih-Chii Liu, "Aer ear: A matched silicon cochlea pair with address event representation interface," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, pp. 48–59, 2005.

[119] Conrad Sanderson, "Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments," 2010.

[120] "QVGE visual graph editor," `https://github.com/ArsMasiuk/qvge`, Accessed: 2019-10-30.

[121] Miklós Ajtai, János Komlós, and Endre Szemerédi, "An o(n log n) sorting network," in *STOC*, 1983.