BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SIMULATION OF BIOLOGICAL PROCESSES USING ASYNCHRONOUS CELLULAR AUTOMATA AND MACHINE LEARNING

SIMULACE BIOLOGICKÝCH PROCESŮ POMOCÍ ASYNCHRONNÍCH CELULÁRNÍCH AUTOMATŮ A STROJOVÉHO UČENÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **VOJTĚCH KALIŠ**
AUTOR PRÁCE

SUPERVISOR **Ing. KAREL FRITZ,**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

156074

Institut:	Department of Computer Systems (DCSY)
Student:	**Kališ Vojtěch**
Programme:	Information Technology
Title:	**Simulation of Biological Processes Using Asynchronous Cellular Automata and Machine Learning**
Category:	Artificial Intelligence
Academic year:	2023/24

Assignment:

1. Study the basic and advanced principles of cellular automata, focusing on asynchronous variants. Ensure understanding of the basic biological processes you plan to simulate (e.g., tumor growth, infection spread, etc.). Explore existing machine learning methods that could be applied to the analysis and optimization of cellular automata models.
2. Conduct a detailed literature review concerning the use of cellular automata in biology and medicine. Identify gaps in current research and choose a specific biological process you plan to simulate.
3. Design and implement an asynchronous cellular automaton that will simulate the selected biological process.
4. Choose an appropriate machine learning method (e.g., neural networks) for the analysis, optimization, or prediction of simulation results. Implement this method and integrate it with your cellular automaton model.
5. Perform a set of experiments with different parameters of the model and the machine learning algorithm. Monitor key indicators such as simulation accuracy, computational complexity, and model robustness.
6. Prepare a comparative study of the results with existing methods or models. Discuss the achieved results, their relevance in the context of the biological process, and the potential for further research. Evaluate the overall results achieved and outline possible directions for future research in this field.

Literature:

- According to the instructions of the project supervisor.

Requirements for the semestral defence:

- Completion of items 1-3, and elaboration of items 4-5.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Fritz Karel, Ing.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	30.10.2023

Abstract

This thesis explores the fusion of asynchronous cellular automata and machine learning techniques for simulating complex biological processes. Its main focus is on showcasing the inherent potential of a computational framework constructed through combining the parallelism of an asynchronous cellular automata updating model with the predictive capabilities of machine learning algorithms. This study aims to demonstrate the qualities of such hybrid approach by implementing three mathematical cellular automata models of increasing complexity—that is, listed based on their level of complexity, Conway's Game of Life, SmoothLife and Lenia—in their basic form and then integrate machine learning into the function of the latter two, comparing the results of both approaches afterwards.

Abstrakt

Tato práce zkoumá spojení asynchronních celulárních automatů a technik strojového učení pro simulaci komplexních biologických procesů. Jejím hlavním zaměřením je předvést vrozený potenciál výpočetního rámce konstruovaného spojením paralelismu aktualizačního modelu asynchronních celulárních automatů s prediktivními schopnostmi algoritmů strojového učení. Tato studie si klade za cíl demonstrovat kvality takového hybridního přístupu implementací tří matematických modelů celulárních automatů s rostoucí složitostí—tj., seřezeny podle stupně složitosti, Conwayova Hra Života, SmoothLife a Lenia—ve své základní formě a následnou integrací strojového učení do funkce dvou posledně jmenovaných, po čemž následuje porovnání výsledků obou přístupů.

Keywords

Cellular Automaton, Asynchronous Cellular Automaton, asynchronism, biological process simulation, complex systems simulation, emergence, Machine Learning, neural networks, PyTorch, Conway's Game of Life, SmoothLife, Lenia, orbium.

Klíčová slova

Celulární Automat, Asynchronní Celulární Automat, asynchronismus, simulace biologických procesů, simulace komplexních systémů, emergence, Strojové Učení, neuronové sítě, PyTorch, Conwayova Hra Života, SmoothLife, Lenia, orbium.

Reference

KALIŠ, Vojtěch. *Simulation of Biological Processes using Asynchronous Cellular Automata and Machine Learning.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Karel Fritz,

Rozšířený abstrakt

Práce nejprve čtenáře uvede do problematik celulárních automatů a strojového učení z hlediska teoretického, se zaměřením zejména na základní koncepty a komponenty, a také na konkrétní algoritmy a struktury používané k řešení složitých úkolů. Zmíněna je také aplikace celulárních automatů i strojového učení na problematiky nejen biologického charakteru, ale i jiných problémů reálného světa. Čtenář se také dozvídá o aktuálních trendech ve vývoji celulárních automatů a především strojového učení, jenž získává v posledních letech na popularitě. Jsou prezentovány nejen pokroky ve výzkumu, ale i překážky a výzvy, jemž momentálně vědecký svět čelí.

Vlastní řešení pak spočívá ve třech etapách—zaprvé, implementace a zaručení funkčnosti všech tří předem zmíněných modelů celulárních automatů, se zaměřením na poukázání na jejich návaznost z hlediska postupného rozšiřování předchůdného modelu a stupňování složitosti i možností modelu. Je prezentován podrobný postup, úvod do zvoleného programovacího jazyka i jeho nástrojů, vysvětlení jednotlivých kroků, i matematický podtext. Druhým krokem je vytvoření modelu strojového učení, k čemuž je jakožto nejpraktičtější možnost zvolen model dopředné neuronové sítě se zaměřením na předpověd budoucích kroků simulace. Takový model je následně úspěšně implementován, vytrénován a následně integrován do funkce simulací SmoothLife i Lenia.

Třetím krokem vlastního řešení je pak postupná kalibrace primárně modelu strojového učení z hlediska jak trénování tak aplikace na námi implementované a zvolené modely celulárních automatů, a to dokud není dosaženo vyhovujících výsledků. Jako primární cíl je určena převážně stabilizace funkcionality modelů celulárních automatů, čímž se myslí zejména prevence jejich předčasného zániku. Jako vedlejší cíl je také dána jejich případná excelerace ve smyslu zrychlení jejich seskupení do stabilizovaného stádia.

V prvním pokusu o integraci natrénované neuronové sítě do zvolených modelů celulárních automatů je vytvořena váha ovlivňující simulaci, ovšem váha je zde prozatím pouze statického charakteru, což vede k selhání u všech třech testovaných simulací. V SmoothLife dochází k postupnému umírání “kluzáků”, zatímco Lenia s náhodnou počáteční konfigurací i s orbium počáteční konfigurací se vůbec ani nedokáže zformovat. K mírnému zlepšení přijde až u druhého pokusu, kdy dojde ke změně váhy na dynamickou, čili měnící se za běhu programu. Zavedena je také lehká podpora stability v místech, kde již momentální stav mřížky je relativně stabilní. SmoothLife i Lenia s orbium počáteční konfigurací sice stále umírají, avšak Lenia s náhodnou počáteční konfigurací se dokáže zformovat, což naznačuje krok správným směrem. Tuhle skutečnost také potvrdí třetí a finální pokus, zaměřený na úpravy a kalibraci již zavedených proměnných, zavedení další podpory stability spočívající v podpoře již existujících vzorů, a také zjemnění váhy za účelem předejití náhlým přechodům. U SmoothLife se “kluzáci” přestanou ztrácet a dokonce získají schopnost rozdvojit se, což udělá simulaci značně živější. U Lenie s náhodnou konfigurací se doba zformování drasticky zkrátí—přesněji čtyřnásobně. Lenia s orbium konfigurací sice stále selhává, ale tentokrát pomaleji a s vizuálně zajímavějším zánikem.

Celkově dojde k usouzení, že propojení strojového učení s celulárními automaty může být prospěšné, ale některé modely, jako Lenia s orbium konfigurací, budou s největší pravděpodobností vyžadovat další stabilizační mechanismy. Budoucí práce v téhle oblasti by se měla zaměřit na exploraci různých přístupů ke stabilitě a také zvážit pokusy s jinými modely strojového učení, aby se předešlo tendencím k selhání.

Simulation of Biological Processes using Asynchronous Cellular Automata and Machine Learning

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Karel Fritz. The supplementary information was also provided by Ing. Karel Fritz. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Vojtěch Kališ
May 8, 2024

Acknowledgements

I would hereby like to express my deepest gratitude to my supervisor, Ing. Karel Fritz, whose expert guidance, insightful feedback, and constant encouragement were instrumental in keeping my sanity intact throughout the writing of this thesis. Your patience and dedication to my success have been invaluable, and I am truly grateful for your mentorship. I would also like to thank my family, who had to endure my endless ramblings about my research and provide encouragement throughout the many late nights and long weekends of hard work. Your assurance in my abilities kept me going, and I couldn't have done it without you. Thank you for being my rock throughout this journey.

Contents

List of Figures

Chapter 1

Introduction

Throughout the extensive span of civilization, man has always sought to understand the intricate and delicate manners in which life as we know it functions and unravels. There are plenty of publications of various kinds and age recorded in the history of humankind that serve as contemporary records of these endeavours, and it is safe to say there will be no shortage of them long into the future still. This wonder already sparked many a research in the field of computational biology, in particular research focused on exploring the intricacies of emergent behaviour, dynamic patterns, and the orchestration of complex systems, and it is this same place of curiosity and excitement from which aspirations of this thesis stem.

The thesis' contents could be divided into two parts; the former, „educational" part strives to present a coherent and concise overview of the progress that had so far been made within the individual domains of cellular automata and machine learning, mainly focusing on introducing both subjects in more of a theoretical sense, as well as on their role and contribution in the quest to unravel the biological tapestry formed of the aforementioned trifecta. Particular emphasis is placed on emergence, or emergent behaviour, which stands as a testament to the wondrous capacity of simple rules and localized interactions to spawn intricate and often unpredictable patterns.

The latter part leans away from theory and delves more into the topic's practical aspects, first introducing the reader into the world of cellular automata programming, as well as later integrating machine learning into the models. It sets a goal to create a computer setup that mirrors how life forms grow and change unpredictably, and it does so by using different cellular automata models of increasing complexity—namely, Conway's Game of Life, SmoothLife, and Lenia—while exploring how to integrate machine learning into the latter two to introduce not just prediction of behaviour, but also adaptive capabilities and possible stability enhancement.

1.1 Motivation and Objectives

Let us first look into one side of the objectives of this work, namely the one focusing on the exploration of cellular automata and machine learning. Both are, individually, well-established and extensively researched concepts; machine learning in particular has seen a surge of interest in recent years. The same cannot, however, be said for the fusion of both fields, which has yet to receive comprehensive attention in the realm of research. The objective of this thesis lies in pointing out the unfortunate nature of the under-exploration

of the potential synergy between these two domains, given the promising avenues it could open for understanding complex systems and, mainly, emergent behaviour.

Which leads us to the other primary objective of this work—unveiling the neglected significance of emergence. In the world of science as well as academia, the concept of emergence always seems to draw the short end of the stick when it comes to any serious conversation or consideration. One cannot help but describe this as strange, considering how fundamental of a role it has in understanding complex systems. Granted, emergence can be rather challenging to work with; its unpredictable nature combined with complex dynamics often make it feel like trying to catch a fleeting shadow, especially so when one is trying to gauge its impact on the system under study. It is not unusual, then, to see it being pushed to the sidelines, and while the pervading aversion towards its study is understandable given the complexity of the subject at hand, this paper seeks to clarify emergence's nuances, advocate for its due consideration and ultimately demonstrate the inherent benefits its exploration brings to the table. Through this display, it seeks to implore the world of research to no longer give emergence the cold shoulder, and acknowledge it as a crucial component of the scientific discourse as well as a challenge.

Chapter 2

Cellular Automata

2.1 Introduction

In the fields of mathematical modelling, computational science and complex systems, cellular automata, or CA for short, represent an important and valuable concept, and as such have, since their inception, been the subject of extensive research. Scientists and researchers alike have used and built upon said concept in order to gain insight into complex, dynamic processes with difficult to predict outcomes or even development, and monitor interactions otherwise near-impossible to foresee. In fact, cellular automata have been instrumental in attempts at mapping one of the most difficult but important concepts, one which touches basically every field of study; we speak, of course, of emergence, or emergent behaviour [15].

Despite what one would assume, the efficiency of a cellular automaton, as far as the aforementioned application is concerned, doesn't come at the expense of simplicity. Some context is, of course, required for such a statement to be made—by simplicity, we mean that of the *conceptual* kind. In its very basic form, a cellular automaton can be described as a grid-based computational model composed of an array of cells. This model evolves over discrete time steps, in which each individual cell exists in one of pre-established states and determines its next state, that is its state in the following discrete time step, based on predefined rules that consider the states of neighbouring cells. As the time progresses, patterns ranging from simple to complex emerge, demonstrating how simple rules can create complex behaviours.

2.2 Basic Concepts

While a very brief explanation has already been provided of what a modern cellular automaton is and how it functions, the subject of this thesis requires a more thorough understanding of the subject at hand. This section's aim is, hence, to introduce the essentials of working with and understanding cellular automata, and provide all the necessary knowledge we need to grasp concepts introduced further down the line.

2.2.1 Components

Cellular automata are comprised of several basic components that work together to facilitate its behaviour. Among the most important of these are the ***cells***, the ***grid***, ***states***, ***neighbourhood***, ***time steps*** and ***rules***. These would all be considered the very building

blocks of a cellular automaton, without which it cannot properly function, and so warrant a thorough understanding. The following text does attempt a proper description of all the aforementioned components except for rules, which are more complex and as such deserve a section of its own.

To begin describing how a cellular automaton functions, it would probably be best to start with the primary piece of the puzzle—the ***cells***. This term refers to the basic units, abstract in nature, used by the automaton to exhibit own behaviour. They are the entities subjected to the individual automaton's pre-defined rules at each discrete time step and whose states are hence updated accordingly. It is worth noting that this change of state is typically synchronous—however, the subject of this thesis does revolve around the opposite approach, meaning asynchronous.

Although we had established cells to be abstract entities, they still do have to exist somewhere, occupy some space within which their states can be saved at each discrete time step so as to enable both their visualisation and subjection to the automata's rules for the next step. The ***grid***, then, typically represented as a one, two or three-dimensional array (or even higher [26]), exists as a spatial domain which serves this very purpose. We can think of it as a canvas upon which the computational model unfolds.

The words ***states*** and ***time steps*** had already been used quite a few times, so let us define their meaning as well. The former encapsulates all the various configurations the automata's cells can and do assume at each discrete time step, distinguishing themselves as a firmly established „player" on the field. In most cases, we can make do with binary configurations—typical example being an „on" and an „off" state. However, states can also be declared as a wide range of values, though one has to expect this would inevitably mean greater computational complexity. The latter of the terms, time step, is used to represent a single unit of time in the automaton's evolution, or, in other words, the progression from one iteration of the updating process (where all cells within the grid are updated according to the automata's rules) to another.

Last but not least, the ***neighbourhood***. A neighbourhood of a cell refers to the surroundings of each individual cell, or more specifically it determines which adjacent cells are to be taken into consideration during the updating process. This range of influence is entirely adjustable, though the two most common configurations (for a two and higher-dimensional grid) are referred to as „von Neumann" and „Moore", where the former establishes only the directly connected cells as a neighbourhood (extending to include all cells orthogonally adjacent to the von Neumann neighbourhood itself in case of bigger depth), whereas the latter takes into consideration all immediately adjacent cells (this reach would, in case of higher depth, extend to include cells immediately adjacent to the ones defined as neighbourhood in the previous depth). The following diagram should clarify further.

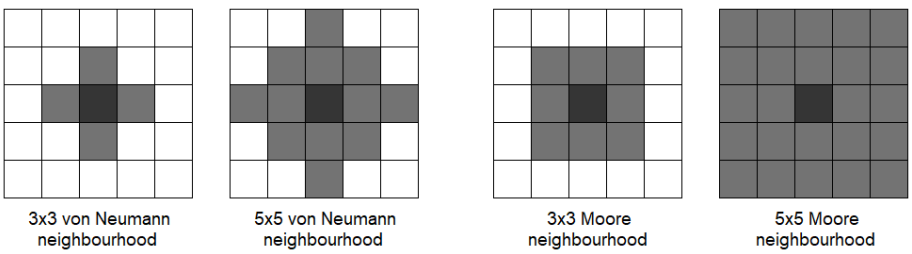

Figure 2.1: von Neumann and Moore neighbourhoods

2.2.2 Rules and Rule Tables

Rules, also called transition rules or transition functions, are the driving force of a cellular automaton. Of course, the components described previously all contribute to a cellular automaton's function, but it is rules which govern the very reason we built it in the first place—its behaviour. They provide what we can look at as a template on which cells base their evolution during each time step, and as such every slight change to these rules can completely alter the course and result of the simulation.

Rule tables, often represented as lookup tables or sets of conditional statements, can essentially be seen as nothing more than collections of these rules—that is, they encapsulate a set of rules that define the behaviour of the automaton. Each entry or row in a rule table acts as input that corresponds to a specific configuration of a cell and the cells within its neighbourhood, and its output then, consequently, defines the new state of the central cell for the next time step.

Deterministic or Stochastic Nature

There are several ways to make distinctions between cellular automaton's rule sets, but one of the most important ones we can make is whether the rule sets are *deterministic* or *stochastic*. The difference here is very simple, and lies in whether a degree of randomness is or isn't incorporated into them and thus into the decision-making during each of the automaton's update steps.

Rules flagged as deterministic are simply that; they follow set, firmly established patterns and rules to which they strictly adhere, ensuring consistency when it comes to the matter of each individual cell's change of state at each discrete time step. These rules will almost exclusively provide the same outcome for the same starting conditions. In other words, a deterministic approach maintains uniformity across the board, which means predictability as well.

It is up to the concrete automaton's task and desired outcome whether it warrants the use of a stochastic approach. Sometimes, some amount of unpredictability is desired or even required to achieve correct results [34]. In such a case, the only option is to turn to stochastic rules, which bring in randomness by various means; be it by involving additional factors in the decision-making process during a cell's update step evaluation, defining probabilities of whether or not the cell will actually even undergo an update at a given discrete time step, and so on. The stochastic approach is much preferred in various fields where randomness, uncertainty, or probabilistic elements play significant roles in modelling or understanding complex systems. This includes fields such as biology, physics, economics and more.

There's also the possibility of combining the aforementioned approaches within one cellular automaton. This hybrid approach would allow to incorporate both deterministic qualities as well as randomness in the system's rules, which would then directly translate to both predictability and variability in the cellular automaton's behaviour. This would then enable modelling scenarios where certain aspects of the system follow clear patterns, while other areas differ between each run of a simulation. It should be clear that such an approach would lead to a richer diversity of exhibited behaviours and patterns.

2.2.3 Boundary Conditions

Whilst this section would technically fall under the topic of rules in the sense that it still deals with the issue of cells' updating process, they're not quite the same. Think about

it; transition rules in of themselves are a great way to define the behaviour of a cellular automaton, but there is an important problem which has, up until now, been ignored. A part of the process, or rather of the grid, for which these rules will simply fall apart. We're of course talking about the cells *at the inner bounds of the grid*. These cells cannot, for their lack of a complete neighbourhood, use exclusively the defined transition rules as guidance, and as such are additionally subjected to what we call boundary conditions.

Boundary conditions aren't rules per se. They do influence the behaviour of cellular automata, but only by defining the behaviour of cells that are missing a part of their neighbourhood due to their position at the edge of the grid. They just set the context or constraints within which transition rules operate. Now that we understand what boundary conditions are, let us move on to more in-depth explanation and classification. There are three kinds of boundaries we will consider here: ***fixed value***, ***periodic***, and ***reflective*** boundaries [2].

Perhaps the simplest and laziest of the three, *fixed* boundary conditions maintain the same state for cells at the grid's edges throughout the whole simulation. This approach is very easy to implement, but can cause static behaviour or specific effects along these edges.

Reflective boundary conditions treat the grid like a loop, connecting opposite edges together and thus creating a continuous environment, letting patterns move smoothly across the grid's boundaries (by „spilling over“ to the opposite side).

Unlike fixed and reflective boundary conditions, *periodic* boundary conditions enable the cells at the grid's edges to interact with the outside environment. Cells at these edges could change according to varied rules, considering information from beyond the grid or reacting dynamically to external influences. These types of boundary conditions prove valuable for simulating systems that have interactions with an external environment.

2.2.4 Emergent Behaviour and Rule Complexity

We've already learned that basic transition rules can come together to create intricate, unpredictable patterns on a larger scale. All of these unexpected patterns can pop up even though they weren't explicitly programmed into the original rules, simply as a result of hard-to-predict interactions between cells themselves; these interactions, or rather their result, is what we refer to as *emergence*.

The idea itself isn't too complicated. Cells simply follow the rules that were set to them and engage with their neighbours. But this suddenly sets off a chain reaction across the whole grid and the result is the creation of intricate patterns and behaviors that go beyond what one might predict by looking solely at just the pre-defined rules. Emergent behaviour surely is a captivating, and perhaps a little scary, concept—scary precisely for the fact that we can rarely ever foresee its results, but captivating for the exact same reason.

However, this idea isn't actually novel, nor is it limited to computational models. Before cellular automata, emergent behaviour could have been observed in many a natural system, and is therefore important for cellular automata's application in the fields of biology and physics [15].

As for *rule complexity*, this term refers to how transition rules and boundary conditions affect the varied and intricate nature of the resulting patterns. Rule complexity exists on an entire spectrum, from really simple rules which create repetitive patterns or patterns that are easily predictable, to complex ones which can contribute to the creation of more unpredictable but interesting results.

The link between rule complexity and emergent behaviour should be fairly obvious. The more complex the rules, the more significantly they can impact the diversity, richness, and unpredictability of emergent patterns. A common misconception is that complexity equals more complicated rules or simply greater amount of rules, but this isn't necessarily the case; simple-sounding rules can introduce great complexity, while a lot of rules which look complicated can, ultimately, only work towards a mundane, unchanging result.

2.2.5 Universality

When learning cellular automata, one might stumble upon the term *universality*, specifically in the context of labeling an automaton as „universal". Universality basically boils down to the ability to perform various tasks by simply being programmed differently while still retaining the same underlying construction. It means the system can mimic diverse and complex computations, akin to a universal Turin machine (so, able to handle any computation), and essentially emulate any other system.

In our case, this means a „universal" cellular automaton can mimic diverse and complex computations and as such become incredibly adaptable whilst still retaining its simplicity.

2.3 Types of Cellular Automata

Though he didn't know it back then, von Neumann's prototype of what he intended to turn into self-replicating machines was the very foundation of what we now know as cellular automata, that much is clear. It was him who planted the seed that others later adapted, albeit in the pursuit of something a little bit different; that is, simulation of complex patterns and behaviours. However, this evolution wasn't a straightforward process, and it most certainly didn't happen overnight. The road to where cellular automata are right now started with von Neumann's idea, but the first big milestone happened in 1968, when Stephen Wolfram introduced one-dimensional—otherwise known as elementary—cellular automata. Since then many adaptations and modifications arose, each paving the way towards bigger and better understanding of the concept's underlying potential.

One could say that 'cellular automata walked so machine learning could run'. This metaphore needs to be taken with a grain of salt, but nevertheless, it perfectly captures the importance of cellular automata research, mainly its contribution in shining light on how simple rules and interactions among individual components can lead to the generation of complex behaviours and patterns [15]. While not a direct precursor to machine learning, the principles and understanding gained from studying cellular automata helped shape the broader field of computational intelligence and thus indirectly contributing to the evolution of machine learning methodologies [4].

From the points raised previously, it is clear how important, crucial even, it is to understand how the evolution of cellular automata happened and what ideas its most significant milestones brought to the table. However, while there have been lots of types of cellular automata devised over the years, we will point out and examine only the four of the most beneficial for our purposes, that is for the purposes of this paper. Those would be ***Elementary cellular automata***, ***Stochastic (Probabilistic) cellular automata***, ***Asynchronous cellular automata*** themselves, and, last but not least, the very interesting ***Neural cellular automata***.

2.3.1 Elementary Cellular Automata

Elementary cellular automata specifically refer to one-dimensional cellular automata with two possible cell states: 'on' or 'off' (labeled 0 and 1). Since they are one-dimensional, meaning that they consist of a linear array of cells, its rules depend only on the state of the currently evaluated cell, and the states of its two immediate neighbours—the one to the immediate left and the one to the immediate right of it specifically.

Because the neighbourhood consists of only 3 cells, and each cell can only be of either the value of '0' or '1', a single elementary cellular automaton has exactly $2^3 = 8$ possible configurations for a cell and its neighbourhood, and each elementary cellular automaton has to also define the resulting cell state for each and every one of these configurations. This makes it so there there are exactly $2^8 = 256$ elementary cellular automata rule sets [15].

Based on a numbering method created by Stephen Wolfram, these rules sets (hereinafter only rules) are categorized as simple binary numbers as follows: each rule is shown as a set of eight numbers (zeros and ones), where each number (from left to right) represents a configuration of a 0, 1, 2,..., 7 written in binary (so 000, 001, 010,..., 111) and the '0' and '1' in the place of a specific configuration then denotes the resulting state for a cell in the case of said configuration. To clarify, we can showcase this numbering system for some rule; let's take Rule 30, for example, where Rule 30 = 00011110_2, for which the configuration is as follows:

Current Configuration	000	001	010	011	100	101	110	111
Next State	0	1	1	1	1	0	0	0

Table 2.1: Rule 30 Elementary Cellular Automaton Rule Set

Another classification we do when it comes to these rules is based on their behaviour, where we, for instance, distinguish between rules which lead to simple, repetitive patterns and those which show results directly opposite. It was again Wolfram who create these four categories:

- Rules leading to simple and uniform patterns which often repeat.
- Rules leading to moderately complex patterns; they usually start as more complex before stabilizing.
- Rules leading to seemingly random patterns without any observable uniformity.
- Rules leading to complex, organized structures without observable repetition and randomness.

The wonder of elementary cellular automata lies in their knack for showing unexpected complexity despite following basic rules at the cell level. Even though these rules are straightforward, the resulting patterns and structures they create can be strikingly intricate and hard to predict. This surprise complexity has sparked lots of exploration into what these automata can do, sparking research in computer science, physics, and biology.

2.3.2 Stochastic Cellular Automata

Stochastic, also known as Probabilistic, cellular automata are a type of cellular automata which introduce probabilistic behaviour (as opposed to the usual deterministic approach)

by employing probabilistic rules (see: 2.2.2). Simply put, instead of strict adherence to the automaton's rules, a degree of randomness in the form of unpredictability and probability distributions is sprinkled in, meaning sometimes, cells simply toss a coin or take a chance instead of doing what they're supposed to. This means we can say that they do not always do what one would expect; same as—you guessed it—processes found in nature.

The reason for bringing this type of cellular automata up in the context of this paper should then be clear. Processes found in the biological field rarely ever are idealistic or predictable; while most organisms have some pre-programmed instincts, they don't always strictly conform to these. A lion could choose to hunt down a gazelle, but he may also decide he'd already eaten enough that day, we just don't know. But before we get into individuality and such, let us stop here, as the contribution-related point should already be made.

2.3.3 Asynchronous Cellular Automata

This type of cellular automata holds a very important role in the evolution of cellular automata for the introduction of a very intriguing concept—cells' *individuality*. Or something close to it, at least. But how?

Let us take a look at the updating process, which has, so far, been pretty straightforward. Every update step, each and every cell is subjected to the automaton's rules and changes its state accordingly. We've introduced stochasticity to shake things up a bit, but nevertheless, the updating process has always had one goal, which was to synchronize the update of every cell across the whole grid—we aptly call this update pattern *synchronous*. But what would happen if we did away with this approach?

Asynchronous cellular automata use the *asynchronous* approach to cell updates, meaning they allow cells to update independently at different times by giving them each their own update schedule. This entirely breaks the uniformity synchronous updating is known for apart and sows seeds of much desired chaos, bringing in irregularity and flexibility of the automaton's behaviour. As every cell is given more freedom in the 'when' department of the updating process, they are, for example, allowed increased flexibility in reactions to internal and external stimuli. This brings us to the word mentioned at the beginning of this subsection. Simply put, where synchronous updating allows for the simulation of collegiality amongst cells, we can safely say that asynchronous updating allows for the opposite, meaning their individuality.

This fascinating concept is very important in nature—not just because it is the subject of this thesis. Therefore, it only seems appropriate that we dedicate an entire separate section (see: 2.4) to it, where we examine asynchrony and asynchronous cellular automata more thoroughly.

2.3.4 Neural Cellular Automata

It felt fitting to also include Neural cellular automata in our list, the reasons for which are numerous; aside from their concept making for a very captivating study, the most significant reason for their inclusion would be their aim to mimic the behaviour of neurons (think: biological neural systems). The concept of Neural cellular automata has been formed as the culmination of various research and studies in the fields of cellular automata and neural networks (among others); it is difficult, then, to attribute their creation to a single person and hence may prove difficult to search for information. However, there is a quite prominent article [23] published in 2020 by Alexander Mordvintsev, which could

be deemed as a perfect introduction into the subject, and as such will be used in further explanation.

Based on the aforementioned article, Neural cellular automata bear a striking resemblence with Convolutional neural networks, allowing for the construction of Neural cellular automata models using components readily available in popular machine learning frameworks. The article also proposes that Neural cellular automata could be referred to as „Recurrent Residual Convolutional Networks with 'per-pixel' Dropout“—this name aims to highlight their similarities and connections to established neural network architectures.

We can infer the function of a Neural cellular automaton as follows: every cell serves to mimic an artificial neuron, and its state then reflects the activation level or behavior of that neuron. These artificial neurons still interact with each other as cells would in a cellular automaton; that is, by updating their states based on local rules and considering the states of nearby neurons, thus mirroring how information is exchanged between neurons in real biological networks.

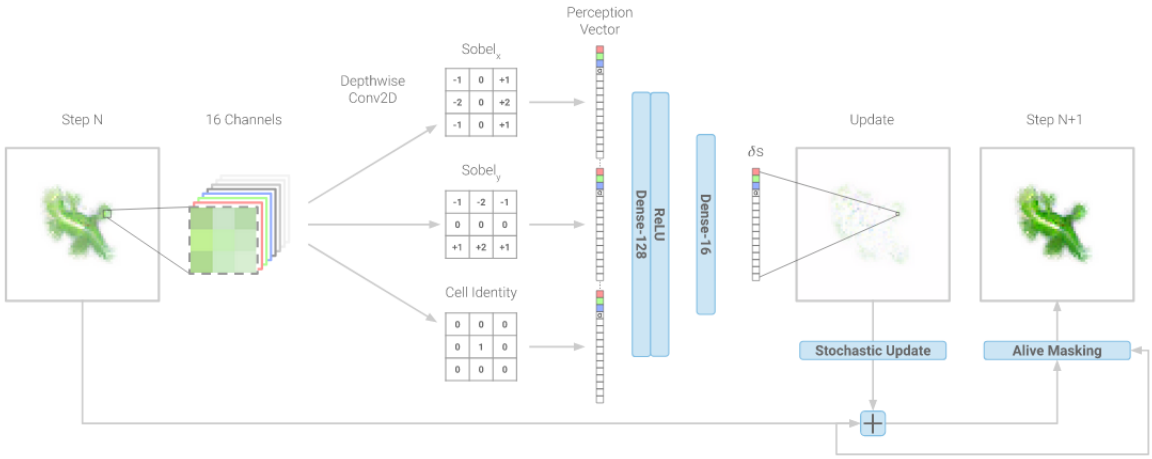

Figure 2.2: A single update step of the Neural Cellular Automata model [23].

Albeit discovered only relatively recently, Neural cellular automata have already gained attention for their potential applications across various fields, particularly for their ability to mimic the behaviour of neural networks whilst operating within the simpler computational framework of cellular automata. A fundamental aspect of Neural cellular automata is also the concept of self-organization, which is noted to be gaining interest in mainstream machine learning, particularly with the popularization of Graph neural network models.

2.4 Asynchronous Approach

Multi-dimensional cellular automata in their rawest form already are amazing tools of complex patterns and behaviour simulation, on top of being not too complicated conceptual-wise. However, while they would suit most needs, we can't always say the same for biological processes; especially when it comes to the manner in which updates occur. After all, most things in nature don't by any means conform to the idea of synchrony. There is no synchronizing „clock“, so to speak, to orchestrate every living being and manage a universal overlook of all processes which may occur, nor is there any proof to the existence of a force

dictating when, where and what happens. Most natural systems act on their own accord, and as such, so they should in our simulations.

Though nature doesn't follow a strict schedule, it doesn't mean every biological simulation requires asynchrony in order to exhibit correct behaviour; in fact, in most cases, the most optimal solution would simply be to test various updating schemes (including synchronous) and decide on which would be the best for our use-case based on observed results [12]. In simulations focusing on phenomena where cell-level dynamics aren't the main focus of study (so, ones where our goal is only to observe broader patterns or behaviours), the synchronous pattern would be more than adequate whilst easier to implement. The choice between synchronized and asynchronous models should match what we're trying to understand about the biological system under study.

Nevertheless, there are simulations where individuality among cells has a part to play in how true the result is to the real deal. For these purposes, we will define what asynchrony is and how to go about implementing it in our systems.

2.4.1 Defining Asynchrony

Asynchrony is easiest to imagine as the opposite of synchrony, or rather anything that doesn't fall under the description of synchrony. Simply put, anything that isn't synchronous is inherently asynchronous. Where synchrony refers to the state of happening at the same time, which is usually achieved either through precise timing or through coordination of the process by which the involved subprocesses are manipulated, asynchrony can be derived as every scenario where that isn't true.

In cellular automata, asynchrony means that cells update independently of other cells and ergo are given a greater degree of autonomy, being provided their own internal clock as well as the ability to update their states without regards to the global system state. This consequently leads to parallelism, where cells can perform computations concurrently, which has a plethora of benefits when one's intent is to involve cell-level dynamics in the simulation.

2.4.2 Fully Asynchronous Updating vs. Alpha-asynchronous Updating

Suppose we have already evaluated our options, and have come to the conclusion that what we're attempting to simulate could benefit from our taking the asynchronous route. We now have our direction, but before we can move on to the implementation itself, we still have a couple more decisions to make; one of which is the choice between *fully asynchronous* and *alpha-asynchronous* updating [29].

If one were to form an abbreviate description of the difference between both options, it would be that fully asynchronous updating takes the idea of asynchrony and applies it in full, with no regards to keeping any conceivable organization, whilst alpha-asynchronous updating adds a bit of structure or rules for the updating process while still keeping the flexibility and dynamic nature that comes with the inclusion of asynchrony.

In other words, the difference between both options has to do with the level of order that the concrete phenomena requires. In systems like cellular automata, fully asynchronous updating means that each and every cell will be updating wholly independently of eachother to achieve a highly decentralized process; meaning, no fixed update sequence. This results in a system that is fully unpredictable and highly complex.

On the other hand, we use alpha-asynchronous updating when we do want to keep the asynchronous updating of cells, but we want to introduce a low degree of structurizing

so the simulation doesn't run completely amok. In this approach, we set a few rules or constraints to the updating process to introduce some level of order, such as, for example, allowing cells of a certain state to update before all others. This way, we can still reap the benefits of asynchrony whilst retaining some authority over how the simulation evolves.

2.4.3 Update Schemes

The other decision we have to make is to choose an update scheme. Update schemes are basically guidelines for how to implement the updating process in regards to either setting the order (or lack thereof) of updated cells, or giving each cell an update clock of its own and thus introducing complete individuality. There exist numerous update scheme expositions; therefore, for the scope of our research, we will adopt the approach outlined in one such paper (Cornforth et al, 2005) [9].

- The *synchronous* scheme - at every time step, all cells undergo simultaneous updates (this is the standard, synchronous model, presented here only for comparison)
- The *random independent* scheme - nodes are selected randomly with replacement at each time step. As stated in the paper, asynchronous random boolean networks could be applicable to biological systems due to the exact points raised previously in this thesis—the perceived absence of a synchronizing clock.
- The *random order* scheme - very briefly, all nodes are updated at each time step in a random order.
- The *cyclic scheme* - a fixed update order is created at random during model initialization, determining the selection of a node at each time step.
- The *clocked* scheme - upon initialization, each cell is given an individual timer set to a random period. When this period reaches zero, the cell undergoes an update and the timer is reset. This achieves autonomous updating which proceeds at different rates for different cells.
- The *self-sync* scheme - this scheme is similar to the clocked scheme, except it also incorporates local synchrony by having the update sequence be influenced by local interactions (neighbourhood).

2.5 Cellular Automata in Biological Systems

We know that cellular automata excel at modeling complex systems—and there are few systems more complex than those found in nature. The initial development of cellular automata was, arguably, also motivated by the prospect of modelling biological phenomena [15], along with mathematical and computational curiosity, so there is no doubt that many knew the significance its discovery would have on this field soon after the concept was laid down—indeed, the first big stepping stone in the exploration of cellular automata capabilities, made mere 4 years after von Neumann's introduction of their concept, was Conway's Game of Life, which aims to simulate natural dynamics. Since then, cellular automata have found extensive applications in modeling biological processes due to their ability to simulate complex systems based on simple rules.

When the terms cellular automata and biology get mentioned, the minds of most would immediately be able to supply many an application, such as tumor growth modeling, ecological systems study (e.g. predator-prey interactions) or morphogenesis. We will, however, primarily focus on the study of pattern formation, as this was among the main concepts which drove forward the development and exploration of cellular automata within the realm of natural systems. Alan Turing's work on reaction-diffusion systems aimed to explain the formation of patterns in biological systems, offering ideas that later contributed to the development of cellular automata models simulating similar phenomena—from his work, a model was derived, one capable of explaining pattern formation without a preformed pattern [3].

Since then, many studies based on cellular automata arose, particularly those focusing on tumor growth simulations as well as others with focus on the study of ecological systems. Even though recent years showed various scientific fields' attentions drifting towards machine learning, including biology, cellular automata still remain valuable tools in the studies of the aforementioned applications, as they offer a different approach; meaning, of course, their emphasis on the simulation of systems based on simple rules and local interactions. As such, cellular automata and their iterations will likely remain a valuable tool in modeling biological processes for the foreseeable future.

Chapter 3

Machine Learning

3.1 Introduction

It is safe to say that everyone who has had any interest in the world of technology and computer science in recent decades has heard of the following two terms: *artificial intelligence* and its subfield *machine learning*. It would be hard not to, as both of these fields have garnered unprecedented attention for their capabilities in solving problems encountered in various aspects of our lives—perhaps most recognizable is the use of machine learning in the fields of e-commerce and marketing, but it's decision-making and data analysis prowess also has other applications, such as in voice or face recognition systems, security, climate data analysis, education and healthcare.

Machine learning is a departure from the traditional ways of computational science, where computers follow explicit instructions written into algorithms for calculations and problem-solving. Machine learning algorithms approach problems from a different angle, rooted in statistical analysis—they allow computers to learn from sample data inputs through training, analyze the data statistically afterwards, and produce output values within specific ranges [31]. This approach speeds up what would otherwise be a time-consuming and tedious process—sometimes drastically.

We can conclude, then, that at its core, machine learning presents a powerful mechanism for extracting knowledge from massive data sets, endowing machines with the capability to learn and adapt autonomously. It's not too far of a stretch then to say that machine learning reflects the complex dynamics of neural networks in the mammalian brain; that is, specifically the behaviour of neurons, which are interconnected in complex ways to process complex data [20]. This process, where the brain learns and enhances its data-processing capabilities by forging new connections between neurons, was the inspiration for artificial neural networks—the backbone of machine learning algorithms. In the realm of machine learning, neural networks stand out thanks to their proficiency in various information-processing tasks, mainly due to their ability to recognize patterns and generalize learning to novel, unseen data.

Depending on how learning is performed and how learning feedback is provided to the developed system, we also classify the machine learning methods used; the two most widely applied methods are supervised and unsupervised learning, where the former trains algorithms on sample input and output data which is labeled, while the latter provides the algorithm with data without labels so as to allow it to find structure in them. Between supervised and unsupervised learning is reinforcement learning, a model in which an agent optimizes its behavior based on positive feedback (rewards) received in response to signals.

As should be evident by now, there is a lot to learn about this ever-expanding field. Even attempting to summarize it all into these few paragraphs has proven quite difficult—and unsuccessful. This chapter sets a goal to properly, in more detail, explore all aforementioned topics as well as those which couldn't be fit into this short summary, comprehensively distilling years of research with the aim to provide an in-depth understanding of the subject as a whole and its contemporary applications.

3.2 A Brief Historical Analysis

The world is filled with examples of machine learning usage, one doesn't need to look very far to notice them; it only takes paying a little bit of extra attention. How does the mail server know which received messages are spam, for example? How do YouTube and Netflix recommend videos and shows based on your recent activity? What about virtual assistance technologies like Siri, or Alexa? Yes, those use machine learning to formulate their responses as well. Over the last couple of decades, the world fell in love with machine learning, and its research has seen a significant boom as a result; though, the history of machine learning development does involve periods of fluctuating interest. There were times when enthusiasm waned, which are often referred to as the „AI winters"—a term first coined in 1984, presented as the topic of a public debate at the annual meeting of the American Association of Artificial Intelligence (AAAI) [28].

But we digress. The fact of the matter is that, at the time of writing this, machine learning research as well as that of artificial intelligence is steadily rising in popularity. There is a myriad of reasons to learn how it got to this point, how machine learning developed over the years, but the main one is that by doing so, we unveil how and why certain methods and techniques came to be. And for that, we best start at the very beginning.

In 1943, Warren McCulloch and Walter Pitts published a seminal paper titled „A Logical Calculus of Ideas Immanent in Nervous Activity" [19], in which they proposed the first mathematical model of an artificial neuron, providing a theoretical basis for *neural networks*. Later on, during a traditional debate on the topic of defining intelligence, Turing introduced the concept for a universal machine capable of mimicking human thinking and behaviour in tasks requiring some degree of intelligent thought. He proposed a practical test for computer intelligence, specifically its capability of becoming equivalent to that of a human [32]. This test—now simply known as the Turing test—became the groundwork for the development of computer science and machine learning.

While the idea of machine learning and artificial intelligence was already present in the minds of the scientific world, it wasn't until 1956, when John McCarthy coined the term „artificial intelligence" during the Dartmouth Conference, that the era of the study of both fields truly began. Envisioned were machines capable of learning and adapting without explicit programming, a departure from traditional computational approaches, and early perceptions of this concept were often fueled by optimism of unlocking the full potential of machines to mimic human intelligence, primarily so in terms of problem-solving, decision-making, and pattern recognition.

Machine learning and artificial intelligence were at the forefront of research. The scientific world became obsessed with the very idea of systems capable of replicating human thought processes, so much so that, as swiftly as the enthusiasm soared, it also dwindled when research started encountering some significant bumps. Early developments slowed, until the early 1970s saw them basically grind to a halt when it became clear that tech-

nology at the time wasn't capable of fulfilling expectations. Challenges such as limited computational power, insufficient datasets, and the complexity of real-world problems contributed to a decrease in funding and interest, as many researchers and companies became skeptical about the practical feasibility of artificial intelligence and machine learning. The following years marked the first true AI Winter [28].

It wasn't until a decade later, in the mid-1980s, that the field experienced a resurgence, a reignition of interest. The time that followed became called the „AI spring," which is a clever play on words in regards to waking up from the previously mentioned AI winter. Advances in neural network research, the development of new algorithms, and increased computational power reignited interest in machine learning, primarily through the fact that researchers were seeking an instrument for solving complex problems, and the aforementioned advancements made it more feasible to explore and apply machine learning techniques in practical scenarios. It wasn't quite the same boom as when the topics of machine learning and artificial intelligence first hit the scientific world, but slowly yet surely, the curve of interest started bending back the other way.

Machine learning continued to be a topic that, while its exploration continued, its popularity was nowhere remotely close to what it is today. The slow and steady rise in research picked up the pace in the 2000s, when scientists turned to machine learning for its capabilities in sorting through a vast amount of data; a skill that was once again becoming relevant when various fields started producing larger data sets which required some sort of handling [13].

A particularly significant role in this time played the return of neural networks, this time in the form of *deep learning*, which checked all the boxes when it came to using the newfound powerful computing resources, breakthroughs in algorithmic efficiency and abundance of data to push forward research in finding a solution to not only the problems machine learning used to be regarded as suitable for before, but the introduction of deep learning also expanded these goals to include tasks such as image recognition, speech recognition and more complex decision-making.

Deep learning did have a rocky start, with many people not believing its usefulness when compared to what could be achieved using heuristic methods, Support Vector Machines (SVM), and other traditional approaches. The results were regarded as too similar, and so the usage of deep learning was called into question when there already existed other, easier, means of achieving the same goals. But all of these apprehensions stemmed from a fundamental misunderstanding of where deep learning truly shines—in dealing with tasks of a greater magnitude. Tasks which require the full use of the modern technologies, discoveries, hardware capabilities. It was only in using deep learning for solving these marginally harder tasks that its place at the forefront of research was truly solidified.

The latest key development in machine learning, particularly important in regard to processing sequences of data (primarily text and audio), are Transformers; who quite literally transformed the way we approach many machine learning tasks. These utilized a mechanism called „self-attention", which is a mechanism that essentially allows a model to weigh different parts of a sequence to understand context. This approach addresses limitations in traditional models, like the vanishing gradient problem that previously made it hard to train very deep networks. The self-attention mechanism in Transformers allows them to consider all parts of a sequence at once, which grants them the ability to handle long sequences more efficiently and as such helps avoid the kind of paralyzation that older models suffered from, where data processing could slow down or become stuck in complex computations. This was huge for tasks such as language processing, and made training

deeper networks more efficient, opening up new avenues for machine learning applications that were once considered too complex or computationally expensive. As a result, the research around Transformers saw a big boom, with 70% of the papers on AI that had appeared on arXiv in the years 2020-2022 having some kind of mention to this type of neural network [21].

As should be apparent by now, machine learning has, in its lifetime, been through a vast amount of development and has experienced significant breakthroughs. The first of such notable advancements is a study called LeNet, developed by Yann LeCun and his colleagues in 1998 [18], which laid the foundation for convolutional neural networks (CNN) and demonstrated the effectiveness of hierarchical feature learning. Fast forward to 2012, AlexNet, a seminal work by Alex Krizhevsky and his team [17], revolutionized image classification by leveraging deep convolutional networks, securing a decisive victory in the ImageNet Large Scale Visual Recognition Challenge. The following year, the Visual Geometry Group (VGG) introduced VGGNet [30], emphasizing the importance of deep architectures with uniform filter sizes, influencing subsequent architectures. Residual Networks, or ResNets, introduced by Kaiming He in 2015 [16], addressed the vanishing gradient problem by incorporating shortcut connections, enabling the training of exceptionally deep networks. Last but not least, 2017 saw the introduction of Transformers by Vaswani [33], which brought a paradigm shift in sequence-based tasks by utilizing the self-attention mechanism, allowing models to process data more efficiently and paving the way for modern language models like BERT and GPT. These milestones showcase the progression from LeNet to increasingly sophisticated architectures like AlexNet, VGG, ResNets and Transformers, each contributing crucial insights and techniques that have become integral to the evolution of machine learning.

As we travel the historical contours of machine learning, we witness not only a story of technological innovation but also a persistent human effort to push the boundaries of what new horizons machines can help us reach. From the early dreams of artificial intelligence to the practical reality of today's deep learning systems, this journey is a testament to human ingenuity, resilience, and the relentless drive for a future where machines and humans collaborate in harmony like never before.

3.3 Neural Networks

Neural networks are physical cellular systems capable of acquiring, storing, and using experiential knowledge [11]. This knowledge takes on the form of steady states or mappings in the network, allowing its retrieval in response to specific cues. Neural network processing often involves solving large-scale problems in terms of dimensionality, data volume, and computational complexity, which is a necessity for real-world applications in various fields.

Neural networks work differently from conventional digital computers—the goals of both approaches don't exactly align. You see, while digital computers excel at executing predetermined sequences of commands with incredible speed and accuracy, they fail to provide any means of solving problems where scenarios are hard to predetermine; unless, of course, one would be willing to write all of the use-cases out themselves, but in certain scenarios, such a feat is bordering on being impossible. This is where neural networks come into play. Their primary goal is to mimic the brain's ability to perform complex tasks, such as understanding speech and processing images, with remarkable efficiency. Even though they operate at speeds much slower than those of computers, their capability of solving problems described previously sets them apart.

Formally, a neural network is a computer system consisting of many interconnected nodes which process information dynamically in response to external inputs. We have already previously likened these nodes to neurons 3.1, stating that, like in the neural systems found in a brain, they collectively contribute to the network's ability to learn from data and make decisions. While this analogy with biological systems may have been the initial inspiration for neural networks, their architecture and operations differ significantly from the complexity of real nervous systems.

Renewed interest in neural networks in recent years can be attributed to both hardware- and software-related advances in computer architecture, especially so the big strides the fields of graphics processing units (GPUs) and computing parallelism have been achieving lately, as well as the availability of large amounts of data. This resurgence has been accompanied by a change in perspective, with a growing awareness of the importance of intermediate representations created in neural networks during the learning process [27]. These intermediate representations, embedded in the network's hidden layers, have potential implications for tasks such as transfer learning, extending the utility and flexibility of neural network architectures.

By their nature, neural networks are powerful tools for approximating complex functions and solving difficult computational tasks, from handwritten character recognition to natural language processing. As the field continues to develop, researchers are exploring new architectures, optimization techniques, and applications, driven by efforts to exploit the full potential of these approaches in addressing real-world problems.

3.3.1 Neural Networks vs. Artificial Neural Networks

First, there is a matter we need to clear up before we continue. The terms „Neural Network“ and „Artificial Neural Network“ are often used interchangeably, and you will see both terms being used from this point on, but there is a subtle nuance in their meaning, or rather what concepts they encompass.

The term ***Neural network*** specifically refers to computational models inspired by the structure and function of biological neural networks—that is, the ones found in a mammalian brain. They are composed of interconnected nodes, which we call neurons, arranged in layers, and it is through connections between these neurons and adjusting said connections through the employment of learning algorithms that they process data. Neural networks can encompass a wide range of architectures and approaches, including but not limited to artificial neural networks.

On the other hand, when we say ***Artificial Neural Network***, we typically refer to a subset of neural networks constructed with artificial neurons. This term encompasses not only neural networks but also other related approaches or systems that mimic aspects of neural processing, such as neuromorphic computing or spiking neural networks. Artificial neural networks typically include mechanisms for learning—a typical example being backpropagation, which is a key algorithm used for training artificial neural networks through adjusting parameters based on observed data in order to improve the network's ability to perform a given task. In other words, artificial neural networks are models which are based on neural networks, but employ additional approaches or systems.

In short, we can say that, while all neural networks are artificial neural networks, the same cannot be said the other way around. The former term encompasses a broader range of computational models inspired by neural brain activity, while the latter refers to net-

works which are a subset of neural networks, structured and trained to perform tasks using artificial neurons and connections while employing additional, advanced mechanisms.

3.3.2 Basic Components

As is true for almost any unique model or system, there are some concepts and components it is comprised of, and which we need to have at least a basic understanding of before we can dive into more elaborate and complex problems.

We should start with the fundamental computational units of artificial neural networks, which would be ***neurons*** (also called units, nodes or perceptrons). The purpose of these neurons is very simple; they process received input signals (often weighted values) by performing mathematical computations on these inputs and producing output signals accordingly. In addition to neurons, neural networks also sport an extensive network of ***weighted connections***, which influence and drive the flow of information between neurons. Think of these connections as paths between each two individual neurons, where each of these paths is given a certain weight. This weight determines the strength of said connection, and is adjusted during the network's training process in order to optimize the network's performance. Along with weights, there are also what we call ***biases***, which are constant values assigned to each individual neuron. These values are used as an additional means of influencing the range of outputs, which comes in especially handy in situations where the inputs are insufficient to trigger activation based on the weights alone. This is done during the computation process, by adding the bias to the weighted sum of inputs for the specific neuron before applying the activation function. This adjustment not only allows for accelerating or delaying the activation of a given neuron, but also provides each neuron with the ability to produce a non-zero output even in the absence of input signals.

Before we continue, we need to clear up a misconception that artificial neural networks don't differ much from their biological counterparts. While the very concept of artificial neural networks is inspired by neurons and neural networks found in biological brains, they quickly diverged from these concepts as the field of machine learning progressed throughout the years, and new complex ideas and techniques have been developed. As had been said by others, „It is easy to draw the wrong conclusions from the possibilities in AI research by anthropomorphizing Deep Neural Networks, but artificial and biological neurons do differ in more ways than just the materials of their containers [24].“ To further showcase the distinction, please refer to figure 3.1, which shows the differences between how a biological and an artificial neuron look and work.

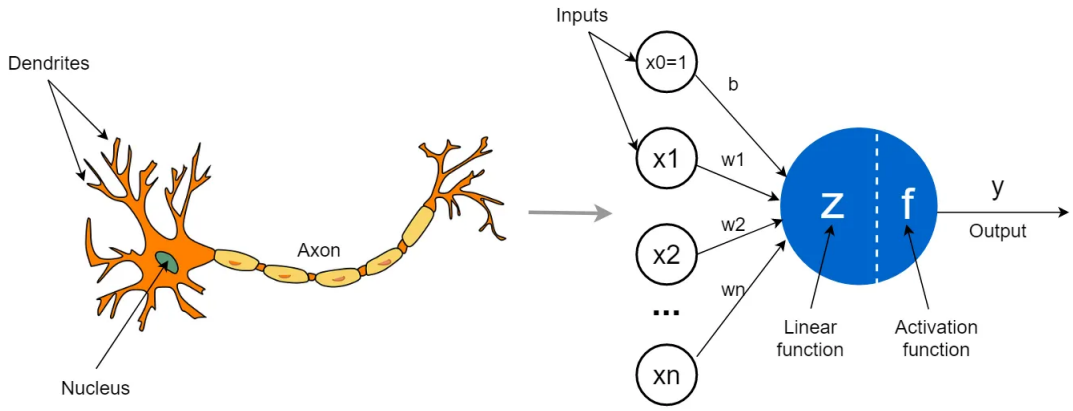

Figure 3.1: A biological neuron and an artificial neuron comparison (via https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc)

Neural networks are organized into ***layers***, each consisting of interconnected neurons and serving a distinct purpose in the information processing pipeline—by which we mean that each layer performs specific computations and processing on the input data. Imagine we have a system in place, as pictured in figure 3.2, consisting of three different types of layers; first and foremost, there's the *Input layer*, whose sole purpose is to accept input data and simply pass them on to the next layer. Each neuron in the input layer represents a feature or attribute that is passed as the input data, which means the number of neurons in this layer is determined by the dimensionality of the input data (how many distinct attributes are passed to the neural network). There are no activation functions associated with the neurons found in the input layer, and no computations are made on the input data either.

Past the input layer lies the realm of *Hidden layers*. Unlike the other two types of layers, this section can hold multiple layers, each consisting of multiple neurons through which they're connected. The number of hidden layers and neurons in each layer can vary depending on the concrete network's architecture. Hidden layers are where all main computations of the neural network take place; that is, where input data passed on by the input layer is evaluated and subjected to computations involving weighted sums of inputs followed by the application of an activation function. They introduce non-linearity into the network's computations, which allows for complex relationships in the data to be observed and learned, enabling the neural network to model intricate patterns and extract meaningful features from the input data.

Once the hidden layers work their magic, the results are passed to the *Output layer*, which is the final layer of the neural network and produces the network's predictions or classifications. The neurons found here compute the final outputs based on the information propagated from the hidden layers, with the option to also apply an activation function (depending on the nature of the task at hand). Each neuron in this layer represents the network's predictions or classifications for the input data.

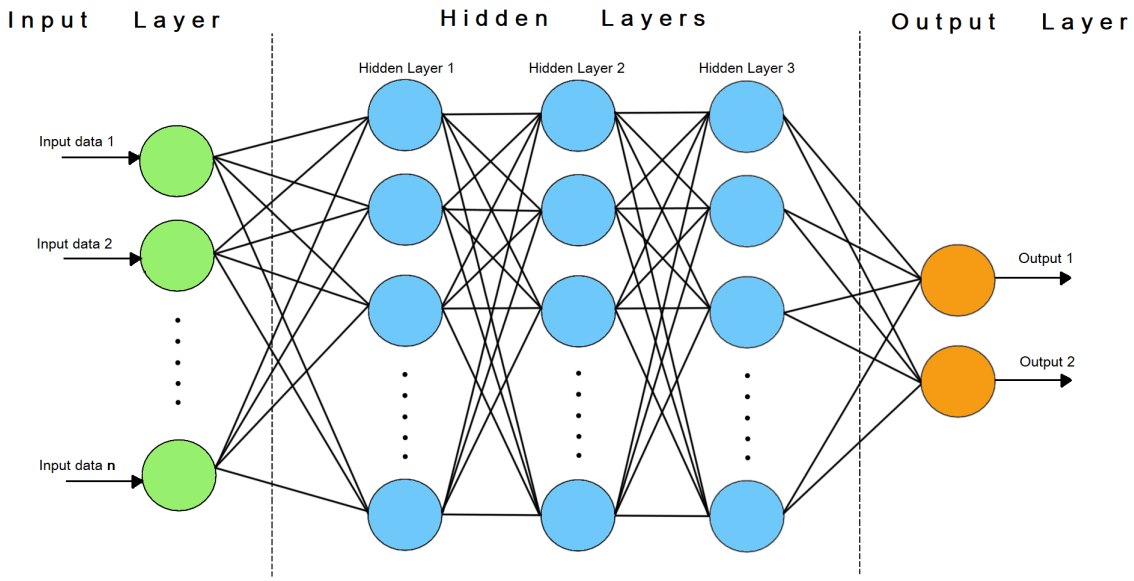

Figure 3.2: A neural networks layers architecture representation

To put this new information into a better perspective, let us draw a (rough) practical example. Suppose, for a moment, that we have a binary classification problem, where we want to predict whether a person will buy a product based on two features: age and income. The input layer will then consist of two neurons, each representing one feature (age and income). This data will subsequently be passed on to the hidden layers for processing—let's say that we will have two hidden layers, one which determines whether the currently processed person will buy our product based on age, and another layer which evaluates based on income. The neurons in these layers use activation functions and weighted connections to produce a real number; for example, if we consider a hidden layer with three neurons and use the ReLU activation function, the output of each neuron could be any real number within the range of the activation function's output (usually $< 0, +\infty$) for ReLU). These outputs are then passed to the next layer, be it another hidden layer or the output layer, contributing to the overall computation of the neural network. The output layer will then take the final values that it gets from the hidden layer, and uses its own computations and activation function to determine the final result („0“ or „1“ in this case, or „will“ or „will not“ buy).

We now have the basic structure of a neural network, and as such can move on to covering components managing its function. We should start this venture with a term already previously mentioned but never really explained, ***activation functions***—non-linear functions applied to the output of neurons in order to transform the input signal into an output signal, which gets then passed on to subsequent layers. In doing so, they introduce the non-linearity phenomenon into the neural network's computations, which allows the network to learn and represent more complex patterns. There are many different common activation functions, but amongst the most well-known are the Sigmoid function, the Tanh (Hyperbolic Tangent) function, the ReLU (Rectified Linear Unit) function, and Softmax (a sigmoid/logistic activation function that works on calculating probability values), each serving specific purposes depending on the network's architecture and objectives.

The core of the process of training a neural network is comprised of three primary components. Starting with ***Forward propagation***, this is the process which handles the

passing of data through the entire neural network, meaning it encapsulates accepting the data passed to the neural network on the input layer, processing the results on the output layer (usually by using an activation function), as well as the entire procedure happening in between, on the hidden layers—computing the weighted sum of inputs at each neuron, applying an activation function to the result, and passing the output to the next layer.

Then there's the ***Loss function***, which is a way of measuring how well the neural network's predictions match the true labels or targets in the training data. It quantifies the difference between the predicted output and the actual output for a given set of input data, with the goal of achieving as small a value of the loss function as possible, because smaller value means a more precise prediction or classification. One can then check the value of the loss function, adjust parameters of the neural network (weights, biases), and check the loss function again afterwards to see if the changes made a more positive or negative impact on the overall accuracy.

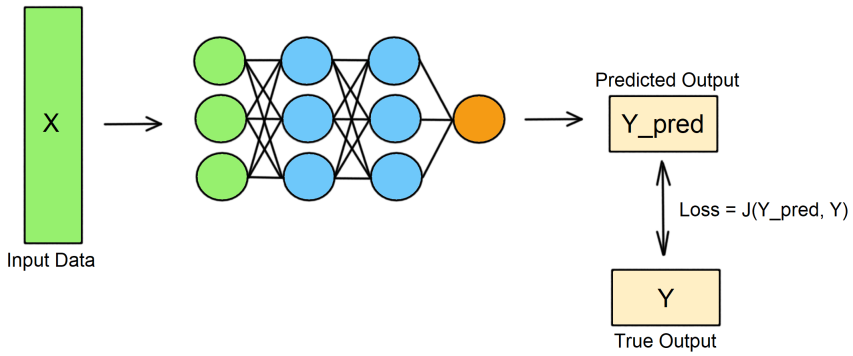

Figure 3.3: Neural Network Loss Visualization

The last of the three components is ***Backpropagation***, which is a little more complicated in comparison to the other two. What it does, as in the end result, is relatively simple—one can think of it like a feedback loop that helps a neural network learn from its mistakes. It's how the network adjusts its internal parameters (like weights and biases) to get better at making predictions, primarily using the previously mentioned loss function. But the way it does this is a whole other beast to tame. In essence, „In the case of neural networks, all the activation units are differentiable, as is the output of the network. Thus, if we can define an error function (eg. sum of squares) that is a differentiable function of the output, we can evaluate the derivatives of this error with respect to the weights, and find weights that minimize the error using gradient descent or other methods. This method of propagating error from the top layers to lower layers is called backpropagation [27].“ If you didn't fully understand that, I don't blame you, and will try to provide an explanation that is a bit simpler. The process of backpropagation starts by taking the loss function values (errors), which are then propagated backwards through the network, layer by layer, calculating gradients using calculus and the chain rule, to determine how much each parameter in the network contributed to the error. Based on the obtained gradients, the algorithm updates the network's parameters (weights and biases) in the opposite direction of the gradient to minimize the errors. These parameter updates are performed iteratively, with each iteration bringing the network closer to minimizing the error and improving its ability to predict more accurately.

The iterative updates are facilitated by a diverse array of ***optimization algorithms***, such as Gradient Descent and its variants, Adam (Adaptive Moment Estimation), Adagrad (Adaptive Gradient Algorithm) and others. These algorithms iteratively update the network's parameters based on the gradients of the loss function with respect to those parameters, effectively guiding the network towards a configuration that minimizes prediction error.

3.3.3 Types of Neural Networks

There is a tremendous amount of neural network types which emerged as results of research in the field over the years, so much so that listing and explaining all of them would be bordering on impossible, not to mention pointless strictly from a practical standpoint. Yes, it is useful to know of some of the options out there, not simply just to broaden one's horizons, but also to know of the possibilities should one find the need for them. However, diving too deeply into the details of each type may not always be necessary or practical, especially for those who are new to the field or who have specific use cases in mind, because each type of neural networks that had been invented over the years excels in a specific scenario, and each architecture has its own strengths and weaknesses; thus, having a broad understanding of the various types allows practitioners to choose the most suitable model for a given problem or application, after which they can further their knowledge pertaining that specific model. With that notion in mind, this section sets a goal to scratch the surface of what the world of neural networks has to offer as of now, without delving too much into details.

Feed-forward Neural Network (FNN)

Also known as the multi-layer perceptrons (MLPs) model, this type of neural network is one of the simplest forms of artificial neural networks, because the input data in this case travels only in one direction through all of the network's layers without forming any feedback loops or cycles within the network.

While feed-forward neural networks are versatile models capable of tackling many a task—they were among the first types of neural networks to demonstrate the capability to approximate complex functions and learn meaningful representations from data—they may struggle with capturing complex patterns in data with high non-linearity. Nevertheless, their simplicity and versatility does still make them an important neural network model, with uses in facial recognition, natural language processing and others, but also primarily as an inspiration for many other, more sophisticated, neural network architectures.

Convolutional Neural Network (CNN)

Convolutional neural networks are a highly popular neural networking model used extensively in computer vision and image or video processing tasks. They are similar to feed-forward neural networks, but are specifically designed to process structured grid-like data such as images or videos by leveraging principles from linear algebra and optimizing filters through automated learning. Additionally, convolutional neural networks use relatively little pre-processing compared to other algorithms, as they can automatically learn and optimize the filters during training.

Unlike feed-forward neural networks, convolutional neural networks use one or more convolutional layers in place of hidden layers (which can be either pooled or entirely con-

nected) to extract hierarchical features from the input. These convolutional layers play a crucial role in the whole process, as it is on these layers that convolution operations are applied to the input data, which involves sliding a set of learnable filters (also known as kernels) over the input data's maps. This allows the network to uncover and extract local patterns and features—such as edges, textures, shapes—from raw data by performing element-wise matrix multiplications and summations, saving us the trouble of extracting these features manually.

There also exist types of neural networks called *Deconvolutional neural networks*. These adopt the same principles as convolutional neural networks, but the other way around. Simply put, they are convolutional neural networks that use transposed convolutional layers and are trained to perform the reverse operation of convolution, also known as transposed convolution or upsampling. While this may not sound very useful at first glance, such an approach proves itself very capable of performing tasks related to the construction or reconstruction of data—such as image generation, image upscaling, or even image inpainting.

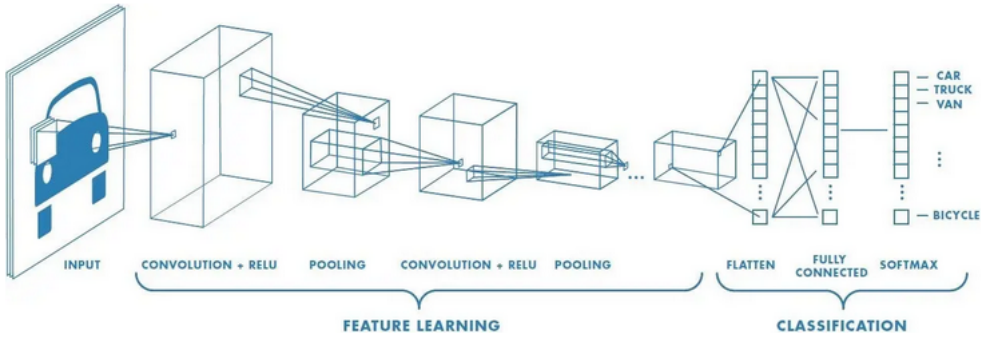

Figure 3.4: Convolutional Neural Network architecture (via https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/)

Recurrent Neural Network (RNN)

This type of neural networks is primarily used in tasks where one needs to process sequences of data; what does that mean, you may ask? Imagine you're reading a sentence, word by word. As you read each word, you're not just processing said word in isolation, but your understanding of it is influenced by the words that came before. For example, if you see the word „mouse" on its own, without knowing what words came before, you might think of an animal. However, if you knew that, before this word, the word „computer" was mentioned as well, your brain would automatically assume we're talking about the piece of hardware. In other words, your understanding of the currently processed word changed through context.

Recurrent neural networks handle this problem by maintaining a memory of past inputs—each neuron acts as a memory cell to retain information across time steps, which enables the performing of intelligent computation and implementation of feedback loops, where the network assesses mistakes and adjusts its predictions accordingly. It does so by utilizing a process that saves the output generated by the network's processor nodes and feeds it back into the algorithm. Instead of just processing one piece of data at a time, the network remembers what it had seen before and uses this knowledge to further its understanding of the current data. This memory helps these types of neural networks tackle tasks where the

order of data matters, like predicting the next word in a sentence, recognizing handwriting, or even generating music.

However, as is true with many other things, these types of networks don't come without any flaws. Their primary flaw stems from the fact that they can face problems holding information when the sequence of data is really long, skewing the predicted results. This is where variants come in; primarily so the *Long Short-Term Memory (LSTM)*, which, in short, is a specialized variant of recurrent neural networks designed to address the aforementioned issue (more specifically the vanishing gradient problem [1]), and as such remember important information for longer periods of time. If we had to describe this variant in more detail, we can say that they introduce memory cells with gates that regulate the flow of information, allowing them to selectively remember or forget information over time. This, as you may have guessed, means they're capable of more effectively capturing important dependencies in sequences (think information required in a multitude of sequential data processings), making them well-suited for tasks such as machine translation and speech recognition.

Another variant, so to speak, of recurrent neural networks, is the *Sequence-to-Sequence (Seq2Seq)* model. This model is also designed for tasks involving sequences of data, but it differs from the ones previously mentioned in that it essentially utilizes the other variants, such as RNNs or LSTMs themselves, to do its work. The Seq2Seq model consists of two main components: an *encoder* and a *decoder*, both of which can be implemented using RNN layers, and we can think of them as translators; the former, encoder, takes an input sequence, scans it for important information, and then generates a vector of a fixed size—which we call *context vector*—representing this information. We can, therefore, say that the input had been „translated" into something more comprehensible. The decoder then uses this context vector, along with what it itself had generated so far, to produce an output sequence, essentially „translating" it into another sequence of data, such as a translation of text in a different language, a summarization of a longer text, or a response in a conversation.

Aside from LSTMs and Seq2Seq models, we should also mention *Hopfield Networks*. These are in no way comparable to the former two in that they do not address the same issue; instead, Hopfield networks are used to solve problems related to associative memory or pattern recognition. They are designed to store patterns as stable states and retrieve them based on partial or corrupted inputs. In other words, they are used for content-addressable memory tasks, where the network learns to associate input patterns with specific output patterns. This makes Hopfield networks useful for tasks such as pattern recognition, optimization, and content-addressable memory.

Deep Neural Network (DNN)

Deep neural networks are among the most popular types of neural networks, for good reasons. They're capable of automatically learning layered representations of raw data by utilizing *deep learning techniques* (more on those later 3.5), where the most important part of that statement is the word „automatically"—one of the key advantages of utilizing deep learning is the ability to find and learn features in raw, unlabeled data, entirely without any manual feature engineering involved. This makes deep neural networks highly adaptable to different domains and reduces the amount of domain expertise required to build effective models.

However, while they are certainly a very interesting and, arguably, powerful type of neural networks, the value of these models comes primarily from the fact that they use deep learning—which is a very important topic in of itself, and as such will be discussed in more detail in a section of its own. By doing so, the inner workings of deep neural networks and their usage shall be explained in that way as well.

Transformers

As of writing this thesis, Transformers are the latest key development in machine learning, especially when it comes to processing sequences of data. Their primary feature is a mechanism called „self-attention“, sometimes called „intra-attention“; which isn't a novel concept and has been used rather successfully in various (primarily text comprehension-related) tasks in the past, but the difference here is that the Transformer is the first transduction model that relies entirely on self-attention in computing representations of its input and output, without needing sequence-aligned RNNs or convolutional layers [33].

Self-attention basically allows the model to weigh different parts of a given sequence to understand how they relate to eachother, assigning scores to these relationships to help determine what part of the sequence is most important. In other words, it is capable of determining context; this is a big step forward to solving limitations in traditional models, like the vanishing gradient problem that previously made it hard to train very deep networks. Self-attention also makes it so the Transformer can consider all parts of a sequence at once, meaning Transformers allow for more computation to be parallelized than older models—like recurrent neural networks, in which data processing could slow down or become stuck in complex computations. By adopting this parallelized approach, Transformers can handle long sequences more efficiently, leading to their widespread adoption in various domains—primarily so text and audio processing, though it can process some types of images as well.

In summary, Transformers are a big deal because they can look at the whole sequence at once, not just one step at a time like older models. This makes them faster and often more accurate for a wide variety of tasks, and as such they had been the subject of thorough study and adaptation since their introduction in 2017. Thanks to the advantages the self-attention mechanism brings to the table, we're seeing Transformers used in things like language translation tools, text generation (GPT), language representation model (BERT) and so on. They are a major step forward in AI development, because they brought a new way to process sequences that is fast, efficient, and pretty adaptable to a lot of different tasks.

3.4 Learning Paradigms

So far, we've established some basics of machine learning models and their components, delving into the rabbit hole that is neural networks. However, there is a major part of the overall machine learning process we've simply skimmed over, with the subtle implication that the topic is of too large a scale to take apart efficiently in just a couple paragraphs—we speak of, of course, the „learning“ itself. It is high time we talked about the primary learning paradigms used in machine learning, what with how much they'd been teased, and unfold the tapestry of possibilities they offer.

What do these learning methods do, exactly? There is a subtle nuance in how one can perceive what the word *learning* actually encompasses, especially when mentioned in

certain scenarios or in combination with certain words. In his paper, Carbonell outlines two basic forms the word can take on: *knowledge acquisition* and *skill refinement* [5], where the former can be boiled down to acquiring new concepts, by which we mean studying and understanding them, to the point where this knowledge can be applied (usually in combination with other knowledge) in an effective manner. The latter one of these terms, on the other hand, refers to the incremental honing of both physical and mental skills via repetitive exercises.

If he had to look at both of these learning forms from the perspective of machine learning, then knowledge acquisition refers to algorithms working with data to learn and understand new concepts or patterns, which happens by picking out important pieces of information from the data and creating models that can apply what they've learned to make predictions or group things together. For instance, in image recognition, a machine learning program gets better at recognizing objects by learning from labeled pictures during its training. On the flip side, skill refinement could still be taken as we've described it before—that is, getting better at a task through continuous practice, or optimizing performance if you will. This tweaking and fine-tuning of the model and its parameters to improve the performance of a machine learning model—at least, until or unless overfitting occurs—is done very simply by running it several times.

In the earlier stages of machine learning development, a much bigger emphasis was put on the study of learning from the perspective of knowledge acquisition. However, nowadays, both forms are regarded as fundamental aspects of machine learning, and the prevalence of one over the other depends more so on the specific task and application, though usually it is regarded as not just recommended, but almost necessary to combine both.

Speaking of, we should really get to introducing some of the most used learning methods; aside from deep learning, that is, as that's one which emerged relatively recently and almost immediately spawned a plethora of research and sparked unprecedented amounts of interest, and as such we will dedicate an entire section of its own to it. Furthermore, as deep learning addresses slightly different issues, it isn't too uncommon to see its combination with the other methods explored in this section.

To commence, we will start with ***Supervised learning***. Supervised learning is essentially a guided type of learning in that an input dataset contains both input data and, mainly, corresponding labels. These labels are primarily what differentiates supervised learning from its counterpart, as it is them that the machine uses as a guide in learning to correctly predict said labels in the future, on data that doesn't have them. This is achieved through running the model on the labeled dataset in an attempt to learn inherent patterns, and the subsequent adjustment of the model's internal parameters (weights) to minimize errors contributing to incorrect predictions, for which it employs various optimization techniques/algorithms. This makes supervised learning perfect for tasks such as image classification; speaking of, there are two main task types supervised learning is used for, which would be *classification* and *regression* tasks. The difference between two is simple; let's say our input data are pictures of people. In classification tasks, the model focuses on learning to separate the input data into predefined classes or categories, so for example in this case, determining whether an image is a picture of a man or a woman. On the other hand, in regression tasks, the model's purpose is to predict a continuous numerical value based on input features—here, it can be predicting the age of the person in the image. No matter which type of task is being handled, though, the method's effectiveness does hinge on the quality of the labeled data.

Next up, we have ***Unsupervised learning*** which is the aforementioned counterpart to supervised learning. As you may have correctly assumed, its primary difference and selling point has to do with labels—or, more specifically, lack thereof. This may not sound very appealing at first glance, but the lack of a teacher isn't always a bad thing. You will definitely not achieve as good results in image classification as with supervised learning, but that really isn't what the unsupervised variant is for, anyway. Instead, it is used when one's goal is to uncover patterns, structures, or any kinds of inherent similarities within the received data; a task in which labels would achieve naught but muddying of the water. Forcing the algorithm to work without the presence of labels allows it to identify natural groupings within the data without any previous knowledge, and as such unsupervised learning excels in tasks like clustering, which is the practice of grouping data together based on shared characteristics or features, even those one could easily miss or dismiss upon first glance.

Now, we come to a type of learning we can most easily describe as a mixture of both of the so-far mentioned learning methods; hence why it is aptly called ***Semi-supervised learning***. In this case, we are given a mixture of labeled and unlabeled data (where usually the pool of unlabeled data is larger, often significantly so), and our goal, or rather the goal of the method, is to leverage both types of data to enhance the learning process by first using the labeled data to build a basic understanding of what we're dealing with, and then turning to the data that is unlabeled to further refine the model or discover additional patterns that may not have been apparent during the previous step. This way, one can efficiently use a small amount of data with labels along with a larger amount of data without, effectively making efficient use of available resources. Semi-supervised learning isn't as widely used as the learning methods it incorporates, but there is no doubt that it can prove useful in fields and situations where it isn't possible to get much data that is or can be labeled due to lacking the means, be it due to time or money constraints.

The last learning method we will introduce in this section is ***Reinforcement learning***, which is an approach inspired by the idea of reinforcement behaviour, or learning through trial and error. Simply put, the model learns by being run over and over whilst trying out different sequential decisions with the goal of finding the most optimal way to reach a desired outcome, learning from its mistakes in the process through environmental feedback in the form of either rewards or penalties. It doesn't truly have any input data to speak of, unlike the previous learning methods; instead, reinforcement learning operates solely based on interaction with the environment. This rids the method of a serious limitation, and as such, its applications are widespread, from things like training a machine to play computer games on a professional level to tasks as big as traffic control. These use-cases broaden even more when one takes into consideration the ability to combine this method with deep learning, into what we call *Deep reinforcement learning*, which applies deep learning to reinforcement learning problems.

3.4.1 Generalization, Overfitting and Underfitting

Training a machine learning model can throw some notably large snags under our feet, and we of course try our best to avoid tripping on them. One such problem stems from the fact that we train our model on training data, which is fine, but the broad goal of our training is not that the model works well on just said training data, but that it is capable of doing the same on data it hasn't seen before [14]. In other words, we want to know how well a model

can learn underlying patterns in the data it was given and transfer this knowledge into recognizing these patterns in new, unexplored data; we call this process ***Generalization***.

This is where we may encounter problems. If we overtrain the algorithm on the training set, we risk it capturing noise or irrelevant patterns which fits the model too closely to the training data, rendering it incapable of generalizing—we call this ***Overfitting***. The inverse can happen as well, meaning training a model with data of inadequate amount or quality so that it essentially learns very little, if anything at all. This is known as ***Underfitting***. Overfitting and underfitting will not be noticeable until tested on new data (a test consisting of examples the model hasn't yet seen), but by that time the trained model is already pretty much useless to us.

Of course, techniques to improve generalization by reducing overfitting and underfitting exist, but there isn't a way to completely erase the problems; at least, not as of now, if there ever will be. „Good models will probably still overfit at least a little bit, and if we try to eliminate overfitting, i.e. eliminate the gap between training and test error, we'll probably cripple our model so that it doesn't learn anything at all [14]."

3.5 Deep Learning

Deep learning is a learning paradigm used with increasing enthusiasm in modern times, particularly so thanks to advancements in hardware technologies (as has been more thoroughly explained before: 3.2). It presents itself as a specialized and advanced technique within the broader field of artificial intelligence, more specifically one that builds upon the foundation of neural networks. In this hierarchical dependency within the field of artificial intelligence (as illustrated in Figure 3.5), each concept builds upon the previous one to achieve increasingly sophisticated levels of intelligence and capability—and deep learning represents the pinnacle of this progression.

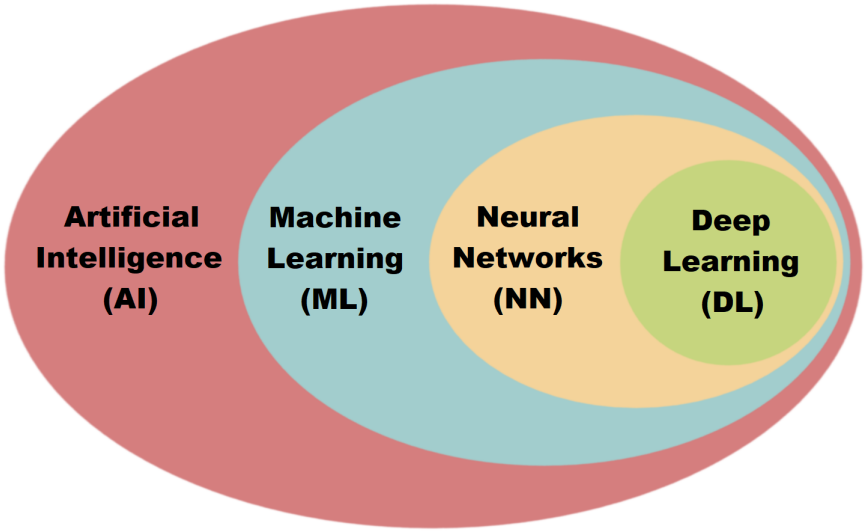

Figure 3.5: Hierarchical dependency within the field of artificial intelligence

As a part of a larger family of machine learning techniques that leverage data representations as a basis for the training of models, deep learning stands out primarily thanks to its usage of artificial neural networks as its primary computational network. They use neural

networks with many hidden layers (a fact based on which they are actually called „deep“), for their capability to learn complex mappings between inputs and outputs through a process known as backpropagation. This practice allows them to automatically—by which we mean without explicit human intervention or manual feature engineering—learn hierarchical representations of data. It was one of the first, and remains one of the most effective, learning techniques capable of training directly on raw data [10]. This, along with the ability to handle large, high-dimensional datasets with complex patterns, is deep learning's forte.

The advantages deep learning brings to the table are only expanded in combination with other learning methods; a practice that isn't all that uncommon. Particularly strong, and widely used in tasks regarding computer vision and image classification, is its pairing with convolutional neural networks, where deep learning enables the network to handle large datasets and learn from raw data directly. Among the wide variety of combinations also stand out two others: first, with recurrent neural networks, where deep learning is used to enrich the network's capability to capture long-range dependencies in sequential data, and second, the combination of deep learning with reinforcement learning, fixing problems which may arise when the latter has to learn complex behaviours autonomously, especially in environments where it's challenging or impractical to provide explicit guidance or supervision to the model.

Although the whole topic of deep learning may seem quite daunting to the newcomer, especially so if they're looking for a way to implement a model themselves, there exist deep learning libraries, such as TensorFlow and PyTorch, which provide pre-built functions and modules designed for efficient and relatively easy deployment of a deep neural network, including building the model and training it, making deep learning accessible even to people new to programming. Python, which is the language used by the aforementioned libraries, is also considered one of the easiest and most efficient programming languages one can start their programming journey with. Even better, PyTorch will be used in this thesis to build our models, meaning a basis for starting out will be provided as well.

Chapter 4

The Trifecta of Dynamic Systems Simulation - Conway's Game of Life and its Evolutionary Continuum to Advanced Iterations

Cellular automata and machine learning are both subjects holding a firm position at the forefront of computational exploration; even more so is their fusion, for the combination of cellular automata's relative conceptual simplicity at next to no expense to its capabilities as far as simulating complex systems goes, and the adaptive, data-driven prowess of machine learning, promises unprecedented potential in understanding and modeling intricate phenomena.

This convergence, however, fascinating as it is, has unfortunately not been explored very extensively as it isn't particularly easy to pick up. To be able to do so, we will progress through three individual cellular automata models of rising complexity so as to build our foundational understanding of the subject at hand—starting with the simple Conway's Game of Life, from whence we will advance on to SmoothLife, and afterwards, finally, we will wander into the intricate landscapes of Lenia. It is in the latter two that we will seek to incorporate machine learning into the ordeal. In doing so, we will present why these advanced versions of cellular automata were carefully chosen as the focus of investigation for this work.

The trifecta of computational models mentioned above starts with Conway's Game of Life, which is celebrated as one of the most important stepping stones in cellular automata history. The model's strict adherence to simple rules, binary state transitions and synchronous updates by the use of discrete time steps make its implementation relatively simple, whilst still proving it to be quite the efficient tool in simulating complex behaviours.

Building on the foundation laid down by Conway's Game of Life, SmoothLife serves as an organic evolution from the aforementioned model, augmenting it trifold; first by switching from the binary nature of states to continuous, second—as a corollary of the previous change—by utilizing a smoother transition function, and lastly by shaking up the standards established in neighbourhood definitions through the application of circular convolution over a neighbourhood in the form of a radius.

As part of this ongoing process, Lenia represents the cutting edge of this evolutionary progression (especially as far as simulating natural systems goes) as it implements all the

upgrades mentioned with SmoothLife, while also introducing asynchronous updates over continuous time, ridding us of the strictness of the discrete time step concept.

This chapter's goal is to provide an analysis of all three models and machine learning techniques, as well as present the results of experiments and observations found whilst working on their implementation and, primarily, integration of machine learning. Please note that any code samples shown henceforth are simply to serve as a showcase of the general way and steps to take in order to implement these models in PyTorch, mainly to provide basic knowledge of PyTorch functions together with insight of how the implementation itself was approached, and as such our goal won't be to create complete and runnable scripts. Should you want that, please refer to the fully working scripts enclosed with this thesis instead.

4.1 Python & PyTorch

The programming language chosen for the purposes of this thesis is Python, for which the reasons are aplenty. First and foremost, Python is generally agreed upon to be one of the easiest programming languages to pick up as a non-programmer or programmer alike, whilst offering a wide range of libraries handling some basic or even complex functionalities that can make our lives significantly easier and trivialize even seemingly challenging tasks. Which brings us to another point; one such library is PyTorch—an open-source deep learning framework containing tools specifically designed for handling various machine learning related tasks, from building a neural network model and training it, to optimizers, loss functions and supporting GPU acceleration. Among the most important tools PyTorch offers, and which we will be using a lot throughout our implementations, are `torch.tensor`, which represents multi-dimensional arrays or tensors, `torch.nn` for building neural networks and various layers, and `torch.fft`, which provides functions for fast Fourier transforms and signal processing.

To install Python as well as PyTorch, you can follow the official PyTorch guide; however, for brevity purposes, a tutorial as concise as possible will be provided in the next subsection. Keep in mind that the practices that will be used aren't future-proof, and as such may become obsolete in the future.

4.1.1 Installing Python and PyTorch

First task to tackle is installing Python on our computer, whilst keeping in mind that the ultimate goal is to use it with PyTorch. This means we have to ensure compatibility issues don't arise. For this, we can refer to PyTorch's Release Compatibility Matrix found at the start of their official RELEASE documentation. Once the preferred version of Python is determined, we can go and download it from the official Python website, install it as per the included instructions (it's advised to check the box to „Add Python to PATH", so you avoid having to do so afterwards) and then do the same with PyTorch. Once again, make sure both the Python and PyTorch versions are compatible. Whether they've been installed correctly can be confirmed using the following commands:

```
python --version

pip list
```

4.1.2 Installing CUDA (+ cuDNN)

Once Python and PyTorch are running, we can also take a look at CUDA (Compute Unified Device Architecture), which is a parallel computing platform enabling the use of NVIDIA GPUs for general-purpose computing tasks, meaning various computationally intensive workloads can be redirected from CPU to GPU. This accelerates the entire process exponentially, which comes in very handy in machine learning (along with others), particularly so for the training process as it includes extensive computation and as such can be quite lengthy. To install CUDA, one needs to first ensure that their GPU supports it, meaning that it is a compatible NVIDIA GPU. This can be done using a simple terminal command:

```
Windows:    nvidia-smi

Linux/Mac:  lspci | grep -i nvidia
```

Running one of the disclosed commands should return details about the NVIDIA GPU on our system; if it does, we can be sure that it is compatible, and may proceed to downloading the NVIDIA CUDA Toolkit and installing it. Once again, we have to make sure that we get the correct CUDA version, compatible with our PyTorch setup. Should an older version of CUDA be needed, we can refer to the archive of previous CUDA releases instead. Go through the regular setup, then check whether it had been installed correctly by running a simple Python script:

```
import torch

#check if CUDA is available
if torch.cuda.is_available():
    print("CUDA is available.")
else:
    print("CUDA is not available.")
```

Last but not least, we have the option to also get cuDNN (CUDA Deep Neural Network), which is a library designed to optimize and further accelearte the deep learning-oriented computing tasks handled by CUDA, by introducing more efficient implementations for its common neural network operations, primarily so convolutions and activation functions. cuDNN is entirely optional, but it is nevertheless a valuable asset to add to our repertoire. Should you choose to get it as well, simply go to the cuDNN download page and proceed from there, similarly to the previous CUDA installation. Once again, a successful cuDNN installation can be confirmed by a simple Python script:

```
import torch

#check if cuDNN is available
if torch.backends.cudnn.is_available:
    print("cuDNN is available.")
else:
    print("cuDNN is not available.")
```

4.2 Conway's Game of Life

Often presented as one of the first important milestones in cellular automata development, Conway's Game of Life aims to simulate emergent complexity similar to that which can observed in biological systems. For our purposes, this automaton will serve as the foundational basis of information upon which the other two models will be built.

4.2.1 Basics

Conway's Game of Life is a two-dimensional cellular automaton with binary cell states and a condition-based rule set. This means that it operates on a two-dimensional grid populated by cells which, at all times, can only be in either one of two states—dead (respectively labeled '0') and alive ('1'). The system evolves over discrete time steps, and evolution of cells is based on a rule set consisting of four fundamental conditional rules:

- If a live cell has two or three live neighbours, it stays alive.
- If a live cell has fewer than two live neighbours, it dies (underpopulation).
- If a live cell has more than three live neighbours, it dies (overpopulation).
- If a dead cell has exactly three live neighbours, it becomes a live cell (reproduction).

In terms of neighbourhood, the model isn't strict as to the preferred configuration, but as the Moore neighbourhood allows for interactions with a large selection of cells and ergo introduces more options, it should come as no surprise that it tends to result in more complex and varied patterns. For this very reason, we will choose to go with this configuration.

4.2.2 Patterns

Conway's Game of Life is well-known for producing many different types of patterns. The emergence of these patterns is a result of the interplay between the initial configuration of cells and the application of the four fundamental rules governing the evolution of the cellular automaton. We classify the patterns based on their behaviour into four main categories:

Still lifes

Still lifes are stable configurations that do not change from generation to generation. Basically, they are a pattern which remains static in between time steps, at least unless another pattern collides with them. There is a handful of Still life patterns which occur in Conway's Game of Life, but the most common include the Block, the Tub, the Beehive, the Loaf and the Boat.

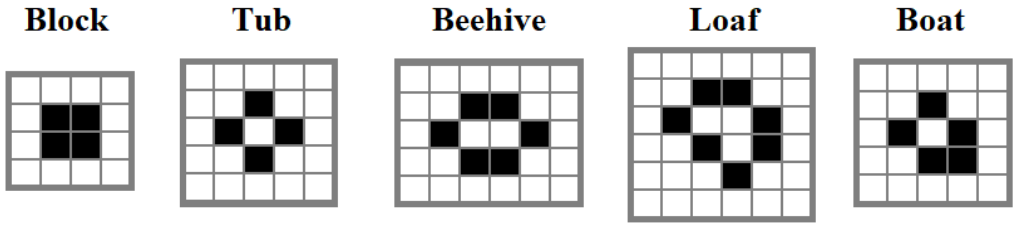

Figure 4.1: GoL: Still life patterns examples

Oscillators

Oscillators are patterns that repeat their configuration after a certain number of generations, while retaining their initial position. They showcase the periodic nature of the Game of Life dynamics. Examples of oscillators include the Blinker and the Beacon, which oscillate between two states, and the Pulsar, which oscillates between three states.

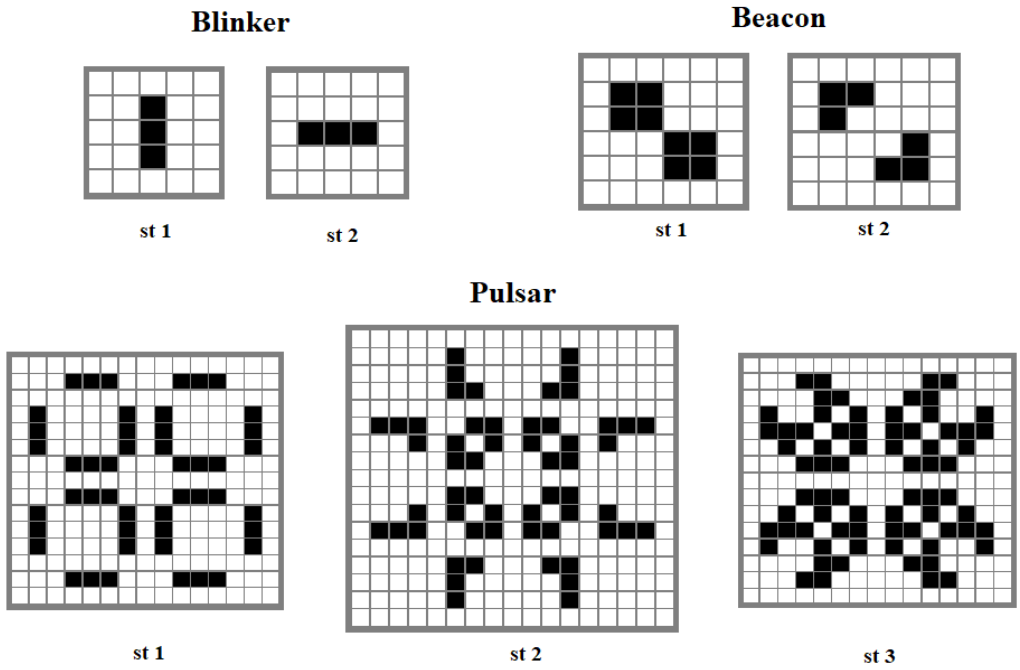

Figure 4.2: GoL: Oscillator patterns examples

Spaceships

One of the most intriguing aspects of Conway's Game of Life is the existence of moving patterns. Gliders are configurations that translate themselves across the grid while maintaining their overall shape. These dynamic patterns illustrate the potential for complexity and movement within the cellular automaton. While these patterns differ drastically in terms of size and number of oscillations, we will showcase them on the most basic of them—the Glider.

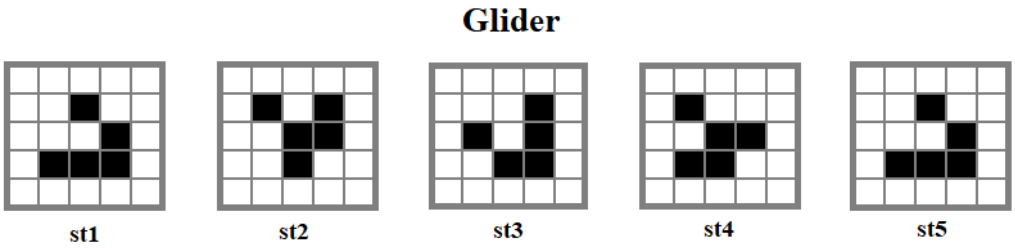

Figure 4.3: GoL: Spaceships example - glider

In summary, the richness of Conway's Game of Life lies in its ability to generate an astonishing variety of patterns, ranging from static and stable to dynamic and complex. These patterns not only provide aesthetic appeal but also serve as a testament to the intricate and unpredictable nature of emergent phenomena in cellular automata.

4.2.3 Simple Implementation

Implementing Conway's Game of Life is relatively simple, but a necessary stepping stone if one wants to delve into programming more progressive types of simulations. This subsection provides a basic analysis of the process, meaning building the simulation components, implementing Conway's Game of Life rules and incorporating everything into the forward pass, all using primarily **PyTorch** functions.

Definitions

First, we need to define our basic components 2.2.1; that means we have to: create a grid of cells, set said cells to binary states, and create a neighbourhood (in our case Moore).

```
#create a randomly populated grid of cells, where low = smallest generated number = 0,
#high = one above the highest generated number = 2, and (height, width) are dimensions
#of the grid
grid = torch.randint(low, high, (height, width))

#create a convolution filter (Moore neighbourhood)
Moore_filter = torch.tensor([[1, 1, 1],
                             [1, 0, 1],
                             [1, 1, 1]])
```

Forward Pass

Next, we have to implement the forward pass, which will be called at each time step. Its purpose is to serve as a function that can be called on each given time step to update the state of each cell based on the number of its alive neighbours, as per the rules of Conway's Game of Life.

We will use the previously defined grid and convolution filter to feed into PyTorch's convolution function. Convolution basically takes in the original grid and a filter, then uses element-wise multiplication followed by summation to produce an output; which is perfect in our case, as this means convolution will basically create a new grid where each cell will contain a value corresponding to the number of alive neighbours in the kernel (our Moore neighbourhood) around that cell.

```
#obtain count of alive neighbours for each individual cell using convolution
alive_neighbours_cnt = torch.nn.functional.conv2d(grid.unsqueeze(0),
                                                  Moore_filter, padding=1)
```

Once we have obtained the count of alive neighbours, nothing's stopping us from implementing the Conway's Game of Life rules 4.2.1. Note that, due to the nature of how we will update the grid (creating a new grid to compute on), we will only really need to compute on old grid and fill the new one with cells which should stay/become alive. This means that, basically, we will only need to implement rules #1 and #4).

```
#convert self.grid into a new tensor where each value is represented as a byte to
#ensure that all elements in self.grid strictly conform to being 0 or 1 to avoid
#unexpected behavior
alive = self.grid.byte()

#implement Rule #1 (so, check if a cell is alive and has 2 or 3 alive neighbours)
new_grid = (alive & (torch.eq(alive_neighbours_cnt.squeeze(), 2) |
                     torch.eq(alive_neighbours_cnt.squeeze(), 3)))
#implement Rule #4 (so, check if a cell is dead and has 3 alive neighbours)
new_grid += (~alive & torch.eq(alive_neighbours_cnt.squeeze(), 3))

#update grid, squeeze() it to remove any added dimensions (make it a 3x3 tensor again)
grid = new_grid.clone().squeeze()
```

4.3 SmoothLife

SmoothLife builds upon the foundation laid by Conway's Game of Life and seeks to push it to greater heights. Its idea was first proposed by Stephan Rafler in his paper titled „Generalization of Conway's "Game of Life" to a continuous domain - SmoothLife“ [25], and it is precisely this paper we will use as a base for our study and, consequently, implementation.

4.3.1 Advancements to Conway's Game of Life

SmoothLife takes all that made Conway's Game of Life, seeking to create a more continuous and therefore natural experience. It does so primarily through the combination of three things:

1. By getting rid of the binary nature of states, instead calculating state as a float valued between 0.0 and 1.0.

2. As a consequence of the previous change, by introducing a smoother transition function, which only serves to further augment the visual appeal and complexity of the simulation.

3. By reforming the concept of a neighbourhood, introducing the application of circular convolution over a certain radius around each cell.

Additionally, SmoothLife incorporates a variety of parameters to control aspects like birth, survival, and range, providing a more flexible framework for exploring emergent behaviours.

4.3.2 Mathematical Background

The previous points make it quite clear that SmoothLife requires a different strategy from the mathematical standpoint; while it still utilizes a condition-based rule set, it requires a more nuanced and continuous approach due to the nature of a continuous state space.

Inner and Outer Fillings

In his paper, Rafler describes the mathematical backbone of his discovery as thus: the proposed model allows cells to have a finite size, typically circular (disk-shaped), instead of infinitesimal points, which then allows us to determine the next state of the cell by „the filling of the circle around that point [25]“ as well as the (ring-shaped) neighbourhood. We can think of them as one big circle, which is divided into two key quantities: the inner circle (cell's filling), and the outer ring (neighbourhood's filling). The typical neighbourhood configuration in SmoothLife involves setting the outer radius to three times the inner radius, creating a 3-cell-wide neighborhood ring.

- **Inner Filling (m)**: The inner filling represents the filling of a circular cell, calculated as the integral of the state function $f(\vec{x}, t)$ over a disk with radius r_i:

$$m = \frac{1}{M} \int_{|\vec{u}|<r_i} f(\vec{x} + \vec{u}, t)\, d\vec{u}$$

  where M is a normalization factor ensuring that m ranges from 0 to 1.

- **Outer Filling (n)**: The outer filling represents the filling of a ring-shaped neighborhood, calculated as the integral of the state function over a ring with inner radius r_i and outer radius r_a:

$$n = \frac{1}{N} \int_{r_i<|\vec{u}|<r_a} f(\vec{x} + \vec{u}, t)\, d\vec{u}$$

  Similarly, N is a normalization factor that keeps n within the range [0, 1].

Transition Function

In the original Game of Life, the state of a cell in the next time-step depends on its current state and the number of live neighbors. In SmoothLife, the state of a cell at position $\vec{x}$ in the next time-step is determined by a transition function $s(n, m)$. This function takes the inner and outer fillings and maps them to a new state, defined on the interval $[0, 1] \times [0, 1]$ with output also in the range [0, 1].

4.3.3 Implementation

Compared to Conway's Game of Life, SmoothLife obviously brings with added challenges; however, if we adhere to the rules set out by Rafler and make use of the tools that were presented to us, our job will be much simpler.

Definitions

We will, again, need to define the most basic structures and values for our model. Some we will have to determine ourselves, while others can be directly taken from our source material,

at the end of Rafler's paper; namely, the birth and death intervals, sigmoid widths, and inner and outer radii.

```
#set the dimensions of the grid
height = 50
width = 50
#create a grid---it will need a more complex way of initialization, more on that later
grid = torch.zeros((height, width))

#values taken out of Rafler's paper
birth_interval = (0.28, 0.37)
death_interval = (0.27, 0.45)
sigmoid_widths = (0.03, 0.15)
inner_radius = 6.0
outer_radius_multiplier = 3.0
```

Grid initialization

First, we have to initialize a grid. However, this time, simply using `torch.rand` or any of its variants won't do; as already mentioned, cells in SmoothLife aren't simply a point on the grid, but instead they are presumed to be of a circular shape. As such, we have to take the idea of what `torch.rand` does and mold it to our situation. We can do so by setting a radius for the cells, and then create random „patches" in the general shape of these cells on the grid. In other words, we have to fill the grid as we did previously with Game of Life, but instead of filling one point at a time, we will be filling a bigger „square" each time. This code showcases the general idea of what we're trying to achieve by this:

```
#set the cell radius (example)
inner_radius = 6

grid_area = height * width
#inner radius = size of patch, so for example inner radius = 6 is a 6x6 patch,
#meaning the area of the patch is 6^2
patch_area = inner_radius ** 2
#cover 30% of the grid space
coverage_goal = 0.3
#calculate how many patches of 1s to create
estimated_patches_cnt = (grid_area / patch_area) * coverage_goal
#round the final number of patches up
patches_cnt = math.ceil(estimated_patches_cnt)

for _ in range(patches_cnt):
    #get a random starting point for the patch
    start_row = torch.randint(0, height - int(inner_radius), (1,))
    start_col = torch.randint(0, width - int(inner_radius), (1,))
    #calculate the ending point for the patch
    end_row = start_row + inner_radius
    end_col = start_col + inner_radius

    #fill the patch on the grid
    grid[start_row:end_row, start_col:end_col] = 1
```

Convolution

Due to the direction in which we're taking our simulation, by which we primarily refer to how forward pass will be defined, we actually need to calculate two separate representations of the smoothed inner filling (`real_M`) and outer filling (`real_N`) in the spatial domain. This raises the question of the best method to realise this, and whether it can be achieved using already available functions. Personal experimentation proved that things weren't so simple, and the readily available functionalities were hard to mold into working for our purpose; that is, until stumbling upon an interesting article detailing a way to achieve precisely what we need [22]. As such, a new plan was devised, and the result would become our own personalised convolution function designed specifically for our purposes. First, let us define a function that will create a circular mask.

```
def generate_circle(size, radius):
    """
    Generate a circular mask with a given radius
    """
       #calculate coordinates
       y_coord, x_coord = torch.meshgrid(torch.arange(size[0]), torch.arange(size[1]),
       indexing='ij')
       #compute distances from the center
       distances = torch.sqrt((x_coord - size[1] / 2) ** 2 +
                              (y_coord - size[0] / 2) ** 2)

       #create circle with 1s inside the radius and 0s outside
       circle = (distances <= radius).float()

       #roll the circle, centering it at extremes (this is better for convolution)
       circle = torch.roll(circle, size[0] // 2, dims=0)
       circle = torch.roll(circle, size[1] // 2, dims=1)

       #result will be a 2D grid with a circle drawn onto it with 1s
       return circle
```

Next, we will use this function to craft our shapes. The idea is to create two circles; one bigger and another smaller one. The smaller one will be our inner filling, and then, by subtracting it from the bigger one, we will also get an area equal to our ring. To prepare both areas for convolution, we will also transform them from the spatial domain to the frequency domain using Fast Fourier Transform. To do so, we will first have to normalize the circles by dividing them by their respective sums using `torch.sum`, to ensure the sum of all elements in each circle is 1. This is important for consistency while using operations like convolution. We also can't forget to cut out the „ring“ shape for our outer area by subtracting the smaller circle from the bigger circle. Last but not least, we finally apply Fast Fourier Transform.

```
#create two circles; the outer and inner one
smaller_circle = generate_circle(size, inner_radius)
bigger_circle = generate_circle(size, inner_radius * outer_radius_multiplier)

#create the final areas
#normalize the circles using torch.sum, then apply fft to transform them into the
#frequency domain. Create the "ring" outer area by subtracting the smaller circle
#from the bigger one
inner_area = torch.fft.fft2(smaller_circle / torch.sum(smaller_circle))
outer_area = torch.fft.fft2((bigger_circle - smaller_circle) /
                            torch.sum(bigger_circle - smaller_circle))
```

Forward Pass

Here, with our convolution ready, we can keep strictly to our blueprint, meaning that we can define the transition function as:

$$s(n, m) = \sigma_2(n, \sigma_m(b_1, d_1, m), \sigma_m(b_2, d_2, m)) \tag{4.1}$$

Where the sigmoid functions (for smoothing) are:

$$\sigma_1(x, a) = \frac{1}{1 + \exp(-(x - a)4/\alpha)} \tag{4.2}$$

$$\sigma_2(x, a, b) = \sigma_1(x, a)\ (1 - \sigma_1(x, b)) \tag{4.3}$$

$$\sigma_m(x, y, m) = x(1 - \sigma_1(m, 0.5)) + y\,\sigma_1(m, 0.5) \tag{4.4}$$

In short, σ_1 calculates the current cell's degree of being alive (creates smooth transition from 0 to 1), which helps avoid abrupt changes (anti-aliasing). σ_m determines the thresholds through linear interpolation between `x` and `y`, controlled by `m`. σ_2 returns a high output if `x` is between `a` and `b`, and is used by the main function $s(n, m)$ to determine whether `n` is between two thresholds created by σ_m. Programatically, we can simply copy what's laying in front of us:

```
def s(n, m):
    new_aliveness = self.sigma_2(n, self.sigma_m(self.birth_low, self.death_low, m),
                                    self.sigma_m(self.birth_high, self.death_high, m))

    #we clamp the result here to make sure values stay between 0 and 1
    return torch.clamp(new_aliveness, 0, 1)

def sigma_1(x, a, filling):
    #x == m | n
    #a == 0.5 | thresholds (sigma_m results)
    #filling == self.N | self.M
    return (1.0 / (1.0 + torch.exp(-4.0 / filling * (x - a))))

def sigma_2(x, a, b):
    #x == n
    #a, b == thresholds (sigma_m results)
    return (self.sigma_1(x, a, self.inner_filling) *
           (1.0 - self.sigma_1(x, b, self.inner_filling)))

def sigma_m(x, y, m):
    #x == self.birth_high|low
    #y == self.death_high|low
    #m == m
    return  (x * (1.0 - self.sigma_1(m, 0.5, self.outer_filling)) +
             y * self.sigma_1(m, 0.5, self.outer_filling))
```

Past this point, the forward pass is relatively simple. We first convert our grid from the spatial domain to the frequency domain through Fast Fourier Transform, in order to perform convolution in a more computationally efficient manner. Then do the convolution itself (by simply multiplying converted grid with the previously prepared convolution masks, the „circle“ and „ring“ ones), using Inverse Fast Fourier Transform to convert our grid back into the spatial domain and `torch.real` to extract the real part. Once that step is done, all that's left to do is to simply apply the SmoothLife rules with smoothing.

```
#convert grid from the spatial domain to the frequency domain for convolution
grid_transformed = torch.fft.fft2(grid)
#do convolution (simply through multiplication), using Inverse Fast Fourier Transform
#to convert grid back into the spatial domain and torch.real to extract the real part
M_real = torch.real(torch.fft.ifft2(grid_transformed * inner_area))
N_real = torch.real(torch.fft.ifft2(grid_transformed * outer_area))

#apply SmoothLife rules with sigmoid smoothing
grid = s(N_real, M_real)
```

4.4 Lenia

Let me preface this section by relaying one important message—compared to the previous models, Lenia is somewhat hard to fully grasp. On paper, this model is relatively simple if we already possess knowledge of the inner workings delegating SmoothLife's functionality; however, the truth is, the changes Lenia introduces into the mix are severe enough to warrant adjustments in nearly every part of the overall process, and fully grasping the mathematical background is quite the feat in of itself. Luckily, all of these are fully detailed in research papers—by people behind Lenia as well as others—that can be found on Lenia's

official site. For our purposes, we will mainly work with the original paper by Bert Wang-Chak Chan [7]. It is still a lot of information, but an effort will be expended to try making it as digestible as possible.

4.4.1 Advancements to SmoothLife

SmoothLife is already capable of wonderful things; capturing complex behaviours and dynamics, studying emergent properties, all whilst allowing for more fluid and organic-like patterns compared to traditional cellular automata. That said, however, it isn't without its constraints. Lenia strives to rid us of these constraints, providing more options for tuning and, in turn, makes it more adept at simulating behaviours which are more complex and life-like.

1. SmoothLife's radial pattern for its kernel is rather straightforward and as such nigh impossible to customize. Lenia uses a more generalised structure, making it more adjustable.

2. SmoothLife generally uses smaller amount of parameters to decide how cell states change, while the way Lenia's growth and convolution processes are set up allows for more fine-tuning.

3. In regards to the previous point, SmoothLife's basic setup might also result in a fewer types of patterns and behaviours, compared to what Lenia's flexible approach has to offer.

4. Where SmoothLife usually uses a simple transformation function only utilizing a single sigmoid curve, Lenia's transformation function makes use of Lenia's flexibility and, though harder to design, enables more intricate patterns and dynamics through this added complexity.

4.4.2 Mathematical Background

A comprehensive mathematical definition of Lenia is provided in the original paper by Bert Wang-Chak Chan [7]. Lenia's mathematical structure revolves around a combination of convolution, kernel functions, and transformation functions, set in a continuous state space; here's a high-level summary of the primary mathematical components, as per the aforementioned paper:

Discrete and Continuous Lenia

Lenia has two primary forms: discrete Lenia and continuous Lenia. Discrete Lenia enhances the concepts of traditional cellular automata, by extending state values from $x \in \{0, 1\}$ to $x \in < 0, P >, P \in \mathbb{Z}$, as well as describing the neighbourhood as a disk-shaped region of size defined by radius R.

Continuous Lenia is the hypothetical limit where the discrete values and discrete space of discrete Lenia become continuous, allowing cells to have any value between 0 and 1 and to exist in a continuous space.

Neighbourhood

Lenia's neighbourhood is defined as a set of cells within a certain radius R from a central cell, mathematically described as:

$$\mathcal{N} = \{\mathbf{x} \in \mathcal{L} : \|\mathbf{x}\|_2 \leq R\} \tag{4.5}$$

Convolution

Here, the formulas vary for discrete and continuous Lenia. For the former, the convolution at a point x to yield the *potential distribution* U^t is defined as:

$$\mathbf{U}^t(\mathbf{x}) = \mathbf{K} * \mathbf{A}^t(\mathbf{x}) = \sum_{n \in \mathcal{N}} \mathbf{K}(\mathbf{n}) \mathbf{A}^t(\mathbf{x} + \mathbf{n}) \, \Delta x^2 \tag{4.6}$$

Whereas in the latter case, the convolution is given by:

$$\mathbf{U}^t(\mathbf{x}) = \mathbf{K} * \mathbf{A}^t(\mathbf{x}) = \int_{n \in \mathcal{N}} \mathbf{K}(\mathbf{n}) \mathbf{A}^t(\mathbf{x} + \mathbf{n}) \, \mathrm{d}x^2 \tag{4.7}$$

where $\mathbf{A}^t(x)$ represents the state of the grid at time t and $\mathbf{K}(n)$ is the kernel function that defines the influence of neighbors.

Kernel Structure

The kernel $\mathbf{K}$ is constructed by *kernel core* $K_C : [0, 1] \to [0, 1]$ which determines its detailed „texture“, and *kernel shell* $K_S : [0, 1] \to [0, 1]$ which determines its overall „skeleton“ [7].„

In other words, kernel core defines the basic shape of the kernel. It is sually a unimodal function with a peak in the middle, and as such can take on various forms, i.e.:

$$K_C(r) = \begin{cases} \exp\left(\alpha - \dfrac{\alpha}{4r(1-r)}\right) & \text{exponential, } \alpha = 4 \\ (4r(1-r))^{\alpha} & \text{polynomial, } \alpha = 4 \\ \mathbf{1}_{[\frac{1}{4}, \frac{3}{4}]}(r) & \text{rectangular} \\ \dots & \text{or others} \end{cases} \tag{4.8}$$

The kernel shell creates concentric rings based on the core, allowing for more complex structures. It is defined as:

$$K_S(r; \beta) = \beta_{\lfloor Br \rfloor} K_C(Br \bmod 1) \tag{4.9}$$

At the end, the kernel also undergoes normalization to ensure the convolution results remain within [0, 1]:

$$\mathbf{K}(\mathbf{n}) = \frac{K_S(\|\mathbf{n}\|_2)}{|K_S|} \tag{4.10}$$

Where the normalization factor $|\mathbf{K_S}|$ is given as $|K_S| = \sum_{\mathcal{N}} K_S \, \Delta x^2$ for discrete Lenia, or $\int_{\mathcal{N}} K_S \, \mathrm{d}x^2$ for continuous Lenia.

Growth Mapping

The growth mapping function determines how the potential $\mathbf{U}^t(x)$ is transformed into a growth value. The function can take various forms:

$$G(u;\mu,\sigma) = \begin{cases} 2\ \exp\left(-\dfrac{(u-\mu)^2}{2\sigma^2}\right) - 1 & \text{exponential} \\ 2\ \mathbf{1}_{[\mu\pm 3\sigma]}(u)\left(1 - \dfrac{(u-\mu)^2}{9\sigma^2}\right)^{\alpha} - 1 & \text{polynomial, } \alpha = 4 \\ 2\ \mathbf{1}_{[\mu\pm\sigma]}(u) - 1 & \text{rectangular} \\ \dots & \text{or others} \end{cases} \tag{4.11}$$

Update Rule

New state is calculated by adding a fraction Δt ($\mathrm{d}t$ in CL) of the growth and clipping to ensure it remains within the range [0, 1]:

$$\mathbf{A}^{t+\Delta t}(\mathbf{x}) = \left[\mathbf{A}^t(\mathbf{x}) + \Delta t\ \mathbf{G}^t(\mathbf{x})\right]_0^1 \tag{4.12}$$

4.4.3 Implementation

The Leania implementation described in this section will be based on the existing implementations [6]. For the purpose of this section, let us define them for the most basic application—with a randomized grid—which will simply showcase continuous space influence.

Definitions

First, let's define the basic Lenia parameters. These parameters are highly malleable, and their values can be changed for different applications and outcomes.

```
#set the dimensions of the grid
height = 64
width = 64
#calculate middle of the grid, rounded down. This will come in handy later
mid = width // 2
#create grid, a simply randomized one will do
grid = torch.rand(height, width)

#scale is kind of useless for us as of now, but let's keep it in for the future
scale = 1
#define radius of kernel
kernel_radius = 20 * scale
#frequency of cell updates
update_frequency = 10
#define the center of the Gaussian distribution = where the kernel or growth '
#function has its maximum value
mean = 0.135
#standard deviation, or spread/width of the Gaussian curve
#(smaller spread value = narrower peak)
spread = 0.015
```

Now we will also define some functions and steps that we will generally keep as they are. These specify main functionalities and objectives of Lenia's framework.

```
#function to create a kernel for convolution where kernel has a smooth distribution,
#returns a Gaussian (or normal) distribution, centered at "m", with a spread defined
#by "s"
def bell(x, m, s):
    return np.exp(-((x - m) / s) ** 2 / 2)

#determines how cells in the cellular automaton should grow or shrink based on the
#"potential" derived from convolution with the kernel
def growth(U):
    return bell(U, mean, spread) * 2 - 1

#now, define the kernel and prepare it for convolution by transforming it into
#the frequency domain using the Fast Fourier Transform
def create_kernel():
    #create two 2D grid tensors representing the x and y coordinates
    x_indices, y_indices = torch.meshgrid(torch.arange(-mid, mid),
                                          torch.arange(-mid, mid), indexing='ij')
    #compute the Euclidean distance from the origin for each point on the grid
    distances = torch.sqrt(x_indices**2 + y_indices**2) / kernel_radius
    #create the kernel
    kernel = (distances < 1) * bell(distances, 0.5, 0.15)

    #pre-calculate FFT of kernel
    fK = torch.fft.fftn(torch.fft.fftshift(kernel.clone().detach() /
                                           torch.sum(kernel.clone().detach())))

    #return the kernel
    return fK
```

Forward Pass

We have prepared most of what we need to execute forward pass, and as such will just have to utilize it. We will also incorporate asynchronicity here, in the form of random independent scheme 2.4.3 (we will create a mask where each cell will have a certain chance to be updated).

```
#call the previously defined function to create the kernel
fK = create_kernel()

#randomly determine whether each cell should be updated or not
#cells with values > 0.3 will be updated (30% chance)
update_mask = torch.rand_like(grid) > 0.3

#compute local neighborhood using convolution
#first compute the FFT of grid, then multiply it with pre-prepared kernel in the
#frequency domain and transform it back to the spatial domain using inverse FFT
U_fft = torch.fft.fftn(grid)
U_fft = torch.fft.ifftn(U_fft * fK)
#now get the real part
U = torch.real(U_fft)

#apply rules, update only the cells where update_mask is True
grid[update_mask] = torch.clamp(grid[update_mask] + 1 /
                                update_frequency * growth(U)[update_mask], 0, 1)
```

4.5 Machine Learning

By this point, we have successfully defined our foundational structure for both the mathematical and programmatic aspects of all three models, through which we can implement them without any bigger problems. Our last step is, hence, to finally also delve into machine learning and its applications on the implemented models—more specifically the latter two. In this section, we will implement a machine learning model, train it on a data set, then employ and integrate into the aforementioned cellular automata models.

4.5.1 Implementing a Neural Network model

For our machine learning model, we will be creating a simple feed-forward neural network with one hidden layer using PyTorch's `nn` module. This neural network will be used to predict future state of the simulation based on an input, after which this functionality will be applied to try and prevent the simulation from dying prematurely, or simply to make it more "alive„ so to speak.

Defining the Predictor

While defining the Predictor neural network, we will mainly be working with the official PyTorch tutorial for its `nn` module [8] and modifying it. We will be using the ReLU activation function over the other options for the single hidden layer to introduce nonlinearity whilst maintaining faster computation due to ReLU's simplicity.

```python
import torch
import torch.nn as nn

#future state predictor neural network model
class Future_state_predictor(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Future_state_predictor, self).__init__()
        #define the input layer, with the expected size of data input as well as
        #the size the hidden layer will expect as its input (so, input layer's output)
        self.fc1 = nn.Linear(input_size, hidden_size)
        #define the hidden layers with a ReLU activation function
        self.fc2 = nn.ReLU()
        #define the output layer, with the expected size of input as well as the size
        #of the data output
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        #simply pass the data input first through the input layer,
        x = self.fc1(x)
        #then the hidden layer (applying the ReLU activation function),
        x = self.fc2(x)
        #and lastly the output layer
        x = self.fc3(x)

        return x
```

Training the Predictor

Next step is to train our neural network on actual data. For this step, let's create a new file and implement our training there. Let us also assume that our cellular automata models (SmoothLife and Lenia) are neatly packed in their own respective Python classes, called `CL_SmoothLife` and `CL_Lenia` respectively, and both of them contain a function called `forward` which manages their forward pass functionality. We will need to take these steps:

1. load the cellular automata model
2. define the predictor, optimizer and loss function
3. generate training data (inputs and targets) from the respective simulation
4. move model and data to GPU, if possible
5. stack train inputs and targets to lighten load
6. train the model (processing the data in mini-batches for faster processing)
7. save the trained model

```
#load the cellular automata model (SmoothLife, for example)
model = CL_SmoothLife()

#define the predictor, as well as optimizer and loss function
predictor = Future_state_predictor(grid_size, 64, grid_size)
optimizer = optim.Adam(predictor.parameters(), lr=0.001)
criterion = nn.MSELoss()

#generate training data from the respective simulation
num_samples = 2000 #number of samples
train_inputs = []
train_targets = []
for h in range(num_samples):
    current_state = model.grid.clone().detach()
    model.grid = model.forward()
    future_state = model.grid.clone().detach()

    train_inputs.append(current_state.view(1, -1))
    train_targets.append(future_state.view(1, -1))

#move model and data to GPU, if possible
if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

#stack train inputs and targets (to lighten load)
train_inputs = torch.stack(train_inputs).to(device)
train_targets = torch.stack(train_targets).to(device)

#train the model for 300 epochs
for epoch in range(300):
    optimizer.zero_grad()

    #process data in mini-batches of 20 (for faster processing)
    for i in range(0, len(train_inputs), 20):
        batch_inputs = train_inputs[i:i + 20].view(20, -1)
        batch_targets = train_targets[i:i + 20].view(20, -1)

        #run inputs through neural network
        outputs = predictor(batch_inputs)
        #calculate current loss
        loss = criterion(outputs, batch_targets)
        loss.backward()

    #if loss value went up, we're encountered overfitting ==> stop training
    if old_loss < loss.item() and old_loss != 0.0:
        break
    old_loss = loss.item()

    #perform a single optimization step
    optimizer.step()

#save the trained model
torch.save(predictor.state_dict(), "SmoothLife_predictor.pth")
```

Loading the Predictor

While loading our saved predictor from inside the cellular automata implementations, we mustn't forget to set it to "testing„ mode using `.eval()` (the default mode is "training„).

```
#initialize predictor
predictor = Future_state_predictor(input_size, 64, output_size)
#load pre-trained predictor
if torch.cuda.is_available():
    predictor.load_state_dict(torch.load("SmoothLife_predictor.pth",
                                         map_location=torch.device("cuda")))
else:
    predictor.load_state_dict(torch.load("SmoothLife_predictor.pth",
                                         map_location=torch.device("cpu")))
#put smooth predictor into eval mode
predictor.eval()

#load the predicted state using our predictor
#we use no_grad to disable gradient calculation, so we don't mess up backward
   propagation
with torch.no_grad():
    predicted_state = predictor(grid.view(-1)).view(height, width)
```

Using the Predictor, Experimenting & Results

Now that we have our predictor loaded, it is time to put it to use. How do we go about doing so, though? Past this point lies a collection of various personal attempts in using the trained neural network model, and their results.

At first, naive thought led to a belief that, after a step was made, one could simply combine the resulting state of the grid with the predicted state and as such have it influence the simulation's growth positively. This attempt included setting a value we will call `alpha` determining how big an influence would the predicted state have on the outcome of the current step. The thought wasn't entirely wrong, but execution was lacking severely, as shown in the result portrayed in figures 4.4 and 4.5, where in all cases the simulation would very quickly start to deteriorate before eventually dying entirely. No amount of alpha adjustment seemed to yield better results.

```
#increase the alpha (weight) to increase the influence of the predictor
alpha = 0.8
#combine original state with predicted state
grid = alpha * grid + (1 - alpha) * predicted_state
```

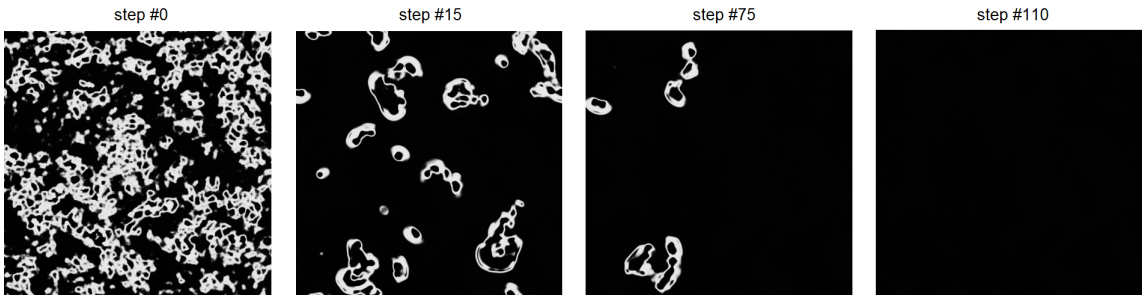

Figure 4.4: SmoothLife results of a simple predicted state influence with a static alpha.

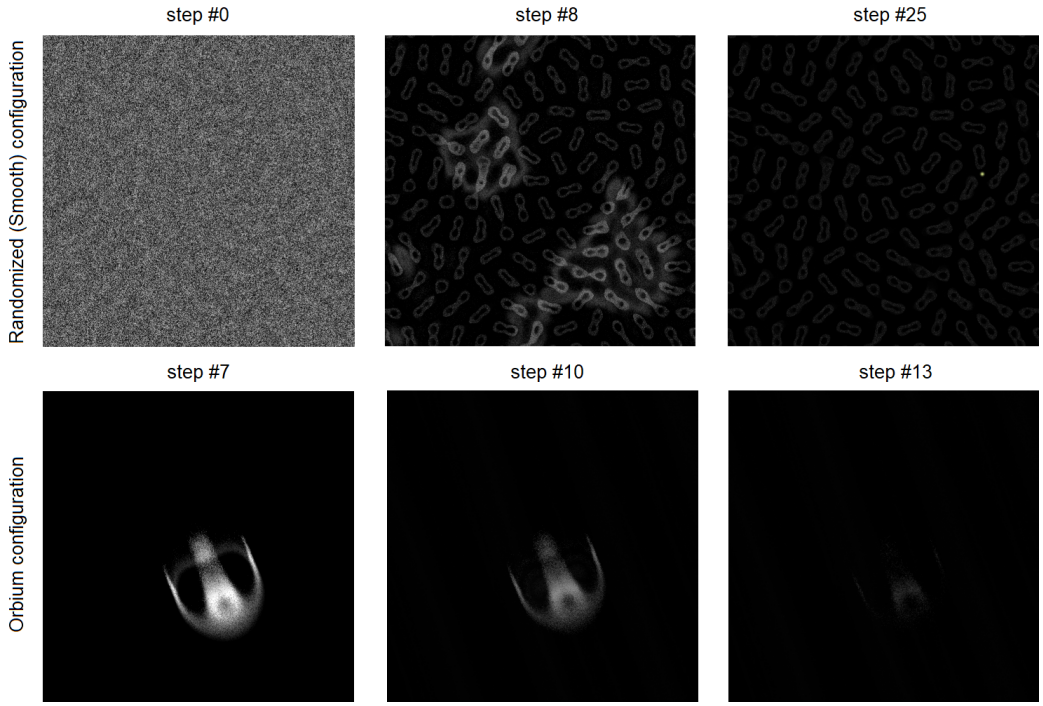

Figure 4.5: Lenia results of a simple predicted state influence with a static alpha.

Afterwards, it became clear that a different approach was required; the thought to somehow adjust alpha during program execution came to mind, and as such a new idea was born; a function was written which would, at every call of the forward pass function, recalculate alpha based on the difference between predicted state and current state, as well as a stability boost meant to give more weight to grid areas where the current state was stable. Alpha was also instead turned into a weighing tensor.

```
def calculate_alpha(current_state, predicted_state):
    #define the lowest and highest possible values for alpha
    alpha_low = 0.001
    alpha_high = 0.5

    #calculate the difference between predicted state and current state
    state_difference = torch.abs(predicted_state - current_state)

    #give more weight to areas where the current state is stable
    stability_boost = 0.2
    #scale the state difference within the alpha range
    diff = state_difference * (alpha_high - alpha_low)
    #calculate new alpha
    alpha = alpha_low + diff - (current_state * stability_boost)

    #ensure alpha is within the specified range
    alpha = torch.clamp(alpha, alpha_low, alpha_high)

    return alpha
```

However, the results were still not very good. Only Lenia with a randomized initial configuration seemed to improve; though if one looks closely, they could notice some abnor-

malities occuring. SmoothLife and Lenia with an orbium initial configuration actually died even faster than before. Some fiddling with the set values improved the models' stability slightly, but it was clear that more advanced adjustment was, indeed, needed.

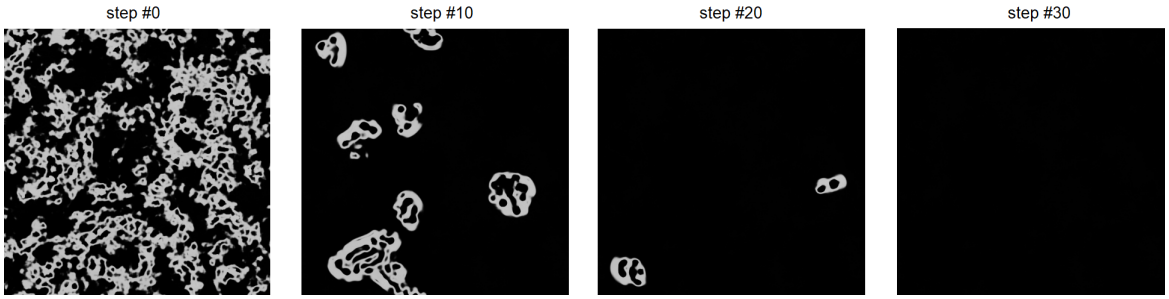

Figure 4.6: SmoothLife results of a predicted state influence with a dynamic alpha.

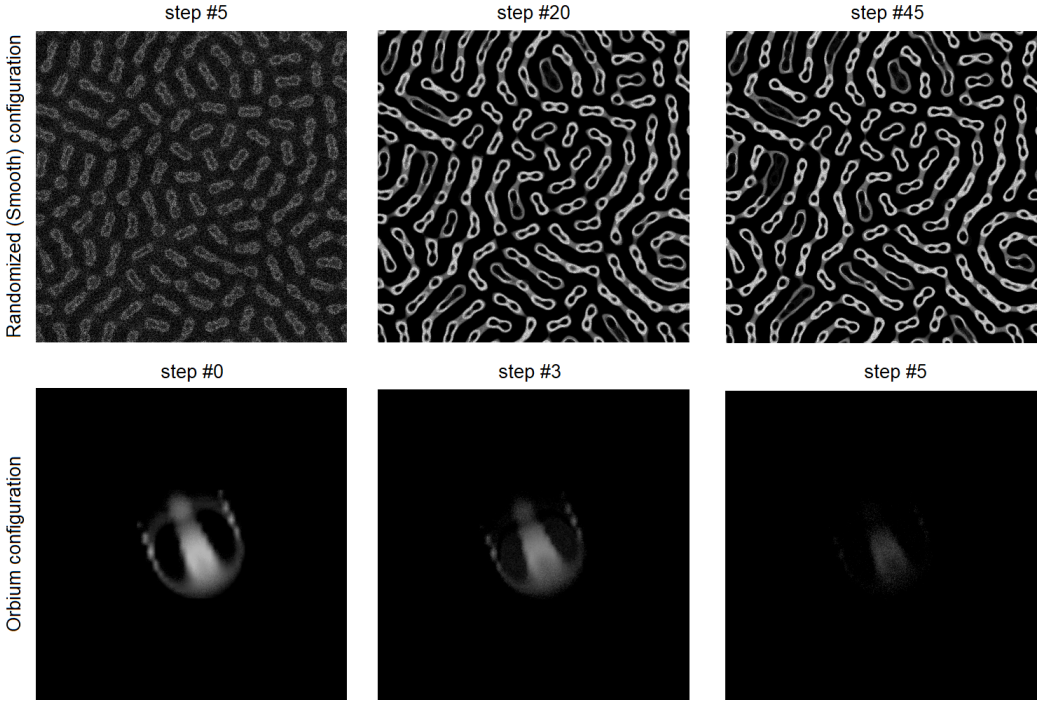

Figure 4.7: Lenia results of a predicted state influence with a dynamic alpha.

This was when the final ideas were put in place. The stability factor received some adjustments, while a boost to maintain already existing patterns was introduced as well. This boost was realized by taking the current state of the grid and subtracting a percentual weight of it from the alpha, to limit its influence in those regions. Lastly, alpha received some smoothing as well, to avoid abrupt transitions.

```
def calculate_alpha(predicted_state, alpha_low=0.05, alpha_high=0.5):
    #calculate the difference between predicted state and current state
    state_difference = torch.abs(predicted_state - grid)

    #promote stability to next state, apply it
    stability_factor = 0.5 - state_difference
    alpha = alpha_low + (stability_factor * (alpha_high - alpha_low))

    #add a small boost to alpha for regions where current state is present
    regulator = 0.05 #NOTE: this boost was lowered significantly for Lenia, to 0.005
    alpha -= self.grid * regulator  #boost to maintain existing patterns

    #apply a smoothing operation to avoid abrupt transitions
    smoothed_alpha = torch.nn.functional.conv2d(alpha.unsqueeze(0).unsqueeze(0),
           weight=torch.ones((1, 1, 3, 3)) / 9,  #3x3 averaging kernel for smoothing
           stride=1, padding=1,).squeeze()

    #clamp alpha to ensure it stays within the allowed range
    alpha = torch.clamp(smoothed_alpha, alpha_low, alpha_high)

    return alpha
```

The results were, finally, satisfactory enough. Lenia with an orbium initial grid configuration sadly still continued to die, though this time more slowly at least, and instead of simply disappearing the orbium "imploded„ inward. No matter how many adjustments to the boosts were made, this situation didn't seem to change. However, more noticeable improvements could be observed in the other simulations, where SmoothLife became more "alive„ and aggressive, primarily so thanks to the fact that the new adjustments made it so gliders, instead of simply continuing to glide in one direction, had gotten more prone to splitting up into two separate gliders, setting off a chain reaction. Lenia with a randomized grid configuration also seemed to assemble much faster than in any previous attempts.

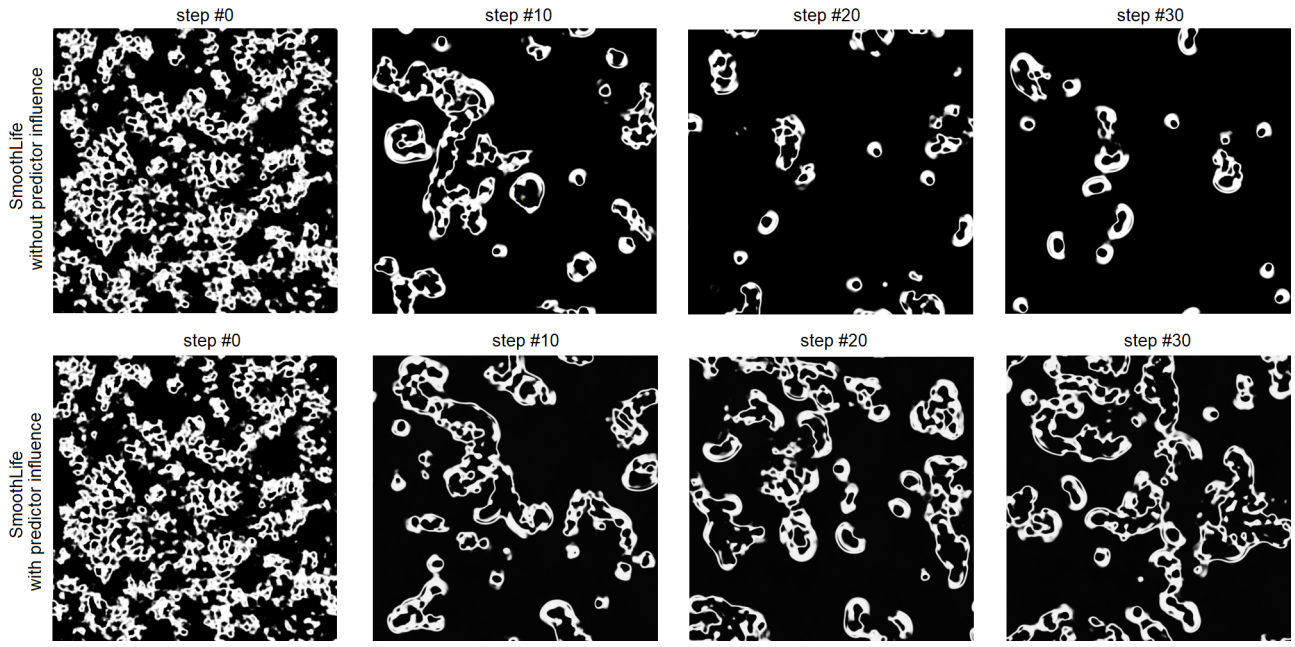

Figure 4.8: Comparison of SmoothLife results with and without a predicted state influence (with an improved dynamic alpha).

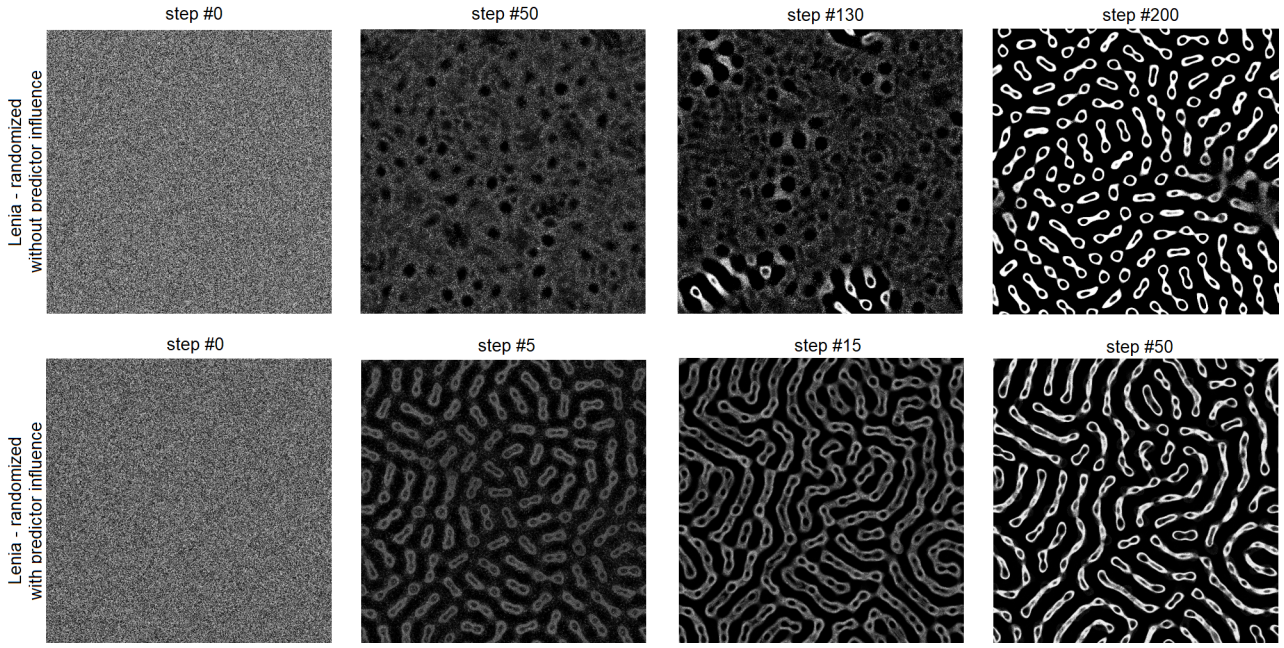

Figure 4.9: Comparison of Lenia (with randomized grid configuration) results with and without a predicted state influence (with an improved dynamic alpha).

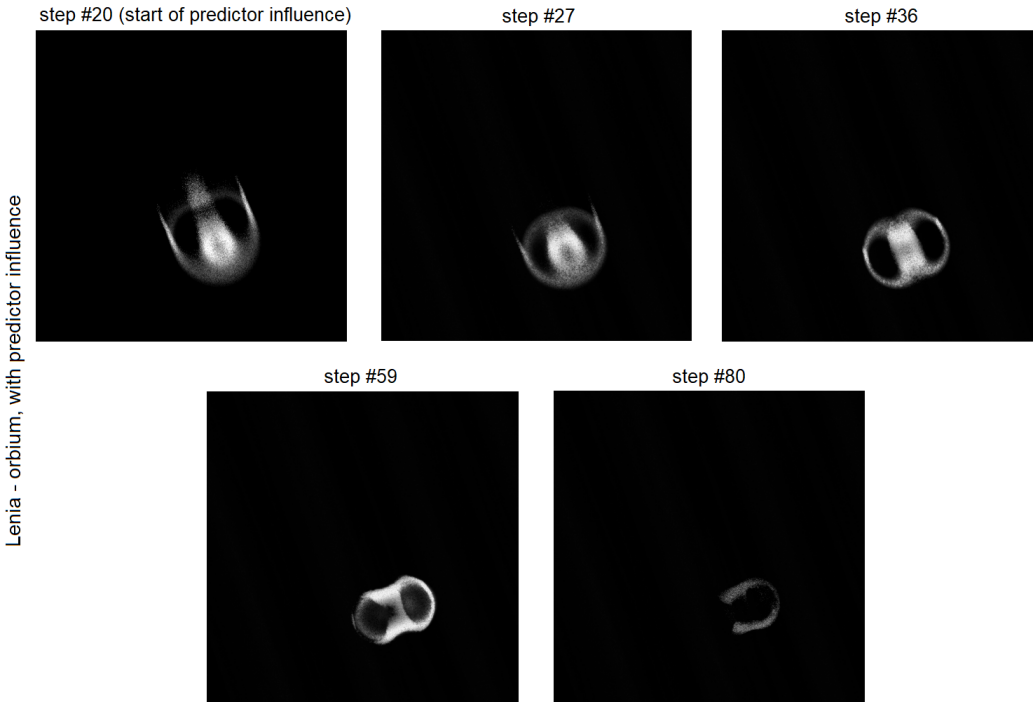

Figure 4.10: Lenia (with orbium grid configuration) results with a predicted state influence (with an improved dynamic alpha). The simulation was allowed to run without the predictor influence for 20 steps to better capture the shown behaviour.

4.6 Results Discussion & Conclusion

Experiments with our trained machine learning model yielded a varied arrangement of observable results of both the positive and negative kind, highlighting the successful approaches as well as methods which required some further refinement. Our first experiment was a testament to the latter; while the idea of a weight dictating the amount of sway the predictor would have over the simulation wasn't, in of itself, a bad one, the fact of said weight being static was truly an exercise in naivety. This showed in the results, where each of the three tested simulations died sooner or later—SmoothLife saw a gradual decrease in population primarily due to gliders disappearing, which can be seen in the second screenshot in figure 4.4, while Lenia practically failed to so much as start, with a randomized configuration failing to truly fully form and orbium configuration simply disappearing altogether.

It wasn't until we decided to switch to dynamically changing alpha during program runtime when we actually started seeing benefits of our cellular automata and machine learning fusion. Granted, our first attempt at doing so ended up still in a not-so-pleasant outcome, with SmoothLife and Lenia with orbium initial configuration even dying faster than previously, while Lenia with randomized grid configuration at least formed but with observable artefacts and inconsistencies. A few adjustments to alpha's calculation process later, however, primarily through boosts to stability and to maintaining already existing patterns, and we started actually seeing great improvements in the overall performance of two of the three models.

SmoothLife actually turned its behaviour around somewhat, with gliders not disappearing anymore and instead taking to multiplying by splitting in two, and the whole simulation visibly becoming more erratic and "alive„, as illustrated in figure 4.8. Lenia with randomized initial grid configuration formed much faster than without the influence of machine learning—concretely, taking four times less time to fully form, as evident in figure 4.9. Orbium still continued to die, however; albeit this time around, its death was much slower and visually interesting, what with it imploding.

Overall, we can conclude that the inclusion of machine learning in simulations using cellular automata can be beneficial, with proof shown as to the notion. We've seen gradual performance improvement in the simulation of both the SmoothLife and Lenia cellular automata models, though when it comes to orbium configuration, some additional stabilization mechanisms might be required to maintain a healthy simulation state. The persistent issues faced in this configuration suggest that future work should explore alternative approaches to stability and consider trying different machine learning models and techniques to avoid inherent tendencies toward collapse.

Bibliography

[1] Bengio, Y., Simard, P. and Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*. 1994, vol. 5, no. 2, p. 157–166, [cit. 2024-03-28]. DOI: 10.1109/72.279181. ISSN 1045-9227. Available at: https://ieeexplore.ieee.org/document/279181.

[2] Burzyński, M., Cudny, W. and Kosinski, W. K. Cellular automata: structures and some applications. *Journal of Theoretical and Applied Mechanics*. Warsaw: [b.n.]. 2004, vol. 42, p. 461–482, [cit. 2024-03-22]. ISSN 1429-2955. Available at: https://api.semanticscholar.org/CorpusID:55638228.

[3] Cable, K. W. *Alan Turing's Reaction-Diffusion Model – Simplification of the Complex* [online]. 1. december 2010. Available at: https://phylogenous.wordpress.com/2010/12/01/alan-turings-reaction-diffusion-model-simplification-of-the-complex/.

[4] Cain, D. *Harnessing Nature's Genius: The Convergence of Evolutionary Algorithms and Cellular Automata in AI* [online]. 7. august 2023. Available at: https://www.linkedin.com/pulse/evolutionary-algorithms-cellular-automata-pioneering-future-cain.

[5] Carbonell, J. G., Michalski, R. S. and Mitchell, T. M. Machine Learning: A Historical and Methodological Analysis. *AI Magazine*. september 1983, vol. 4, no. 3, p. 69, [cit. 2024-04-02]. DOI: 10.1609/aimag.v4i3.406. Available at: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/406.

[6] Chan, B. W.-C. *From Conway to Lenia* [online]. 2024-04-15 [cit. 2024-05-04]. Available at: https://colab.research.google.com/github/OpenLenia/Lenia-Tutorial/blob/main/Tutorial_From_Conway_to_Lenia.ipynb.

[7] Chan, B. W.-C. Lenia: Biology of Artificial Life. *Complex Systems*. Wolfram Research, Inc. october 2019, vol. 28, no. 3, p. 251–286, [cit. 2024-05-04]. DOI: 10.25088/complexsystems.28.3.251. ISSN 0891-2513. Available at: http://dx.doi.org/10.25088/ComplexSystems.28.3.251.

[8] Chintala, S. Neural Networks Tutorial. *Neural Networks* [online]. 2024. Available at: https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html.

[9] Cornforth, D., Green, D. G. and Newth, D. Ordered asynchronous processes in multi-agent systems. *Physica D: Nonlinear Phenomena*. may 2005, vol. 204, no. 1, p. 70–82, [cit. 2024-03-11]. DOI: https://doi.org/10.1016/j.physd.2005.04.005. ISSN 0167-2789. Available at: https://www.sciencedirect.com/science/article/pii/S0167278905001338.

[10] David, E. *Deep Learning For Dummies®* [online]. Deep Instinct Special Editionth ed. 111 River St., Hoboken: John Wiley 'I&' Sons, Inc., 2018 [cit. 2024-04-02]. ISBN 978-1-119-48358-8. Available at: https://info.deepinstinct.com/hubfs/TOF/Deep%20Learning%20for%20Dummies.pdf?

[11] Engineering, V. C. of. *Neural Networks* [online]. Hyderabad: Vardhaman College of Engineering, march 2021 [cit. 2024-03-13]. Available at: https://vardhaman.org/wp-content/uploads/2021/03/Neural-Networks.pdf.

[12] Fatès, N. A. Asynchronous cellular automata. In: Meyers, R., ed. *Encyclopedia of Complexity and Systems Science* [online]. Springer, December 2017, p. 21, 2023-09-11 [cit. 2024-03-10]. DOI: 10.1007/978-3-642-27737-5_671-1. Available at: https://inria.hal.science/hal-01653675.

[13] Fradkov, A. L. Early History of Machine Learning. *IFAC-PapersOnLine*. 2020, vol. 53, no. 2, p. 1385–1390, [cit. 2024-04-03]. DOI: https://doi.org/10.1016/j.ifacol.2020.12.1888. ISSN 2405-8963. Available at: https://www.sciencedirect.com/science/article/pii/S2405896320325027.

[14] Grosse, R. *Lecture 7, Part 1: Generalization* [online]. Toronto: Department of Computer Science of the University of Toronto, 2020 [cit. 2024-04-02]. Available at: https://csc413-2020.github.io/assets/readings/L07a.pdf.

[15] Hanson, J. E. Cellular Automata, Emergent Phenomena in. In: Meyers, R. A., ed. *Encyclopedia of Complexity and Systems Science* [online]. Yorktown Heights, USA: Springer, New York, NY, January 2009, p. 768–778 [cit. 2024-03-08]. ISBN 978-0-387-30440-3. Available at: https://doi.org/10.1007/978-0-387-30440-3_51.

[16] He, K., Zhang, X., Ren, S. and Sun, J. Deep Residual Learning for Image Recognition. *CoRR*. 2015, abs/1512.03385, [cit. 2024-04-20]. DOI: https://doi.org/10.48550/arXiv.1512.03385. Available at: http://arxiv.org/abs/1512.03385.

[17] Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In: Pereira, F., Burges, C., Bottou, L. and Weinberger, K., ed. *Advances in Neural Information Processing Systems* [online]. Curran Associates, Inc., 2012, vol. 25 [cit. 2024-04-20]. Available at: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[18] Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*. 1998, vol. 86, no. 11, p. 2278–2324, [cit. 2024-04-20]. DOI: 10.1109/5.726791. Available at: http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf.

[19] McCulloch, W. S. and Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*. december 1943, vol. 5, no. 4, p. 115–133, [cit. 2024-04-20]. DOI: 10.1007/BF02478259. ISSN 1522-9602. Available at: https://archive.org/details/a-logical-calculus-of-ideas-immanent-in-nervous-activity.

[20] Mehlig, B. Artificial Neural Networks. *CoRR*. arXiv. january 2019, abs/1901.05639, [cit. 2024-03-14]. DOI: https://doi.org/10.48550/arXiv.1901.05639. Available at: http://arxiv.org/abs/1901.05639.

[21] Merritt, R. *What Is a Transformer Model?* [online]. 25. march 2022. Available at: https://blogs.nvidia.com/blog/what-is-a-transformer-model/.

[22] mikolalysenko. *Conway's Game of Life for Curved Surfaces (Part 1)* [online]. 19. november 2012. Available at: https://0fps.net/2012/11/19/conways-game-of-life-for-curved-surfaces-part-1/.

[23] Mordvintsev, A., Randazzo, E., Niklasson, E. and Levin, M. Growing Neural Cellular Automata. *Distill*. february 2020, [cit. 2024-03-08]. DOI: 10.23915/distill.00023. Available at: https://distill.pub/2020/growing-ca.

[24] Nagyfi, R. The differences between Artificial and Biological Neural Networks. *The differences between Artificial and Biological Neural Networks | by Richard Nagyfi | Towards Data Science* [online]. 4. september 2018. Available at: https://towardsdatascience.com/the-differences-between-artificial-and-biological-neural-networks-a8b46db828b7.

[25] Rafler, S. Generalization of Conway's "Game of Life„ to a continuous domain - SmoothLife. *ArXiv*. Ithaca, New York: Cornell University. november 2011, [cit. 2024-05-01]. DOI: https://doi.org/10.48550/arXiv.1111.1567. Available at: https://arxiv.org/pdf/1111.1567.

[26] Rampe, J. *4D Cellular Automata* [online]. 15. december 2017. 2017-12-15. Available at: https://softologyblog.wordpress.com/2017/12/15/4d-cellular-automata/.

[27] Roth, D. *Neural Networks* [online]. Philadelphia: Penn Engineering University of Pennsylvania, School of Engineering and Applied Science, Department of Computed and Information Science, october 2016 [cit. 2024-03-13]. Available at: https://www.cis.upenn.edu/~danroth/Teaching/CS446-17/LectureNotesNew/neuralnet1/main.pdf.

[28] Schuchmann, S. *Analyzing the Prospect of an Approaching AI Winter* [online]. [cit. 2024-04-17]. Bachelor's thesis. Available at: https://www.researchgate.net/publication/333039347_Analyzing_the_Prospect_of_an_Approaching_AI_Winter.

[29] Sethi, B. *Theory and Applications of Fully Asynchronous Cellular Automata* [online]. Shibpur, 2017. [cit. 2024-03-12]. Dissertation. Indira Gandhi Institute of Technology. Available at: https://www.researchgate.net/publication/332901217_THEORY_AND_APPLICATIONS_OF_FULLY_ASYNCHRONOUS_CELLULAR_AUTOMATA.

[30] Simonyan, K. and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In: [online]. September 2014 [cit. 2024-04-20]. Available at: https://arxiv.org/pdf/1409.1556.

[31] Tagliaferri, L. *An Introduction to Machine Learning* [online]. 31. may 2022. 2022-05-31. Available at: https://www.digitalocean.com/community/tutorials/an-introduction-to-machine-learning.

[32] Turing, A. M. Computing machinery and intelligence. *Mind*. 1950, vol. 49, p. 433–460, [cit. 2024-04-20]. DOI: https://doi.org/10.1093/mind/LIX.236.433. Available at: https://redirect.cs.umbc.edu/courses/471/papers/turing.pdf.

[33] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L. et al. Attention Is All You Need. *CoRR*. 2017, abs/1706.03762, [cit. 2024-05-07]. DOI: https://doi.org/10.48550/arXiv.1706.03762. Available at: http://arxiv.org/abs/1706.03762.

[34] Winkler, R. L. *"Risk„ and Energy Systems: Deterministic Versus Probabilistic Models* [online]. Laxenburg, Austria: International Institute for IIASA Applied Systems Analysis, september 1973 [cit. 2024-03-17]. Available at: https://pure.iiasa.ac.at/id/eprint/60/1/RM-73-002.pdf.