UNIVERSITY OF COLOGNE

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

IN COOPERATION WITH
JÜLICH SUPERCOMPUTING CENTRE (JSC)

MASTER'S THESIS

FEBRUARY 12, 2021

Deep Learning for Prediction and Control of Cellular Automata in Unreal Environments

MARCEL AACH

First Examiner: Prof. Dr. Ulrich Lang
Second Examiner: Dr. Jenia Jitsev
Advisor: Jens Henrik Göbbert

Abstract

In this thesis, we show the ability of a deep convolutional neural network to understand the underlying transition rules of two-dimensional cellular automata by pure observation. To do so, we evaluate the network on a prediction task, where it has to predict the next state of some cellular automata, and a control task, where it has to intervene in the evolution of a cellular automaton to achieve a state of standstill. The cellular automata we use in this case are based on the classical Game of Life by John Conway and implemented in the Unreal Engine. With the usage of the Unreal Engine for data generation, a technical pipeline for processing output images with neural networks is established.

Cellular automata in general are chaotic dynamical systems, making any sort of prediction or control very challenging, but using convolutional neural networks to exploit the locality of their interactions is a promising approach to solve these problems. The network we present in this thesis follows the Encoder-Decoder structure and features residual skip connections that serve as shortcuts in between the different layers. Recent advancements in the field of image recognition and segmentation have shown that both of these aspects are the key to success.

The evaluation of the prediction task is split into several levels of generalization: we train the developed network on trajectories of several hundred different cellular automata, varying in their transition rules and neighborhood sizes. Results on a test set show that the network is able to learn the rules of even more complex cellular automata (with an accuracy of $\approx 93\%$). To some extent, it is even able to interpolate and generalize to completely unseen rules (with an accuracy of $\approx 77\%$). A qualitative investigation shows that static rules (not forcing many changes in between time steps) are among the easiest to predict.

For the control task, we combine the encoder part of the developed neural network with a reinforcement agent and train it to stop all movements on the grid of the cellular automata as quickly as possible. To do so, the agent can change the state of a single cell per time step. A comparison between giving back rewards to agents continuously and giving them only in the case of success or failure shows that Proximal Policy Optimization agents do better with receiving sparse rewards while Deep Q-Network agents fare better with continuously receiving them. Both algorithms beat random agents on training data, but their generalization ability remains limited.

Acknowledgements

I would like to thank my advisors Dr. Jenia Jitsev and Jens Henrik Göbbert for giving me the opportunity to write my thesis at the Forschungszentrum Jülich, employing me as a student assistant at the JSC, and introducing me to the interesting topic of cellular automata. I especially appreciated the freedom to try out my own ideas during the course of this thesis and learned a lot from your feedback. I also want to thank Prof. Dr. Ulrich Lang for serving as the first examiner of this thesis.

List of Abbreviations

CA	**C**ellular **A**utomata
CNN	**C**onvolutional **N**eural **N**etwork
DQN	**D**eep **Q** **N**etwork
DL	**D**eep **L**earning
PPO	**P**roximal **P**olicy **O**ptimization
RL	**R**einforcement **L**earning
UE	**U**nreal **E**ngine

Contents

Chapter 1

Introduction

The field of *Deep Learning*, especially of *Deep reinforcement learning*, is currently one of the most active research areas, producing exciting advancements almost every week. The AlphaGo algorithm, which beat a human player in a very complex board game, and one of its successors, AlphaFold, apparently solving the problem of protein folding, are just two examples of the exciting results that labs like Google DeepMind released in the past years. However, problems solved with deep learning usually require a lot of fine-tuning, so we end up with a model that is narrowly tailored to the problem. While a general artificial intelligence is still in the distant future, would it not be great to **show** a model a particular problem - for example, a chemical reaction - and let it learn by just observing?

The motivation of this thesis is to put forward such a kind of model, a neural network that is able to extract the underlying laws in processes from watching them. As these laws may be of very different kinds, the focus is on the generalization ability of neural networks here: our model should at least be applicable to a greater class of processes, and not just to one. Of course, observing such a process in nature would be far too complex, so we rely on a game engine to artificially generate a huge amount of image training data based on predefined equations. This data is used to train and evaluate the network on a prediction task, where we show it a number of steps and expect it to calculate the next one, as well as on a control task, where we let the network make changes to the process it sees.

In this thesis, the processes that are to be studied with deep learning are cellular automata (CA, often also CAs in the plural). These very simple dynamical systems exhibit interesting properties: although their interactions are all basic and spatially local, they can produce chaotic and truly unpredictable patterns, thus making them good candidates for modeling any operation in nature. In the next chapter, we will give an introduction to the research conducted in this field and present its most famous offspring: the Game of Life by John Conway [Gar70]. While he unfortunately passed away due to an infection with Covid-19 in 2020, his game has entertained mathematicians and computer scientists for the past decades and will continue to do so for sure. We focus on some of the dynamics and patterns that emerge from the Game of Life.

In the third and fourth chapter, we cover the basic theory that is required to understand the algorithms used in this thesis and also take a look at state-of-the-art methods in the field of deep learning: the usage of residual connections in U-shaped deep neural networks and the algorithms that can play video games (and beat human players!). In both chapters, we will also briefly cover some papers that have already applied deep learning methods to cellular automata.

The last part of this thesis will focus on the implementation of the experiments in C++ and Python and present and evaluate the results. The utilization of the Unreal Engine to embody the cellular automata and its rules makes the start. After generating image data and transferring it to a script in Python via an interface, we develop a neural network that can learn the rules of the underlying cellular automata, just by looking at the pictures. This is the prediction task, as the rules are used to model the next time step in the evolution of the cellular automata. Finally, we use the network and the reinforcement learning algorithms to control the outcome of a cellular automaton. The last section summarizes the findings and presents an outlook for future research.

Chapter 2

Cellular Automata as a Model of the Real World

Cellular automata were first developed by mathematicians John von Neumann [Neu66] and Stanislaw Ulam [Ula62] while working at the Los Alamos National Laboratory in the 1940s and 1950s. The motivation was to find a simple system that was capable of "self-replication", as von Neumann believed that nature and any living organism were also just following some very basic logical rules and Ulam was working on growing crystals. Implementing these thoughts into a program running on a computer would make this machine similar to a real living being, at least to some extent. In the following years, some other scientists conducted further theoretical research in the area, but it was John Conway who pushed the topic to a broad audience with his simple yet fascinating Game of Life [Gar70]. Based on very elementary rules even a child could understand and every software engineer could code in a matter of minutes, it creates an unlimited number of patterns to look at. Almost 40 years later, searching for the Game of Life on YouTube still brings up hour-long videos of designs to look at. However cellular automata still remained a mostly unstudied scientific discipline [Sch07].

The next big popularity boost came after Stephen Wolfram, a physicist at that time studying particle physics at CalTech, published a paper connecting cellular automata with statistical mechanics in 1983. Since then, he has made the study of cellular automata one of his main goals in life, concluding his more than 20 years of research with a volume called "A new kind of Science". What made him curious about the subject in the beginning, was how simple rules do not (as the widespread belief was) lead to simple behavior. Instead, these policies generated behavior that was as complex as he had ever seen, see fig. 2.1. In the book, he lays out his ideas on how the complete universe and everything in it could be based on simple cellular automata, following basic logical rules, also giving several examples from all different kind of scientific fields, which we will take a look on later in this section. As the title suggests, Wolfram sees cellular automata as a new kind of model, which may replace the older ones used in the natural sciences. [Wol02]

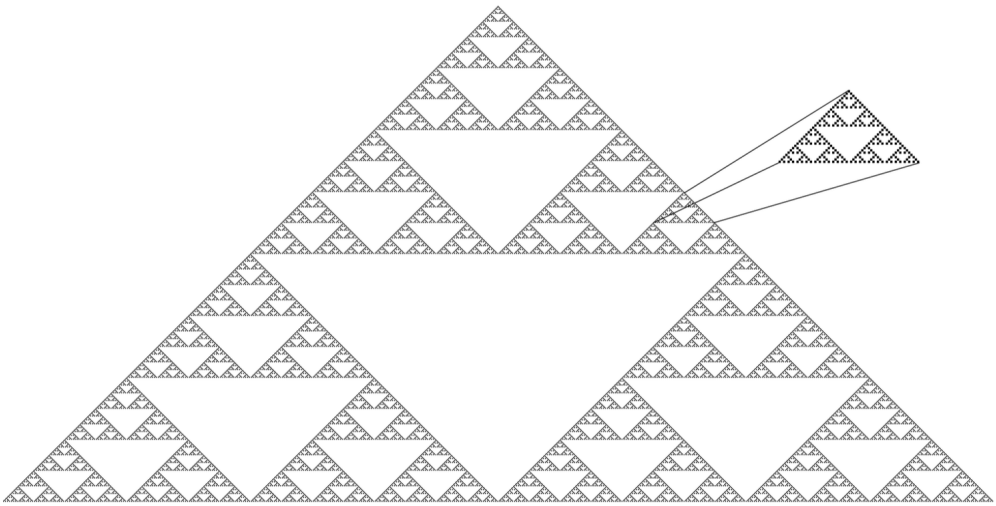

FIGURE 2.1: A cellular automaton that produces a nested, self-similar pattern after 500 steps, from [Wol02], Page 26

2.1 Definition

A cellular automaton can be described as an array of cells in a space that are connected to each other. Following [Ger95] closely in this section, we formally define a cellular automaton as a discrete, dynamical system with:

- **cell space:** the n-dimensional grid
- **cell states:** possible states a cell can take
- **cell neighborhood:** the number of neighbor cells that the state of the cell depends on
- **cell transition rules:** a local function that defines the transition of one cell state to another

Cell Space

The cell space is a discrete grid, which every CA lives on. However, there are variations when it comes to its geometry and especially dimension. While most applications use a $2D$ square grid, as these are easy to implement and to visualize on a screen, some real-world models (in physics or chemistry) may require a $3D$ space to be more flexible, see fig. 2.2.

Cell States

As all components of a CA, also the cell states are discrete and finite. They are described using a number, which depends on the problem. If there are only two cell states, the CA is also called a *binary automaton*. While such a tiny number might sound boring, for n cells there are 2^n possible total states the CA can take. So even a small 10×10 grid has 2^{100} possible configurations,

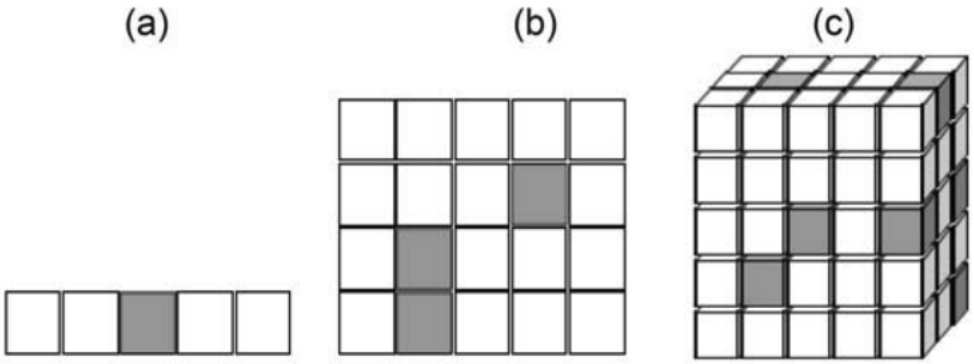

FIGURE 2.2: Different dimensions of cell spaces: $1D$, $2D$ and $3D$, taken from [Wei98]

which is an astronomical number. It is important to notice that not all of these states can be reached, depending heavily on the transition rules. [Ger95]

Cell Neighborhood

Depending on the geometry of the cell space, the cell neighborhood is already pretty straightforward. If we are looking at a one-dimensional CA, each cell only bounders one neighbor on the left and one on the right side. It is of course possible to increase this neighborhood size, by taking the next $2, 3 \ldots r$ cells into account. For a $2D$ CA, the most common choice is the *Moore neighborhood*, for which each cell the number of neighbors is the number of all cells it is adjacent to. The *von Neumann neighborhood* is a slight variation of this, by only considering the direct, "fully" adjacent cells 2.3. In the course of this thesis, we will make use of the Moore neighborhood with different sizes.

Cell Transition

The choice of "rules" is probably the most important component of a CA. To archive interesting or realistic behavior, the rules have to be similar to the rules observed in nature, at least to some extend. Apart from that, in his book [Sch07] proposes three fundamental features that apply:

- ***homogeneity:*** the cell states are all updated using the same transition rules
- ***parallelism:*** the cells are all updated at the same time
- ***locality:*** the cell transition rules are local

While these features apply to all CA in the "original" sense, there are however variations that break with some of these fundamentals, as it might be necessary to use asynchronous cell updates to better model a natural process.

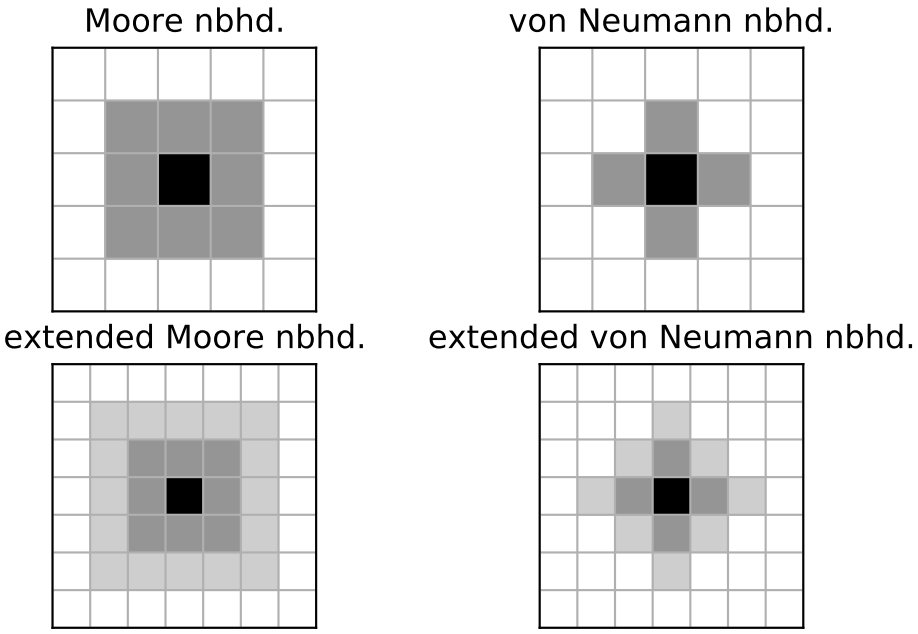

FIGURE 2.3: Different neighborhood concepts and sizes

We will be focusing on two-dimensional cellular automata in this thesis. While these are easy to implement and calculate, they still provide behavior and patterns that are interesting to study. In most of these cases, the cell space will consist of a square grid and there are only two different cell states: **True**$(= 1)$ and **False** $(= 0)$.

Classification

In his book [Wol02], Wolfram proposed the following classification system, based on the evolutionary behavior the cellular automata exhibited:

- *Class I:* nearly all initial patterns evolve to a fixed state (similar to a stable, stationary attractor). These are the simplest ones.
- *Class II:* nearly all initial patterns evolve to a stable, but periodic or oscillating state. These are similar to cyclic dynamical system.
- *Class III:* nearly all initial patterns evolve to an apparently random or chaotic state. Stable structures, that may arise are destroyed by noise. These are similar to chaotic dynamical system.
- *Class IV:* nearly all initial patterns evolve to localized structures, that interact with each other. This is probably the most interesting one and has no dynamical system equivalent.

These classifications are however not formally rigorous, as the same cellular automata can belong to two categories, depending on how many time steps are calculated.

2.2 Conway's Game of Life

In this thesis, we will focus on the "Game of Life", developed by the English mathematician John Conway and first described in October 1970 in an issue of the *Scientific American* [Gar70]. Conway's goal was to further simplify the rules for interesting cellular automata. While von Neumanns proposed concept included close to 30 different cell states, for Conway two cell states were sufficient. As computers were not that powerful and widely available at that time, it is told that the common rooms at the Mathematical Faculty of the University of Cambridge were used by him, to model his thoughts with lots of game tokens.
The Game of Life has two cell states, which correspond to a cell being **alive** or **dead**, with some very simple rules (using the direct Moore neighborhood) applied in each timestep:

- a cell comes to life if it has exactly three (of the eight) neighbors that are alive (reproduction)
- a cell with two or three neighbors that are alive stays alive (survival)
- if a cell has less than two neighbors that are alive or more than three, it dies (underpopulation and overpopulation)

Mapping the very basic principles of "real" life in nature and the terms of population increase, decrease and stagnation is where the Game of Life got its name from. The phrase "Game" might be a little misleading, as this is (in its original form) not a game where a player has any impact on the evolution (apart from defining the initial number and state of cells). We will later introduce a playable version of the Game of Life.

Over time many different variants of rule sets emerged. On the internet, there are several forums dealing with interesting discoveries based on the game. It is usually played with periodic boundary conditions, meaning that the left boundary of the grid is connected to the right boundary (and the top boundary to the bottom one). It belongs to Class IV automaton. Conway was looking to create a cellular automaton that featured patterns that disappear quickly but also growing ones. We will take a look at some of these interesting patterns now.

Lifeforms and Unpredictability

Starting out from an initial gameboard, we will probably reach a state where most of the cells have died out and only some of these, either periodic or static (still) patterns survive. The most fascinating thing about the Game of Life is, that there is no way to tell exactly how a configuration of cells is going to end (apart from computing every step). Therefore it is **unpredictable** [BCG04]. Some smaller patterns can however be studied: Following [Ger95]

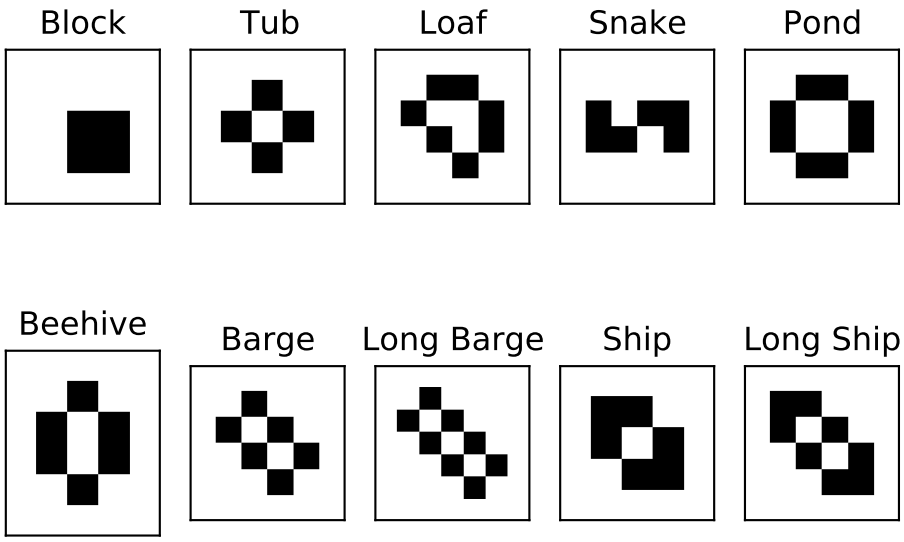

FIGURE 2.4: Some Still Life forms in Game of Life

the smallest viable organism or pattern has to have at least three alive neighbors in its 3×3 Moore Neighborhood. While there are several possible organisms, only two of them are capable of survival in the long term: the "Blinker" and the "Block". While the first one switches between two states endlessly, the other develops into a 2×2 block which does not change anymore.

Looking at some other simple starting conditions, a straight line of n life cells yields the following:

number of alive cells	**evolution over time**
n = 1 or 2	dies immediately
n = 3	is the Blinker
n = 4	becomes the Beehive after 2 iterations
n = 5	turns into the Traffic Lights
n = 6	dies after 12 iterations
n = 7	terminates in the Honey Farm after 14 iterations
n = 8	turns into 4 Blocks and 4 Beehives
n = 9	delivers two Traffic Lights after 20 iterations
n = 10	turns into pentadecathlon

TABLE 2.1: Lifeforms and their evolution, taken from [BCG04].
See figure 2.4 for visualization of the patterns.

If a pattern does not change from one generation to the next one, it is called Still Life. These patterns are very interesting to study as they provide a "final state" for the Game of Life. Some typical Still Life patterns are shown in figure 2.4.

While Still Life constellations are static and "dead", there is also the exact opposite: structures, that generate an endless amount of cells over time. After Conway offered $50 in 1970 for the discovery of such a structure, people

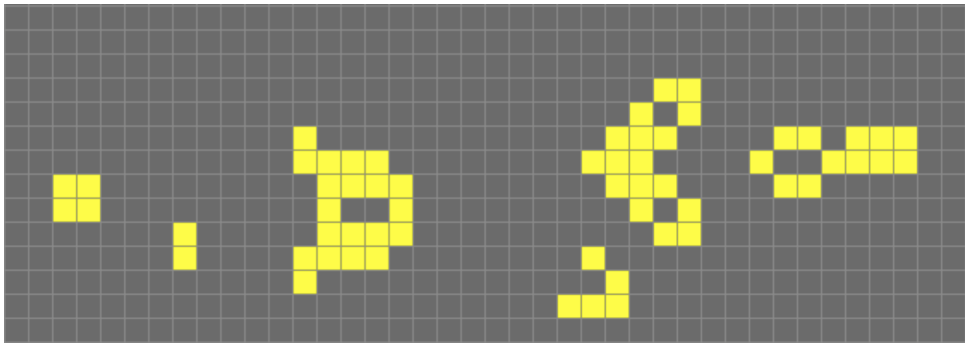

FIGURE 2.5: The Glider Gun shoots a glider, generated with https://playgameoflife.com/

quickly came forward with the concept of a **Glider Gun**, see Figure 2.5. This fascinating pattern births a new glider every 30 iterations and sends it across the map. The gun never runs out of power and thus is generating new life indefinitely. This is probably what von Neumann dreamt of when he thought about self-replication.

2.3 Applications

Though the concept of using cellular automata as models for real-world applications has not been as widespread as Wolfram had predicted in [Wol02], there still exist several scientific disciplines that use cellular automata for modeling purposes, some of them we will look at now:

Physics

Modeling a gas or a fluid in physics can be done by describing the behaviour of the underlying small physical particles that it consists of. If these particles all behave the same way, it is obvious that a cellular automaton would be a good model choice, using the cells as particles and the physical laws (like conservation of momentum) as transition functions. The "HPP-Lattice-Gas-model", introduced in 1973 by Hardy, Pomeau and de Pazzis [HPP73] is the perfect example for this.

Biology

When it comes to the simulation of population dynamics, biologists often use differential equations to discover information about the number of predators and prey in certain environments. While these equations are good in estimating the timeline, they do not give away the spatial component. A simple automaton provides a remedy: WATOR is a model to describe the life cycle of fish (prey) and sharks (predators) on a $2D$ grid. While the fish move randomly around, they are chased by sharks and eaten if they do not escape fast enough. If the sharks do not eat enough fish, they eventually also die. This

simple CA is similar to the Game of Life, though a little more complex, and was introduced by Dewdney and Wiseman in 1984 [Dew84].

Traffic and Pedestrian Dynamics

Predicting the density and the flow of car or pedestrian traffic also presents itself for modeling via cellular automata. We can divide the street or the space into small cells, which take the state "occupied" if there is a car or a pedestrian and "unoccupied" if not. Cars move along the grid with a specified velocity which is depending on their local neighborhood. For example, you would usually drive fast if the road is empty and slower if there are a lot of cars. As for pedestrian traffic, the models can be used to better simulate the behavior of large crowds in the event of an emergency for example.

Chapter 3

Deep Learning for Image Segmentation and Prediction

Of all areas related to Machine Learning, the field of *Deep Learning* has probably seen the biggest rise in popularity in the last few years. Especially its contribution to the fields of image recognition, the related autonomous driving and beating human players in all sorts of difficult games were the reason. When talking about Deep Learning, what one refers to is the development and training of Artificial Neural Networks (ANN) with several so-called *layers* of *neurons*. These neurons are the basic element of every ANN and are in their functionality similar to the human brain. If they receive a signal that is strong enough, they "fire" and thus are able to transmit the signal further.

The *Perceptron*, one of the first mathematical models of a neuron, was introduced by Rosenblatt in the late 1950s [Ros58]. While one of these Perceptrons alone is not very powerful, combining them with several other ones and connecting them in separate layers, lets them unfold their true power. In fact, Cybenko proved in 1989 that a very small ANN with just two layers (but a huge number of neurons in each layer) is capable of approximating any continuous function [Cyb89]. This *Universal approximation theorem* is the theoretical foundation of the ability of Artificial Neural Networks to represent very broad function families. In practice, however, it is usually necessary to build models with a lot more hidden layers than just two. Since these deep networks require a lot of data and a lot of computational power to train, it mainly was the technological progress that pushed the topic. With the introduction of Distributed Computing and Graphical Processing Units (GPU) it is now possible to train networks efficiently and fast.

3.1 Feedforward Networks

In *Supervised Learning* the data consists of an input vector $x \in \mathcal{X}$ that maps to some target vector $y \in \mathcal{Y}$ via a function $f : \mathcal{X} \to \mathcal{Y}$. For example, in the case of an image detection problem, where we would want to distinguish between an apple and a banana, the input space would be the image dimensions: $\mathcal{X} = \mathbb{R}^{n \times m}$ and the target space would be $\mathcal{Y} = \{0, 1\}$, where 0 means

apple and 1 means banana. The goal now is to find the best possible approximation of that function so that $\hat{f} \approx f$.

The simplest choice for an approximation is a linear function:

$$\hat{f} = \omega^T x + b \tag{3.1}$$

where ω is a weight vector and b is the bias. The Perceptron, which we mentioned earlier, is doing exactly that.
So for a binary classification problem (like the apple-banana one) this would yield:

$$\hat{f}(x) = \begin{cases} 0 & \text{if } \omega^T x < b \\ 1 & \text{if } \omega^T x \geq b \end{cases} \tag{3.2}$$

Represented graphically, this would be the same as drawing a straight line into a data diagram.
By definition, a linear model cannot describe non-linear relationships. However, as most relations observed in the real world are non-linear, we have to make some changes to the Perceptron model to accommodate for this. For this we introduce a non-linear *activation function* $\sigma(x)$, transforming the model function (3.1) into:

$$\hat{f} = \sigma((\omega^T x + b)) \tag{3.3}$$

For the activation function, there are several choices. We will introduce the most common ones now:

- **Sigmoid**
  The Sigmoid is a non-linear monotonic function with the following form:

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

  It takes any value (out of $\mathbb{R}$) and maps it onto the interval $[0, 1]$. By looking at the graphs in figure 3.1, we can however see that the function produces some very small numbers close to zero for a lot of values. Especially for deeper networks, this can lead to the problem of the "vanishing gradient".

- **Hyperbolic Tangent**
  The hyperbolic tangent is similar to the Sigmoid function, as it is just a scaled version:

  $$\sigma(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

  Mapping to an interval of $[-1, 1]$ it suffers from the same problems as the original function as well.

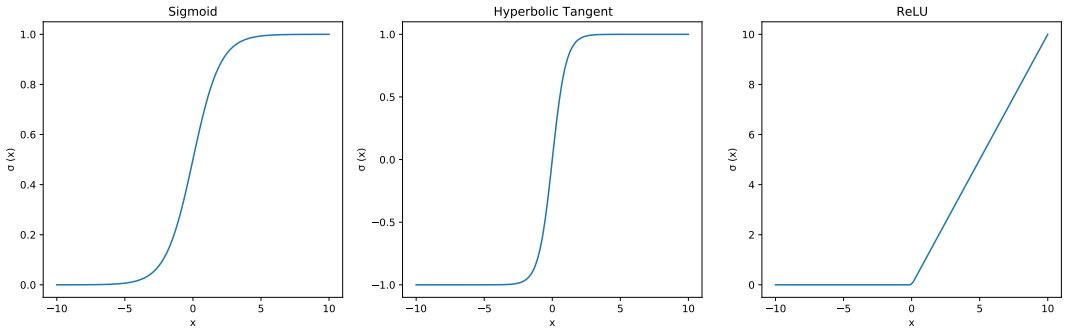

FIGURE 3.1: The most common activation functions

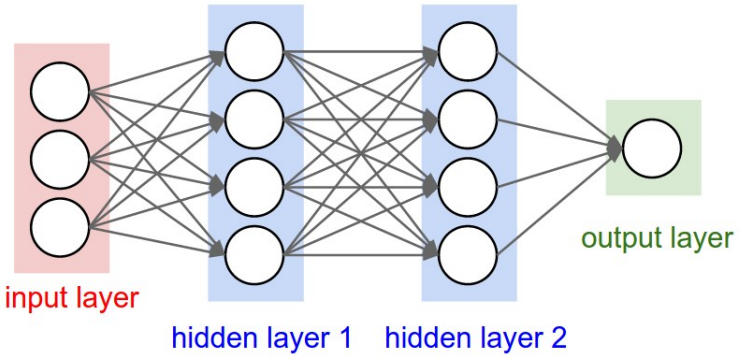

FIGURE 3.2: A neural network with two hidden layers, taken from [Kar]

- **Rectified Linear Unit** The ReLU function fixes the problem of the "vanishing gradient" because it does not saturate at some point. Also, it is very easy to calculate (thus reducing the computational demand) as its form is just:

$$\sigma(x) = \max(0, x)$$

  Especially for deep, dense networks it works very well. It can however result in too many "dead" neurons, as every negative number is mapped to zero.

In a classical fully-connected feedforward network, the neurons are arranged in several layers, with each neuron being connected to each neuron from the following and previous layer. However, as the name states, it can only transmit data further down the network and there are no connections going back to previous neurons. The first layer is always the input layer that takes the input data and has to match it in terms of size. The last layer of a network is the output layer, which puts out the results, see figure 3.2.

Training a Network

After talking about the general structure of Artificial Neural Networks, we will now take a look at the process which is used to train the parameters of a

network. The goal of the training process is to find the optimal parameter set θ (containing all the weights and biases of the neurons), that gives the best approximation of the real function.

First of all, we need to define a *Loss function*, that serves as a metric to tell us how far away from the desired outcome we are. A simple choice is the *Mean Squared Error* defined as the following (let T be the training set and N be the size of the training set):

$$L(\theta) = \frac{1}{N} \sum_{x \in T} ||f(x) - \hat{f}_\theta(x)||^2$$

with θ being the parameters of the network (weights ω and biases b). The loss function will be zero if the approximated values are the same as the exact values. So what we are looking to do is minimize the loss function, depending on the parameters θ. For classification problems, especially binary ones, the *Binary cross-entropy* is the loss function of choice. It looks like this:

$$L(\theta) = -\frac{1}{N} \sum_{x \in T} f(x) \log(\hat{f}(x)) + (1 - f(x)) \log(1 - \hat{f}(x))$$

and computes the difference between two probability distributions. The loss is computed after each *forward pass* (processing the input and predicting an output) through the network. Once we have a measure of how wrong the prediction is, we compute the loss of the global gradient. The choice of optimizer for this kind of problem is the gradient descent method, for which parameters are increased by a small value (the stepsize α) in the direction of the steepest descent of the gradient. This process is repeated for a huge number of steps:

$$\theta_{new} = \theta_{old} - \alpha \frac{\partial L}{\partial \theta}$$

Computational results have shown that it is easier to not calculate the gradient for the whole data set, but instead randomly choose just a fraction of the data points (called a mini batch) and perform the gradient descent on these. This method is called stochastic gradient descent [KW52].
One setback of this method is, that the step size is not adaptive to the problem. While there are implementations that reduce the step size over time or introduce a "momentum", the optimizer can still get stuck with a too-small step size. In this thesis we will use the Adam (adaptive moment estimation) optimizer, which is an improved form of the stochastic gradient descent [KB17].

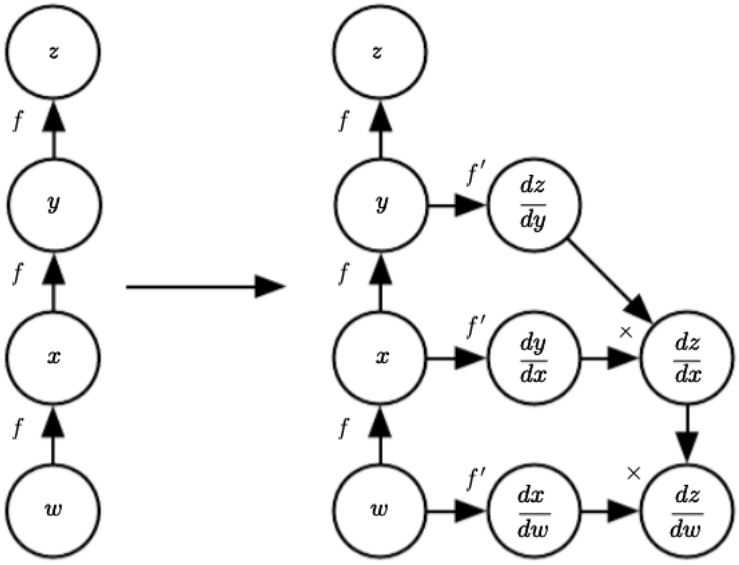

FIGURE 3.3: Automatic Differentiation approach to computing derivatives, taken from [GBC16]. A graph of the function $z = f(f(f(w)))$ on the left and the derivation of $\frac{dz}{dw}$ on the right.

Backpropagation

We still have to explain how to compute the gradient and apply the correction with regard to each individual weight. This is done using the backpropagation method, which is based on the chain rule. Given two functions $z = f(y)$ and $y = g(x)$ we can compute the derivative:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \tag{3.4}$$

Using this formula for recursive backward calculation we get the error for every single weight. As manually deriving the derivatives for each weight would require too much work, the backpropagation method makes use of *Automatic Differentiation*: In the implementation package (Tensorflow) we will be using later, all mathematical operations in the neural network are represented in a computational graph. The graph has additional nodes for the derivatives that contain a symbolic description of the derivatives [GBC16]. This means that no numeric values are required for the differentiation, until it is time to evaluate the symbolic terms, see figure 3.3. The graph architecture also makes Automatic Differentiation very computational efficient.

3.2 Convolutional Neural Networks

The problem of fully connected neural networks (FCNN) we have learned about in the previous section is that they do not offer good scaling properties when it comes to images. As each neuron of a layer is connected to each neuron of the next layer, even an image with a small number of pixels would

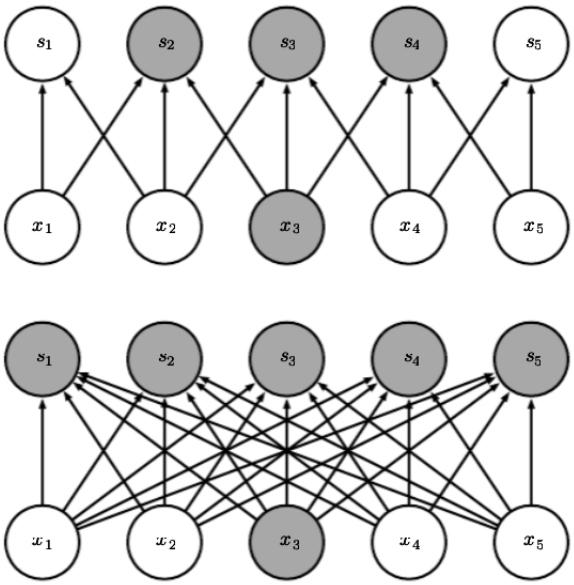

FIGURE 3.4: Top: Sparse connectivity in a convolutional layer for a kernel of size 3. Bottom: connectivity in a fully connected layer, taken from [GBC16], page 326

generate a huge amount of parameters. For example, an image of the size 100×100 with 3 color channels would yield $100 \cdot 100 \cdot 3 = 30000$ parameters (plus the bias) for each neuron in the layer after the input layer, raising the computational demands by a lot.

To solve this problem, Convolutional Neural Networks were introduced. Among others, they offer two key advantages [GBC16]:

- **sparse interactions** ensure every neuron is only connected locally to a subset of the neurons of the previous layer, see figure 3.4. This reduces the number of parameters in a model.
- **parameter sharing** means that instead of learning a different set of parameters for every location in the input data, we "reuse" the same set, thus only learning one set.

Simply put, a CNN exploits the local properties of images (pixels that are close to each other, are usually in some way related to each other) and reduces the number of parameters. To do so, in the mathematical theory the convolution of two continuous functions x and w is defined as :

$$s(t) = (x * w)(t) = \int x(a) \cdot w(t-a)da \tag{3.5}$$

So practically speaking the convolution for a specific point t is the average of the function x combined with a weight function w around t [GBC16]. As we are looking at discrete data points, equation 3.5 changes to:

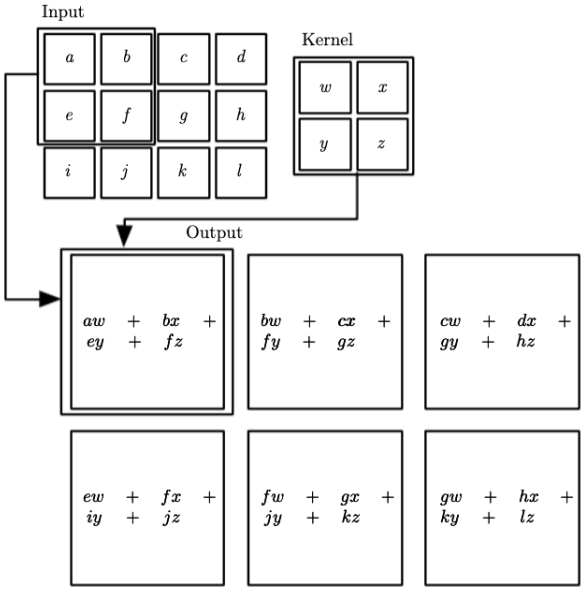

FIGURE 3.5: Example of a 2D convolution applied to input data. The result is the feature map. Taken from [GBC16], page 325

$$(x * w)(t) = \sum_{a=-\infty}^{\infty} x(a) \cdot w(t-a) \tag{3.6}$$

or in two dimensions, using an Image I and a kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_{m} \sum_{n} I(i-m)(j-n) \cdot K(m,n) \tag{3.7}$$

Practically speaking this means we are moving a filter K of dimension $n \times m$ across the input image and compute a multiplication of the image data and the weights of filter matrix, resulting in the output feature map, see figure 3.5. As the weights and biases are shared for each filter, this reduces the parameter complexity of the model. The following hyperparameters can be modified in a convolutional model:

- a **kernel size** K or filter size, depending on how big the "view" of the kernel should be.
- a **stride s** which defines how much space is left between each application of the filter.
- **Padding p** which extends the boundaries of the image, if the choice of the kernel and stride do not fit perfectly. For example, it is possible to pad along the boundaries with zeros.
- **filters f** the number of filters or feature maps created.

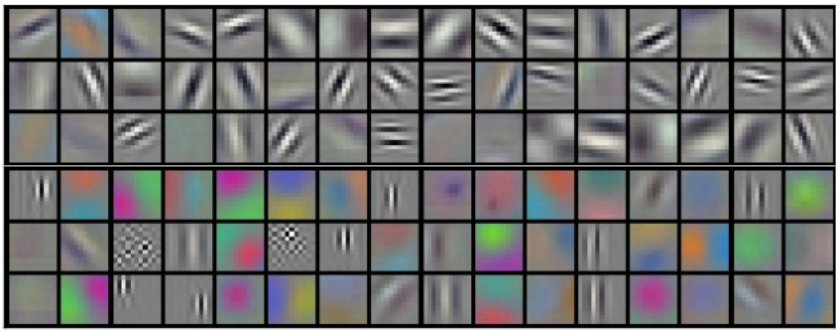

FIGURE 3.6: Example filters learned. Shapes and colors can be detected. Taken from [Kar]

Of course, the weight sharing feature of CNN's means, that the features we are looking to extract from a picture, should be present in all areas of the image. If they are only visible for example in the center of the image (like portraits), it does not make sense to use weight sharing.

In image recognition, the filters in the first layers of a CNN usually extract basic features like straight shapes or colors. The deeper the layer's position is in the network, the more complex the extracted features are. This can be visualized, see figure 3.6.

Pooling Layers

To further reduce the number of parameters in a CNN, there are Pooling layers. These are usually put behind a convolutional layer and, like the name suggests, pool the data. One can specify the pool size (e.g. 2×2) and the pooling action to perform (e.g. max pooling or average pooling [ZC88]). The pooling window then glides across the output feature map and performs a down-sampling operation. They do not have any trainable parameters on their own.

Batch Normalisation

To improve the convergence behavior and stability of CNNs, Ioffe and Szegedy introduced Batch Normalisation layers in 2015 [IS15]. They act as a normalization of layer activations, by shifting the activations of each layer in the network to zero mean and unit variance over the input mini-batches. This reduces the internal covariate shift.

3.3 ResNet

Using the Deep Learning methods, especially the CNN layers, we later want to build a network that is capable of extracting the rules from the Game of Life. For this, we take a look at the state-of-the-art methods used in the field

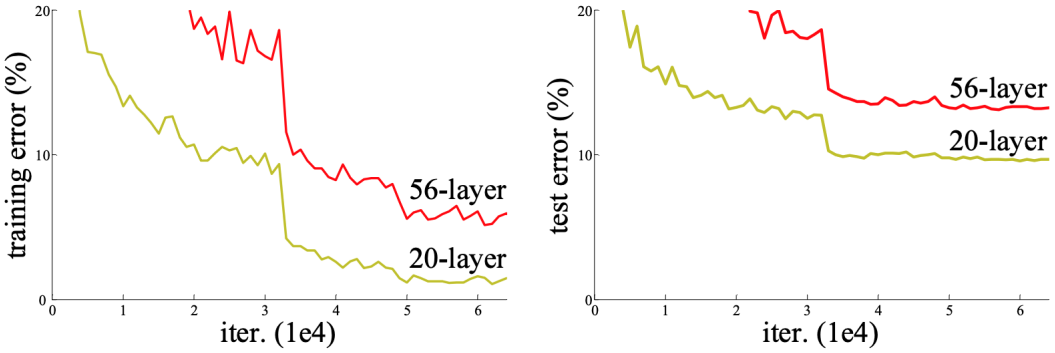

FIGURE 3.7: The degradation problem: The deeper network has the worse performance on the CIFAR-10 dataset. Taken from [He+15]

of image recognition.

Stacking a lot of convolutional layers on top of each other and thus generating very deep neural networks has been the main strategy in recent years to improve accuracy for Image Recognition tasks. Each year at the ImageNet Large Scale Visual Recognition Competition (ILSVRC) researchers can submit their networks and see how well they generalize. A good performing, very deep network is the VGG-net for example, with an error rate of about seven percent [SZ15]. It has however been shown, that just stacking layers on top of each other does not always work. Actually, the performance drops with the addition of more layers (degradation problem, see figure 3.7). This can lead to problems, as the deeper layers are required to extract hidden features from the input images. In 2015 the research team at Microsoft therefore introduced a Deep Residual Network (ResNet) to address this problem [He+15].

In a ResNet-architecture the desired output mapping $\mathcal{H}(x)$ is not fitted to the input x, but instead a transformed mapping $\mathcal{F}(x) = \mathcal{H}(x) - x$ is fitted. $\mathcal{F}(x)$ in this case is the residual mapping, so the desired output mapping becomes $\mathcal{H}(x) = \mathcal{F}(x) + x$. In terms of the structure of the network, this is realized using a *shortcut connection*, which performs an identity mapping, see figure 3.8. It is interesting to note that the original idea of solving the residual problem comes from the field of numerical mathematics and the multigrid method [BHM00].
The mapping simply adds the input x to the output of the weight layers $\mathcal{F}(x)$. Of course (as we are adding the values in this case), the dimensions of the data have to be the same, so a convolution with a stride is not possible. There are however workarounds, for example, one could use the concatenation or also down-sample the identity mapping. The authors in the paper use padding with zeros or projection (which however introduces more parameters). All of this has no influence on the training methods and standard optimizers like Adam can still be used. Stacking multiple residual blocks on top of each other or implementing shortcuts that skip more than just one or

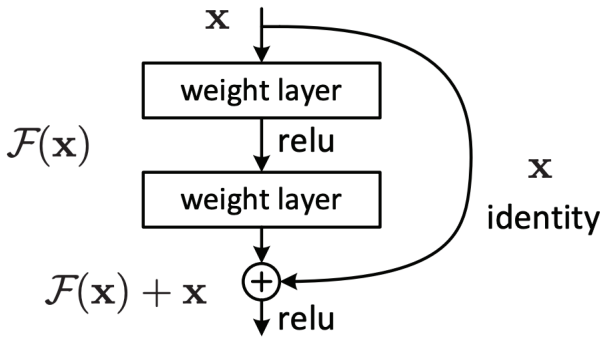

FIGURE 3.8: A building block in the ResNet, taken from [He+15]

two layers is also possible and often done in practice. On the ILSVRC in 2015 the ResNet architecture was able to archive an error as low as 3.57%. The structure of the ResNet in comparison to "plain" networks is shown in figure 3.9. The authors use 31 convolutional layers (with an increasing number of filters), two pooling layers, and a fully connected dense layer for the output at the end. There are several modifications of the original ResNet by now, see for example [Xie+17].

3.4 Encoder-Decoder Structures and the U-Net

The goal of the ResNet is to predict the content of a whole image (for example if it shows a car or an apple). Especially in biomedical applications, however the output should not only be a single class label for the whole picture but instead a label for each part or even pixel of the input image. This type of problem is referred to as *image segmentation*. Consistent with this is, that the output dimension usually has to be the same or very similar to the input dimension.
The SegNet [BKC16] delivers great results for this type of task. It is capable of pixel-wise semantic segmentation using an Encoder-Decoder architecture. For these type of models, there is an encoder network in the first part, that takes the input and processes it through multiple layers. The processed input results in something we call *latent space*, which means that although we usually cannot directly understand the representation in this space, the network has mapped the input data to some result. It means that in this "hidden" space the network has already learned "what" the input data is. If we would add a suitable output layer at this point, we could probably extract this kind of information too, which is why often the structure of models that have already succeeded in this first task (image recognition), is taken for the encoder. In the case of the SegNet the encoder is similar to the VGG network [BKC16]. On the other side of the network, we have the decoder. The job of the decoder is to map the representation of the data in the latent space back to the dimension of the input data. In the case of the SegNet, the decoder is basically an inverse of the encoder: a convolutional layer becomes

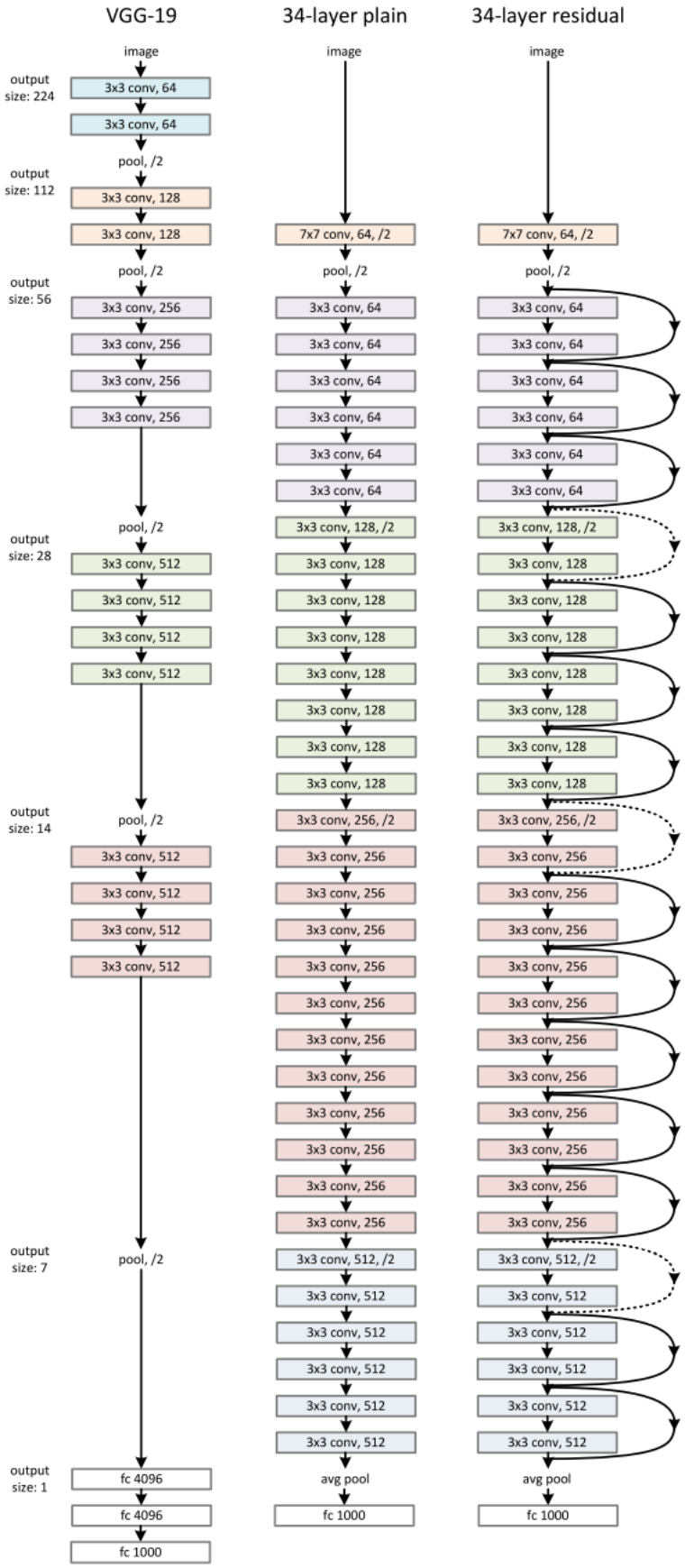

FIGURE 3.9: Structure of a 34 layer ResNet compared to a VGG net and a 34 layer plain net, taken from [He+15]

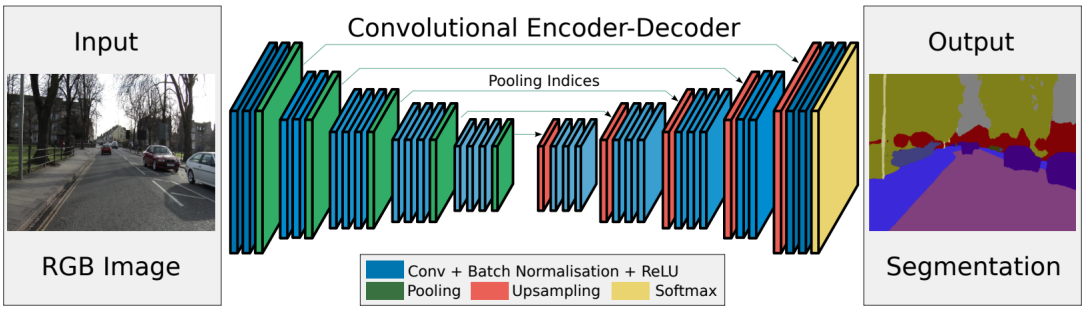

FIGURE 3.10: Architecture of the SegNet, with an example input and output. Taken from [BKC16]

a deconvolutional layer, a pooling layer becomes an upsampling layer (all with the same parameters), see figure 3.10. Training these two parts (ciphering and deciphering) as a whole network from End-to-End, lends the ANN a free hand in the latent space representation of the data and thus improves the performance.
It is also important to note, that the SegNet is a fully convolutional network, so there are no fully connected dense layers, making it possible to use input data of variable size [LSD15]. The function of dense layers can however be replicated with 1×1 convolutional layers.

As creating the training data for segmentation tasks often requires a lot of work by hand, since each pixel has to be annotated, the amount of training data is not as great as for other classification tasks and networks should be able to learn from a few samples. Because the models that involve image processing often have a lot of parameters and thus need a lot of data, this can lead to problems. An efficient network that can work with less data is the U-Net. It was introduced by scientists at the University of Freiburg [RFB15]. Based on the architecture of the SegNet, it uses novel skip connections between layers. In this way, it is similar to the ResNet, but the connections are not "shortcuts" that only skip a few layers, but instead "long connections", that skip not only a lot of layers but also the latent space. So they go from each stage of the encoder to each stage of the decoder to preserve data. This way the decoder has access to the same information as the encoder (for example in regard to edges in the image) and no information is lost in the latent space. Other than that the U-net architecture consists of 3×3 convolutional layers, pooling and upsampling layers, and a 1×1 convolutional output layer, see figure 3.11.

3.5 Applications to Cellular Automata

The locality of convolutional neural networks makes them great candidates for applying them to dynamical systems with local interactions (as CA are). Because neural networks can (in theory) represent any function, we would expect near-optimal results when trying to predict the next state of a CA.

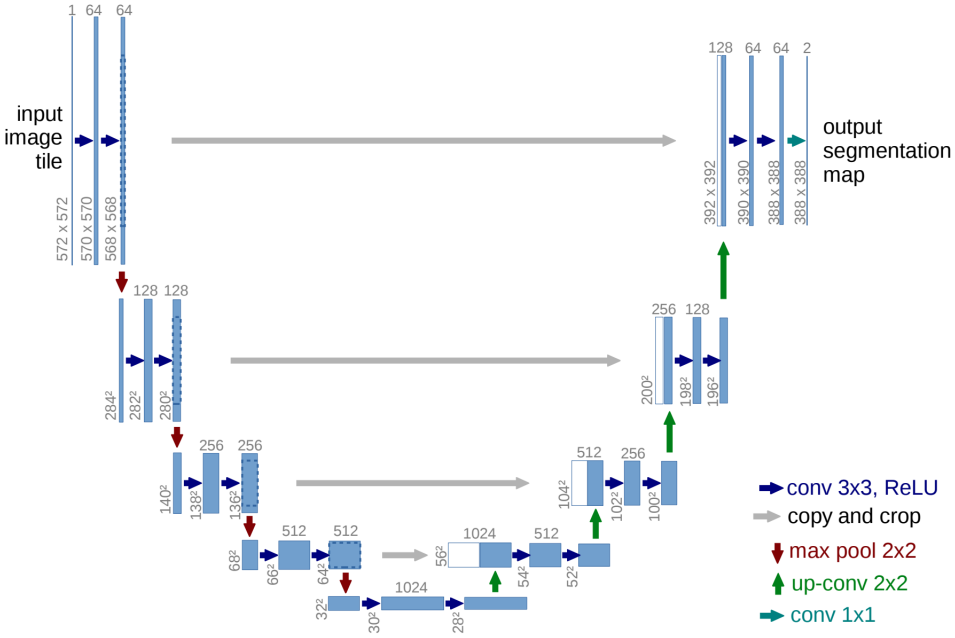

FIGURE 3.11: The structure of the U-net. The blue box corresponds to the number of filters used in the conv layers, doubling with each stage. Taken from [RFB15]

While this has been shown to work for very chaotic, turbulent systems [Pat+18], not much work has been done in the area of simple dynamical systems. A paper studying the ability of CNNs to learn the Game of Life (and related CA) was only published last year [Gil19]. The author demonstrates, that using just one convolutional layer with a kernel size of 3×3 with enough filters, followed by several 1×1 convolutional layers, is capable of predicting a binary cellular automaton with a Moore neighborhood of size $3 \times 3 = 9$ with an accuracy of 100% (after rounding). Analysis of the layers shows that the first layer counts the number of live or dead cells in the neighborhood, while the consecutive layers extract the rules of the CA with this knowledge. The resulting network is therefore fully convolutional and no fully connected layers are used.

While these results are very promising, it has also just recently been shown that learning the **exact** representation of the Game of Life for the prediction of one or several time steps is very hard [SK20]. The authors use a minimal neural network, made of two 3×3 convolutional layers, forcing the network to learn the explicit rules and not some other representation, by strictly limiting the number of filters and therefore the number of parameters. Predicting the next three, four and five time steps completely failed, and predicting the next one or two time steps only succeeded if the network was "overcomplete", meaning that it had five to ten times more parameters than it would theoretically need. The authors point out that this might be a general problem of CNNs, also referred to as the "lottery ticket hypothesis" [FC19].

Chapter 4

Deep Learning for Control

4.1 Reinforcement Learning

Unlike in *Supervised Learning*, where we have a labeled training data with the input and output results known, *Reinforcement Learning* takes a different approach. Here the learner is expected to discover the best "approximation" by trying-out. Generally speaking there is an agent, that is placed in an environment, trying to archive some goal by taking specific actions and getting feedback. This way of learning is very similar to the way an infant learns. By interacting with the world and getting either a positive or negative reaction for its behavior, the human starts to memorize and understand which action allows him to advance. For this introduction, we will be following the book written by Sutton and Barto [SB18]. Later in this chapter, we will focus on Deep Reinforcement Learning.
A Reinforcement Learning problem formally is defined by an environment and an agent. At each time step t, the agent is in a state $s_t \in \mathcal{S}$ and chooses an action $a_t \in \mathcal{A}(s)$, with $\mathcal{S}$ and $\mathcal{A}$ being the possible states and actions. As a consequence of the action, the environment will change to the state s_{t+1} and the agent receives a numerical reward $r_{t+1} \in \mathcal{R}$. This interaction is shown in figure 4.1.
The agent performs a mapping from the states to the probabilities of selecting each available action, called the *policy* π. Here $\pi_t(s, a)$ is the probability that action a is chosen if the current state is s. The goal of the agent is to maximize the total reward it gets. In this thesis, we confine ourselves to discrete,

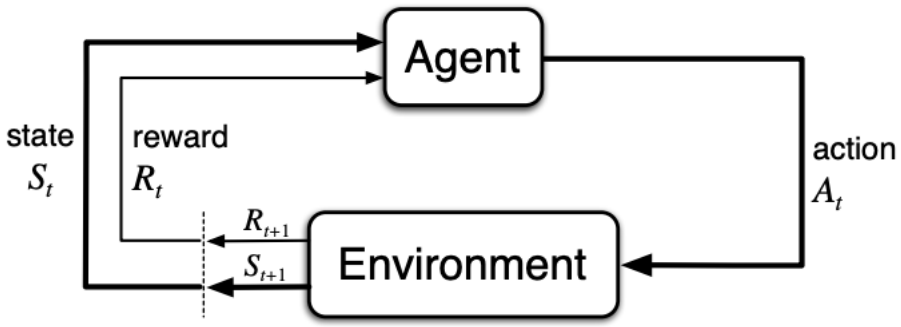

FIGURE 4.1: Interaction between the environment and the agent, taken from [SB18]

.

episodic problems.

Rewards

However for an agent to be efficient, he should not only take into account the instant reward he gets but instead the reward he gets over the course of the whole time it takes to solve the problem. So in time step t the agent should be maximizing the *expected reward*:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} \dots r_T \tag{4.1}$$

with T being the final step. Usually, the training of reinforcement learning agents is split into training episodes, where an episode ends if a certain terminal state is reached and the agent is reset to the beginning. But as the agent has to decide which action to take, the moment he sees the current environment, we need the concept of discounting the help with that decision. The *discounted expected return* at time step t is defined as:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{4.2}$$

where $0 \leq \gamma \leq 1$ is called the discount rate. It determines the present value of all future rewards. If γ is close to 1, the agent will act more farsighted, considering future rewards more strongly. On the other hand, if γ is close to 0, the agent will only try to maximize the reward in the next couple of steps.

Value Functions

Dependent on its policy π, there is a value function $v_\pi(s)$, giving the agent an estimate of how "good" it is to perform a certain action in the current state. So $v_\pi(s)$ is the expected discounted return when starting in state s and behaving according to policy π thereafter:

$$v_\pi(s) = \mathbb{E}_\pi \left\{ R_t | s_t = s \right\} = \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \Big| s_t = s \right\} \tag{4.3}$$

Similarly define the *action-value function* q^π as:

$$q^\pi(s,a) = \mathbb{E}_\pi \left\{ R_t | s_t = s, a_t = a \right\} = \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \Big| s_t = s, a_t = a \right\} \tag{4.4}$$

which gives the expected discounted return when starting in state s, taking action a, and then follow policy π. Both v^π and q^π are estimated from experience.
An important property of value functions is, that they can be formulated recursively. This results in the *Bellmann equation for* v^π, which expresses the

value of the current state s' in relationship to the values of the possible upcoming states. The successor states are weighted by its probability of occurring $P^a_{s,s'}$ and the expected reward $R^a_{s,s'}$:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi \left\{ R_t | s_t = s \right\} && (4.5)\\
&= \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \Big| s_t = s \right\} && (4.6)\\
&= \mathbb{E}_\pi \left\{ r_{t+1} + \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \Big| s_t = s \right\} && (4.7)\\
&= \sum_a \pi(s,a) \sum_{s'} P^a_{s,s'} \left[R^a_{s,s'} + \gamma\, \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \Big| s_{t+1} = s' \right\} \right] && (4.8)\\
&= \sum_a \pi(s,a) \sum_{s'} P^a_{s,s'} + [R^a_{s,s'} + \gamma v^\pi(s')] && (4.9)
\end{aligned}
$$

So solving a reinforcement learning problem means finding the optimal policy π^*. With the help of the value function we can formulate what an optimal policy is and how a policy π is better than another policy π':

$$\pi \geq \pi' \iff v^\pi(s) \geq v^{\pi'}(s) \tag{4.10}$$

We say it is better or equal if the expected return is better or equal for all states. We denote all optimal policies by π^* (there might be more than one) and they all have the same optimal value function v^* and the same optimal action-value function q^*:

$$
\begin{aligned}
v^*(s) &= \max_\pi v^\pi(s) \quad \forall s \in S && (4.11)\\
q^*(s,a) &= \max_\pi q^\pi(s,a) \quad \forall s \in S, a \in A && (4.12)\\
&= \mathbb{E}[r_{t+1} + \gamma v^*(s_{t+1}) | s_t = s, a_t = a] && (4.13)
\end{aligned}
$$

Using this, we can derive the Bellman optimality equations for $v^*(s)$ and $q^*(s,a)$:

$$v^*(s) = \max_{a \in A(s)} q^{\pi^*}(s,a) \tag{4.14}$$

$$= \max_a \mathbb{E}_{\pi^*} \left\{ R_t | s_t = s, a_t = a \right\} \tag{4.15}$$

$$= \max_a \mathbb{E}_{\pi^*} \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\} \tag{4.16}$$

$$= \max_a \mathbb{E}_{\pi^*} \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \middle| s_t = s, a_t = a \right\} \tag{4.17}$$

$$= \max_a \mathbb{E}_{\pi^*} \left\{ r_{t+1} + \gamma v^*(s_{t+1}) | s_t = s, a_t = a \right\} \tag{4.18}$$

$$= \max_a \sum_{s'} P^a_{s,s'} [R^a_{s,s'} + \gamma v^*(s')] \tag{4.19}$$

$$q^*(s,a) = \sum_{s'} P^a_{s,s'} [R^a_{s,s'} + \gamma \max_{a'} q^*(s',a')] \tag{4.20}$$

These equations show, that the value of a state under an optimal policy has to be the same as the expected discounted return for the optimal action from that state.

Estimation Methods

The goal of an agent is now to estimate the best value function and find the optimal policy for a given problem. We confine ourselves to methods that do not need to have any additional knowledge of the environment, apart from the state and actions. A simple method, that converges towards the optimal policy, if the number of iterations is large enough, is a Monte Carlo method. Usually, there is a Q-table that has to be learned. In this Q-table, we store the reward for each possible state-action combination. In the Monte Carlo method, these rewards are averaged over all episodes. Every time the agent visits a certain state and takes a certain action, the corresponding entry is updated with the new reward. For this, the method has to wait till the end of an episode to get the corresponding reward. Expressed in the term of the value function, a simple Monte Carlo update rule would look like this:

$$V(s_t) = V(s_t) + \alpha [R_t - V(s_t)] \tag{4.21}$$

with R_t being the actual return after time t and a weighting parameter α. The value R_t is only known after the episode. In this thesis however we will be focusing on *Temporal Difference Learning*. The main difference is that these methods only have to wait until the next time step to update their value function:

$$V(s_t) = V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{4.22}$$

as r_{t+1} is known and an estimate for all the future rewards is calculated. An important Temporal Difference method, which does not use the policy to update its Q-table is called Q-learning. Here the learned action-value function Q directly estimates the optimal action-value function Q^*:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \Big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \Big] \tag{4.23}$$

To force the agent to also try other actions, and not only exploit the actions where the reward is known, in ϵ percent of the time the agent has to explore other actions than the ones he already knows. This scheme is referred to as exploration vs. exploitation.

Deep Reinforcement Learning

In general we can split the ways to solve RL problems into two areas: the pure policy based methods that focus on directly finding a policy function and the pure value based methods that focus on optimizing the value function and derive the policy from it. While the *Deep Q-Network* method is value based, the *Actor-Critic* method is a hybrid of both. We will introduces these methods now.

For reinforcement learning problems with a large number of states and actions, it can be a problem to create a $Q-$ table as it would quickly reach computational limits in terms of storage on a computer. And visiting all states to update the table would also take a very long time. The deep reinforcement learning approach makes use of a neural network to approximate this Q table. In each iteration, the network then generates a probability for which action to take. Usually, there is a policy layer on top of it, which sometimes chooses a different action, to make exploration possible.

Deep Q-Network: First proposed by scientists at DeepMind in 2013 [Mni+13], the DQN algorithm showed great success in learning to playing Atari video games and beating human players in most of them [Mni+15]. Its main features are a deep neural network with some convolutional layers, which learn just from the shown frames of the video games. As it is important for deep learning that the input data is independent, the *experience replay* method was used to avoid correlation and stabilize the learning process. For this, we use a buffer of a predefined size, that holds a number of experienced state transitions with its actions, rewards, and the subsequent state. From this buffer, we randomly sample some data and feed them to the network. This way the network does not only see the correlated data from each episode but has enough uncorrelated data as well to learn from. The size of the buffer is a hyperparameter that can be tuned.
The second novel approach that the DQN algorithm took, is the fixed or *frozen target network*. Namely, the loss function in the DQN case is the squared error between a prediction and a target network with parameters θ and θ':

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

FIGURE 4.2: The DQN algorithm, taken from [Mni+15]

$$L(\theta_t) = \mathbb{E}[(Q_{target} - Q_{predict})^2] \tag{4.24}$$

with

$$Q_{target} = r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1}, \theta_t') \tag{4.25}$$

$$Q_{predict} = Q(s_t, a_t, \theta_t) \tag{4.26}$$

The structure of the networks is identical, only the parameters are different. At each time step the $Q_{predict}$ network is updated with the given loss function, but the Q_{target} only gets an update $n_{frequency}$ time steps (by setting the parameters $\theta' = \theta$). This is especially important to smooth oscillations in the training process, which are often induced by fast-changing Q-values [Mni+15]. Here the frequency of the target network updates is a tunable hyperparameter. The complete algorithm is shown in figure 4.2.

Proximal Policy Optimization: Policy Gradient methods focus on direct optimization of the policy π instead of the optimization of a value function. Measuring the performance with a scalar performance measure $J(\theta)$ with θ being the policies parameter, we can reformulate the reinforcement learning problem as finding:

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, J(\theta) \tag{4.27}$$

An advantage of Policy Gradient methods is that they can be applied to continuous action spaces. Also, the learning process is usually smoother, as the policy is incrementally updated in small steps (and not the value function, which can have a lot of fluctuation).

The main idea of Policy Gradients methods, based on the REINFORCE algorithm [Wil92] is to compute the expected reward of multiple trajectories.

A trajectory τ with length H is a sequence of states and actions and can be shorter than a full episode:

$$\tau = (s_0, a_0, s_1, a_1 \dots s_H, a_H) \tag{4.28}$$

The expected reward over all trajectories is computed by weighting the reward of each trajectory $R(\tau)$ with its occurrence probability $\pi_\tau(\theta)$. We use this to measure our performance:

$$J(\theta) = \mathbb{E}\left[\sum_\tau R(\tau)\pi_\theta(\tau)\right] \tag{4.29}$$

$$\nabla J(\theta) = \mathbb{E}\left[\sum_\tau R(\tau)\nabla \log \pi_\theta(\tau)\right] \tag{4.30}$$

Proof for the calculation of the derivative can be found in [SB18]. Instead of calculating the reward $R(\tau)$, in our case we will be using a *Generalized Advantage Estimation(GAE) A* [Sch+18]. The advantage uses the value function as a critic for our policy function, resulting in an *Actor-Critic architecture*. This means using the policy to compute an action a for a state s (actor) and then calculate the advantage according to the value function (critic). We define the loss as:

$$\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{4.31}$$

and the advantage as:

$$A_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{4.32}$$

with discount factor γ and GAE hyperparameter λ. Altogether, for our gradient (for a single trajectory) we get:

$$\nabla J(\theta) = \mathbb{E}\left[\sum_{t=0}^{H} A_t \nabla \log \pi_\theta(s_t, a_t)\right] \tag{4.33}$$

The optimization technique used in this case is the gradient ascent, which works in the opposite way as the gradient descent, by going along the direction of the steepest ascent of the gradient (with step size α):

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{4.34}$$

The PPO algorithm we will be using in this thesis was introduced by a team at OpenAI in 2017 [Sch+17a]. It is a very stable and simple implementation, making it possible to apply it to a lot of problems. The stability of this method is based on the concept of a *Trust region*, meaning that the step size of the gradient ascent is constrained in such a way, that it remains "trusted". For this we use a *Surrogate Advantage* L, that gives an estimate of how different a policy π_θ is in regard to another policy $\pi_{\theta_{old}}$:

$$L(\theta, \theta_{old}) = \mathbb{E}\left[\frac{\pi_\theta}{\pi_{\theta_{old}}} A\right] \tag{4.35}$$

While the *Trust Region Policy Optimization* algorithm [Sch+17b] uses the *Kullback Leibler Divergence* as penalty factor in the equation, the OpenAI-PPO algorithm uses a *Clipped Surrogate Objective*. It clips the ratio of the different policies $\frac{\pi_\theta}{\pi_{\theta old}}$ to lie in between $[1-\epsilon, 1+\epsilon]$. This results in the loss equation:

$$L_{clipped}(\theta) = \mathbb{E}_t\left[\min\left[\frac{\pi_\theta}{\pi_{\theta old}} A_t, clip(\frac{\pi_\theta}{\pi_{\theta old}}, 1-\epsilon, 1+\epsilon) A_t\right]\right] \tag{4.36}$$

4.2 Learning to Play ATARI Video Games

After having talked about the methodology and algorithms used in reinforcement learning, we now want to look at an approach for evaluating their performance. A common way of assessing the performance of agents is to train them to play video games. This, unlike a robot, for example, does not require an investment in another technical object apart from a computer and the underlying rules of a game are usually not too hard to understand. A framework often used is the *Arcade Learning Environment* [Bel+13], providing an interface to hundreds of Atari 2600 games. The main reason for using these kinds of games is that many of them are in $2D$, thus limiting the "visual space" and the set of input actions a player can perform is also rather small (18 possible actions). While these two features keep the games "simple", the genre of games is very diverse: it includes shooters, action-adventures, and sports games to only name a few. To transform the game environment into a reinforcement learning problem, the framework takes the virtual input of a joystick as an action, returns the current state with a resolution of 210×160 pixels and up to 60 frames per second and the current in-game score. It also communicates if an episode has terminated or not.

The main challenge for the reinforcement learning community was to develop a single architecture, that was capable of playing all (or a huge number) of the Atari games successfully. This model should not require any fine-tuning towards a specific kind of game, instead, it should be capable of general competency and learn just from raw input data [Bel+13]. The first promising results were delivered with the earlier mentioned DQN algorithm by DeepMind, achieving human-like behavior in six games [Mni+13]. The

serious breakthrough for this task however came two years later, when the agent achieved scores comparable to a human professional player in a set of 49 different games, all with the same network architecture and hyperparameters [Mni+15]. It incorporates the two novel approaches of the *frozen target network* and the *experience replay*, with a buffer size of ten million frames, to stabilize the training process. Interestingly, the agent performs very well on a broad spectrum of the games (even some $3D$ racing games) and is able to follow a long-term strategy for several games that require it (but not all).

The structure of the Q-Network used with the agent consists of three convolutional layers and two fully connected layers, that output a single neuron for each possible action. As shown in the paper [Mni+15], this "deep" structure is crucial to the success of the network and is therefore widely adopted now for DQN agents. There are however some other types of architecture, for example an Encoder-Decoder neural network [Oh+15], similar to the SegNet from the previous chapter. The application in this case is also Atari games, but the network is used more like a prediction network (replacing the original game), for the RL agent to learn from. The use of recurrent neural networks, that possess some kind of memory about the past, is also very popular to solve difficult environments that only return a reward after a long period of time. Nonetheless, results have shown, that even these complex environments can be solved up to a certain degree with basic fully connected convolutional networks [Ple+20].

4.3 Applications to Cellular Automata

To turn a cellular automaton like the Game of Life into a playable game, it is first necessary to let the player have some influence on the development. This is usually done by giving the agent the ability to choose one or more cells on the grid and change its state.

Several approaches have been tested in literature. In [Ear20] the author uses fractal convolutional neural networks in combination with actor-critic methods to play simple strategy-based games like SimCity and test their agent on the Game of Life as well, optimizing for the number of cells alive. To check the generalization ability, the scale of the game board is increased, which (in theory) should not be a big problem for an agent that relies only on local interactions. The results show, that the algorithm usually waits for all chaos to settle down (the point where the "Still Life" from Chapter 2 is achieved) and then fills the game board with cells. This is an indication, that taking controlled actions, while there is a lot of chaos happening might be very challenging for an RL agent.

In another paper [Luc+19], the authors use a similar approach to playing the Game of Life and training the agent to either extinguish or preserve as much life as possible. A neural network is used to learn the cell transitions, but the agent is a *Rolling Horizon Evolution* agent, based on evolutionary algorithms

(which we will not dive any further into here). Results show that for the agent to act successfully, it is crucial to have a good network, that is capable of learning the game rules.

Chapter 5

Implementation and Results

In this chapter, we will show the results of the numerical experiments we have performed. In the first part, we will focus on the prediction of future states of the Game of Life and some closely related cellular automata in two dimensions. In the second part, we will show some methods to control the behavior of 2D cellular automata.

5.1 Why Predict and Control Simple CA?

Applying Deep Learning to simple two-state cellular automata might sound like a "toy problem", given that the cell transition rules are usually very simple. Intuitively one might think that neural networks can be used for much more challenging tasks than the prediction of the next state of a CA. However, as we have already stated in Chapter 1, the behavior of CA is very complex, even if the underlying local rules are simple. In this thesis, we do not assume any knowledge about the rules, except that they are local and feed the neural network nothing more than images of the different states of the CA as input. The main task of the network is therefore to analyze this input data and extract the transition rules from it, which is a very ambitious task. Also, the behavior of these dynamical systems is highly chaotic, so controlling them is challenging as well. In the end, a network that was trained to understand and control a system just by observing is a very powerful tool and can be applied to several problems. For example, just filming a chemical reaction with a camera and feeding this data into a neural network could be enough to understand the concealed process behind this reaction. We therefore implemented the whole thing in the Unreal Engine as well, to be able to generate photorealistic images for the neural network to train on. Summarising one could say that the prediction and control of a simple system is not that simple at all.

5.2 The Unreal Environment

In this thesis, we want to use the Unreal Engine not only as a tool for the visualization of results (which is usually the last step in the development process) but instead also for the generation of data (which is usually the first step). This is motivated by the huge improvements Game Engines have made

in the last years to create digital environments that are in no way inferior to the real world. In fact, it is often hard to tell apart images that are real photos and that are generated (see for example https://vimeo.com/122314583). Furthermore, Game Engines have a Physics engine implemented, which can simulate certain effects in real-time. Using those features, it is possible to recreate a digital version of a scene in the real world, while being capable of easily modifying it in any way that is needed. Taking autonomous driving cars or flying drones as an example, we would need a lot of real-world scenery input data to train the underlying networks correctly. While this is very difficult and expensive to archive, recreating a model of a real road in the Unreal Engine takes way less time. Using Supercomputers, hundreds of thousands of images can be generated in the split of a second and then be used for the training of the autonomous driving neural network.

Unreal Engine for 2D cellular automata: Using the Unreal Engine to generate data of a cellular automaton in just two dimensions may seem like "using a sledge-hammer to crack a nut". It would also be possible to generate the data with a python script, so we are using a very powerful tool for a very simple problem. However with the applications we mentioned in the previous paragraph it is important to have a technical pipeline and workflow ready to use, which is what this thesis provides. For further studies a switch to three-dimensional CA could be made using this framework but the focus of this thesis are the deep learning methods developed to work with the two-dimensional data.

With this motivation we will proceed to the next step: we are also simulating what could be a process (based on cellular automata) from the real world in the Unreal Engine. For this, we have implemented a new version of the Game of Life (and similar CA) in C++ in the Unreal Engine from scratch, and call this version the *Unreal Life*. As cells, we are using 3D cubes, arranged in a 2D square grid. The implementation consists of two main classes:

CellActor: The Cell Actor class implements a single cell of the cellular automaton. It has a Static Mesh in the form of a cube and takes a black material if it is dead and a red material if it is alive.

MainActor: The Main Actor class is responsible for all computations that are based on the logic of cellular automation. The input variables it takes from the user are:

- **rows, columns:** the number of rows and number of columns define the size of the cell space. While we only use square grids, it is possible to modify these values for any rectangular girds.
- **neighborhood size:** the size of the Moore neighborhood (see Chapter 1) can be defined by the user.

As the Unreal Engine does not offer a native 2D array datatype, for the implementation of the cell space we had to use *TArrays* containing the Cell Actor objects, stacked into another *TArray*. Consequently, the number of objects in the first *TArray* is the number of the elements in each row, while the number of total *TArrays* of the first kind is the number of the columns.

The important public functions implemented are:

- ***GenerateNewGameboard(int seed):*** calling this method generates a new initial configuration for the cellular automaton, by randomly setting the state of 20% of the cells to alive, and the rest to dead. To be able to later retrieve the initial state, it is possible to set a certain random seed. For this task, we fix the size of the cell grid to 32×32.
- ***ComputeNextStep(int rule):*** this method computes the next iteration of CA, by applying the transition rules and iterating over all cells. Depending on the new state of a cell, a black or red skin is assigned to it. By passing an index to the function, a certain set of rules can be taken from the stack of possible rules.
- ***Action(int x_index, int y_index):*** calling this method and passing a row and columns index, we can manipulate the current state of the CA. The cell at the position of the indices changes its state from dead to alive. This artificial form of birth turns the CA into a one-player game, by enabling user input. After the cell is brought to life, the ComputeNextStep function is called.
- ***GenerateRules(int amount):*** with this function, it is possible to randomly generate reasonable transition rules for the CA. The rules are stored inside of two *TArrays*, that can be accessed with an index. In the next chapter, we will elaborate on how the rules are generated.

For the illumination of the scene, a single *SkyLight* is used.

5.2.1 UnrealCV

Now that we have created the Unreal Environment, we will take a look at how to extract images from the Unreal Engine. For this purpose, we make use of the UnrealCV plugin. It was introduced in 2017 and provides an interface between the Unreal Engine and a Python Script, enabling communication and the transfer of image data [al17]. After installing the plugin from the repository (`https://github.com/unrealcv/unrealcv/releases`), we can start the Unreal environment and connect to it from a Python script via a socket, see figure 5.1.

For the extraction of images, we have to add a *Fusion Camera Actor* to the UE environment. Setting its location in the world, as well as changing the scene illumination and taking the photo in the *.png* format can then be done from the Python script with predefined commands:

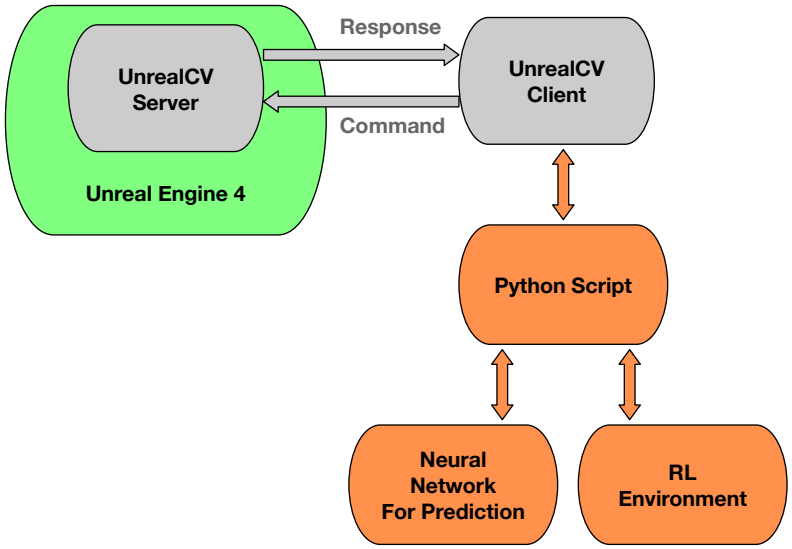

FIGURE 5.1: Communication between the Unreal Engine and a Python Script via UnrealCV, modified version from [al17]

```
vset /camera/[camera id]/location [x] [y] [z]
vget /camera/[camera id]/[viewmode] [image filename]
```

See http://docs.unrealcv.org/en/latest/reference/commands.html for all predefined commands. However, as we also want to call our MainActor environment functions from the client, we have to add some more commands, which is done by modifying the **ObjectHelper** class in the UnrealCV plugin folder, to include the following commands:

```
vset /action/new gameboard [seed]
vset /action/take action [x_index] [y_index]
vset /action/next iteration with rule [rule_index]
```

These functions map directly to the functions we have implemented in the MainActor class we described earlier.

5.2.2 Image Preprocessing

The camera returns the picture in the RGB color space, plus the Alpha channel as the fourth dimension (RGBA order). Since the alpha channel is only important for the transparency values, and these are not necessary for the CA (every cell is either alive or dead), we drop the fourth dimension. We colored the live cells in red (instead of white) to differentiate them better from the bright background. For the following tasks, we want to first map the images created in the Unreal Engine to a $2D$ representation of 1 and 0. While it is possible to directly use the raw image data as input to these networks, the resulting outcomes were very mixed, so we chose to implement this preprocessing step to stabilize the results. Taking this modular approach also makes debugging easier.

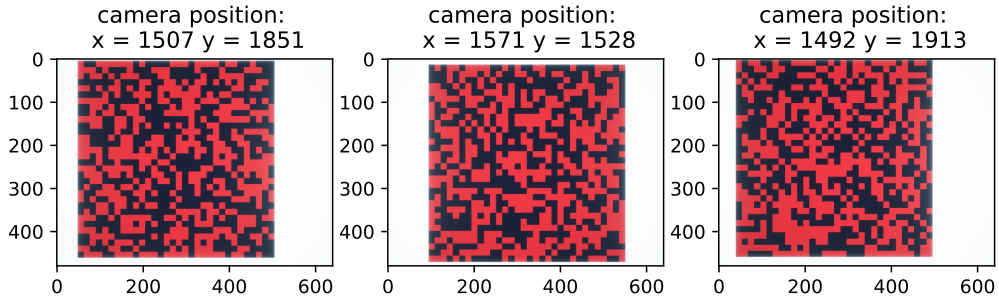

FIGURE 5.2: The red and black gameboard, generated with the Unreal Engine is shown from different camera positions.

For this pre-processing step, we have two options: we can use a neural network, that takes the large images from the Unreal Engine and transforms them into a smaller matrix. As we want to be as close to reality as possible, we have to account for the fact, that the camera that is used to take the pictures will have some changes to its position. Even if it is firmly mounted on a stand, very small perturbations could still occur, and our network should not be prone to such errors. Therefore the network is trained on several camera angles, to improve its generalization ability to address other positions, than the ones seen during training. Structure-wise such a model would look like the U-Net we have seen in Chapter 3. We have developed such a model (see Appendix) as a fully convolutional neural network and trained it on different camera angles, randomly moving the camera a little to the left and right, and up or down. While we do achieve good results ($\approx$ 94% accuracy on training and validation data), we never reach a perfect 100% accuracy. For this reason, the following experiments were done using the second option: an image mask, that directly maps the input image to a $2D$ matrix. This mapping is exact and has no errors.

5.3 Encoder-Decoder for Prediction of 2D Cellular Automata

When we say prediction in the connection with cellular automata, we mean the prediction of the next CA state(s) on the basis of the current or previous game CA state(s). As we have stated earlier, the rules for the Game of Life are very simple, yet can lead to very chaotic and unpredictable behavior. Once you know the rules of the Game, you can easily predict the next state, if you however do not know them, the prediction task becomes a very hard one. So the goal of any neural network we want to use as a predictor for the Game of Life should be to extract the underlying rules, just by looking at the generated states. We will however allow ourselves some knowledge about the cellular automata, that is that its interaction is all local and it therefore makes sense to use convolutional layers. As others have stated [SK20], for the Game of Life, the next n states can be predicted by a convolutional neural network with $2n + 1$ layers, as the rules can be summarized into a 3×3 convolutional

operation (although hard to predict). For this thesis, we want to go a step further and not only predict the original Game of Life but also some modified versions. The modifications we apply are:

- **Modification of the cell neighborhood:** we change the size of the Moore neighborhood that influences the status of the cell, so the locality becomes bigger
- **Modification of the cell transition rules:** in accordance with the increased cell neighborhood we also modify the number of alive neighbors for a cell to die or live. This number is chosen randomly from a plausible set of numbers.

The generation of rules is performed by randomly sampling integer numbers for the number of cells required for a cell to be born or to survive. We call this the Born/Stay notation (see https://www.conwaylife.com/wiki/Cellular_automaton#Rules), the classical Game of Life for example would be B3/S23. The numbers of course have to be plausible in the sense that for a cell neighborhood of size 3×3 the maximum of cells can be 8 and the minimum should be at least 1 to not immediately overpopulate the grid. Though one might expect that randomly sampling cell transition rules in this way, to result in a lot of "boring" rules (completely filled or completely empty grids), this did not happen in practice.

5.3.1 Tensorflow and Keras

For the implementation of the neural networks the famous deep learning library Tensorflow was used. Originally initiated by Google in 2015, it is an open source project that is the most frequently used framework, providing APIs in Python and C. With the versatile architecture it can be deployed effortlessly on multiple CPUs and GPUs. As the name says, the concept behind Tensorflow is the usage of tensors (multi-dimensional arrays), which are then connected by a Graph. Since September 2019 Tensorflow 2.0 is available (see https://www.tensorflow.org/), which offers tight integration with the python package Keras. Its main purpose is to provide an interface to the backend of Tensorflow to make the development process easier for humans, see https://keras.io/. Apart from TF, it also supports several other deep learning backends.

5.3.2 Training and Network Architecture

The goal of our prediction network is to extract the local cell transition rules from the seen image data and then apply them to the game board to get the next state of the CA. For this it makes sense to use the encoder-decoder structure (like in the SegNet and U-net). The job of the encoder part is to obtain the rules and the current state of the CA and pass them on to the latent space. Then the decoder should use this information and generate the next state of the CA. To account for the locality of the cell transitions and

Hyperparameter	Value
batch size	256
epochs	5000
learning rate	0.0001
optimizer	Adam
activation function	ReLu (sigmoid in last layer)
loss function	binary crossentropy
layer initializer	Glorot Uniform
layer regularizer	L2(0.0005)

TABLE 5.1: The hyperparameters used for training

because we are working with image data, we use convolutional layers. The layers are organized in several building blocks, with each block consisting of three convolutional layers. The first and last layer of each convolutional layer has a kernel size of 1×1, the middle layer of either 4×4 or 2×2. At the beginning and at the end we use an 8×8 sized kernel. For the other parameters see figure 5.3. To avoid over-fitting we implement Batch Normalization layers after each convolutional layer and use a Dropout layer in the latent space. The structure of the decoder is similar to the encoder but using deconvolution layers (the inverse of convolution). For the optimization, the Adam optimizer with a learning rate of $lr = 0.0001$ was selected. See table 5.1 for the other hyperparameters.

Skip Connections: As we are stacking several convolutional layers on top of each other, we also make use of the skip connections from the ResNet. This is to ensure that no information gets lost. To accommodate skip connections between layers with different data dimensions, we will not only use identity skip connections ($\mathcal{F}(x) + x = \mathcal{H}(x)$), but also convolutional skip connections ($\mathcal{F}(x) + conv(x) = \mathcal{H}(x)$). This provides an easy way to avoid using techniques like interpolation or zero padding.

To generate and evaluate the results, we will use a 3-step process:

- **1. Train the network:** The network is trained on some input/output data. The input data are three consecutive time steps of an initial configuration, stacked on top of each other. The output data is the fourth time step.
- **1.1. Validate the network:** During the training process, we constantly validate the performance of the network on a validation data set, different from the training set. Then we save the weights of the network that has the best performance (the smallest loss) on the validation data. This is to ensure that we do not suffer from an overfit in the training process.
- **2. Test the network:** We load the weights of the best performing model from the previous step and test it on a different set of data (the test set).

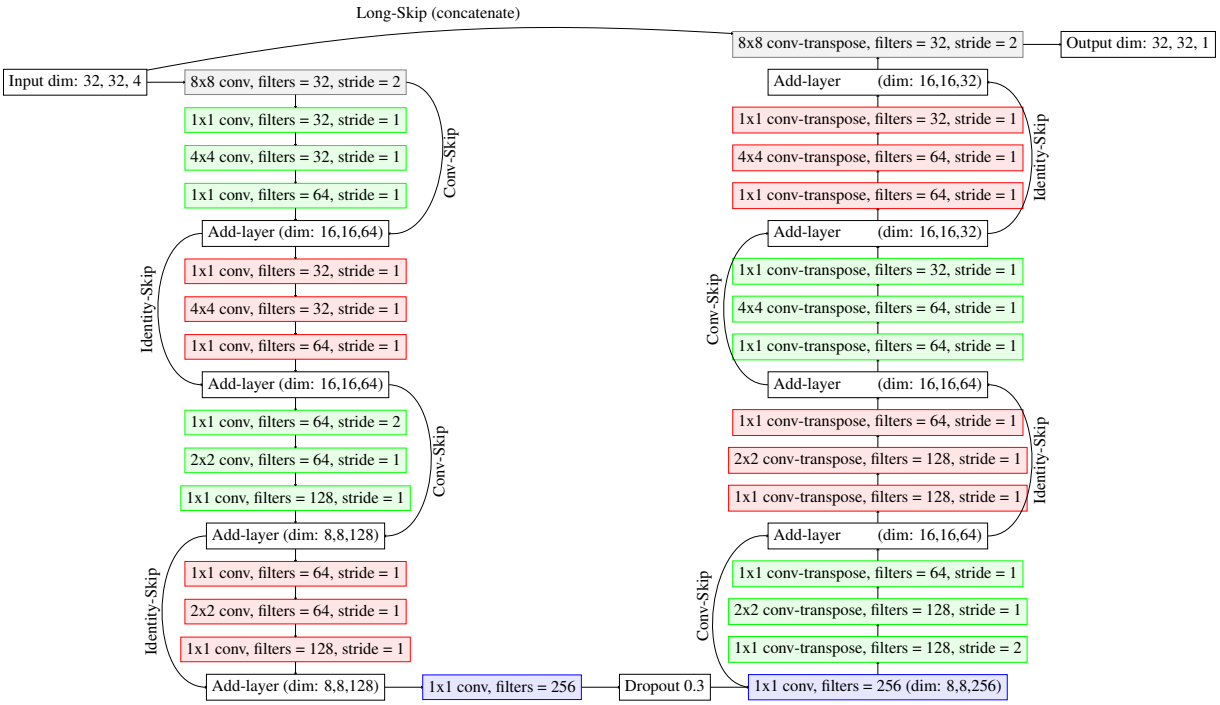

FIGURE 5.3: Architecture of the developed fully convolutional neural network. The encoder is on the left side, the decoder on the right side and in between is the latent space. Identity- and Convolutional-blocks are alternated and each block has a shortcut.

5.3.3 Expectation

As described before, the main goal of our network should be generalization ability. To check if this is achieved, we test for several levels of generalization. For this we have generated several sets of data:

- **Simple generalization:** we train the network on a randomly generated number of transition rules of neighborhood size 3×3, then validate on unseen initial configurations and test on more unseen initial configurations (generated with the same rules). If the network can extract the rules, we expect a very good accuracy for this task.

- **Level 1 generalization:** we train the network on a randomly generated number of transition rules of neighborhood sizes 3×3 and 5×5 and 7×7, then validate on unseen initial configurations and test on more unseen initial configurations (generated with the same rules). As the network has to learn the transition rules of different neighborhood sizes, this task is harder than the previous one, but if the network is able to extract these rules, we also expect good accuracy for this task.

- **Level 2 generalization:** we train the network on a randomly generated number of transition rules of neighborhood sizes 3×3 and 5×5 and 7×7, then validate on unseen initial configurations and test on more

unseen initial configurations (generated with different randomly generated rules). This task is a lot harder because the network has to adapt to a completely new set of rules in the "test" step.

- **Level 3 generalization:** we train the network on a randomly generated number of transition rules of several neighborhood sizes, then validate on unseen initial configurations (generated with the same randomly generated rules) and then test them on unseen generated rules from a different neighborhood size. This is by far the hardest task, as the network has not only to adapt to new rules, but also to new rules from different neighborhood sizes. This could be seen as a "Transfer Learning" task. We also train a Level 3 modified version where we put the size of the test neighborhood in between the sizes of the training neighborhoods, to see if this improves results.

See table 5.2 for a tabular overview of the data.

train data:

Level of generalisation	neighborhood sizes	rules per size	# of datasets
simple	3x3	100	100*100 trajectroies = 10000
Level 1	3x3, 5x5, 7x7	100	3*100*100 trajectroies = 30000
Level 2	3x3, 5x5, 7x7	100	3*100*100 trajectroies = 30000
Level 3	3x3, 5x5, 7x7	100	3*100*100 trajectroies = 30000
Level 3 modified	3x3, 5x5, 9x9	100	3*100*100 trajectroies = 30000

validation data:

level of generalisation	neighborhood sizes	rules per size	# of datasets	same rules as train data
simple	3x3	20	20*100 trajectories = 3000	yes
Level 1	3x3, 5x5, 7x7	20	3*20*100 trajectories = 6000	yes
Level 2	3x3, 5x5, 7x7	20	3*20*100 trajectories = 6000	yes
Level 3	3x3, 5x5, 7x7	20	3*20*100 trajectories = 6000	yes
Level 3 modified	3x3, 5x5, 9x9	20	3*20*100 trajectories = 6000	yes

test data:

level of generalisation	neighborhood sizes	rules per size	# of datasets	same rules as train- and val data
simple	3x3	10	10*100 trajectories = 1000	yes
Level 1	3x3, 5x5, 7x7	10	3*10*100 trajectories = 3000	yes
Level 2	3x3, 5x5, 7x7	10	3*10*100 trajectories = 3000	no
Level 3	9x9	30	30*100 trajectories = 3000	no
Level 3 modified	7x7	30	30*100 trajectories = 3000	no

TABLE 5.2: The train, validation and test data used for the network.

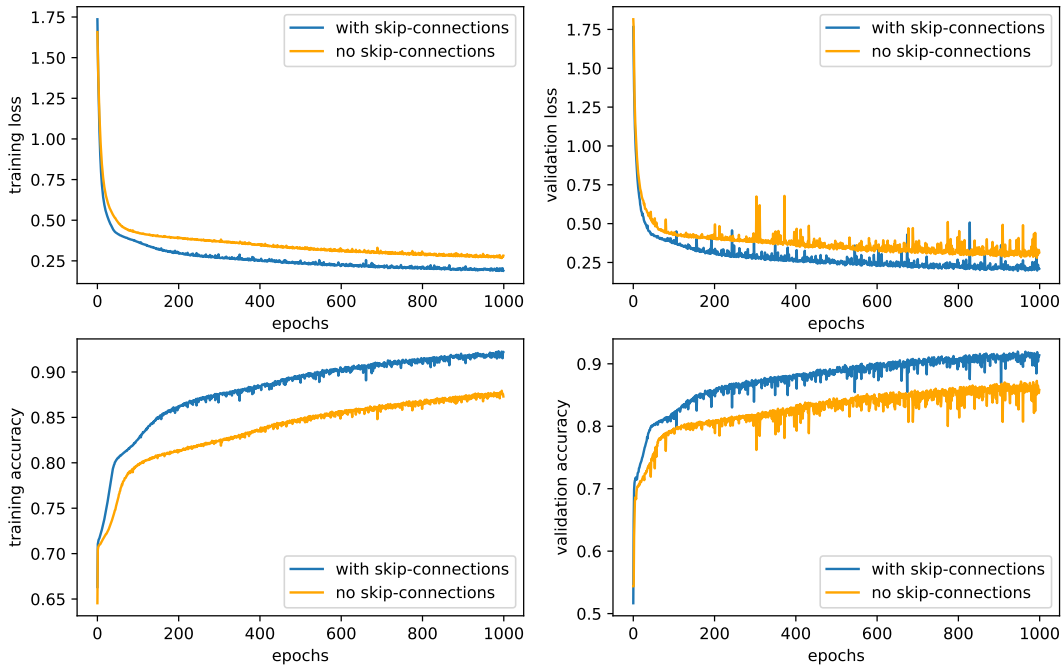

FIGURE 5.4: Comparison of training and validation results (for Level 1 generalization data) when enabling and disabling Skip Connections.

5.3.4 Results and Evaluation

Skip Connections: Comparing the performance of a neural network with and without skip connections, we can see in figure 5.4 that they play an important role in reducing the error. While they behave similarly during the first hundred epochs, the no-skip network suffers from the degradation problem (mentioned in Chapter 3) later on. Here the skip connections can show their strength in further reducing the error both in the training and validation set.

Simple generalization: As expected we do see very good results (see table 5.3) when it comes to these tasks, with the accuracy averaging over 98% across all datasets. The network seems to be very capable of learning the shown rules with some minor errors. For a Moore neighborhood of 3 there are $3 \times 3 = 9$ cells that have two possible states, so we count $2^8 = 512$ possible cell transition rules. This "low" number can apparently easily be learned. Therefore we can take the calculated results as an upper boundary and best-case reference for all other levels of generalization. See 5.3 for results.

Level 1 generalization: For the Level 1 generalization, we can see that the overall accuracy decreases, as the network has to adapt to a greater number of and more diverse transition rules. However, the results are still very good with an averaged accuracy of $92 - 93\%$ over all data sets. This shows the ability of the network to really learn a good number of the given rules, and apply them to new initial configurations, even for different neighborhood sizes. If

	train		val		test	
	loss	acc	loss	acc	loss	acc
mean	0,0729	98,74 %	0,0820	98,44%	0,0800	98,49%
standard deviation	0,0421	1,15 %	0,0435	1,20%	0,0390	1,09%

TABLE 5.3: Simple generalization results averaged over 5 runs.

we predict all cells to be dead ($= 0$), we get a baseline accuracy of $\approx 51\%$. The number of possible transition rules becomes incredibly large ($2^{(5\times5)}$ and $2^{(7\times7)}$), so the drop in accuracy compared to the simple generalization makes sense. Evaluating which neighborhood sizes influence the error the most, we split the validation and test data by neighborhood sizes again: about 98% accuracy is achieved on the 3×3 data, 93% on the 5×5 data, and 88% on the 7×7 data. This also supports the argument that the larger the neighborhood size, the harder to predict.

	train		val		test	
	loss	acc	loss	acc	loss	acc
mean	0,1808	93,00 %	0,2041	92,16 %	0,2049	92,19%
standard deviation	0,0206	1,00 %	0,0232	1,03 %	0,0250	1,12%

TABLE 5.4: Level 1 generalization results averaged over 5 runs.

Level 2 generalization: While the results for the training and validation dataset stay the same (table 5.5), we can see that the test loss and accuracy deteriorate down to 77%. Apparently the network, although being shown a diverse set of rules of a similar kind, is not able to adapt very well to completely unseen rules. In real terms this means, that such a network would not be able to understand and extract completely new phenomena just by observing them, but instead would have to be shown (and trained on) the exact same phenomena to detect them. However the results for the test set are still a lot better than the accuracy of the baseline, so it does provide some advantage.

	train		val		test	
	loss	acc	loss	acc	loss	acc
mean	0,1669	93,65%	0,1923	92,76%	0,8514	77,65%
standard deviation	0,0048	0,40%	0,0054	0,465	0,0799	1,32%

TABLE 5.5: Level 2 generalization results averaged over 5 runs.
Baseline accuracy is still $\approx 51\%$

Level 3 generalization: In these results (table 5.6) we observe a similar effect as for the Level 2 generalization, with a huge drop in accuracy for the test set. The loss is even larger than in the previous generalization step because the rules in the test set now are not only completely new but also based on a different neighborhood size. While still a lot better than the baseline accuracy, this task is apparently too hard for the network. It is however interesting to observe, that there is a way to increase its robustness by about 7%: as the results in table 5.7 show, we can increase the test outcome if we train on neighborhood sizes 9×9, that are larger than the test neighborhood size 7×7. As some cell transition rules in the 9×9 dataset will probably be very similar to the ones in the 7×7 one, the network is able to "interpolate" them.

	train		val		test	
	loss	acc	loss	acc	loss	acc
mean	0,1903	92,63%	0,2183	91,50%	0,7646	70,50%
standard deviation	0,0340	1,35%	0,0329	1,20%	0,0465	1,77%

TABLE 5.6: Level 3 generalization results averaged over 5 runs.

	train		val		test	
	loss	acc	loss	acc	loss	acc
mean	0,2113	91,16%	0,2336	90,40%	0,5650	77,49%
standard deviation	0,0206	1,03%	0,0173	0,83%	0,0970	4,31%

TABLE 5.7: Level 3 modified generalization results averaged over 5 runs.

While a qualitative analysis of the results is hard to perform since the underlying patterns are hard to spot with the naked eye, we want to however show some interesting observations we made when looking at the rules where the network had he highest accuracy: It seems that "static" rules, that do not exhibit many changes in between time steps are very easy to predict. Looking at figure 5.5 we can see that the images look very similar and consequential most of the cells are correctly classified. It makes sense that a rule like this is very easy to extract for the neural network, as it can just copy the last iteration of the data and only make minor modifications to it. Another set of rules that the Encoder-Decoder can successfully predict are what we call "alternating" rules. These kinds of functions basically just switch the state of all cells from iteration to iteration, as all alive cells die from overpopulation and all dead cells are reborn. See figure 5.6 and 5.7 for two examples. In this case, it is also logical that the error rate is so low because taking the last iteration and just switching it is also rather easy. Still, for the given rules the alternating behavior is not valid for all cells and when taking a closer look at the pictures, we can spot multiple of these cells, that the network also predicted right.

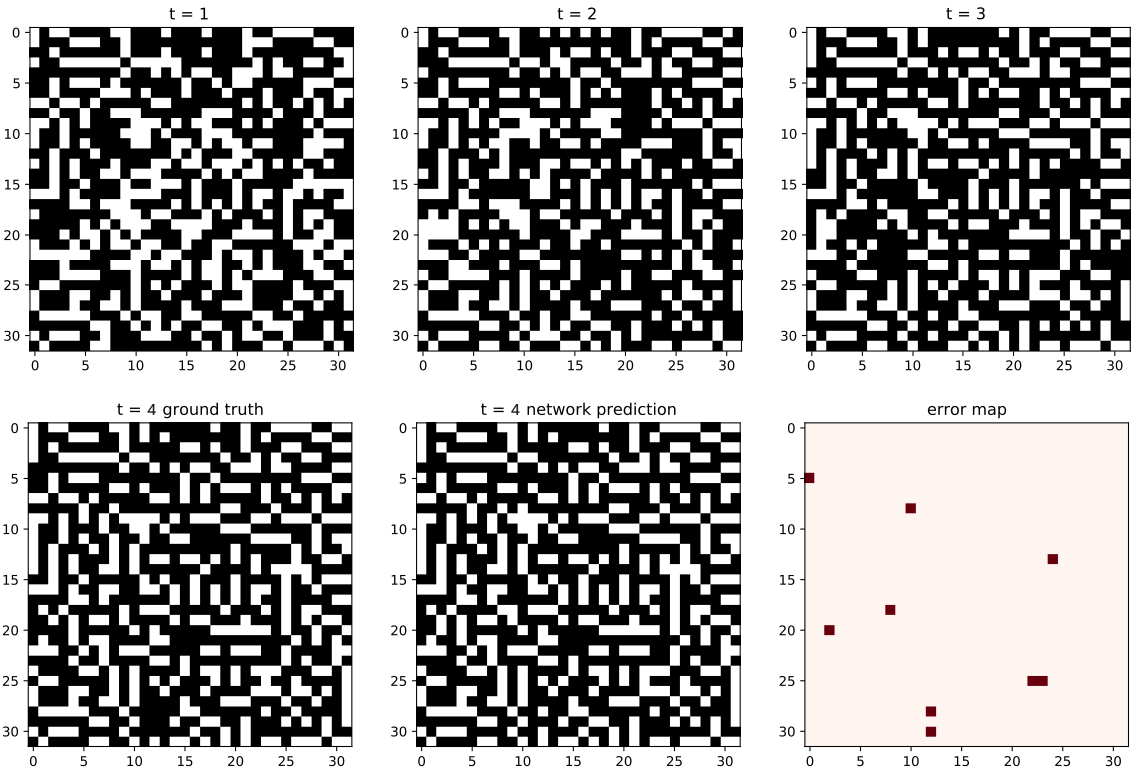

FIGURE 5.5: Trajectory of a "static" rule, that does not evolve much in between the time steps. The error map shows the difference between ground truth and prediction.

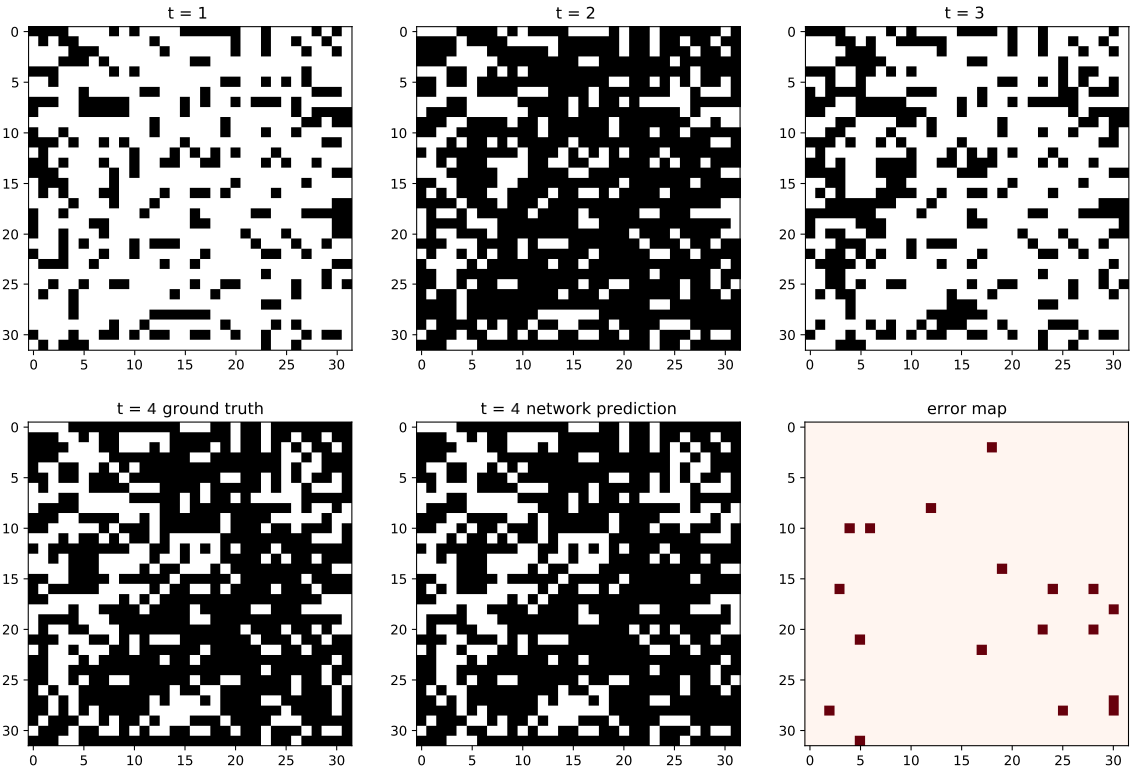

FIGURE 5.6: Trajectory of a "alternating" rule. What was alive dies in the next time step and vice versa.

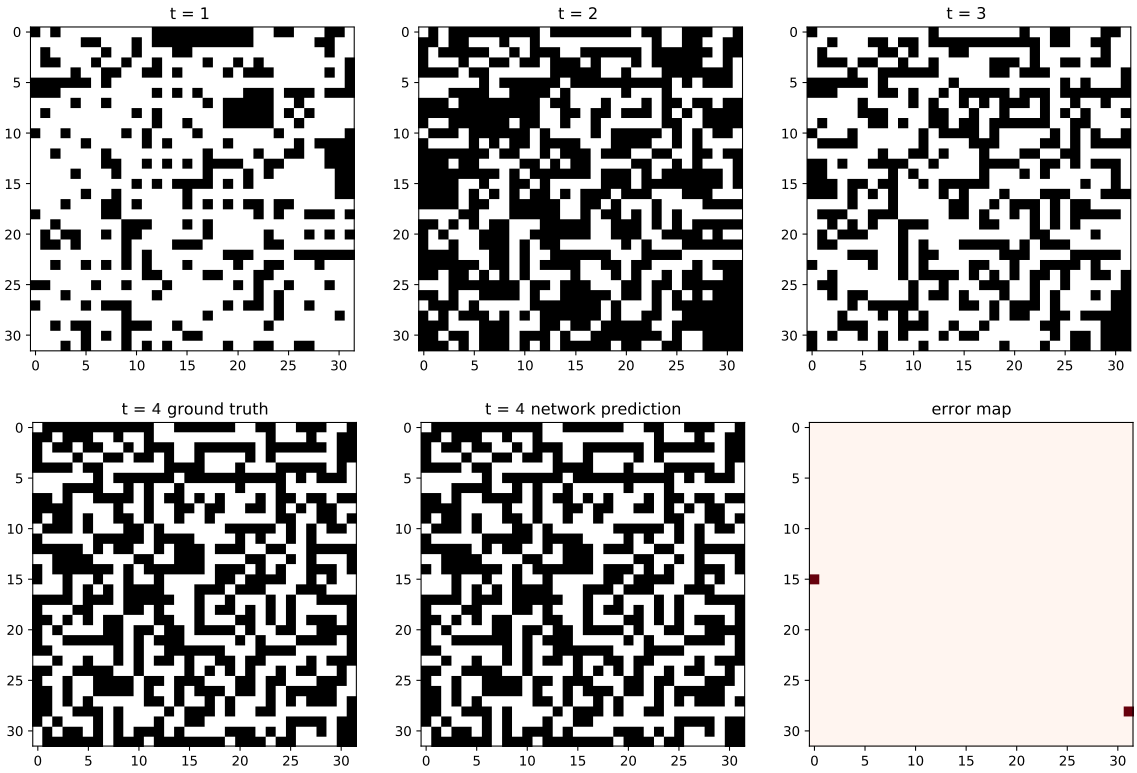

FIGURE 5.7: Trajectory of an "alternating" rule, that exhibits maze-creating behaviour.

Summary: Overall we can say that combining several convolutional layers for the extracting of rules with the structure of the ResNet and U-Net to avoid degradation, is a promising approach for predicting the consecutive iteration of a cellular automaton. The developed network was able to successfully learn a lot of different rules and even to generalize them to some extend.

5.4 RL Algorithms for Controlling a Cellular Automaton

In this section we will focus on controlling a cellular automaton with the help of the reinforcement learning algorithms, we presented earlier. The term "control" in this case means working towards some goal or final state of the CA, and influence its evolution over time to reach this goal faster or better. The final state we want to arrive at in this thesis is the so-called "still life", from chapter 1. It is the destination point in the progression of a CA, where all changes are done and the cells have taken their ultimate state. Once this state is achieved, it cannot be left, as the transition rules are the same for each time step. Therefore it is a very interesting phenomenon to observe and study. In order to do this, we will explore possible reinforcement learning agents and the actions they can take.

5.4.1 OpenAI Gym and RLlib

The tools used for the reinforcement learning part of this thesis are OpenAI Gym and RLLib. OpenAI Gym is a framework specially designed for the development and comparison of RL algorithms, see `https://gym.openai.com/`. The most important feature is the implementation of an *Environment* class that encapsulates the dynamics of an RL environment. There are lots of predefined environments, but it is possible to build a custom one, by implementing the following methods:

- `Env.init()`: this method initializes all important variables. Here we have to define the `action space`, for describing all valid actions and the `observation space` for describing the valid states of the environment. We can choose between a `Discrete` and a `Box` (continuous) space.
- `Env.reset()`: for resetting the environment to its initial state and start a new episode.
- `Env.render()`: for rendering the current state of the environment for debugging and visualization purposes.
- `Env.step(action)`: the probably most important function, taking the chosen action as an input and computing the subsequent state of the environment. This is returned as the `obs` value. The other return values are the `reward` we got from the action, a `done` parameter to indicate if an episode has terminated and an `info` parameter for debugging usage.

RLlib is an open source library that offers implementations of several state-of-the-art RL-algorithms. As it is part of Ray, a framework for building scaleable, distributed applications these algorithms are easy to scale as well. The behavior of the RL agent is defined by policies, which are the core part of RLlib and can be modified by using a deep learning library like Tensorflow. Overall, in combination with OpenAI-Gym, it provides a high-performing,

easy to implement solution to test and improve RL agents. In this thesis, we will be using the DQN and PPO algorithms.

5.4.2 Training Architecture

To enable reinforcement learning, we first have to turn our (zero-player) cellular automata into a one-player game. For this we just perform a simple modification: in each iteration, the player can change the state of a single cell on the grid (we will perform a comparison between an agent that can only kill cells and one that can only birth cells). Our expectation is for the agent to influence the behavior of the CA through this action, so it can achieve the earlier mentioned Still Life faster. An episode starts with an initial observation, then the agent takes its action and the next time step is calculated. One episode ends if Still Life (success) is achieved ($obs_{t-1} = obs_t$) or 20 time steps are reached (failure). We try two different reward functions:

- **sparse reward:** returns a reward of 1 at the end of the episode if it was successfully, otherwise a -1 and during the episode a reward of 0.
- **non-sparse reward:** calculates how different the previous observation and the current are, by summation over all different cells. This sum is multiplied by -1 and returned to the agent after each action. In the case of success, a compensation of 1000 is granted. The goal of this reward function is to guide the agent in the right direction, as for the Still Life, the difference of the game boards would be zero ($obs_{t-1} = obs_t$). It is worth mentioning, that because of the chaotic behavior of CA, this might not always point in the right direction.

As reinforcement learning is a more complex task than "just" prediction we have to limit two parameters for it: First we reduce the game board size to 10×10 cells. This is necessary to limit the action space to $10 \times 10 = 100$ possible actions. This is already very large but should be enough for the algorithms to handle. While there are ways to handle even bigger action spaces (see for example [Dul+16] and [Gau+19]) this is not the focus of this thesis. Second, we will limit the neighborhood sizes (and therefore the number of rules) as well, to only the classical 3×3 Moore neighborhood. On a small grid of only 100 cells, interactions between cells would no longer be considered "local", if we would use larger neighborhoods. The integration of the *Unreal Life* with OpenAI-Gym environment is straightforward: the `Env.init()` function calls the `GenerateNewGameboard()` function in the Unreal Engine, and the `Env.step()` functions calls the `Action()` function in the Unreal Engine.

The neural network we use to approximate the Q-table and the policy function with is shown in figure 5.8. Especially for the DQN agent, we needed to modify the hyperparameters (big experience buffer, less frequent update of the target network) to facilitate a stable learning process. We also only use

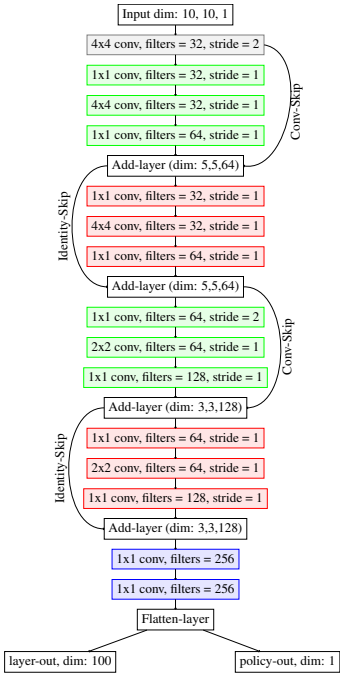

FIGURE 5.8: The neural network used as a function approximator in the RL agents (here shown for the PPO with two outputs). It is just the Encoder part of the network shown earlier, with some dimension modifications

one time step as input for the neural network and do not stack them (like in the prediction task).
A split in training and test data is also important in reinforcement learning to evaluate the performance and generalization ability on unseen inputs. To do so we will split our data in the following way:

- **train data:** we use 100 different initial configurations for the network to train on. While this may seem to be a very small number, there is still enough data to be learned on, as with every different step the agent takes, the trajectory changes. Also, our experiments did not converge when using larger numbers, as it made learning too hard.

- **test data 1:** to test how well the agent does on configurations that are unseen, but in some way still similar to the training data, we make *small perturbations* on the game boards used in the training set. This means changing the value of only one or two cells.

- **test data 2:** this set contains completely unseen and randomly generated initial configurations, that have no (forced) similarity.

We will be using the DQN and the PPO algorithm introduced earlier and compare the results. Both agents are trained for twelve million frames. We observed this to be the optimal training time, as results got very noisy afterwards. In literature, agents are trained between 10 million and 50 million frames [Mni+13]. As cellular automata exhibit chaotic behavior, we expect

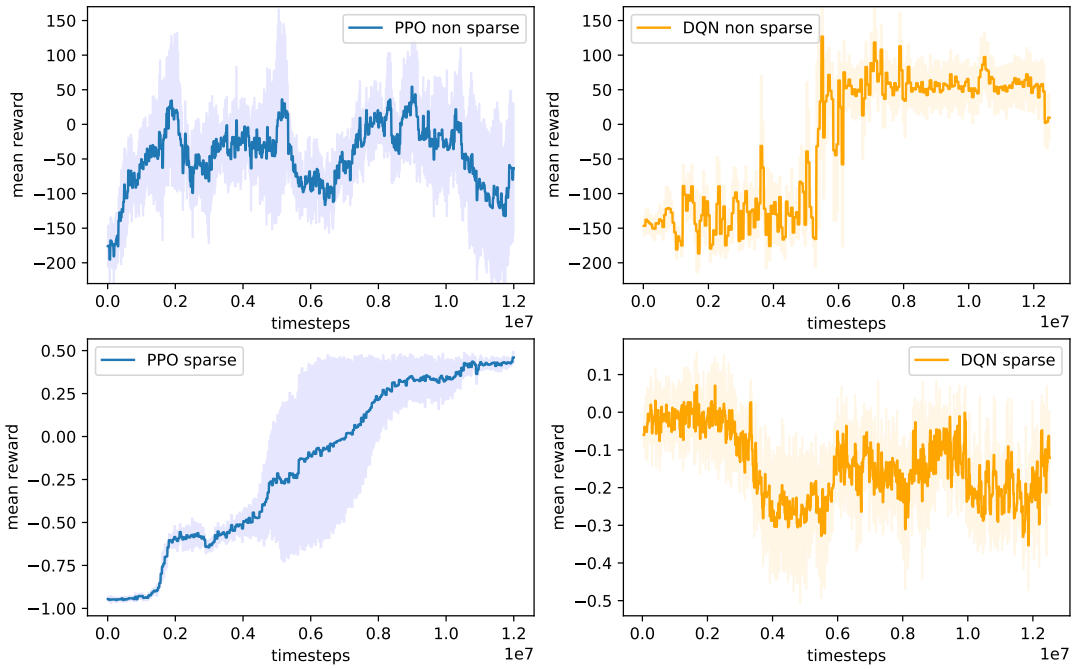

FIGURE 5.9: The results of the killer agent plotted over the number of frames averaged over 5 runs and with their standard deviation. The sparse reward are bounded by -1 and 1.

to see noisy results in general. The Proximal Policy Optimization algorithm with its small steps in the gradient ascent will probably be a little more stable, while the Deep-Q-Network with its less regular target function updates will oscillate more.

5.4.3 Results and Evaluation

Killer Agent: For the first experiment we let the agent kill one cell in each time step. The results for the DQN and PPO agent are shown in figure 5.9. At first glance, we can see that the plots are indeed oscillating a lot. We can however say that we see a very bad performance for the sparse DQN agent, as the mean rewards drops over time. The non-sparse -DQN and the non-sparse-PPO both do overall improve over time, but the heavy oscillations make the results for the non-sparse-PPO unreliable. The non-sparse-DQN at least continuously stays positive after rising above zero. For the sparse-PPO we do see the typical steep increase (from -1 to ≈ -0.6) in the beginning, once the agents acquire some understanding of the environment. Out of all four variations, it is the best performing one.

When looking at the number of iterations the agent needs to reach the Still Life, a comparison of the respective graphs in figure 5.10 (only showing the two best performing agents) to the ones in figure 5.9 show an inverse correlation. For the sparse reward function, this is expected, as the only way for the agent to increase its reward, is reaching the goal. The non-sparse-DQN-agent however could also increase his reward (or put another way: less decrease his

agent	small perturbations		unseen configurations	
	faster than random agent	reached goal	faster than random agent	reached goal
PPO (sparse)	98,8 %	72,2 %	97,6 %	56 %
DQN (non-sparse)	85 %	67,1 %	75 %	48,2 %

TABLE 5.8: Generalization performance of the killer agent. "Faster than random agent" means the algorithm needed fewer time steps to achieve Still Life and "reached goal" indicates how often the agent was able to achieve it in 20 iterations or less

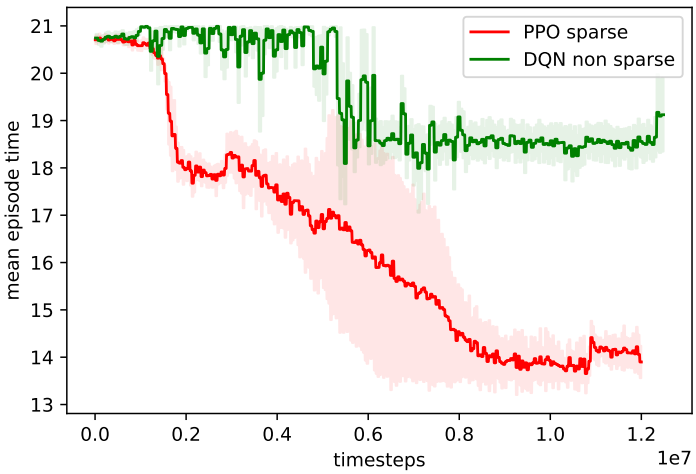

FIGURE 5.10: Number of time steps the killer agents took to finish an episode. If the Still Life is not achieved after 20 iterations, the episode stops.

accumulated reward, as he only gets negative ones during the iteration) by limiting the number of changing cells. As the mean average time also appears to be inversely correlated to the mean reward, we can conclude from this that the non-sparse reward function is a good indicator towards the Still Life. Nonetheless, overall the sparse reward function yields better results.

This is also the case for the generalization results shown in table 5.8. The PPO agent is almost in all cases faster than a random agent in achieving the Still Life. The random agent is our best baseline model, as there is no way to finding out the optimal trajectory towards Still Life (except for trying every possible combination). Overall we see better performance on the first test set (small perturbations) which supports our hypothesis, that the agent can better control a similar environment than one he has not seen before. Both agents were able to achieve the Still Life in less than 20 time steps in $\approx 70\%$ of the cases. On the completely unseen data, this rate drops to 48% for the DQN agent, indicating that generalization to new data is very hard.

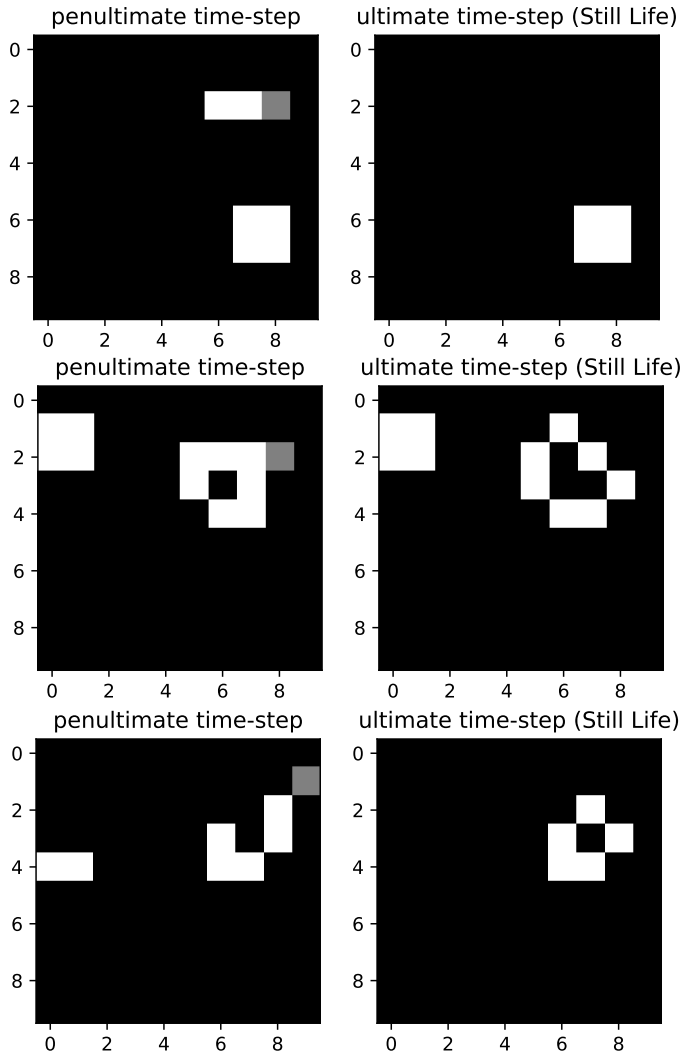

FIGURE 5.11: Top: The agent extinguishes what would become a Blinker (action is shown in grey). The Block is already Still Life. Middle: The agent generating a Loaf and a Block. Bottom: The agent generating a Ship.

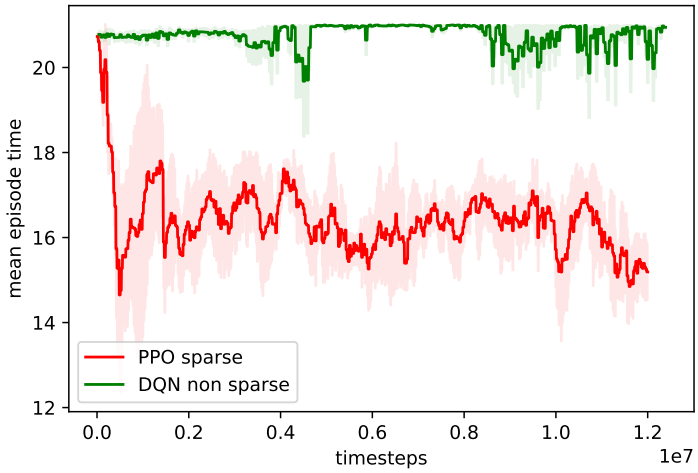

FIGURE 5.12: Number of time steps the rebirth agents took to finish an episode.

When looking at some patterns the agents generate, we see the familiar figures we have already seen in Chapter 2, see figure 5.11. As the agents (due to their architecture) can only look one iteration ahead, we only plot the penultimate and ultimate state of the trajectory. We see that mostly the simpler Still Life patterns are generated, and not the more complex ones (like the Beehive, see figure 2.4).

Rebirth agent: In the first experiment, we only gave the agent the possibility to extinguish life. We now want to check how good the opposite would work: giving an agent the possibility to rebirth a dead cell. Figure 5.12 shows the time the algorithms took to finish an episode. While the PPO agent actually seems to learn a lot faster in the beginning, in the end he does not achieve better results than the killer agent. The DQN agent seems to fail at learning. The reason for the killer agent yielding better results can be found when looking at the Still Life patterns shown in Chapter 2 (see figure 2.4). These patterns are very small and local, so it makes sense, that the agent needs *less* life on the game board, instead of more. Rebirthing dead cells probably only leads to more chaos and is thus counterproductive.

Continuous reward function: The Proximal Policy Optimization agent can also facilitate a continuous action space, instead of a discrete one. The advantage of continuous action spaces is, that the agent has some understanding of the arrangement of actions. Here we map the interval of $[0, 99] \in \mathbb{R}$ to the 10×10 grid of cells (rounding the real number to integers). By design this is not possible with the DQN agent (except with a couple of modifications [Lil+19]). The results of the PPO agent can be found in figure 5.13. After a good start, there is a sudden drop down to the minimum of -1 and it continues to stay in this range. We could not verify if this is due to some numerical issues within the agent (e.g. issues with weights being too small), but have to

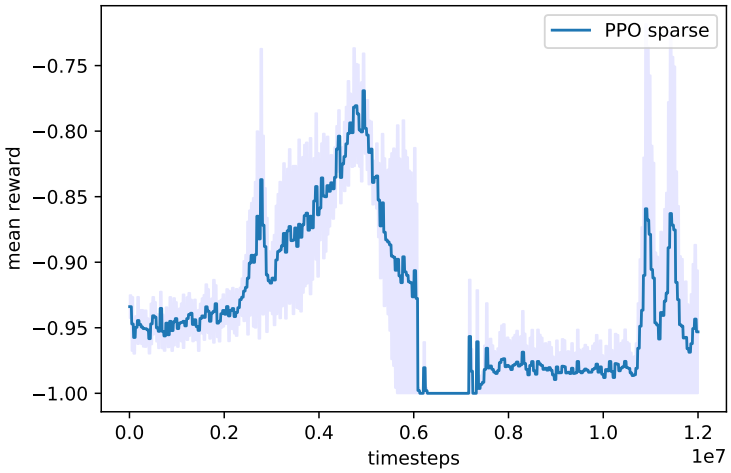

FIGURE 5.13: Mean reward achieved by the continuous PPO agent (with standard deviation).

conclude that in this case transforming a discrete environment (like the cell grid is) into a continuous one did not work.

Learning multiple rules: As the last step we want to check how well the agent is able to control not just the Game of Life rules, but also some similar 3×3 rules. For this, we (similar to the prediction task) randomly generate ten different rules and assign each rule to ten different initial configurations (so we still have 100 different initial configurations in total). These assignments are fixed, so the agent has the possibility to deduce the game rules from seeing the game board. The plots in figure 5.14 show that this is however too hard for both agents, and both fail at learning. The lack of learning progress can be explained by the sheer complexity of the task, that even after showing the agents about thirty million frames they are unable to extract any rules.

Summary: Overall we could see that both, the PPO and the DQN agent were able to reach their goal: the creation of Still Life in less than 20 iterations and thus control the Game of Life. However, the generalization remained very limited, both in regard to unseen initial configuration and to other rules.

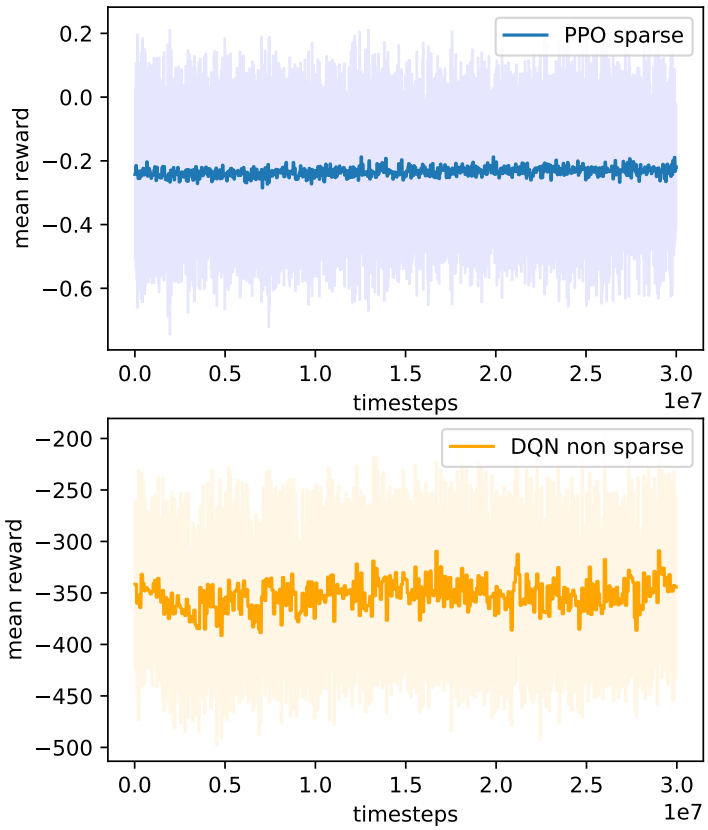

FIGURE 5.14: Mean reward for simultaneously training with 10 different cell transition rules.

Chapter 6

Conclusion and Outlook

Starting out with the motivation to have a neural network extract the underlying rules of processes just by observing them, we quickly decided to choose cellular automata as the objects to study. Two tasks were posed: the passive prediction, which requires watching the cellular automata, and the active control task, which also involves interaction with them. After giving a short introduction to cellular automata at the beginning of this thesis, we concentrated on the most famous of all: the Game of Life. The key takeaway was that these kinds of local dynamical systems sometimes exhibit chaotic behavior and therefore their outcome cannot be predicted. However, some patterns, like the Still Life patterns in the Game of Life, do appear more regularly and can be studied.

As a next step, we took a look at the main methods being used: neural networks and deep learning. For the work with images convolutional neural networks are well suited and recent advances in the field of image recognition have proven that these can be made very deep if "skip connections" are applied. These act as shortcuts between layers and result in the residual formulation of a problem. In general, a U-shaped Encoder-Decoder structure is the standard approach when working with images where the input and output dimensions are the same. For the control task, we also looked at deep reinforcement learning agents, like the Deep Q-Network and the Proximal Policy Optimization algorithms, which have shown great success when it comes to playing video games. On the whole, the literature on the intersection of deep learning and cellular automata was published only quite recently.

In the implementation part of this thesis, we have developed a complete framework for the completion of the prediction and control tasks. With a custom environment, we used the Unreal Engine as a data generator and processed this data in combination with tools like UnrealCV. Although we are using 3D cubes as cells in the Unreal Engine, we are still only looking at a two-dimensional grid. At this point, we are not using the full power the UE, offers which originally lies in its ability to render photo-realistic 3D environments. However three-dimensional cellular automata have a big problem: not all cells are visible from all camera-angles (the cells in the front hide the cells in the back). But to succeed in the given deep learning tasks, we need to know the position of every cell. To tackle this issue, a future version could

focus on predicting a 3D-version of *Unreal Life* by observing it from multiple viewing angles and using multiple images as input for the neural network.

For our task in two dimensions: Implementing a deep convolutional neural network following the Encoder-Decoder structure with skip connections, we were able to achieve very good results for the prediction of the next state of several cellular automata. Generating training data with several hundred different rules, the neural network managed to not only learn a huge part of these rules, but could to some extend also interpolate to completely new and unseen rules. Future research could focus more on which of these rules can be learned (and which not) and why. Overall, it would also be interesting to study if networks are able to make at least some predictions about the long-term evolution of a cellular automaton.

For the control task, we implemented an environment in OpenAI-Gym and connected it to the Unreal Engine. Training agents that were capable of extinguishing or creating life, implemented to achieve the goal of a stable, static environment, worked well on the training data. Both the PPO and the DQN agent were able to create a static environment in a given number of iterations, thus controlling the Game of Life. Generalization, especially in regard to different rules, resulted in poor performance. This can be explained by chaotic behaving systems being hard to control in general. Especially in contrast to video games, where a bad action might be compensated by two good actions, cellular automata stand out, as a bad action here can send us down a completely different trajectory.

The inability to generalize to different rule sets is probably largely due to the structure of the network we use. As it only embodies feedforward layers, there is no way for the latent space to access any information that lies in the past: it only sees the current state and knows nothing about the earlier trajectory. It cannot send back information to an earlier layer, just pass it forward. Consequently, there is no way for the network to know which rule it should apply to a game board if it is shown just the current state. And it is also not able to plan ahead into the future, apart from the next time step. Future research in this area will need to focus on developing long-term strategies, for example, with the help of recurrent neural networks (RNNs). These networks have hidden internal states that carry forward information through time. Problems that deal with time-dependent data series usually need this kind of structure to succeed. In the prediction task we have tried to circumvent this problem by stacking three time steps on top of each other. With some modifications, this would also be possible for the control task, but was not done in this case (we only trained on a single image per time step). One advantage of feedforward models lies in their performance: RNNs have higher computational costs and are thus harder to train. Because our whole OpenAI environment is already developed with RLlib, scaling the training to be able to run on Supercomputers with concurrent learning should be the next step.

Altogether, one could say that this thesis has confirmed that deep learning methods are well suited to further explore the field of cellular automata.

Appendix A

Code

The code for this thesis can be downloaded from sciebo from this url: https://uni-koeln.sciebo.de/s/DN21OMZw2cpyxsJ It features several jupyter notebook files, with some pure python files as backbone. The code folder is split into a **prediction, control, tools and plots folder**. In the Unreal Engine folder the whole Unreal Life project can be found, as well as an executable in the WindowsNoEditor folder. To run the code, make sure to change the path to this executable in the UnrealCV_helper python file. The whole folder has a size of about 13 GB, so (unless you are interested in the Unreal project data) you should just download the python code and the WindowsNoEditor folder.

Appendix B

Neural Networks Graphs

B.1 A Vision-Net for preprocessing

B.2 The Encoder-Decoder for Prediction Network

A detailed visualization of the Encoder-Decoder Network used for the prediction task would be too big to present at this point. Instead see the Encoder_Decoder_appendix.pdf file in the code/plots folder.

B.3 Hyperparameters for RL

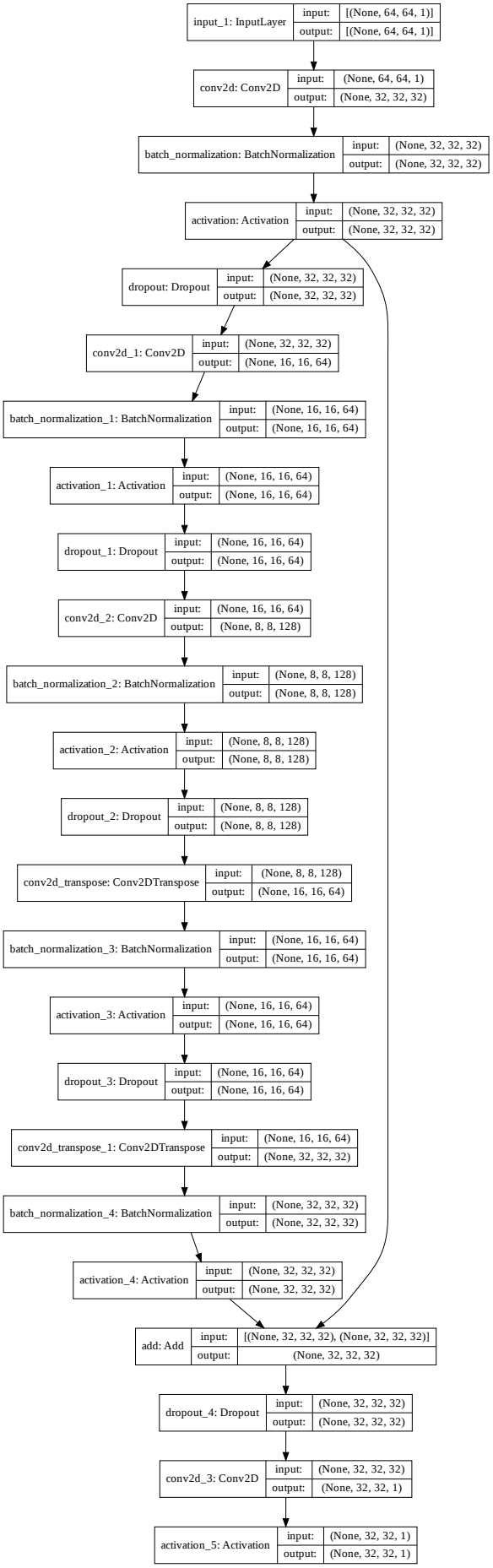

FIGURE B.1: Vision-Net for preprocessing UE images

Hyperparameter	**Value**
gamma	0.99
lambda	1.0
kl_coeff	0.2
rollout_fragment_length	200
train_batch_size	4000
sgd_minibatch_size	1000
shuffle_sequences	True
num_sgd_iter	30
lr	5e-7
lr_schedule	None
vf_share_layers	False
vf_loss_coeff	1.0
entropy_coeff	0
entropy_coeff_schedule	None
clip_param	0.3
vf_clip_param	10
grad_clip	None
kl_target	0.01

TABLE B.1: Hyperparameters for the PPO-algorithm. Most are standard from ray.io, but learning rate was choosen to be small and sgd-minibatch-size to be larger, to get a smooth training process.

Hyperparameters	Value
initial_epsilon	1.0
final_epsilon	0.02
epsilon_timesteps	1000000
timesteps_per_iteration	1000
target_network_update_freq	10000
buffer_size	500000
prioritized_replay	True
prioritized_replay_alpha	0.6
prioritized_replay_beta	0.4
final_prioritized_replay_beta	0.4
prioritized_replay_beta_annealing_timesteps	20000
prioritized_replay_eps	1e-6
lr	0.0000001
lr_schedule	None
adam_epsilon	1e-8
grad_clip	40
learning_starts	10000
rollout_fragment_length	4
train_batch_size	32

TABLE B.2: Hyperparameters for the DQN-algorithm. Most are standard from ray.io, but learning rate was choosen to be small and target-network-update intervall and buffer to be large, to get a smooth training process as well.

Appendix C

Declaration

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Cologne, February 12, 2021

Marcel Aach

List of Figures

Bibliography

[KW52] J. Kiefer and J. Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function". In: *Ann. Math. Statist.* 23.3 (Sept. 1952), pp. 462–466. DOI: 10.1214/aoms/1177729392. URL: https://doi.org/10.1214/aoms/1177729392.

[Ros58] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain". In: *Psychological review* 65 No. 6 (1958).

[Ula62] S. Ulam. "On some mathematical problems connected with patterns of growth of figures". In: (1962). Ed. by E. Bellman, pp. 215–224.

[Neu66] J. von Neumann. *Theory of Self-Reproducing Automata*. Champain, IL: University of Illionois Press, 1966.

[Gar70] Martin Gardner. "MATHEMATICAL GAMES". In: *Scientific American* 223.4 (1970), pp. 120–123. ISSN: 00368733, 19467087.

[HPP73] J. Hardy, O. De Pazzis, and Y. Pomeau. "Time evolution of two-dimensional model system. I. Invariant states and time correlation functions,J". In: *Math. Phys* 14 (1973), pp. 1746–1759.

[Dew84] A. K. Dewdney. "Sharks and fish wage an ecological war on the toroidal planet Wator". In: (1984).

[ZC88] Y. Zhou and R. Chellappa. "Computation of optical flow using a neural network". In: *IEEE 1988 International Conference on Neural Networks* (1988), 71–78 vol.2.

[Cyb89] G. Cybenko. *Approximation by superpositions of a sigmoidal function*. Math. Control Signal Systems 2, 1989, pp. 303–314.

[Wil92] R. J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8 (1992), pp. 229–256.

[Ger95] Heike Gerhardt Martin; Schuster. *Das digital Universum - Zelluläre Automaten als Modelle der Natur [The digital universe - cellular automata as models of nature]*. Friedr. Vieweg Sohn Verlagsgesellschaft mbH, 1995.

[Wei98] J. R. Weimar. *Simulation with Cellular Automata*. 1. Logos Verlag Berlin, 1998.

[BHM00] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial (2nd Ed.)* USA: Society for Industrial and Applied Mathematics, 2000. ISBN: 0898714621.

[Wol02] Stephen Wolfram. *A New Kind of Science*. English. Wolfram Media, 2002. ISBN: 1579550088. URL: https://www.wolframscience.com.

[BCG04] Elwyn R. Berlekamp, John H. Conway, and Richard K Guy. *Winning ways for your mathematical plays, volume 4*. A K Peters, Ltd., 2004, pp. 927–961.

[Sch07] Joel Schiff. *Cellular Automata: A Discrete View of the World*. 1. ISBN 978-1-118-03063-9. Wiley-Interscience, 2007.

[Bel+13] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279. ISSN: 1076-9757. DOI: 10.1613/jair.3912. URL: http://dx.doi.org/10.1613/jair.3912.

[Mni+13] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (Dec. 2013).

[He+15] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *ICML'15: Proceedings of the 32nd International Conference on International Conference on Machine Learning* 37 (2015), pp. 448–456.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. *Fully Convolutional Networks for Semantic Segmentation*. 2015. arXiv: 1411.4038 [cs.CV].

[Mni+15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (Feb. 2015), pp. 529–33. DOI: 10.1038/nature14236.

[Oh+15] Junhyuk Oh et al. *Action-Conditional Video Prediction using Deep Networks in Atari Games*. 2015. arXiv: 1507.08750 [cs.LG].

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].

[SZ15] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[BKC16] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*. 2016. arXiv: 1511.00561 [cs.CV].

[Dul+16] Gabriel Dulac-Arnold et al. *Deep Reinforcement Learning in Large Discrete Action Spaces*. 2016. arXiv: 1512.07679 [cs.AI].

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[al17] Weichao Qiu Et al. "UnrealCV Virtual Worlds for Computer Vision". In: *ACM Multimedia Open Source Software Competition* (2017).

[KB17] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[Sch+17a] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[Sch+17b] John Schulman et al. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG].

[Xie+17] Saining Xie et al. *Aggregated Residual Transformations for Deep Neural Networks*. 2017. arXiv: 1611.05431 [cs.CV].

[Pat+18] Jaideep Pathak et al. "Model-Free Prediction of Large Spatiotemporally Chaotic Systems from Data: A Reservoir Computing Approach". In: *Phys. Rev. Lett.* 120 (2 Jan. 2018), p. 024102. DOI: 10.1103/PhysRevLett.120.024102. URL: https://link.aps.org/doi/10.1103/PhysRevLett.120.024102.

[Sch+18] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018. arXiv: 1506.02438 [cs.LG].

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.

[FC19] Jonathan Frankle and Michael Carbin. *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks*. 2019. arXiv: 1803.03635 [cs.LG].

[Gau+19] Jason Gauci et al. *Horizon: Facebook's Open Source Applied Reinforcement Learning Platform*. 2019. arXiv: 1811.00260 [cs.LG].

[Gil19] William Gilpin. "Cellular automata as convolutional neural networks". In: *Physical Review E* 100.3 (Sept. 2019). ISSN: 2470-0053. DOI: 10.1103/physreve.100.032402. URL: http://dx.doi.org/10.1103/PhysRevE.100.032402.

[Lil+19] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG].

[Luc+19] Simon M. Lucas et al. *A Local Approach to Forward Model Learning: Results on the Game of Life Game*. 2019. arXiv: 1903.12508 [cs.AI].

[Ear20] Sam Earle. *Using Fractal Neural Networks to Play SimCity 1 and Conway's Game of Life at Variable Scales*. 2020. arXiv: 2002.03896 [cs.LG].

[Ple+20] Marco Pleines et al. *Obstacle Tower Without Human Demonstrations: How Far a Deep Feed-Forward Network Goes with Reinforcement Learning*. 2020. arXiv: 2004.00567 [cs.LG].

[SK20] Jacob M. Springer and Garrett T. Kenyon. *It's Hard for Neural Networks To Learn the Game of Life*. 2020. arXiv: 2009.01398 [cs.LG].

[Kar] Andrej Karpathy. *Lecture CS231n Convolutional Neural Networks for Visual Recognition*. URL: https://cs231n.github.io/ (visited on 12/11/2020).