UTRECHT UNIVERSITY

Applied Data Science Master Thesis

Integrating a Machine Learning Model Stored in the Open Neural Network Exchange format within a Physical-Based Model and Executed using a Custom-Built Backend

Supervisor(s) & Examiners:
Derek Karssenberg, Kor de Jong & Oliver Schmitz

Author:
Youssef Fatah (0498157)

July 7, 2024

Department of Information and Computing Sciences
Department of Physical Geography
Utrecht University

Table of Contents

Acknowledgments

I would like to thank my supervisors Derek Karssenberg, Kor de Jong, and Oliver Schmitz for providing me with the tools and facilities to conduct my research, as well as for their invaluable guidance on improving various aspects of my work during this thesis period. I would also like to extend my gratitude to my thesis partner, Rui Chen, for collaborating with me on the codebase for this thesis and for the various challenges we overcame together. This research has broadened my knowledge and has been a highly enjoyable experience. I look forward to applying the knowledge and skills I have gained in this field in future projects.

Glossary of Terms

Open Neural Network Exchange (ONNX): An open-source format for representing machine learning models. It enables models to be used across various frameworks, optimizing computational performance during inference.

PCRaster: A specialized framework for environmental modeling, facilitating the simulation of natural processes using a physically-based approach.

Cellular Automata (CA): A discrete model consisting of a grid of cells, each of which can be in one of a finite number of states. The state of each cell changes over discrete time steps according to a set of rules based on the states of neighboring cells.

Machine Learning (ML): A subset of artificial intelligence involving the development of algorithms that allow computers to learn from and make decisions based on data.

Surrogate Machine Learning Model: A data-driven approximation of the deterministic cellular automata (CA) model. The surrogate model learns the underlying dynamics from data rather than relying solely on pre-defined rules

Convolutional Neural Network (CNN): A type of deep learning model particularly effective for processing grid-like data such as images. It uses convolutional layers to automatically learn spatial hierarchies of features from input data.

Recurrent Neural Network (RNN): A type of neural network particularly effective for sequential data, where connections between nodes form a directed graph along a sequence, allowing it to exhibit temporal dynamic behavior.

Epoch: An epoch is one complete pass through the entire training dataset. For one epoch, the model sees and processes every training sample exactly once.

Batch size: The batch size is the number of training examples used in one iteration before the model's internal parameters are updated. It determines how many samples are processed together in a single forward and backward pass.

Geographic Information System (GIS): A framework for gathering, managing, and analyzing spatial and geographic data.

Land Use Change (LUC): The process by which human activities transform the natural landscape, affecting land cover and ecosystem functions.

Life, the Universe and Everything (LUE): A modeling framework for simulating large systems of agents and fields, primarily in environmental and land use studies.

Software-Defined Accelerator (SODA): A framework used to design custom accelerators for spatiotemporal inference workloads, enhancing computational performance and efficiency in handling spatial and temporal data.

MaxPooling: A down-sampling technique used in (CNNs) to reduce the spatial dimensions of feature maps. It involves partitioning the input into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value. This operation helps to reduce computational complexity and control overfitting.

Reshape: An operation in neural networks and data processing that changes the dimensions of a tensor without altering its data. This is often used to prepare data for input into different layers of a neural network, ensuring compatibility and correct dimensionality for subsequent operations.

Multithreading: A programming technique that allows for the concurrent execution of multiple threads within a single process. Each thread runs independently while sharing the same resources, such as memory, which can lead to more efficient use of CPU resources and improved computational performance of applications.

Parallelization: The process of dividing a computational task into smaller, independent subtasks that can be executed simultaneously on multiple processors or cores. This technique is used to speed up computations and improve computational performance, especially in large-scale data processing and machine learning model training.

Microsoft ONNX Runtime: A high-performance inference engine for Open Neural Network Exchange (ONNX) models, developed by Microsoft. The onnxruntime library is designed to accelerate the deployment of machine learning models across various platforms and devices, providing optimizations and support for running models efficiently on both CPUs and GPUs.

Physical Laws: Fundamental principles, typically expressed mathematically, that describe the behavior of physical systems under various conditions. In the context of physically-based models, these laws simulate real-world processes accurately. For example, in Conway's Game of Life, the rules of the simulation—such as survival, death by underpopulation, death by overpopulation, and birth—govern the evolution of the simulation over time. These rules ensure that the model behaves in a way that reflects the underlying physical phenomena it aims to represent

1 Introduction

Recent research has demonstrated the impressive capabilities of deep neural networks in learning predictive representations of dynamical systems (Carleo et al., 2019; Jin et al., 2023). These networks can rival traditional simulation techniques and predict general features of dynamical systems (Carleo & Troyer, 2017). Neural networks have also been effective in applied domains like speech generation (Rabiner, 1989) and video prediction (Karpathy et al., 2014). However, challenges remain in understanding how the underlying structure of a physical process affects its representation by a neural network trained using standard optimization techniques.

We explore these challenges in the context of cellular automata (CA), simple dynamical systems known for their complex behaviors and use in simulating land use change (LUC) dynamics. (*Stephen Wolfram*, 2002; Xing et al., 2020).

For example, Xing et al., (2020) developed a novel CA model integrated with deep learning techniques to simulate dynamic spatiotemporal land use changes. Their approach employed a CNN to capture spatial features and a Recurrent Neural Network (RNN) to incorporate temporal dependencies. Jr & Qiu, (2012) combined CA with decision tree algorithms to predict land use changes by deriving transition rules from historical data and applying these rules within a CA model. Similarly, other studies have utilized different machine learning algorithms and frameworks to enhance CA models, demonstrating the potential for improved accuracy and realism in simulations.

Building on previous approaches, this study proposes an alternative method by integrating a machine learning model into the CA framework using ONNX (Open Neural Network Exchange) (*ONNX.Ai*, 2024). This integration leverages the ML model's ability to recognize detailed spatial patterns, which allows for capturing intricate variations in land use changes more effectively. The ML model learns from historical data, identifying latent spatial features and relationships that traditional CA models might overlook (Yeh & Li, 2002). This approach aims to enhance the accuracy and realism of land use change simulations by using a surrogate machine learning model as a data-driven approximation of the deterministic CA model.
The surrogate model learns the underlying dynamics from data rather than relying solely on pre-defined rules (Kudela & Matousek, 2022).

This research explores this integration within the context of PCRaster, a specialized framework for environmental modeling, and evaluates the applicability of ONNX (Open Neural Network Exchange) for this purpose.

To contextualize this research, it is essential to understand the underlying frameworks and models involved. PCRaster is a dynamic modeling framework specifically designed for environmental and geographical processes. It allows for the development and execution of spatiotemporal models, often utilizing cellular automata concepts. PCRaster's strength lies in its ability to handle large-scale, spatially explicit environmental models, making it an invaluable tool for researchers in physical geography. It facilitates the simulation of natural processes such as hydrology, soil erosion, and land use changes through a physically-based approach, providing accurate and detailed insights into environmental dynamics (Karssenberg et al., 2001).

A Convolutional Neural Network (CNN) is a type of deep learning model designed to process and analyze visual data by automatically detecting patterns and features. It consists of multiple layers, including convolutional layers that apply filters to the input data to create feature maps, pooling layers that reduce the spatial dimensions, and fully connected layers that perform the final classification or prediction tasks. CNNs are widely used in image recognition, object detection, and various other applications involving spatial data. By integrating deep learning techniques into traditional CA models, researchers aim to enhance their predictive capabilities by leveraging the pattern recognition strengths of machine learning. For instance, in a study by Xing et al., (2020), a novel model combining deep learning techniques with cellular automata was developed to simulate land use changes more accurately than traditional methods. This approach employed a CNN to capture spatial features and a Recurrent Neural Network (RNN) to incorporate temporal dependencies, allowing for machine learning to improve dynamic spatial simulations.

Conway's Game of Life is a well-known example of a cellular automaton, a discrete model studied in computational mathematics. Created by mathematician John Conway in 1970, it consists of a grid of cells that can be in one of two states: alive or dead. The state of each cell in the next generation is determined by a set of simple rules based on the states of its eight neighbors. Despite its simplicity, Conway's Game of Life can simulate a variety of complex behaviors and patterns, illustrating the potential of cellular automata for modeling dynamic systems. (*Conway's Game of Life: Scientific American, October 1970*, n.d.)

ONNX (*ONNX.Ai*, 2024) is an open-source format designed to represent machine learning models across various frameworks, facilitating interoperability and optimizing computational performance during inference (Bai et al., 2019). In the context of spatiotemporal modeling, ONNX can standardize the deployment of machine learning models, enabling seamless integration with existing frameworks like PCRaster. Jain et al.,(2024) highlighted ONNX's role in optimizing inference workloads for applications such as climate forecasting and infrastructure optimization, demonstrating computing performance improvements through graph-level transformations.

To make sure we will be able to integrate and execute the ONNX model within the PCRaster framework, a custom backend needs to be developed. This approach offers several key advantages. First, it optimizes computational performance for the specific hardware and software environment, resulting in possible faster inference times and more efficient resource utilization. Second, it ensures seamless integration with PCRaster, facilitating efficient data exchange and interaction between the ONNX model and PCRaster's native functionalities. Finally, the custom backend supports any custom operations not covered by the standard ONNX runtime (*ONNX.Ai*, 2024), ensuring full compatibility and functionality for the specific needs of this study.

By leveraging the capabilities of ONNX and developing a custom backend, this research aims to demonstrate a novel approach to enhance the functionality and performance of physical models. The findings from this study could pave the way for more sophisticated and efficient modeling techniques, bridging the gap between data-driven and physical-based approaches.

1.1 Research question

The primary research question addressed in this project is:

How can a surrogate machine learning model be integrated into and replace the cellular automata simulation of a PCRaster physically-based model, and how effective is ONNX in facilitating this integration?

To thoroughly investigate this, the following sub-questions are formulated:

1. *Is ONNX a suitable format to represent a machine learning model within a physically based model?*

2. *Can parallelization over multiple CPU cores improve the execution time of an ONNX-based surrogate machine learning model and how does it compare to the physically based model?*

3. *What are the essential operators in the ONNX model format required for executing a CNN-trained machine learning model within physically-based cellular automata models?*

4. *Is a Convolutional Neural Network (CNN) an appropriate architecture for modeling the physical laws represented in a PCRaster physically-based model?*

By addressing the research questions, this thesis aims to evaluate the suitability of ONNX as a technology for integrating machine learning models into physically-based models that process spatial rasters, particularly within the PCRaster framework. This evaluation will involve selecting a machine learning architecture, training the model, developing a custom ONNX backend capable of executing surrogate machine learning models within PCRaster and assessing the practicality and effectiveness of this integration in enhancing the accuracy and efficiency of physically-based environmental models.

The insights gained from this research are expected to inform the design of ONNX backends for similar applications and contribute to the broader field of computational geography, ultimately supporting the goal of the Computational Geography group within the Department of Physical Geography at Utrecht University to integrate trained machine learning models in the (LUE) modelling framework for simulating large systems of agents and fields.

2 Literature review

The integration of physically based cellular automata spatiotemporal models with surrogate machine learning models has been widely researched. In this literature review, we will explore a selection of research in this area. In the first part, we will cover the integration of cellular automata models with machine learning models. In the second part, we will review the use of machine learning in spatiotemporal models. Additionally, we will cover work involving the use of ONNX with spatiotemporal models and parallelization methods.

Integrating Physically Based Cellular Automata Spatiotemporal Models with Surrogate Machine Learning Models

In the study "A novel cellular automata model integrated with deep learning for dynamic spatiotemporal land use change simulation", Xing et al., (2020) developed a novel cellular automata (CA) model integrated with deep learning (DL) techniques to simulate dynamic spatiotemporal land use change (LUC). Traditional CA-based methods often treated LUC dynamics as Markov processes, failing to capture long-term temporal dependencies and the complex spatial heterogeneity of land use. To address these limitations, the proposed DL-CA model employs a convolutional neural network (CNN) to capture latent spatial features, offering a comprehensive representation of neighborhood effects, and a recurrent neural network (RNN) to extract historical information from time-series land use maps. Additionally, a random forest is used as a binary change predictor to mitigate the imbalanced sample problem during training. The model was validated and evaluated using land use data from Dongguan City, China, collected between 2000 and 2014. The input data from 2000 to 2009 were used for training, 2010 data for validation, and data from 2011 to 2014 for evaluation. The study also compared the DL-CA model's accuracy against four traditional CA models: multilayer perceptron (MLP)-CA, support vector machine (SVM)-CA, logistic regression (LR)-CA, and random forest (RF)-CA. The results showed that the DL-CA model significantly improved prediction accuracy by 9.3% to 11.67% for the years 2011 to 2014 compared to the traditional models, demonstrating its effectiveness in capturing long-term spatiotemporal dependencies for more accurate LUC predictions.

Aburas et al., (2019) conducted a comprehensive review of spatiotemporal land-use change modeling, simulation, and prediction, emphasizing the critical nature of these tasks due to factors such as uncertainty, structure, flexibility, accuracy, and the potential for improvement and integration of existing models. Their study covers a range of modeling approaches, including dynamic, statistical, and machine learning (ML) models within the geographic information system (GIS) environment, aiming to meet high-performance requirements for land-use modeling. The review details the general characteristics of both conventional and ML models, categorizing the various techniques employed in their design. Through an analysis of these models' strengths and weaknesses, the authors found that ML models stand out as the most effective for simulating land-use change, thanks to their ability to incorporate all driving forces of land-use change and to handle both linear and non-linear phenomena, which conventional dynamic and statistical models struggle with. Despite their advantages, ML models do have limitations, such as complexity, rigid simulation rules, and a lack of transparency in their operations. The authors suggest that these limitations can be mitigated by using programming

languages like Python, which can enhance the simulation capabilities and flexibility of ML models.

Jr & Qiu, (2012) developed an integrated parcel-based land use change model combining cellular automata and decision tree algorithms to address the ecological impacts driven by land use changes. The model was designed to predict spatial land use changes and was developed using historical land use data from Hunterdon County, New Jersey, USA. The researchers collected and processed this historical land use data along with various driving factors influencing land use changes using a Geographic Information System (GIS). To illustrate the land use change processes from 1986 to 1995, they employed the decision tree J48 Classifier to derive a set of transition rules. These rules were then applied to the 1995 land use data within a cellular automata model, specifically using Agent Analyst, to predict future land use patterns. The predicted patterns were validated against actual land use data from 2002, showing an overall accuracy of 84.46 percent and a Cohen's Kappa Index of 0.644. The model demonstrated a higher capability in predicting quantitative changes compared to locational changes in land use. Additionally, sensitivity analysis revealed that altering the neighborhood size had minimal impact on simulation results and did not significantly affect the model's accuracy.

ONNX in spatiotemporal modelling

Jain et al., (2024) conducted an in-depth analysis of inference workloads for spatiotemporal modeling, focusing on applications such as power grid resiliency, climate forecasting, and transportation infrastructure optimization. They highlighted the challenges of deploying trained models for near real-time inference due to varying performance across environments, computing resources, and result ambiguity tolerance. The authors developed several deep learning applications using spatiotemporal data and examined their computational patterns during inference. They evaluated models like Long Short-Term Memory (LSTM), Convolutional Neural Networks (CNN), and Spatiotemporal Graph Convolution Networks (STGCN) across different DL frameworks (TensorFlow, PyTorch), precisions (FP16, FP32, AMP, INT16, INT8), inference runtimes (ONNX, AI Template), post-training quantization techniques (TensorRT), and platforms (Nvidia DGX A100, Sambanova SN10 RDU).
The study found that models converted to the ONNX format can benefit from performance optimizations during inference. This is because ONNX facilitates the use of tools and runtimes, such as ONNX Runtime, which can apply graph-level transformations (e.g., node eliminations and fusions) to improve efficiency. By using ONNX, these optimizations can be more easily implemented across various frameworks and hardware platforms, enhancing the overall computational performance of inference workloads. ONNX's standardized format allows seamless model sharing across various frameworks, enhancing efficiency. However, despite these benefits, achieving optimal performance on contemporary GPU systems remains challenging. The research suggests using optimized High-Level Synthesis frameworks like SODA (Software-Defined Accelerator) to design custom accelerators tailored to spatiotemporal inference workloads. These custom accelerators are specialized hardware solutions that improve the efficiency and performance of analyzing data that changes over space and time. By leveraging tools like SODA, it's possible to create hardware specifically to handle spatiotemporal data.

Multithreading and parallel processing

Kumawat, (2023) investigated various methods for accelerating model inference through parallel processing, leveraging Python's threading module and *concurrent.futures* library (*Python*, 2024). While Kumawat also discusses other tools like Dask and PySpark (*Dask*, 2024; *PySpark*, 2024), this study specifically focuses on using python's *concurrent.futures* library for implementing multithreading. Kumawat provides practical examples and code snippets, showcasing how to utilize these techniques to speed up model inference for different use cases, and the importance of choosing the appropriate approach based on factors such as computational resources and task dependencies.

In another study, Bayyat et al., (2024) investigated the impact of parallel hyperparameter optimization on machine learning model performance. The study utilized parallel processing capabilities to tune a Random Forest classifier for fake news detection. They found that the parallel implementation yielded substantial speedups, achieving over 5x faster CPU times and 3x faster total run times compared to sequential tuning. While a minor decrease in test accuracy was observed with parallelism, the authors concluded that the significant computational gains outweighed this slight trade-off in model accuracy, enabling broader hyperparameter exploration within acceptable timeframes.

3 Method

This chapter outlines the methodology employed to achieve the integration of a machine learning (ML) model with a physical-based model.

3.1 Cellular automata simulation model

To enable the integration of a machine learning (ML) model within a physically-based model, we need to simulate the physically-based model using cellular automata based on physical laws. These laws are the rules that determine how the simulation behaves. The data generated from this simulation will be used to train a surrogate ML model. This surrogate ML model aims to replicate the dynamics of the simulation, effectively capturing its behavior. This approach allows us to explore how ML can enhance traditional simulation models, particularly in capturing complex spatial patterns and long-term temporal dependencies.

For this project, the cellular automata simulation model selected is Conway's Game of Life created by mathematician John Conway. This model was introduced in the journal "MATHEMATICAL GAMES The fantastic combinations of John Conway's new solitaire game "life" " in the Scientific American. (*Conway's Game of Life: Scientific American, October 1970*, n.d.) This cellular automaton will set the physical laws that determine the simulation's behavior.

The Conway's Game of Life cellular automata simulation was chosen for its simplicity and not requiring many computational resources to run on limited hardware.

The cell's state of evolution is determined by its initial state, requiring no further input. The game takes place on a two-dimensional grid of cells, where each cell can be in one of two states: alive or dead. The state of the cells evolves through discrete time steps according to a set of simple rules:

Survival*:*
An alive cell with two or three living neighbors stays alive to the next generation.

Death by Underpopulation:
An alive cell with fewer than two living neighbors dies in the next generation.

Death by Overpopulation:
An alive cell with more than three living neighbors dies in the next generation.

Birth:
A dead cell with exactly three living neighbors becomes an alive cell in the next generation.

These rules are applied simultaneously to every cell in the grid, creating the next generation of cells. The simplicity of these rules and setting the probability of a cell to be alive at a random value like 0.15 can lead to surprisingly complex and varied behaviors, making the Game of Life a suitable choice in this study.

3.2 Spatiotemporal physically based model

According to the literature review, the physically based model used in the paper by (Xing et al., 2020) that was integrated with a deep learning technique was a traditional Land Use Change Cellular Automata (LUC-CA) model. The specific name of the framework was not explicitly stated and may be proprietary.

For this study, the framework for the physically based model selected is PCRaster: a dynamic modeling tool specifically designed for spatial and temporal modeling. It supports various types of spatial analyses, including environmental modeling, hydrological modeling, and land use change simulations. Its scripting language allows for the implementation of custom models, and its compatibility with GIS software enhances its functionality in spatial data handling. This framework was chosen for its ease of use, clear documentation, accessibility, and its capability to handle raster data and perform spatial analyses. The modelling framework was created at Utrecht University (Karssenberg et al., 2001).

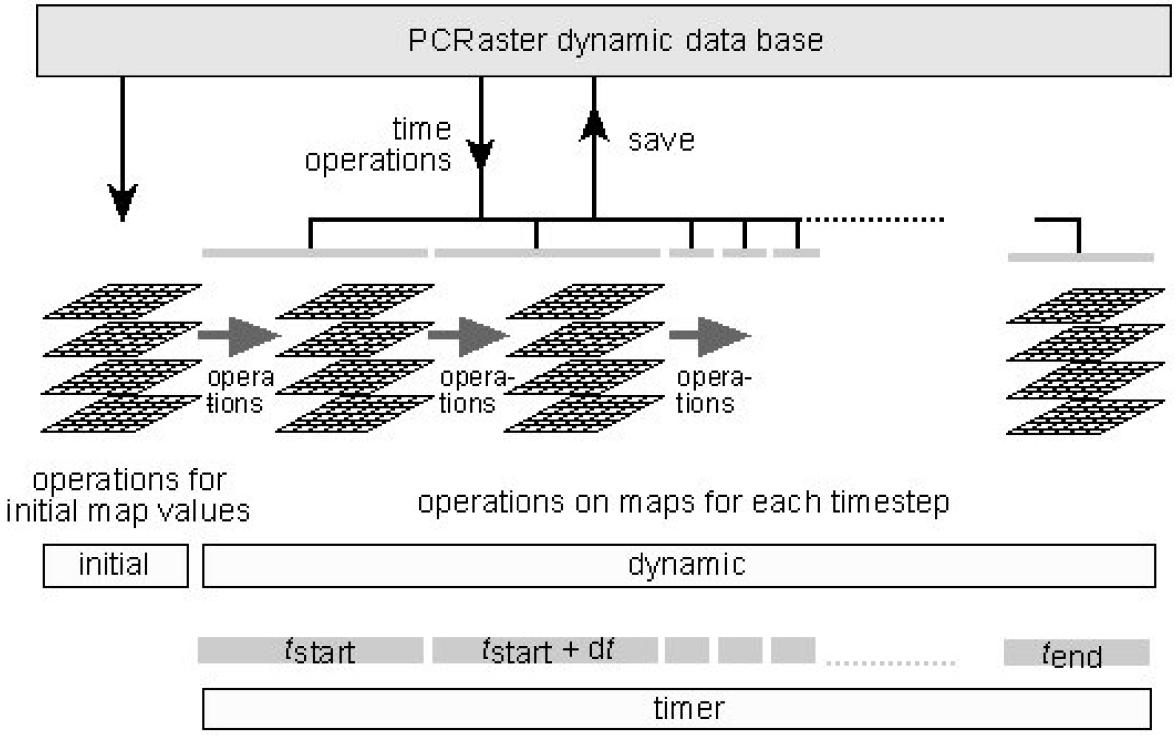

Figure 1: internal process of physically based dynamic modeling (Karssenberg et al., 2001)

The cellular automata functions happen within the dynamic part of the PCRaster model. The dynamic modeling capabilities allow for the inclusion of cellular automata functions, enabling the simulation of spatiotemporal processes. The dynamic section of a PCRaster script file handles the temporal iterations necessary for cellular automata, where each cell's state can change over time based on the predefined rules and the states of neighboring cells.

3.3 Construction of the physically based model

The following steps outline the model construction process: The model initializes a 20x20 grid using the setclone function, setting up the spatial extent and resolution for the simulation. The initial state of the grid is defined using a probability distribution for three states: alive (1), dead (0), and missing (2). This is achieved using NumPy to generate a random initial state based on specified probabilities. The initial state is then converted to a PCRaster field using the numpy2pcr function.

The dynamic behavior of the model is implemented in the dynamic method of the *GameOfLife* class. This method simulates the evolution of the grid over time based on predefined rules. The number of alive neighbors for each cell is calculated using the *windowtotal* function, which sums the values in a 3x3 window around each cell. The rules for cell transitions are defined as outlined in section 3.1, the state of each cell is updated based on these transition rules using the *if then else* function.

The model's state at each time step is stored as a NumPy array for visualization purposes. Matplotlib is used to create plots of the grid at each time step, with a custom colormap to distinguish between dead, alive, and missing cells. Finally, the model is executed for a specified number of steps using the DynamicFramework class from PCRaster, which manages the simulation loop and calls the initial and dynamic methods at each time step.

3.4 Machine Learning model selection

Based on the literature, a Convolutional Neural Network (CNN) can capture latent spatial features, allowing for representations of neighborhood effects (Xing et al., 2020). Using CNNs, you can even build deep learning applications with spatiotemporal data and examine their computational patterns during inference (Jain et al., 2024).

Figure 2 illustrates a Convolutional Neural Network (CNN) architecture for image recognition. This CNN architecture efficiently captures and processes spatial features through its convolutional layers. The process begins with an input image of 28x28 pixels with a single channel. The first convolutional layer applies a 5x5 kernel to detect basic spatial features like edges, followed by a 2x2 max-pooling layer that downsamples the feature maps, reducing their dimensions while retaining significant features. A second convolutional layer with a 5x5 kernel captures more complex patterns, followed by another max-pooling layer for further downsampling. The resulting feature maps are flattened into a one-dimensional vector for the fully connected layers. The first fully connected layer uses ReLU activation to integrate the spatial features into higher-level representations. The subsequent fully connected layer includes dropout to prevent overfitting. Finally, the output layer uses a softmax activation function to produce a probability distribution across the possible classes, with the highest probability indicating the predicted class (Nguyen, 2022).

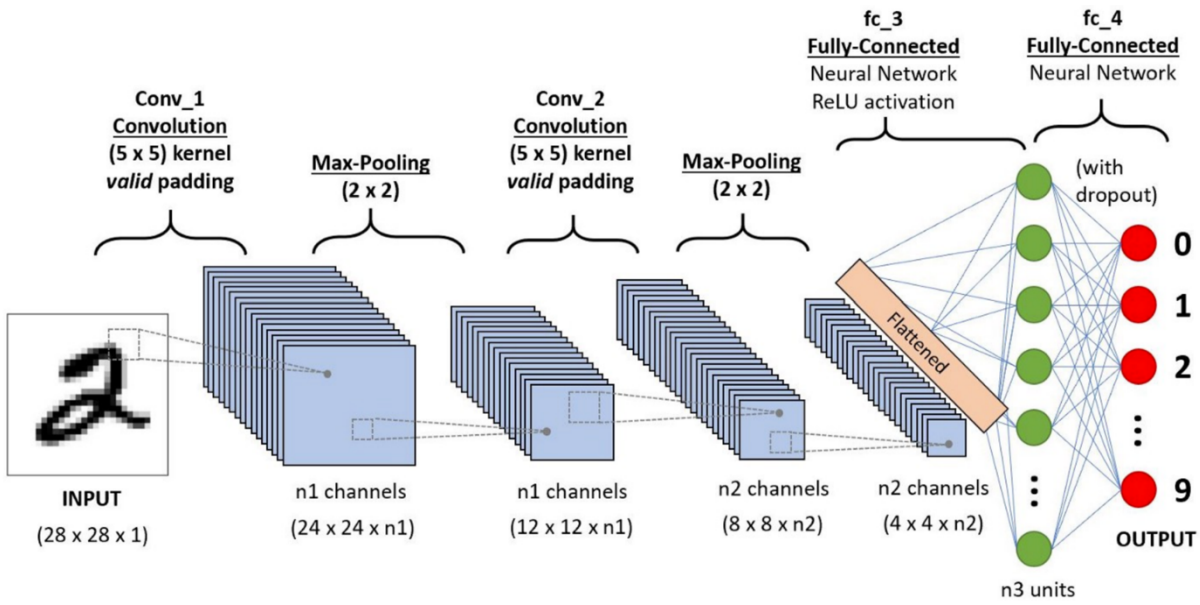

Figure 2: Convolutional operation of a Deep Convolutional Neural Network (Nguyen, 2022)

Convolutional Neural Networks (CNNs) are also used in Geographic Information Systems (GIS) for tasks such as land cover classification and change detection (Lee, 2023).
By training on labeled satellite images, CNNs learn to recognize features associated with various land cover types, including forests, grasslands, and urban areas.
This training enables the accurate classification of new images. Additionally, CNNs can analyze time-series satellite images to detect changes in land cover over time, providing valuable insights into environmental changes (Lee, 2023). The CNN was selected for this project because of these advantages.

3.4.1 Building the Convolutional Neural Network

To build the CNN, TensorFlow was chosen as the framework due to its flexibility and robustness in handling deep learning models. Specifically, Keras, a high-level API within TensorFlow, was utilized for its simplicity and ease of use in constructing and training neural networks. TensorFlow's extensive support for various layers, optimizers, and utilities makes it an excellent choice for developing and deploying CNNs in research and production environments (Chollet, 2021; *Keras: Deep Learning for Humans*, 2024; *TensorFlow*, 2024).

3.4.2 Model construction and data generation:

The physical model's Game of Life simulation was used to generate initial frames of the grid. By running the simulation multiple times, these frames were used to create training and validation datasets, which were stored in .npy files to serve as input data for the CNN .
These datasets were then loaded from the .npy files, and the data was reshaped to include a single channel, as required by the CNN architecture. Additionally, a function, life_step, was implemented to compute the next state of the grid based on the Game of Life rules (see Appendix A) (Fatah & Chen, 2024).

The CNN architecture was built using Keras with the TensorFlow backend. The model comprised convolutional layers to detect spatial features, followed by dense layers for classification. An epoch refers to one complete pass through the entire training dataset, meaning the model sees every training example once per epoch. The batch size is the number of training examples utilized in one iteration before the model's internal parameters are updated. The training process used a dataset of 8000 samples with a batch size of 50, meaning the model updated its parameters after every 50 samples. The model was trained over 50 epochs, meaning it saw each sample 50 times during training. A validation dataset of 2000 samples was used to track how well the model is learning through comparing its training and validation accuracy and prevent overfitting, meaning the model remembers the training data too well. After the training process, the model was saved for future use, ensuring its availability for subsequent inference tasks (*Deep Learning*, n.d.; *Keras: Deep Learning for Humans*, 2024; *TensorFlow*, 2024).

Accuracy is a metric that measures the number of correct predictions made by the model divided by the total number of predictions. It is used to evaluate how well the model predicts the correct classification and is defined as:

$$\text{Accuracy} = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions} \quad \text{or} \quad = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP (True Positives) represents the number of positive samples correctly predicted by the model, TN (True Negatives) represents the number of negative samples correctly predicted by the model, FP (False Positives) represents the number of negative samples incorrectly predicted as positive by the model, and FN (False Negatives) represents the number of positive samples incorrectly predicted as negative by the model (*Accuracy in Machine Learning*, 2024).

Loss or Binary Cross-Entropy Loss measures the difference between the actual and predicted probabilities of the positive class. It is used as the loss function during training and is defined as:

$$\text{Loss} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)]$$

where N is the number of samples, y_i is the actual label (0 or 1), and p_i is the predicted probability of the positive class (*Loss in Machine Learning*, 2024).

3.5 Selecting format to represent machine learning model

Meta (Previously Facebook), Microsoft, and a community of partners developed ONNX as an open standard for representing machine learning models. Models from many frameworks, including TensorFlow, PyTorch, scikit-learn, Keras, Chainer, MXNet, and MATLAB, can be exported or converted to the ONNX format as shown in figure 3 (fkriti, 2024). Once the models are in the ONNX format, they can be run on various platforms and devices. The ONNX format was selected because of its use in inference workloads for spatiotemporal modeling and its promising possible optimizations for these inferencing tasks (Jain et al., 2024). Additionally, ONNX is an open standard that is supported by a wide range of machine learning frameworks and devices, making it interoperable and deployable in many environments

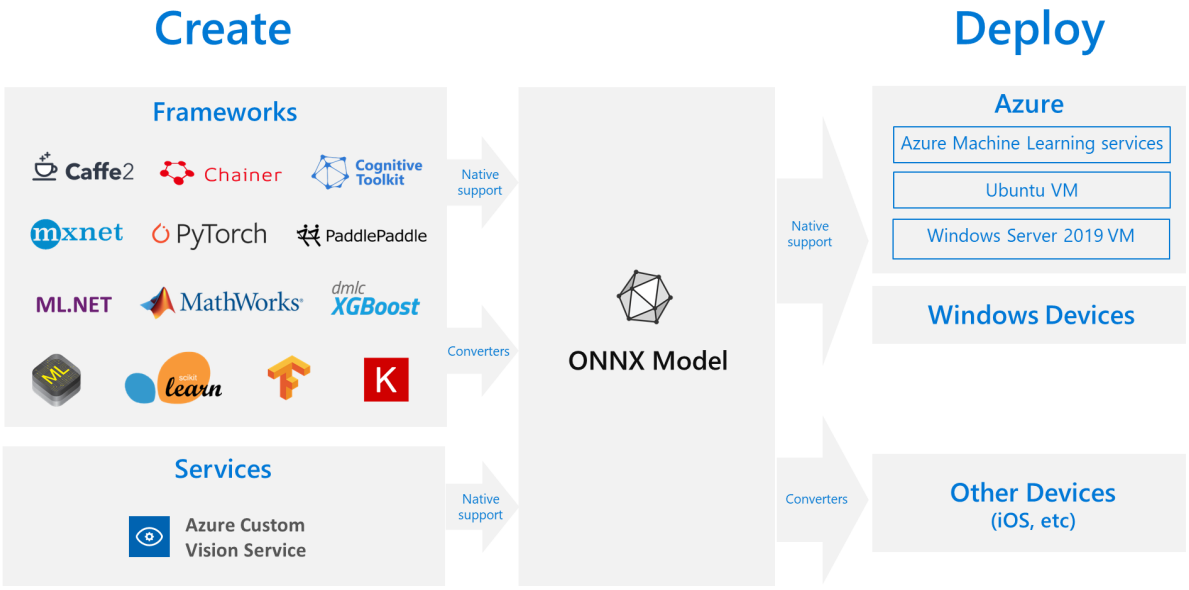

Figure 3: ONNX supported by various frameworks and deployment on multiple environments (fkriti, 2024)

3.5.1 Converting the model

After the CNN model was trained, it was first saved in HDF5 format and then converted to the ONNX format to ensure compatibility and facilitate deployment across different platforms. While TensorFlow provides capabilities for direct conversion to ONNX, the conversion process can sometimes encounter compatibility issues, especially with complex models or custom layers. Using the HDF5 format as an intermediate step ensures the model is fully compatible with the tf2onnx conversion tool, allows for continued training from checkpoints if necessary, and reduces the risk of conversion errors.

The model was saved to disk as ./game_of_life.h5. Then, using the tf2onnx library, TensorFlow, and an input specification, the Keras model was converted to ONNX format. The input specification was defined to match the model's input dimensions. The conversion was performed with the tf2onnx.convert.from_keras function, specifying the input signature and the ONNX opset version 13. The resulting ONNX model was saved to disk as game_of_life.onnx**.**

3.5.2 Analyzing machine learning operators

The operators were visualized using NETRON to understand the models requirement and determine which operators were essential for execution (*NETRON*, 2024). This involved reviewing the model architecture and identifying the individual operations.
The process included running preliminary tests such as creating multiple types of CNN models and examining the model's computational graph to pinpoint the necessary operators (see at onnx_experiment) (Fatah & Chen, 2024).

3.6 ONNX backend

The integration of a CNN into a traditional CA model for the Game of Life simulation poses specific challenges, such as ensuring efficient execution of operations and maintaining compatibility across deployment environments. Existing solutions do not adequately address these challenges, particularly in the context of leveraging ONNX models for CA simulations.

To address these challenges, we will develop a custom backend capable of executing ONNX models. This backend implementation provides several key advantages. By utilizing ONNX, we can apply advanced graph-level optimizations, such as node eliminations and fusions, which enhance computational efficiency. Additionally, the custom backend will be designed to enable parallel processing, thereby potentially improving computational performance on high-performance computing environments. The backend can be constructed by defining an operator_map that maps the ONNX operators to their corresponding NumPy implementations. The ONNX operations from the ONNX documentation can be referenced for accurately mapping and executing these operators within the custom backend (see Appendix B) (Fatah & Chen, 2024).

3.7 Physically based and machine learning model integration approach

The figure below outlines the approach and process of integrating the Convolutional Neural Network (CNN) machine learning model within the PCRaster physically based dynamic framework. The process begins with providing the initial rasterized input, which is a grid-based representation of spatial data. This rasterized input is then converted into numpy arrays, a format suitable for numerical and matrix operations, transforming it into an initial map that can be processed dynamically. The initial state of the simulation is read from this input map, which is then reported as the output map. The initial raster data is further converted into a boolean array indicating the presence (alive) or absence (dead) of cells, preparing the model for dynamic updates according to the rules of the Game of Life.

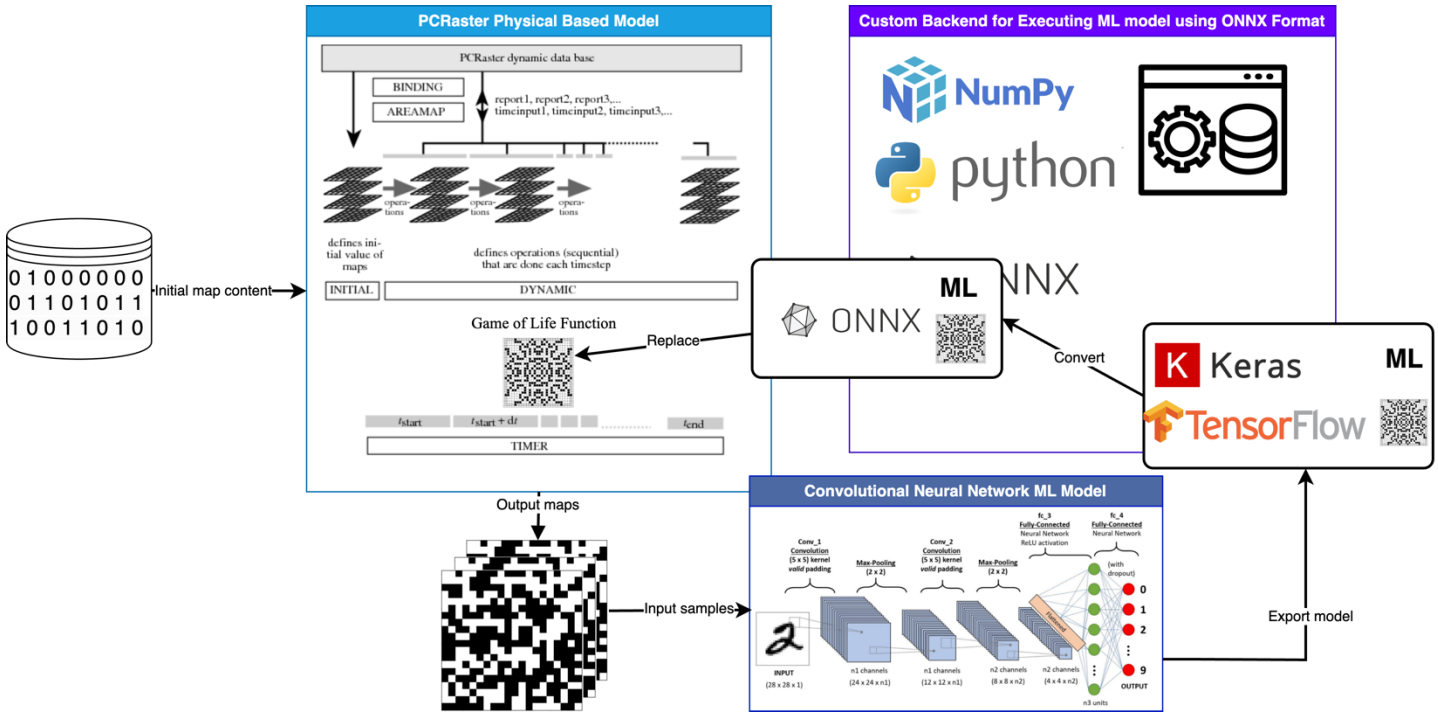

Figure 4: integration process, incorporating figures from Karssenberg et al., (2001) & Nguyen, (2022)

The dynamic method is then used to simulate the evolution of the system over time. In this method, the number of alive neighbors for each cell is calculated using a moving window total. Based on this count, the Game of Life rules are applied: a cell is born if it has exactly three alive neighbors, and a cell survives if it has two or three alive neighbors.

To ensure that only the defined cells are updated, the *ifthen* function is used. The updated state of the cells is then reported, allowing the simulation to progress step by step. Throughout the simulation, the output maps generated at each time step are stored. These time-step output maps will serve as training data for the CNN machine-learning model.

The CNN machine learning model is constructed and trained on these output maps, as explained in section 3.4. Once the CNN model is trained, it is exported as a Keras model and then converted to the ONNX format.

Using the customized backend described in section 3.6, the existing function of the Game of Life can be replaced with this machine-learning variant. Specifically, the *GameOfLife* class initializes the ONNX runtime session and loads the ONNX model during the initial method. In the dynamic method, when the model_type is set to 'backend', the model's graph is executed using the custom backend to compute the next state of the cells. The boolean state of the cells is converted to numpy arrays and passed through the *execute_graph* function, which runs the model and retrieves the output tensor. This output, representing the predicted next state of the cells, is then converted back to the PCraster format for further processing and reporting. This approach allows the replacement of the traditional rule-based updates with predictions from the machine learning model (see at onnx_experiment), (Fatah & Chen, 2024).

3.8 Model evaluation and comparison for Game of Life simulation

To compare how the surrogate machine learning model performed compared to physically based model, we will evaluate the models using several metrics. First, we will simulate both models over multiple time steps up to 100 and record the number of alive cells at each step. We will then calculate the Mean Squared Error (MSE) and accuracy at each time step by comparing the alive cell grids of the machine learning model to those of the physically based model. The MSE will provide a measure of the average squared differences between the models, while accuracy will indicate the proportion of cells that are in the same state in both models.
Additionally, we will plot the number of alive cells over time for both models to visually assess their scientific performance. We will also calculate the average number of alive cells for each model to compare their overall behavior throughout the simulation.
Accuracy is defined as mentioned in section 3.4.2.

Mean Squared Error (MSE) is used to measure the average squared difference between the predicted values and the actual values and is defined as:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)$$

where N is the number of samples, y_i is the actual value, and $\hat{y}_i$ is the predicted value (Frost, 2021).

3.9 Parallelization and measuring execution time

To evaluate the computational performance of the PCRaster physically-based model and the surrogate machine learning model, we measured the execution times for various configurations. The PCRaster model's execution time was recorded. For the surrogate machine learning model, we measured execution times for three different versions: the standard backend, a parallelized backend, and the Microsoft ONNX runtime.

The execution times for each model and configuration were recorded using Python's time module. For each configuration, we performed multiple simulation runs to ensure accuracy and consistency in our measurements. The backend model was parallelized by utilizing Python's *concurrent.futures* module as mentioned in the article by (Kumawat, 2023) to distribute computational tasks across multiple threads, specifically targeting computationally intensive operations like convolution, max-pooling, matrix multiplication, and other operators.

To parallelize the operators using Python's threading and concurrent features, the *ThreadPoolExecutor* from the *concurrent.futures* module was employed. This approach allows multiple threads to run concurrently, leveraging multiple CPU cores for improved computational performance, especially for operations that can be executed independently in parallel. For convolution (conv) and max pooling (*maxpool*) operations, the input tensor is divided into smaller slices corresponding to the output positions. Each slice is processed independently in parallel using worker functions (*conv_worker* and *maxpool_worker*). In the convolution operation, the input tensor is first padded as needed, and an output tensor is initialized. A *ThreadPoolExecutor* is then created to manage the parallel threads. For each position in the output tensor, a task is submitted to the executor, involving slicing the padded input tensor and processing it using the *conv_worker* function, which computes the convolution result for that slice. As each task completes, the results are collected and placed into the appropriate position in the output tensor. Similarly, in the max pooling operation, the input tensor is padded, and an output tensor is initialized. A *ThreadPoolExecutor* is created, and for each position in the output tensor, a task is submitted to the executor. Each task involves slicing the padded input tensor and processing it using the *maxpool_worker* function, which computes the max pooling result for that slice. The results are then collected and placed into the appropriate position in the output tensor. For other operations like *reduceprod*, *concat*, and *gemm*, the parallelization strategy is similar. A *ThreadPoolExecutor* manages the parallel threads, tasks corresponding to parts of the operation that can be executed independently are submitted, and the results of each parallel task are collected and combined to form the final output. This approach ensures that each slice is processed concurrently, which should result in significant computational performance improvements for these operations The parallelization strategy is adaptable to different input graphs with the same set of operations. Since it relies on dividing input tensors into slices and processing these slices independently, the strategy can accommodate variations in the input graph structure. As long as the operations remain consistent, the parallelization strategy will be effective across different input configurations (see Appendix B).

The execution times were collected for single, 10, and 100 simulation runs to provide a thorough analysis across different scales of execution. This approach allows assessment of computational performance over various scales (Jack Dongarra, 2003) By examining multiple scales, we can identify computational performance trends and potential bottlenecks that may not be apparent in single-run evaluations.

4 Results

This chapter presents the results of integrating a Convolutional Neural Network (CNN) model trained on the Game of Life cellular automata with the PCRaster physical-based model.

4.1 Integrated model

A Python package was developed to integrate the physically based model and the machine learning model, including a backend for model execution. Installation instructions are provided in the package's README file. To utilize the package, install it using pip install .
and then import the *simulate_process* function. Pseudo-code of implemented integration:

```
class GameOfLife:
    initialize model with input_map, output_map, model_path, model_type
    set model parameters

    function initial():
        load input_map
        report initial state
        if model_type is 'backend':
            load ONNX model

    function dynamic(output_type):
        print running model type

        # Assume there are more processes above this

        if model_type is 'pcraster':
            run_pcraster_model()
        elif model_type is 'backend':
            run_backend_model(output_type)
        else:
            print unknown model_type

        # Assume there are more processes below this

    function run_pcraster_model():
        calculate number of alive neighbors
        determine birth and survival
        update alive cells
        report new state

    function run_backend_model(output_type):
        prepare input data for custom backend
        execute custom backend graph
        update alive cells based on backend output
        report new state
```

```
function simulate_process(input_map, output_map, model_type, model_path, time_step):
    initialize GameOfLife model
    if input_map not found or model_path not found for 'backend':
        raise error
    run the game of life model for given time steps
    plot the results

# Example call to the function
if __main__:
    simulate_process('alive.map', 'result', 'backend', 'game_of_life.onnx', 100)
```

The pseudo-code above outlines the integration of a machine learning model into a dynamic simulation framework. The *GameOfLife* class initializes the model with input parameters and sets model-specific configurations. The initial function loads the input data and, if the model type is ‘backend’, loads the ONNX model. The dynamic function conditionally runs either the PCRaster model or the ML-based model based on the specified model_type, while assuming there are additional processes before and after this section. The *simulate_process* function runs the model for a given number of time steps and plots the results. The full source code is available on Github at onnx_experiment.

4.2 Model assessment

In this section, the results for the scientific performances will be presented.

4.2.1 Train history

The training process was monitored over 50 epochs, with both training and validation accuracy reaching 99% and 100% respectively from the first epoch. This rapid convergence is likely due to the small size of the dataset and the relatively simple nature of the Game of Life simulation, which the model can quickly learn to replicate.

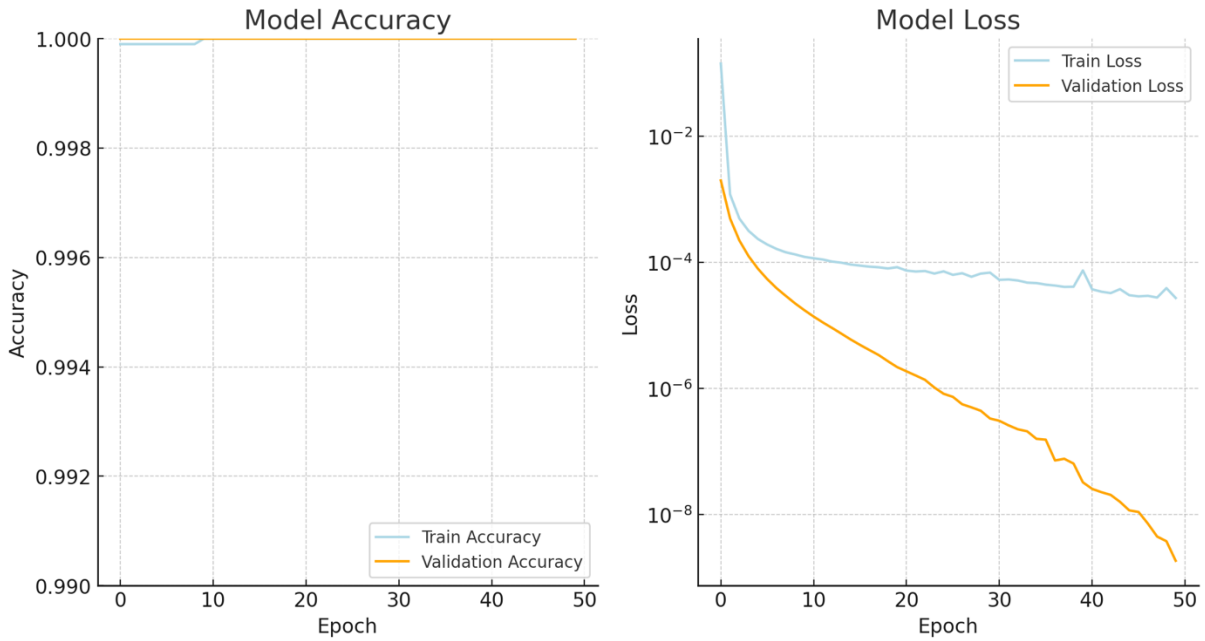

Figure 5: Train history of surrogate machine learning model with the left plot presenting accuracy and the right plot the loss, the blue line is the training set and the orange line the validation set.

Figure 5 shows the training and validation accuracy and loss curves. Despite the high accuracy metrics, the surrogate machine model, through multiple practical tests indicated that training for exactly 50 epochs yielded the most accurate simulation of the CA from the physically-based model. Training for more or fewer epochs resulted in less optimal outputs, likely due to overfitting or underfitting, respectively.

4.2.2 Alive cells over time

Figure 6 shows the number of alive cells of the ONNX surrogate machine learning model (blue line with crosses) compared to the PCRaster physically based model (orange line with squares). Both models start with the same number of alive cells. However, the PCRaster model quickly stabilizes, with the number of alive cells dropping to around 15 within the first 20-time steps and remaining constant thereafter. In contrast, the ONNX surrogate machine learning model exhibits fluctuations, with the number of alive cells oscillating between 20 and 50 throughout the entire simulation. The average number of alive cells for the PCRaster model is 17.73, while for the ONNX surrogate machine learning model it is 32.65.

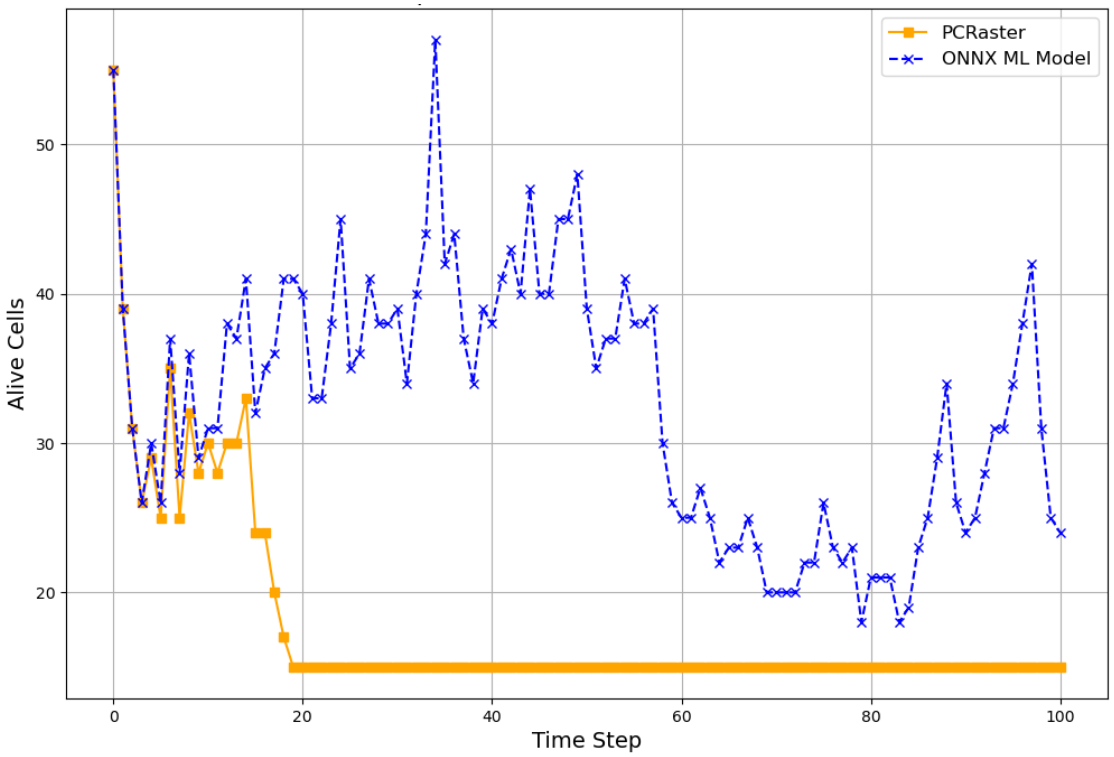

Figure 6: Comparison of alive cells over time, the blue line is the surrogate ONNX ML model and the yellow line the PCRaster model

4.2.3 MSE and Accuracy

Figure 7 illustrates the scientific performance metrics of the ONNX surrogate machine learning model compared to the PCRaster physically based model. Over time, the Mean Squared Error (MSE) (blue line with circles) and Accuracy (green line with squares) are tracked across 100-time steps. The MSE starts low but quickly rises, reaching its peak around 60-time steps before fluctuating and eventually decreasing towards the end. On the other hand, Accuracy shows a rapid initial drop, suggesting that the ONNX surrogate machine learning model's predictions quickly diverge from those of the PCRaster model. Despite fluctuations, Accuracy improves slightly mid-simulation but never fully recovers to the initial level. The average MSE over the 100-time steps is 0.086, and the average accuracy is 0.914.

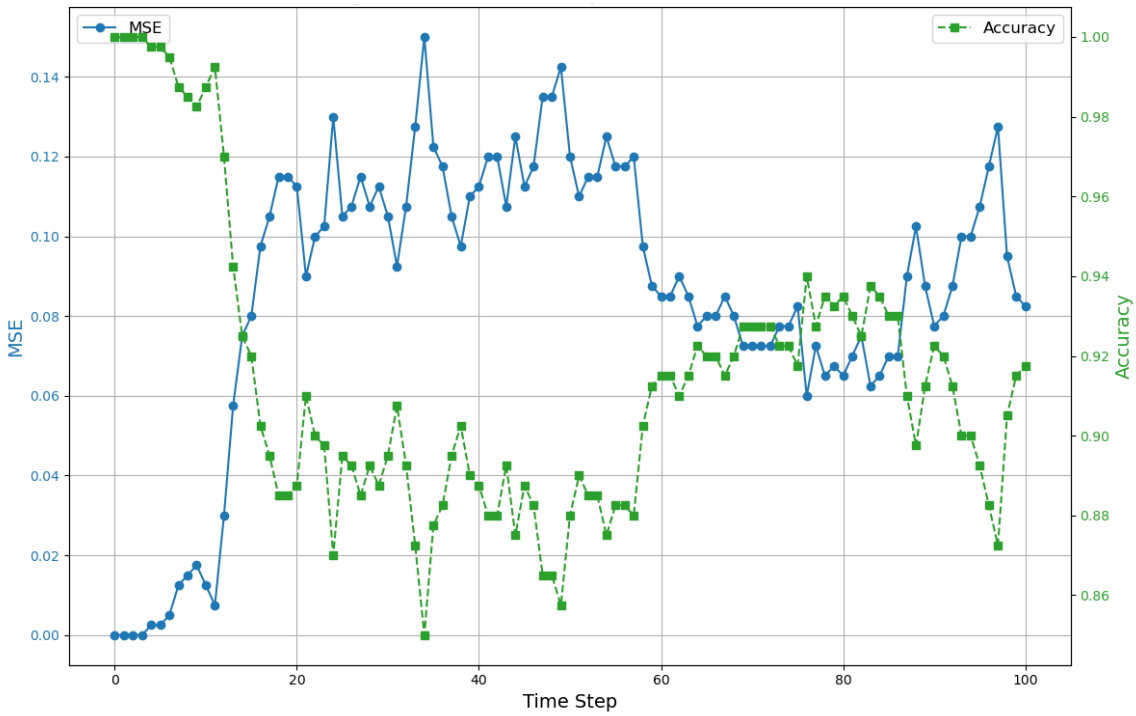

Figure 7: MSE and Accuracy of ONNX model compared to PCRaster over time, the green line is the accuracy and the blue line is the MSE (mean squared error).

4.3 Execution Time

Table 1, presents the average execution times for different model types and simulation runs, including single, 10, and 100 simulations. In this context, a single simulation run corresponds to a single time step of the model's execution. This comparison shows the execution time differences between the ONNX model running on the custom ONNX backend, the model running on the parallelized custom ONNX backend, the PCRaster model, and the model running on the official ONNX runtime.

Table 1: Execution Time Comparison for Single, 10, and 100 Simulations

Model Type	Simulation Runs	Avg Execution Time (seconds)
Custom ONNX Backend	1	0.045205
P. Custom ONNX Backend	1	0.088812
PCRaster	1	0.000625
Microsoft ONNX Runtime	1	0.001169
Custom ONNX Backend	10	0.045627
P. Custom ONNX Backend	10	0.083872
PCRaster	10	0.000197
Microsoft ONNX Runtime	10	0.000262
Custom ONNX Backend	100	0.044897
P. Custom ONNX Backend	100	0.085236
PCRaster	100	0.000182
Microsoft ONNX Runtime	100	0.000229

The execution times for single, 10, and 100 simulation runs of both models were recorded and analyzed. The PCRaster model demonstrated much faster execution times compared to the backend machine learning model across all scenarios.

4.4 Model output comparison of Game of Life simulation

In this section, we compare the outputs of the physically based and surrogate machine learning models. As shown in Figure 8, yellow cells represent alive cells, while purple cells represent dead cells. The figure illustrates the cellular evolution over time steps, with the first row displaying the physically based model and the second row showing the surrogate machine learning model. Both models start from the initial state at time step 0 and show comparable outputs. The cellular states in both models remain identical in the first- and second-time steps.

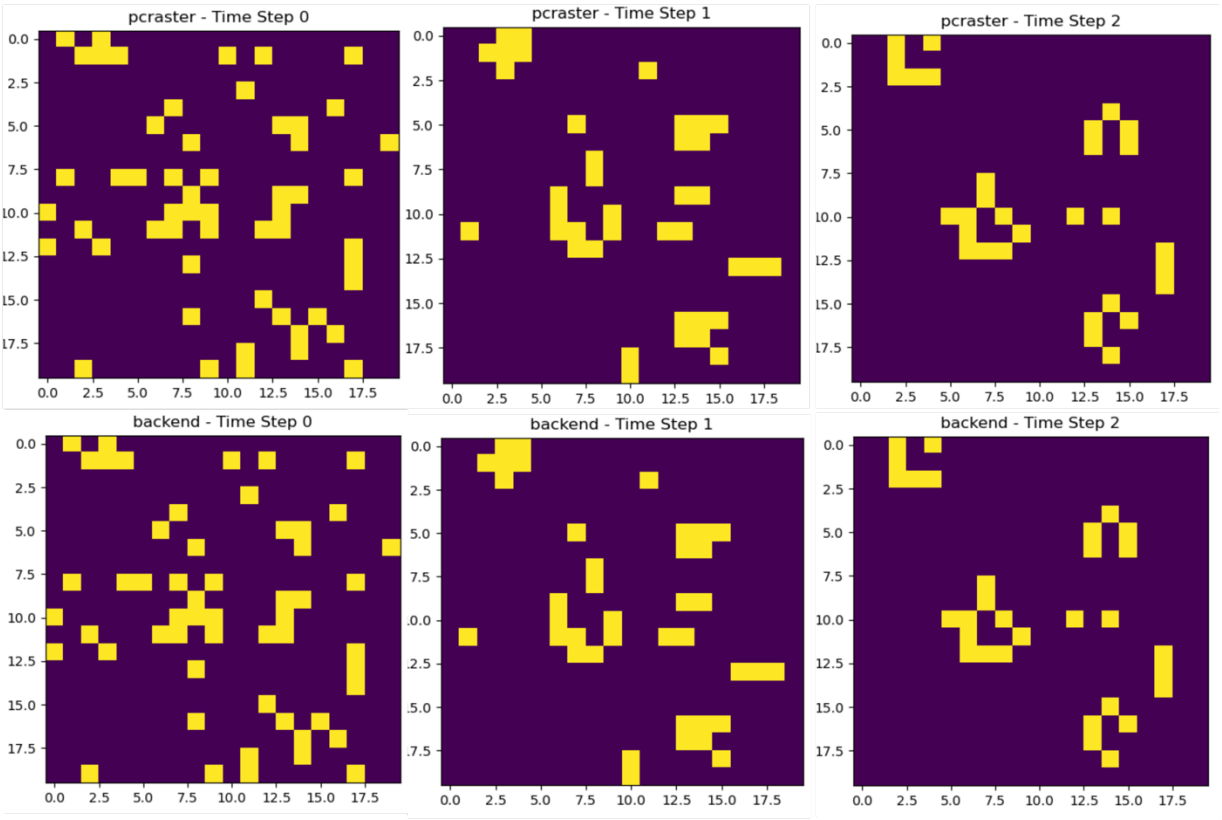

Figure 8: cellular evolution over time steps first row showing the physically based model and second row the surrogate model.

In Figure 9, we observe that by the 4th time step, there is one cell difference in the upper left corner between the two models; the surrogate machine learning model birthed one extra cell compared to the physically based model. Fast-forwarding to the 10th time step, the differences become more pronounced as the cell structure in the corner takes on a different shape. By the 100th time step, almost all cells look different between the two models, except for a single rectangle in the right corner, which remains similar in both models.

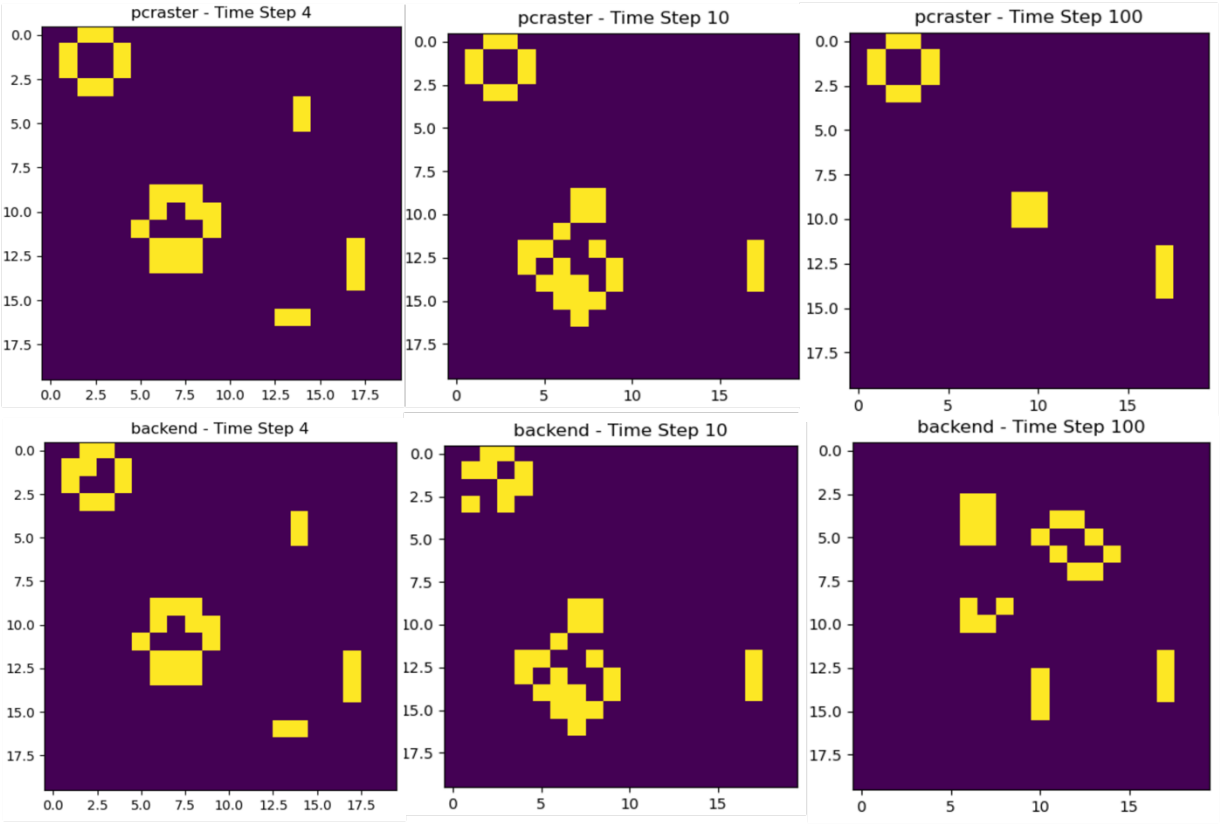

Figure 9: cellular evolution over time steps first row showing the physically based model and second row the surrogate model.

4.5 Supported ONNX operators

The backend development incorporated a variety of essential operations to support the execution of the surrogate machine learning ONNX model. Activation functions like relu and sigmoid were used for non-linear transformations, while matrix operations such as matmul for matrix multiplication and add for element-wise addition were implemented. Convolution operations (conv) and max pooling (maxpool) were used for processing image data and functions like reshape and transpose were used for manipulating tensor dimensions. Additionally, softmax was utilized for output normalization, and gemm handled generalized matrix multiplication.

The backend also included support for other operations such as shape to obtain tensor dimensions, gather for selecting elements along an axis, cast for changing data types, reduceprod for calculating the product of elements along an axis, unsqueeze for adding new dimensions to tensors, and concat for concatenating tensors along a specified axis these operators were needed to enable a CNN-type cellular automata simulation and CNNs in general.

To execute the model, each operator's functions were defined to process each node of the ONNX graph, interpreting and executing operations based on the operator_map. This involved extracting input tensors, applying the corresponding operations, and managing the flow of data through the model. The execute_node function handled individual node execution, while execute_graph managed the overall graph execution, feeding inputs through the model to generate outputs as shown in figure 4. (see onnx_experiment) (Fatah & Chen, 2024; *NETRON*, 2024).

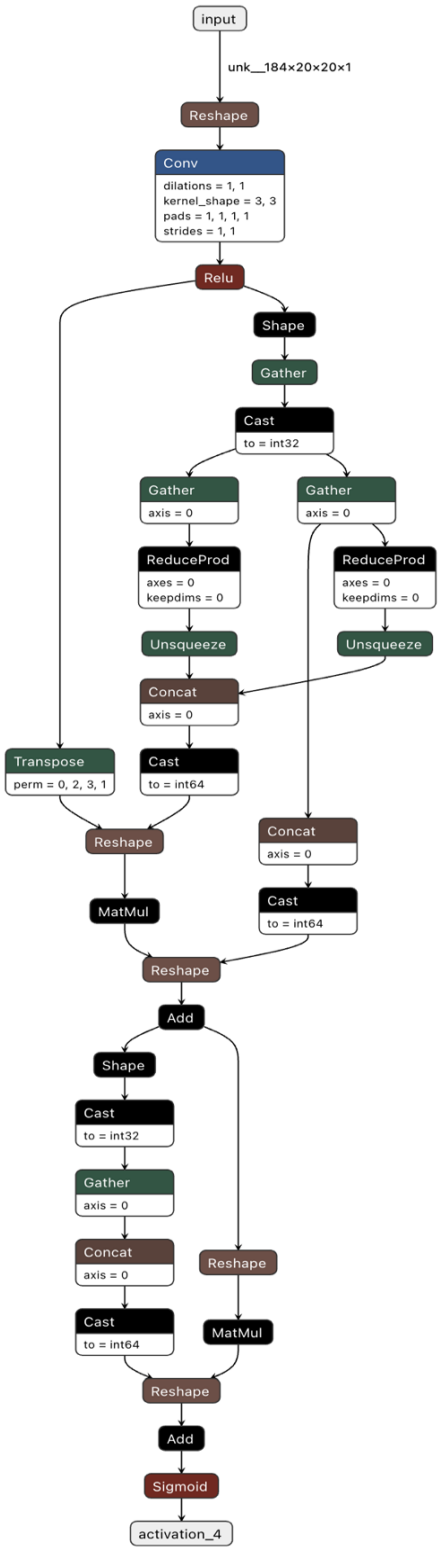

*Figure 10: Figure 4 ML ONNX model operators of the graph visualized using (*NETRON*, 2024)*

5 Discussion & Conclusion

This chapter summarizes the key findings of this research, discusses the broader implications of the results, acknowledges the limitations encountered during the study, and provides suggestions for future research directions based on the insights gained.

This research has demonstrated the feasibility of integrating a surrogate machine learning model into a physically-based model (PCRaster) using a custom-built ONNX backend. The results showcase the potential of this approach to enhance the accuracy and efficiency of simulations, particularly in the context of cellular automata models like Conway's Game of Life.

The custom ONNX backend could successfully be implemented using essential ONNX operators, enabling the execution of the CNN model within the PCRaster framework. The backend supported various operations such as convolution, pooling, activation functions, and tensor manipulations, ensuring compatibility to execute the CNN and ONNX based surrogate machine learning model.

In terms of execution time and efficiency, the PCRaster model significantly outperformed the custom ONNX backend across all scenarios. Although parallelization within the ONNX backend was implemented, it did not provide the expected computational performance improvements and, in some cases, resulted in slower execution times compared to the non-parallelized version, possibly due to increased overhead.

The slower execution time of the different model approaches could be due to differences in the underlying architectures. The PCRaster framework and the onnxruntime, built by Microsoft and Facebook, were developed in C++, which is a compiled language, compared to the custom-built runtime, which was made in Python, an interpreted language. This may explain the considerable differences in execution times (*Python*, 2024; *Standard C++*, 2024).

While the CNN based surrogate machine learning model did not outperform the PCRaster physically based model in execution time for the Game of Life simulation model, this may not necessarily be the case for more complex models, especially when using a compiled language like C++.

The ONNX model integrated within the PCRaster framework achieved scientific performance comparable to the traditional PCRaster model, particularly in shorter simulation runs. However, over 100-time steps, the ONNX model exhibited greater variability and less stability, as evidenced by the fluctuations in the number of alive cells over time. While the PCRaster model quickly stabilized with the number of alive cells dropping to a constant value, the ONNX model's alive cells oscillated significantly throughout the simulation. This inconsistency is further seen in the MSE and accuracy metrics; the MSE of the ONNX model rose and fluctuated before decreasing towards the end, indicating periods of deviation from the PCRaster model. Additionally, the accuracy showed an initial drop and fluctuated without fully recovering, suggesting that the ONNX model's predictions diverged from the PCRaster model. Despite these differences, the ONNX model's average number of alive cells over time was higher than that of the PCRaster model. These observations indicate that while the ONNX model can capture some

dynamics of the PCRaster model, it struggles with maintaining precision and stability over longer simulations.

The comparison of the physically-based and ONNX-based surrogate models in simulating Conway's Game of Life reveals initial consistency showcasing that the convolutional neural network is able to quickly learn the dynamics of the physically based PCRaster model . The ONNX-based surrogate machine learning model, using the CNN architecture, can closely mimic the physically-based model in the short term, although variations accumulate over time. This indicates that the surrogate model's ability to generalize the Game of Life rules is strong but may require further refinement to achieve long-term consistency with the physically-based model. However, the persistence of certain structures across both models, combined with the potential for improved accuracy through further training on larger datasets, suggests that ONNX-based models using CNN architectures could be suitable replacements for physically-based simulations in specific scenarios.

The CNN based surrogate machine learning model achieved 99% accuracy on the training and 100% accuracy on the validation sets almost immediately, which can be attributed to the small dataset and the simplicity of the Game of Life task. However, practical testing revealed that the model's ability to replicate the outputs of the physically-based model was optimal at 50 epochs. This indicates that while the accuracy metrics are high, the number of epochs plays an important role in achieving the best generalization scientific performance for this specific application.

During the conversion of the CNN model to ONNX format, it was noted that the ONNX model defines specific input dimensions it can handle. This shape dependency can be a usability issue, especially for simulation models using frameworks like PCRaster that require flexibility in handling datasets of varying shapes. Initial tests indicate that the ONNX model cannot handle inputs of different shapes without preprocessing. Which means the ONNX model is shape-dependent.

Because of time constraints and the scope of this thesis project, it was not possible to account for every type of spatial machine learning model and each models own selection of operators. This requires finding already built machine learning models which would be either already converted to the ONNX format or need to be manually converted which would be quite time consuming and outside the scope of this research.

Future research could explore several avenues to build upon these findings. One area of focus could be handling unmodeled areas in the input data, which are represented as missing values in the pixel data. A critical distinction must be made between no-data cells and dead cells, as conflating the two leads to incorrect results. To accurately address missing values, support for no-data must be built into the runtime through the ONNX operators. If the current cell is no-data, it must be skipped by most or all operators, resulting in a no-data in the output. Additionally, if a neighboring cell is no-data, convolution operators must also skip these cells. Implementing these strategies could be essential for improving the model's generalizability and accuracy. These additions could potentially enable the model to be applied to more complex and realistic scenarios, addressing gaps and enhancing the overall robustness of the simulations.

Additionally, future work could focus on developing preprocessing techniques to reshape input data dynamically or enhancing the model's architecture to accept variable input dimensions. Investigating alternative machine learning architectures, such as recurrent neural networks (RNNs) or graph neural networks (GNNs), could be valuable for capturing temporal dependencies and complex spatial relationships in cellular automata simulations. Exploring different parallelization strategies, hardware acceleration techniques, or implementing the models in a compiled language like C++ could significantly improve the execution time of the machine learning model. Finally, applying this approach to more complex and realistic environmental models could provide further insights into its potential for enhancing the accuracy and efficiency of physically based simulations in various domains (*Standard C++*, 2024).

6 Contributions

The code for this thesis was a collaborative effort between my thesis partner, Rui Chen, and myself. While we both contributed to all aspects of the project, our specific responsibilities were divided to leverage our individual strengths effectively.

Rui Chen primarily focused on creating the physically-based model. He was responsible for developing the physical simulation model using PCRaster, implementing the Game of Life simulation, and ensuring its accurate representation of physical laws.

On my part, I was primarily responsible for building the ONNX backend, creating and training the ONNX surrogate machine learning model, and integrating and packaging the entire codebase. This included developing the custom backend using Python and NumPy, constructing the CNN model, generating training and validation datasets, and training the model using TensorFlow and Keras. I also combined the physically-based model and the machine learning model into a single model (*Keras: Deep Learning for Humans*, 2024; *NumPy*, 2024; *TensorFlow*, 2024) .

7 Bibliography

Aburas, M. M., Ahamad, M. S. S., & Omar, N. Q. (2019). Spatio-temporal simulation and prediction of land-use change using conventional and machine learning models: A review. *Environmental Monitoring and Assessment*, *191*(4), 205. https://doi.org/10.1007/s10661-019-7330-6

Accuracy explained. (2024). https://www.evidentlyai.com/classification-metrics/accuracy-precision-recall

Bai, J., Lu, F., & Zhang, K. (2019). *Onnx: Open neural network exchange*. https://scholar.google.com/scholar?cluster=8367393578902923505&hl=en&oi=scholarr

Bayyat, S. A., Alomran, A., Alshatti, M., Almousa, A., Almousa, R., & Alguwaifli, Y. (2024). Parallel Inference for Real-Time Machine Learning Applications. *Journal of Computer and Communications*, *12*(1), Article 1. https://doi.org/10.4236/jcc.2024.121010

Carleo, G., Cirac, I., Cranmer, K., Daudet, L., Schuld, M., Tishby, N., Vogt-Maranto, L., & Zdeborová, L. (2019). Machine learning and the physical sciences. *Reviews of Modern Physics*, *91*(4), 045002. https://doi.org/10.1103/RevModPhys.91.045002

Carleo, G., & Troyer, M. (2017). Solving the Quantum Many-Body Problem with Artificial Neural Networks. *Science*, *355*(6325), 602–606. https://doi.org/10.1126/science.aag2302

Chollet, F. (2021). *Deep learning with Python* (Second edition). Manning.

Conway's Game of Life: Scientific American, October 1970. (n.d.). Retrieved June 24, 2024, from https://www.ibiblio.org/lifepatterns/october1970.html

Dask. (2024). https://www.dask.org/

Deep Learning. (n.d.). Retrieved July 7, 2024, from https://www.deeplearningbook.org/

Fatah, Y., (first), & Chen, R. (2024). *Lambdacreator/onnx_experiment* [Jupyter Notebook]. https://github.com/Lambdacreator/onnx_experiment

fkriti. (2024, March 18). *ONNX models: Optimize inference - Azure Machine Learning*. https://learn.microsoft.com/en-us/azure/machine-learning/concept-onnx?view=azureml-api-2

Frost, J. (2021, November 12). *Mean Squared Error (MSE)*. Statistics By Jim. https://statisticsbyjim.com/regression/mean-squared-error-mse/

Jack Dongarra. (2003). Sourcebook of parallel computing. *Choice Reviews Online*, *41*(01), 41-0348-41–0348. https://doi.org/10.5860/CHOICE.41-0348

Jain, M., Agostini, N. B., Ghosh, S., & Tumeo, A. (2024). *Analyzing Inference Workloads for Spatiotemporal Modeling*. https://doi.org/10.2139/ssrn.4772671

Jin, L., Liu, Z., & Li, L. (2023). Prediction and identification of nonlinear dynamical systems using machine learning approaches. *Journal of Industrial Information Integration*, *35*, 100503. https://doi.org/10.1016/j.jii.2023.100503

Jr, F. B., & Qiu, Z. (2012). *An integrated parcel-based land use change model using cellular automata and decision tree*.

Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., & Fei-Fei, L. (2014). *Large-scale Video Classification with Convolutional Neural Networks*. 1725–1732. https://openaccess.thecvf.com/content_cvpr_2014/html/Karpathy_Large-scale_Video_Classification_2014_CVPR_paper.html

Karssenberg, D., Burrough, P. A., Sluiter, R., & de Jong, K. (2001). The PCRaster Software and Course Materials for Teaching Numerical Modelling in the Environmental Sciences. *Transactions in GIS*, *5*(2), 99–110. https://doi.org/10.1111/1467-9671.00070

Keras: Deep Learning for humans. (2024). https://keras.io/

Kudela, J., & Matousek, R. (2022). Recent advances and applications of surrogate models for finite element method computations: A review. *Soft Computing*, *26*(24), 13709–13733. https://doi.org/10.1007/s00500-022-07362-8

Kumawat, T. (2023, November 21). Accelerating Model Inference Through Parallel Processing for Enhanced Speed. *Medium*. https://medium.com/@tejpal.abhyuday/parallel-processing-at-model-inferencing-for-speed-up-53662ef716d0

Lee, S. W. (2023). *Exploring the Potential of Convolutional Neural Network (CNNs) for GIS Applications*.

Loss in Machine Learning. (2024). https://www.datacamp.com/tutorial/loss-function-in-machine-learning

NETRON. (2024). https://netron.app/

Nguyen, Y. H. (2022). *Scoliosis Detection using Deep Neural Network* (arXiv:2210.17269). arXiv. http://arxiv.org/abs/2210.17269

NumPy. (2024). https://numpy.org/

ONNX.ai. (2024). https://onnx.ai/

PySpark. (2024). https://spark.apache.org/docs/latest/api/python/

Python. (2024, July 2). Python.Org. https://www.python.org/

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257–286. https://doi.org/10.1109/5.18626

Standard C++. (2024). https://isocpp.org/

Stephen Wolfram: A New Kind of Science. (2002). https://www.wolframscience.com/nks/

TensorFlow. (2024). https://www.tensorflow.org/

Xing, W., Qian, Y., Guan, X., Yang, T., & Wu, H. (2020). A novel cellular automata model integrated with deep learning for dynamic spatio-temporal land use change simulation. *Computers & Geosciences*, *137*, 104430. https://doi.org/10.1016/j.cageo.2020.104430

Yeh, A. G.-O., & Li, X. (2002). Urban Simulation Using Neural Networks and Cellular Automata for Land Use Planning. In D. E. Richardson & P. van Oosterom (Eds.), *Advances in Spatial Data Handling* (pp. 451–464). Springer. https://doi.org/10.1007/978-3-642-56094-1_33

Appendix A

Code listing: Building ML Model

```
# Load data from 8000 train and 2000 valid samples stored in .npy files from
physically based model outputs
import numpy as np
import onnx

X_train = np.load("X_train.npy")
X_val = np.load("X_val.npy")

def reshape_input(X):
    return X.reshape(X.shape[0], X.shape[1], X.shape[2], 1)

X_train = reshape_input(X_train)
X_val = reshape_input(X_val)

def life_step(X):
    live_neighbors = sum(np.roll(np.roll(X, i, 0), j, 1)
                     for i in (-1, 0, 1) for j in (-1, 0, 1)
                     if (i != 0 or j != 0))
    return (live_neighbors == 3) | (X & (live_neighbors == 2)).astype(int)
y_train = np.array([life_step(frame) for frame in X_train])
y_val = np.array([life_step(frame) for frame in X_val])

# CNN Properties
filters = 50
kernel_size = (3, 3) # look at all 8 neighboring cells, plus itself
strides = 1
hidden_dims = 100

model = Sequential()
model.add(Conv2D(filters, kernel_size, padding='same',
activation='relu',strides=strides, input_shape=(board_shape[0], board_shape[1], 1)))
model.add(Dense(hidden_dims))
model.add(Dense(1))
model.add(Activation('sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

def train(model, X_train, y_train, X_val, y_val, batch_size=50, epochs=5,
filename_suffix=''):
    model.fit(
        X_train, y_train,
        batch_size=batch_size,
        epochs=epochs,
        validation_data=(X_val, y_val) )
```

```
train(model, X_train, y_train, X_val, y_val, filename_suffix='_basic')



model_save_path = './keras_model.h5'
model.save(model_save_path)

input_spec = [tf.TensorSpec([None, board_shape[0], board_shape[1], 1], tf.float32,
name='input')]
onnx_model, _ = tf2onnx.convert.from_keras(model, input_signature=input_spec,
opset=13)

# Save the ONNX model to disk
onnx_model_path = 'game_of_life.onnx'
with open(onnx_model_path, "wb") as f:
    f.write(onnx_model.SerializeToString())
```

Appendix B

Code listing: Multithreading Backend

```
import numpy as np
import onnx.numpy_helper as numpy_helper
from concurrent.futures import ThreadPoolExecutor

def relu(x):
    return np.maximum(0, x)

def add(a, b):
    return np.add(a, b)

def matmul(a, b):
    return np.dot(a, b)

def reshape(tensor, shape):
    return np.reshape(tensor, shape)

def conv_worker(x_slice, w, b, i, j, out_channels):
    result = np.zeros((x_slice.shape[0], out_channels), dtype=np.float32)
    for k in range(out_channels):
        result[:, k] = np.sum(x_slice * w[k, :, :, :], axis=(1, 2, 3))
    if b is not None:
        result += b.reshape(1, -1, 1, 1)[:, :, 0, 0]
    return i, j, result

def conv(x, w, b=None, strides=(1, 1), pads=(0, 0, 0, 0)):
    batch_size, in_channels, in_height, in_width = x.shape
    out_channels, _, kernel_height, kernel_width = w.shape
    stride_height, stride_width = strides
    pad_height_begin, pad_width_begin, pad_height_end, pad_width_end = pads

    out_height = (in_height + pad_height_begin + pad_height_end - kernel_height) //
stride_height + 1
    out_width = (in_width + pad_width_begin + pad_width_end - kernel_width) //
stride_width + 1

    y = np.zeros((batch_size, out_channels, out_height, out_width), dtype=np.float32)

    x_padded = np.pad(x, ((0, 0), (0, 0), (pad_height_begin, pad_height_end),
(pad_width_begin, pad_width_end)), mode='constant')

    with ThreadPoolExecutor() as executor:
        futures = []
        for i in range(out_height):
            for j in range(out_width):
```

```
                h_start = i * stride_height
                h_end = h_start + kernel_height
                w_start = j * stride_width
                w_end = w_start + kernel_width
                x_slice = x_padded[:, :, h_start:h_end, w_start:w_end]
                futures.append(executor.submit(conv_worker, x_slice, w, b, i, j,
out_channels))

        for future in futures:
            i, j, result = future.result()
            y[:, :, i, j] = result

    return y

def maxpool_worker(x_slice, i, j, in_channels):
    result = np.zeros((x_slice.shape[0], in_channels), dtype=np.float32)
    for c in range(in_channels):
        result[:, c] = np.max(x_slice[:, c, :, :], axis=(1, 2))
    return i, j, result

def maxpool(x, kernel_shape, strides=(1, 1), pads=(0, 0, 0, 0)):
    batch_size, in_channels, in_height, in_width = x.shape
    kernel_height, kernel_width = kernel_shape
    stride_height, stride_width = strides
    pad_height_begin, pad_width_begin, pad_height_end, pad_width_end = pads

    out_height = (in_height + pad_height_begin + pad_height_end - kernel_height) //
stride_height + 1
    out_width = (in_width + pad_width_begin + pad_width_end - kernel_width) //
stride_width + 1

    y = np.zeros((batch_size, in_channels, out_height, out_width), dtype=np.float32)

    x_padded = np.pad(x, ((0, 0), (0, 0), (pad_height_begin, pad_height_end),
(pad_width_begin, pad_width_end)), mode='constant')

    with ThreadPoolExecutor() as executor:
        futures = []
        for i in range(out_height):
            for j in range(out_width):
                h_start = i * stride_height
                h_end = h_start + kernel_height
                w_start = j * stride_width
                w_end = w_start + kernel_width
                x_slice = x_padded[:, :, h_start:h_end, w_start:w_end]
                futures.append(executor.submit(maxpool_worker, x_slice, i, j,
in_channels))
```

```
        for future in futures:
            i, j, result = future.result()
            y[:, :, i, j] = result

    return y

def transpose(x, perm):
    return np.transpose(x, perm)

def softmax(x, axis=-1):
    e_x = np.exp(x - np.max(x, axis=axis, keepdims=True))
    return e_x / e_x.sum(axis=axis, keepdims=True)

def shape(x):
    return np.array(x.shape)

def gather(x, indices, axis=0):
    return np.take(x, indices, axis=axis)

def cast(x, to):
    onnx_to_numpy_dtype = {
        1: np.float32,
        2: np.uint8,
        3: np.int8,
        4: np.uint16,
        5: np.int16,
        6: np.int32,
        7: np.int64,
        8: np.str_,
        9: np.bool_,
        10: np.float16,
        11: np.double,
        12: np.uint32,
        13: np.uint64,
        14: np.complex64,
        15: np.complex128
    }
    return x.astype(onnx_to_numpy_dtype[to])

def reduceprod_worker(x_slice, axes):
    return np.prod(x_slice, axis=axes, keepdims=False)

def reduceprod(x, axis=None, keepdims=False):
    if axis is not None:
        with ThreadPoolExecutor() as executor:
            futures = [executor.submit(reduceprod_worker, x, ax) for ax in axis]
            results = [future.result() for future in futures]
        for res in results:
```

```
            x = res
        return x
    else:
        return np.prod(x, keepdims=keepdims)

def unsqueeze(x, axes):
    for axis in sorted(axes):
        x = np.expand_dims(x, axis)
    return x

def concat_worker(tensor, axis):
    return np.atleast_1d(tensor)

def concat(*tensors, axis=0):
    with ThreadPoolExecutor() as executor:
        futures = [executor.submit(concat_worker, tensor, axis) for tensor in tensors]
        tensors = [future.result() for future in futures]
    return np.concatenate(tensors, axis=axis)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def gemm_worker(a, b, alpha, beta, trans_a, trans_b):
    if trans_a:
        a = a.T
    if trans_b:
        b = b.T
    return alpha * np.dot(a, b)

def gemm(a, b, c=None, alpha=1.0, beta=1.0, trans_a=False, trans_b=False):
    with ThreadPoolExecutor() as executor:
        future = executor.submit(gemm_worker, a, b, alpha, beta, trans_a, trans_b)
        y = future.result()
    if c is not None:
        y += beta * c
    return y

# Map ONNX operator names to functions
operator_map = {
    'Relu': relu,
    'Add': add,
    'MatMul': matmul,
    'Reshape': reshape,
    'Conv': conv,
    'MaxPool': maxpool,
    'Transpose': transpose,
    'Softmax': softmax,
    'Shape': shape,
```

```
    'Gather': gather,
    'Cast': cast,
    'ReduceProd': reduceprod,
    'Unsqueeze': unsqueeze,
    'Concat': concat,
    'Sigmoid': sigmoid,
    'Gemm': gemm,
    # Add more operators
}

def execute_node(node, inputs):
    input_tensors = [inputs[input_name] for input_name in node.input]

    if node.op_type == 'Reshape':
        shape_tensor = input_tensors[1]
        if isinstance(shape_tensor, list):
            shape_tensor = np.array(shape_tensor)
        output_tensors = operator_map[node.op_type](input_tensors[0], shape_tensor)
    elif node.op_type == 'Conv':
        attrs = {attr.name: attr.ints for attr in node.attribute}
        strides = tuple(attrs.get('strides', [1, 1]))
        pads = tuple(attrs.get('pads', [0, 0, 0, 0]))
        b = input_tensors[2] if len(input_tensors) > 2 else None
        output_tensors = operator_map[node.op_type](input_tensors[0],
input_tensors[1], b, strides, pads)
    elif node.op_type == 'MaxPool':
        attrs = {attr.name: attr.ints for attr in node.attribute}
        kernel_shape = attrs.get('kernel_shape', [2, 2])
        strides = tuple(attrs.get('strides', [1, 1]))
        pads = tuple(attrs.get('pads', [0, 0, 0, 0]))
        output_tensors = operator_map[node.op_type](input_tensors[0], kernel_shape,
strides, pads)
    elif node.op_type == 'Transpose':
        perm = node.attribute[0].ints if node.attribute else []
        output_tensors = operator_map[node.op_type](input_tensors[0], perm)
    elif node.op_type == 'Gather':
        indices = input_tensors[1]
        axis = node.attribute[0].i if node.attribute else 0
        output_tensors = operator_map[node.op_type](input_tensors[0], indices, axis)
    elif node.op_type == 'Cast':
        to = node.attribute[0].i  # The data type to cast to (ONNX data type enum)
        output_tensors = operator_map[node.op_type](input_tensors[0], to)
    elif node.op_type == 'ReduceProd':
        axis = node.attribute[0].ints if node.attribute else None
        if axis is not None:
            axis = list(axis)  # Convert to list if it's a RepeatedScalarContainer
        keepdims = node.attribute[1].i if len(node.attribute) > 1 else False
        output_tensors = operator_map[node.op_type](input_tensors[0], axis, keepdims)
```

```
    elif node.op_type == 'Unsqueeze':
        axes = node.attribute[0].ints if node.attribute else []
        output_tensors = operator_map[node.op_type](input_tensors[0], axes)
    elif node.op_type == 'Concat':
        axis = node.attribute[0].i if node.attribute else 0
        output_tensors = operator_map[node.op_type](*input_tensors, axis=axis)
    elif node.op_type == 'Sigmoid':
        output_tensors = operator_map[node.op_type](input_tensors[0])
    elif node.op_type == 'Gemm':
        attrs = {attr.name: attr for attr in node.attribute}
        alpha = attrs['alpha'].f if 'alpha' in attrs else 1.0
        beta = attrs['beta'].f if 'beta' in attrs else 1.0
        trans_a = attrs['transA'].i if 'transA' in attrs else 0
        trans_b = attrs['transB'].i if 'transB' in attrs else 0
        c = input_tensors[2] if len(input_tensors) > 2 else None
        output_tensors = operator_map[node.op_type](input_tensors[0],
input_tensors[1], c, alpha, beta, trans_a, trans_b)
    else:
        output_tensors = operator_map[node.op_type](*input_tensors)

    if not isinstance(output_tensors, tuple):
        output_tensors = (output_tensors,)

    for output_name, output_tensor in zip(node.output, output_tensors):
        inputs[output_name] = output_tensor

def execute_graph(graph, input_data):
    inputs = {}

    # Provided input data
    for input_tensor in graph.input:
        input_name = input_tensor.name
        inputs[input_name] = input_data

    # Initialize tensors for constants (initializers)
    for initializer in graph.initializer:
        tensor = numpy_helper.to_array(initializer)
        inputs[initializer.name] = tensor

    # Execute nodes
    for node in graph.node:
        execute_node(node, inputs)

    # Extract outputs
    outputs = {output.name: inputs[output.name] for output in graph.output}
    return outputs
```