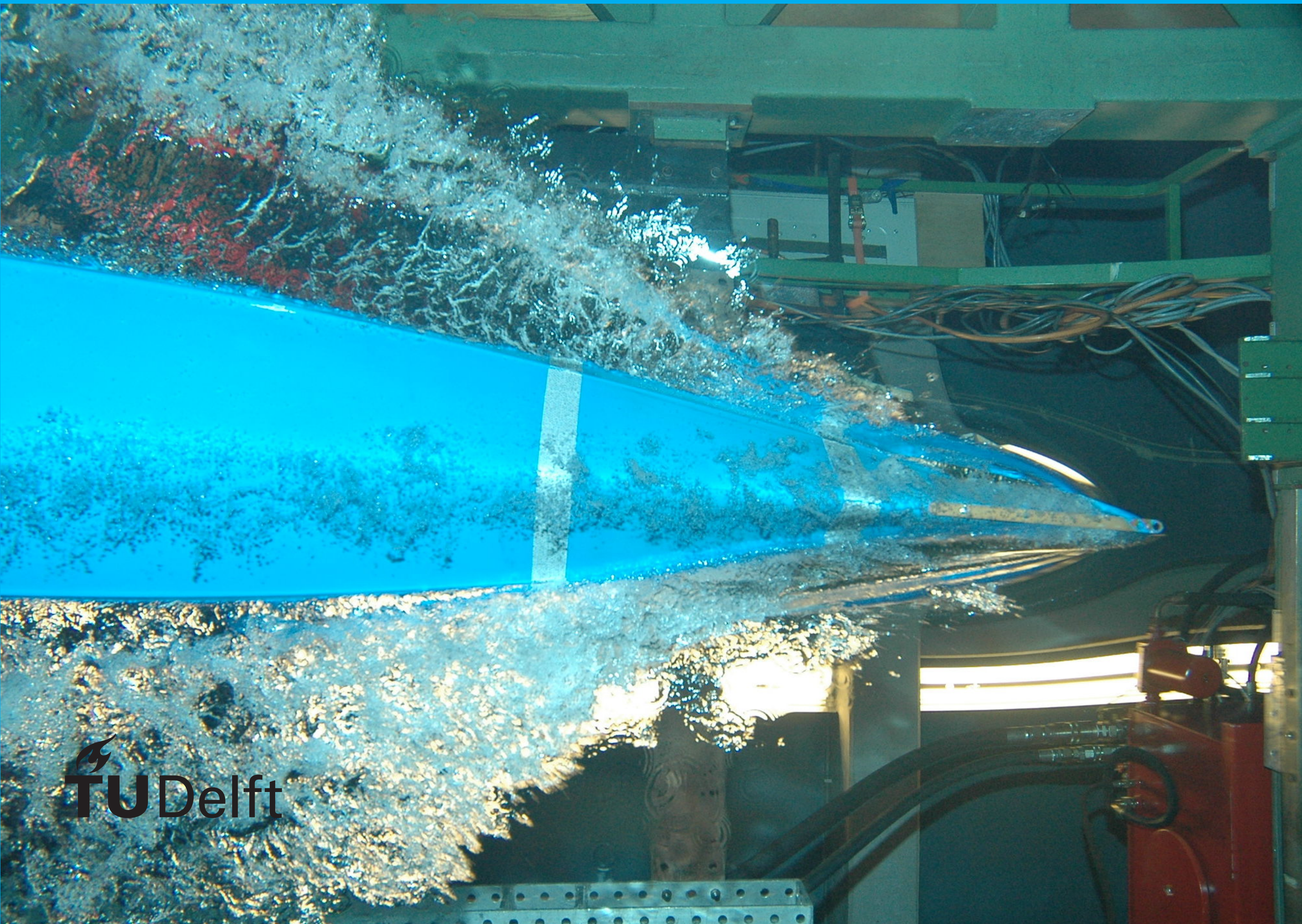
Observable Simulator of SNNs

Sander Renger



Observable Simulator of SNNs

by

Sander Renger

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday August 22, 2023 at 10:30 AM.

Student number:	4483510	
Project duration:	September 1, 2023 – August 23, 2024	
Thesis committee:	Prof. dr. ir. G. Gaydadjiev,	TU Delft, supervisor
	Dr. R. Venkatesha Prasad,	TU Delft

Acknowledgments

Progressing on this thesis has been quite the challenge sometimes. To ensure this thesis would meet its deadline I'd like to thank a few people for both their technical and mental support.

First, I'd like to thank my supervisor Prof.dr.ir. Georgi Gaydadjiev from Delft University of Technology. The weekly meeting proved very helpful for both the progress of the thesis and keeping a positive spirit.

Secondly, I'd like to thank Theofilos Spyrou. His insights into Spiking Neural Networks were a great boon.

Thirdly, I'd like to thank all my friends, who kept encouraging me to keep working on my thesis.

Lastly and maybe most importantly, I'd like to thank my parents for not just their support during the thesis but the entirety of my university journey.

Sander Renger
Delft, August 2024

Contents

List of Figures

List of Tables

Abbreviations

ABC	Atanasoff–Berry computer
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
ARCO	Automatische Relais Calculator voor Optische berekeningen
CPU	Central Processing Unit
DVS	Dynamic Vision Sensor
ENIAC	Electronic Numerical Integrator and Computer
EWI	Elektrotechniek, Wiskunde en Informatica
FET	Field-Effect Transistor
IC	Integrated Circuit
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
GPU	Graphical Processing Unit
MSE	Mean Square Error
PSP	Post Synaptic Potential
SNN	Spiking Neural Network

Abstract

With the rise of the artificial intelligence starting in the late 2010s there has been a great increase of demand for neural networks. It seems AI is omnipresent these days, whether it is OpenAI releasing the famous or infamous large language model chatGPT, Google following by adopting a similar approach with integrating Artificial Intelligence (AI) into it's search engine or help desks becoming entirely controlled by AI. It is clear, AI is here to stay. NVIDIA normally a company thought of selling Graphical Processing Units (GPUs) for gaming purposes has shifted its focus to making GPUs used for running neural networks, this shift has shown its returns as it is now the most valuable company in the world.

The increased interest in AI does leave a lot of academia wondering: What actually is artificial intelligence and especially what are neural networks. With the introduction of deep learning, a machine learning method that searches for features in a dataset, neural networks have become more and more abstract. Developing an appropriate solution to better interpret, visualize and adjust the inner workings of a neural network is not just needed to keep track of the underlying dynamics of a neural network, but also to allow the user to go outside of the standardized framework to fine tune a network.

This thesis explores the functionality of neural networks, more specifically spiking neural networks, and describes the workings and functions of the created simulator. Multiple helper functions, that allow the user to both visualize and adjust the network outside of the standardized framework, are introduced that distinguishes the created simulator from the already present SNN simulators. Afterwards, five small simulations are done to validate the basic functionality of the simulator. Following these simulations the simulator is validated using two large datasets and comparing the simulator with the results from a well known spiking neural network library SLAYER. The first validation is done with a smaller data set containing hand written images of the numbers zero to nine. The second validation is done with a larger data set containing multiple different kind of performed hand gestures. Both validations show that the simulator has a high accuracy with the maximum mean square error between the SLAYER output and the simulators output being less than 2%.

1 Introduction

Within this chapter the topics of the thesis, the goal of the thesis and other related works, are introduced. Section 1.1 talks about the motivation for the thesis, the shortcomings of Spiking Neural Network simulators, or (SNN) simulators, and possible improvements. Section 1.2 details the contributions done by this project. Lastly, in final the section 1.3 the thesis layout is described.

1.1. Motivation

What started as simple neural networks able to beat amateur checkers players in 1959 [17], has now more than 50 years later turned into the fastest growing industry in the world [9]. The application of Artificial intelligence (AI) as a whole is growing exponentially, even more so SNNs. SNNs operate in a similar manner to the functioning of a human brain, which suggests their far reaching potential. Currently, however, SNNs are used primarily in the field of image recognition, where the ability to quickly adapt to change is vital.

The advancement of image recognition technology is paramount for the future of many autonomous machines. As seen in figure 1.1 the self driving car market is growing rapidly, doubling its size in less than 10 years. With more and more self driving cars on the road [8], safety is an ever growing concern [20]. To ensure the safety of people inside and the people outside self driving cars, we have to guarantee that image recognition is infallible.

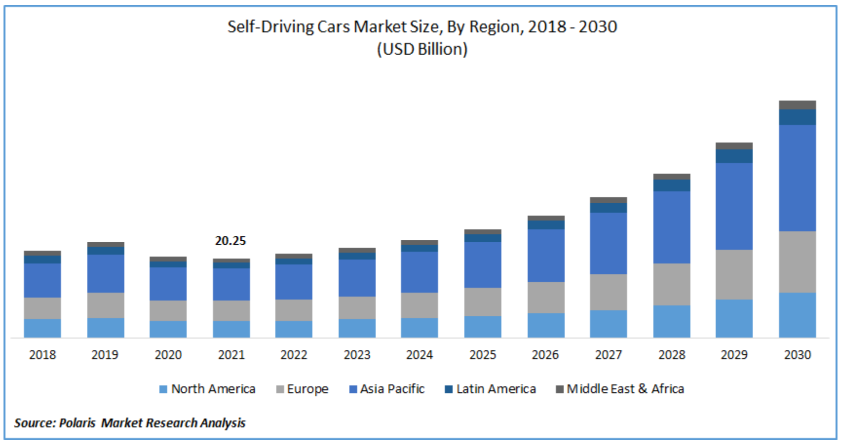


Figure 1.1: The market growth of self driving cars [1]

Surveillance is another field where image recognition is important, The surveillance camera market has seen a significant increase in sales and is only expected to increase more, as seen in figure 1.2. The surveillance market is not just restricted to government surveillance but also smart doorbells, highway speed controls or general building security.

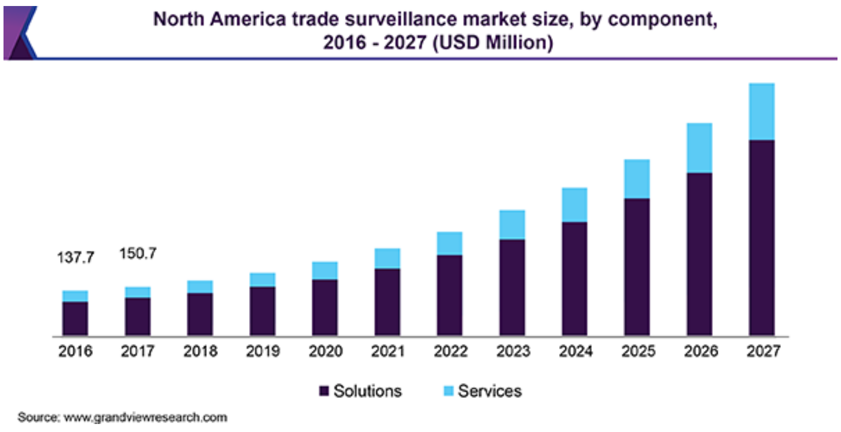


Figure 1.2: The market growth of surveillance cameras in North America [2]

The growth of real-time image recognition demands is supported by the application of advanced spiking neural networks. With SNNs become more advanced, it becomes difficult to see what happens inside the black box of an SNN. This lack of observability can be detrimental for both fine tuning and debugging of an SNN. In order to streamline the development of SNNs, state-of-the-art SNNs use a standardized framework of layers, a grouping of neurons. Adjusting neurons in a way that would make the layer no longer comply with the framework is difficult within state-of-the-art SNNs.

1.2. Contribution

In this thesis, the objective is to increase the observability and control of a trained SNN. The developed simulator, aptly named SNNSim, has multiple added functions that both help to visualize the spiking functions of the neural network and allow for a better view and control of the connectivity inside the neural network.

The main contributions of the thesis are:

- **Increasing the observability of connectivity between neurons:** The added observability functions allow the user to more easily understand the mapping of SNNSim. By requesting the connectivity lists of the interested neurons the user is able to see the mapping of the requested neuron.

- **Increasing the control of connectivity between neurons:** Multiple control functions allow for both adding, deleting and editing existing connections between neurons, outside of the more established framework that current state-of-the-art SNN use.

- **Visualization of the spike response functions:** By extracting the spike response function of a neuron, the potential of the specified neuron over the duration of the entire simulation can be shown in a graph.

1.3. Thesis Organization

The remainder of the thesis is divided up into several chapters. The second chapter provides an introduction into Machine Learning, Spiking Neural Networks and their simulators. The third chapter entails the design of my simulator. The fourth chapter validates the functionality of SNNSim with small scale simulations. The fifth chapter explains the method of validating SNNSim against a state-of-the-art SNN. Finally, the last chapter concludes the thesis. The following list provides a more detailed explanation of every chapter:

Chapter 2: A quick introduction of Machine Learning and Spiking Neural Networks. It introduces the different types of layers in a Spiking Neural Network and the way spikes are transmitted. Finally, two simulators, Pytorch and Tensorflow, are introduced and compared.

Chapter 3: Introduces the different components of SNNSim and its design choices. Secondly, the different observability and control functions are introduced.

Chapter 4: The functionality of SNNSim is validated with multiple simulations. The first simulation is about testing the connectivity of the neurons and validates the ability for neurons to spike. The second simulation validates the ability to change the weights and threshold of a neuron. The third simulation is about different spike arrival times. The fourth simulation is on inhibitory connections. The fifth and final simulation validates the refractory function of the neuron.

Chapter 5: SNNSim is validated with two SNNs both provided by the SLAYER library for Pytorch, the first network is validated with Multilayer Perception and the second network with a DVSGesture data set. The last part of the chapter gives a conclusion on the validation of SNNSim.

Chapter 6: A summary of the chapters within the thesis is given and the thesis is given a conclusion. Lastly, the chapter provides suggestions for possible future works on SNNSim.

2 Introduction to Neural Networks and Simulators

The first section 2.1 introduces neural networks and gives a short background on both AI and neural networks. The second section 2.2 describes the inner workings of a neural network. The third section 2.3 goes into further detail on the specifics of an SNN. The third and final section 2.4 introduces and compares the two most prominent neural network libraries, Pytorch and TensorFlow.

2.1. Background

Back in the 17th century a computer meant someone who computes. In 1820 the English inventor Charles Babbage invented the first modern mechanical computer, the Difference Engine. These computers worked on intricate gear systems used to do simple arithmetic calculations. Not long after the second world war the first electronic computers came to the forefront.

The Atanasoff–Berry computer (ABC), developed in 1942 by is the first fully electronic computer to be invented. The ABC is the first electronic computer to introduce arithmetic logic unit (ALU). An ALU allows a different kind of operations to be executed on inputs depending on a secondary signal that determines the type of operation to be performed, for instance: addition, subtraction, multiplication and division.

TUDelft did not shy away during the early development electronic computers either. The Automatische Relais Calculator voor Optische berekeningen (ARCO), better known as the Testudo, is an early computer designed by the Dutch computer scientist Willem van der Poel. The Testudo was a relatively slow computer, but it was capable of executing calculations day and night. This was an impressive feat compared to other computers of the time, which required maintenance every few hours. The slow speed yet long endurance of the computer explains the name 'Testudo', the latin name for tortoise. Willem van der Poel was also responsible for developing other notable computers such as the ZEBRA, the ZERO, and the PTERA. Each of these machines contributed to advancements in computing during that era. Most of Van der Poel's computers are still visible in the cellar museum below the TUDelft faculty of Elektrotechniek, Wiskunde en Informatica (EWI).

Starting with the ABC computer, vacuum tubes, which function as electronic switches without moving parts, became crucial in the early development of computers. The Electronic Numerical Integrator and Computer (ENIAC) was the first large-scale implementation of a computer with vacuum tubes. The ENIAC however, did also show the problems vacuum tubes have. A computer with 18.000 vacuum-tubes, a relatively small number of switches by today's standards, required 150 kW of power and occupied a space of 28 m^2. Both these problems, high power usage and large size, caused a so to speak 'computer winter' with respect to the development of computers.

The 'computer winter' saw its first signs of ending with the rapid development of the first transistors. One of the earliest transistors, the Bipolar Junction Transistor, functions by applying a constant current to control the conductivity of a semi-conductor, allowing it to act as an electronic switch. The first transistors were made with the semi-conductor germanium. Germanium is fragile and difficult to work with, on top of that the workable temperature range is less than ideal. The transition from germanium as a substrate to silicon, a semi-conductor with a higher temperature range and a lower voltage leakage when turned off, was the second sign the 'computer winter' was ending. The final sign was the introduction of the Field-Effect Transistor (FET). Compared to the BJT the FET uses an electric field to control the conductivity of a semi-conductor, thereby eliminating the need for a continuous current flow and significantly reducing overall energy consumption. The integration of silicon as substrate with the FET technology allowed for the development of Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET). With the advent of MOSFETs the development of integrated circuits (ICs), an electronic device of multiple interconnected components like transistors etched into a semiconductor, started to accelerate. Not long after the introduction of IC, Gordon Moore, one of the founders of intel, observed that the components in IC seemed to double every year [13]. The term now known as Moore's law, slightly adjusted in later years to doubling every other year, seems to still hold up to this day.

Nowadays, computers in the modern sense are everywhere, from the mobile phones in our pockets to the large data centers scattered around the country, supporting everything from personal devices to large scale online computing services that drive global communication and industry.

The emergence of artificial intelligence shares a similar story as the rise of computers, not just because both fields are related but also in the ways the rose to prominence.

During the second world war the Germans had invented the then most heavily encrypted messaging device, the Enigma machine. With this computing unit the Germans were capable of sending encrypted messages all across the globe with impunity.

Shortly after the outbreak of the war Alan Turing started working as a codebreaker at Bletchley Park. In 1940 Alan Turing developed his new code breaking machine the Bombe, an electro-mechanical machine. The Bombe saw its first success with the breaking of a German submarine messages. Alan Turing kept improving his code breaking machine and played a very important role in winning the second world war.

In his 1950 paper: "Computing Machinery and Intelligence" [19] Alan Turing starts the field of artificial intelligence by asking himself: Can you build a machine that is capable of being trained in a similar way as how humans learn new information. With this thought in mind Alan Turing proposed the Turing Test. In a Turing test a participant is having a conversation with either a machine or a human, if the participant cannot tell whether it is a person or a machine, the machine passes the Turing test. The publication of this paper is often considered as the starting point of the field of machine learning.

The first neural network that was capable of learning from data provided to it, is the Neural Network named Perceptron, developed in 1957 by Frank Rosenblatt [15]. The Perceptron is the simplest form of a Neural Network that takes inputs and weights these inputs to create an output. The network is trained by adjusting the weights in such a way to get a more desirable output, this all was done by hand. In 1969 both Minsky and Papert published a paper outlining the limitations of the Perceptron and Simple Neural Networks like the Perceptron in general [12]. With the publication of this paper the development of artificial intelligence and neural networks entered a winter period with little to no new development.

The winter ended with the invention of the back propagation algorithm developed by Rumelhart, Hinton, and Williams in 1986 [16]. This algorithm allowed for the training of more complex neural networks, removing the limitations that stagnated the field of artificial intelligence.

The supercomputer Deep Blue caught the world by storm as it beat the world champion chess player, Garry Kasparov, in a six-game chess match in 1997. The Chinese board game Go on the other hand remained elusive for a much longer period. The difficulty in creating an AI capable of beating a professional go player becomes clear when comparing the two games: Chess has an approximate 10^{40} possible positions in go on the other hand there are an approximate 10^{170} positions. On top of the difference in amount of positions, the complexity in the value of a move is a lot higher in Go compared to chess. Despite the elusiveness of Go the victory against Garry Kasparov marks a significant win for artificial intelligence. The confidence that there is a future in AI started to take off.

The introduction of more advanced Graphical Processing Units (GPUs) gave way to the current AI revolution. GPUs allow for more and faster computations than what a central processor unit (CPU) is capable off. This coincided with the emergence large scale cloud computing. Cloud computing grants a person the ability to access the large computing capabilities present in data centers over the internet. Togehter both the development of more advanced GPUs and the increased accessibility cloud computing provides, gave rise to the current state-of-the-art neural networks, such as like OpenAI's ChatGPT and Google's Gemini.

2.2. Neural Networks

In the section Neural Networks the functionality of a neural network is introduced. The first part 2.2.1 introduces the modules and functions of a neural network. In the second part of the section 2.2.2 layers are introduced with their functionalities. In the final part 2.2.3 the method of training a neural network is discussed.

2.2.1. Components

A Neural Network [10] is a computational model inspired by the human brain's processing abilities. It consists of neurons, which are organized in groups also known as layers. The data flows from one layer of neurons to another until it reaches the output layer. A neural network can be divided into multiple components and functions. The Components of a Neural Network are as follows:

- **Neuron:** The main and basic computing unit in a Neural Network. All the calculations happen inside this unit. The neuron both receives and sends information from one neuron to another neuron.

- **Layers:** A Neural Network is divided up into multiple layers. The input layer takes in the data, usually from either a data set or real-life data. The hidden layers are the intermediate layers where the calculations happen. The output layer is the final layer inside a Neural Network and describes the output of a Neural Network.

- **Connections:** The neurons inside the layers of a Neural Network are connected to neurons inside the next layer in various ways depending on the type of layer used. These connections all have different weights and biases. The weights are variable to differentiate the importance of each connection. The biases on the other hand are present to counteract the over reliance on too highly weighted connections.

The functions of a basic neural network are as follows:

- **Forward Propagation.** The transfer of data between the layers is done using forward propagation. From the input data at the input layer through the different weights from each connection the final output data is generated with this forward propagation.

- **Backwards Propagation.** The process of learning is done with backwards propagation. Backwards propagation is an algorithm that calculates the loss function of a connection. The loss function is a formula that compares the current output with the desired output. A common type of loss function is the mean squared error (MSE) loss function. The common MSE loss function is in the format of $\frac{1}{n}\sum_{n=1}^{n}(y_{desired} - y)^2$. Using the loss function the weights or gradients are adjusted to reduce the loss of each connection.

2.2.2. Types of layers

Multiple types of layers are used in neural networks. The most commonly used layer types in a neural network are: The Pooling layer, a layer used to condense the data from the previous layer with the intention to lower the data points without losing too much of the information. The Convolution layer, a layer used to select different type of characteristics from the previous layer. The fully connected, a layer used to combine all the previously collected information and condense it into a usable output.

Pooling Layer

The Pooling layer is a layer used to compress the spatial dimensions of a layer without losing much of the data. The inputs of a pooling layer are determined by the pooling window.

As an example, the first neuron of a pooling layer where the indexes represent the xy-axis is as follows: $Neuron[y][x]$. In a pooling layer, with window size of two by two, this neuron takes the following outputs of the neurons in the previous layer as inputs:

$$Neuron[y][x], Neuron[y][x+1],$$
$$Neuron[y+1][x], Neuron[y+1][x+1]$$

There are multiple types of pooling layers as seen in figure 2.1, these types of layers are: Average, Summation and Maximum.

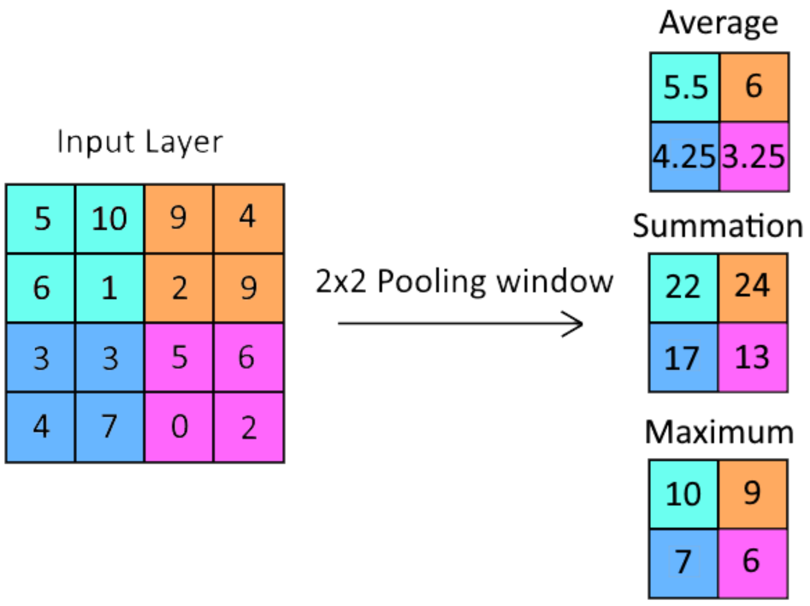


Figure 2.1: Different kinds of Pooling Layers

The average pooling layer takes the average value of the values inside the pooling window as its output. The Summation pooling layer sums the values inside the pooling window. The maximum pooling takes the highest valued number inside the pooling window and uses it as its output.

Convolution Layer
The convolution layer has the function of selecting desired characteristics from a layer using a window, better known as a mask. The weights of a convolution layer window are, unlike the pooling layer, not all the same. Because the weights of different positions on the mask differ, a mask can specifically select for desired characteristics from the previous layer. A convolution layer can exist of multiple different weighted masks and can therefore create multiple different output dimensions, where each mask creates a different dimension. The figure 2.2 shows this clearly. The input image of a bag is taken through a convolution layer with 32 masks that all have different weights. These masks all search for different characteristics creating 32 distinct new images of a bag.

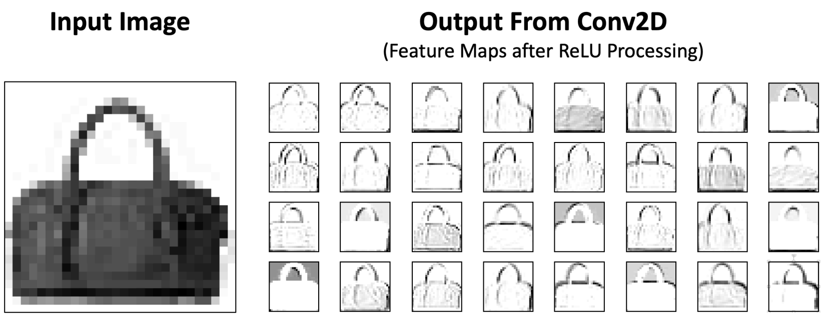


Figure 2.2: 32 different kind of masks applied on the input image of a bag[3]

The convolution layer spatial mapping works on the same principle as the pooling layer, but whereas in the pooling layer the window moves with the size of the pooling window, the convolution layer does not. As shown in table 2.1 in the convolution layer inputs are centered around the outputs neuron window, compared to the pooling layer where the output neuron is located on the top left of the window.

Table 2.1: The inputs neurons of a convolution layer with a window size of 3

Neuron[y-1][x-1]	Neuron[y-1][x]	Neuron[y-1][x+1]
Neuron[y][x-1]	Neuron[y][x]	Neuron[y][x+1]
Neuron[y+1][x-1]	Neuron[y+1][x]	Neuron[y+1][x+1]

As seen in figure 2.3, a convolution layer using a three by three convolution window, the layer give a problematic output. The border regions of the output layer are undefined, shown in the figure as white. This problem arises when for instance both x and y are zero and the convolution layer tries to access the $Neuron[y-1][x-1]$, resulting in negative indexes.

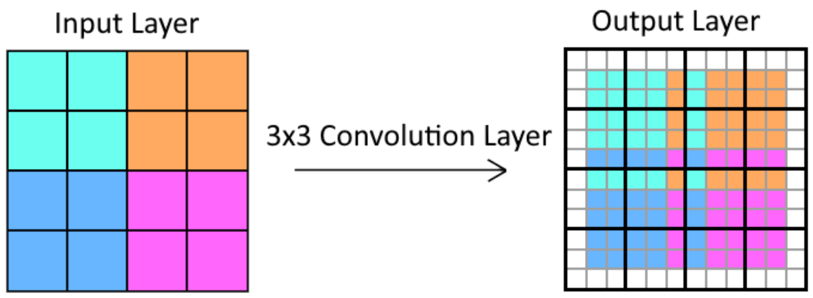


Figure 2.3: A Convolution Layer

As the Neurons with negative indexes do not exist, taking the convolution of all border neurons is impossible. To circumvent this problem, zero padding is used. With zero padding temporary neurons are created around the border with a connection weight of zero. To ensure that the outgoing spatial dimensions are the same as the incoming spatial dimensions, there should be as much padding as the convolution window size divided by two rounded down.

Fully Connected Layer

A fully connected layer or a dense layer, is a layer where every neuron has a connection with all neurons from the previous layer. This layer is used to combine all gathered information from the previous layer, and all its different masks, and condense it towards a single presentable output layer. As shown in figure2.4 the first layers neurons are all connected to the second layers neurons. Th different width and boldness of the lines represent the different weights assigned to each connection.

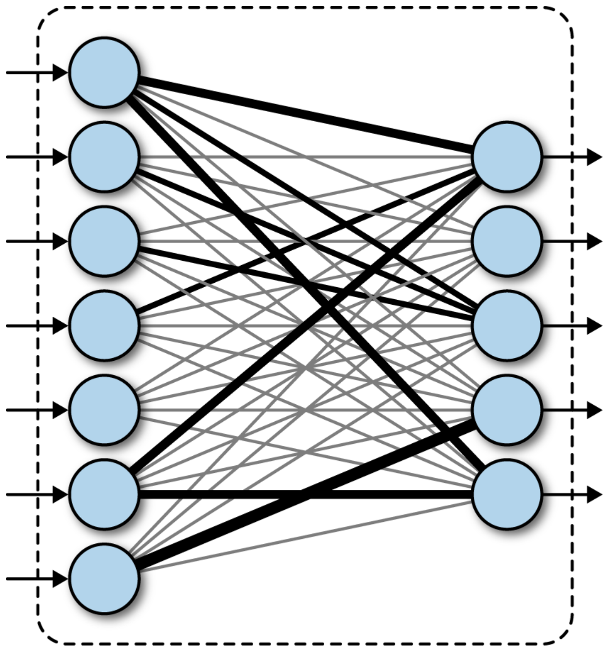


Figure 2.4: A Fully Connected Layer [4]

Dropout

To combat over learning specific characteristics in a dataset the dropout function can be used. During every learning cycle, when dropout is applied, all neurons have a chance p to be deactivated. To keep the functionality of the network, the remaining neurons have their output strength increased by $\frac{1}{1-p}$. This way of randomly deleting different neurons every learning cycle is a tool to not just prevent the over learning of specific characteristics, but it will also create a more robust network that relies on every aspect of the data set. Once the learning process of the network is complete every neuron is turned on again and the output strength of every neuron is scaled back down. In figure 2.5 the dropout function is seen in action. As seen in the figure every neuron has an approximate $p = 0.5$ chance of deactivation. The second and third neuron the lowest layer are during this specific learning cycle deactivated forcing the network to rely on the first, fourth and fifth neuron.

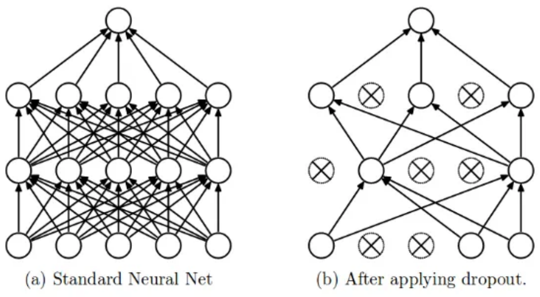


Figure 2.5: The dropout function in action [5]

The dropout function has both upsides and downsides. On one side it decreases the over usage of specific characteristics in a dataset, but on the other side it might not learn important characteristics that are present in the dataset that are vital for the output.

2.2.3. Training

With the use of the previously described functions a neural network is able to be trained for a desired output. In order to train a network on a specific dataset, the dataset has to be divided up into two components, the training set and the test set. The training set is used to train the neural network and the test set is used to confirm the functionality of the trained neural network on data that was not used in the training period.

In the first generation the weights are normally randomly distributed. After the first generation the loss function is calculated. Before rerunning the neural network for its second generation the weights are adjusted. After the second generation the new loss function is calculated, if the loss function is lower compared to the first generations loss function, the new generations weights are saved as the new optimal generation. This process continues until the maximum desired generation has been reached. Letting a network train for too long however causes the network to overfit itself to the training data. This overfitting will often result in a high accuracy on the training data, yet a low accuracy on the testing data. To combat this overfitting the a commonly used strategy is to forcefully stop the training the moment the test performance starts to degrade.

2.3. Spiking Neural Networks

an SNN differs from a more standard and well known artificial neural network or (ANN) in that it takes both information and the time into account instead of just the information as in ANNs. In an SNN the output data from a neuron is not transmitted as a single instance of information, but a potential spike that slowly decays over time. Image recognition software uses Neuromorphic datasets, these data sets only record changes in the image compared to sending the entire image every frame. Neuromorphic datasets are very suitable for SNNs as the changes in pixels can be well represented as spikes in an SNN. On top of that an SNN is a lot more energy efficient due to the nature of only responding to changes in the image instead processing the entire image every time step. These differences make an SNN very suitable for image recognition.

2.3.1. Spike Response

When a neuron outputs a spike, it is represented as a response kernel ϵ. This response function will then be multiplied by the weight of the connection to generate the incoming spike response function $a(t)$ at the input of the next neuron. The combined incoming spikes response functions are called the Post Synaptic Potential or (PSP). The neuron will output a spike itself the moment the PSP crosses the threshold ϑ. As would happen with a real-life neuron, a neuron has a refractory response $V(t)$ to prevent itself from immediately firing again.

2.4. Simulators

There are many different kinds of machine learning libraries, the main two neural network libraries currently available are TensorFlow and Pytorch. The SNN used for validation in this thesis is a library based on Pytorch, namely SLAYER.

2.4.1. Pytorch

Pytorch [14] is an open-source machine learning library written in either Python or C++. It is based on the much older Lua based machine learning library Torch. Originally developed by META AI in 2016 it is now part of the Linux foundation umbrella. This up-and-coming Machine Learning library is extensively used in the academic field and is especially popular among researchers. The Python friendly way of describing networks and intuitive flow of data makes this simulator one of the most popular simulators.

2.4.2. TensorFlow

TensorFlow [6] is an open-source machine learning library written for Python, JavaScript, C++ and Java. Developed by the Google Brain team in 2015 it is one of the older still up-to-date machine learning library. TensorFlow focuses on the industrial deployment of machine learning and is optimized for large data set handling, training and evaluating networks.

2.4.3. Comparison

Both Pytorch and TensorFlow have their similarities and differences. The comparison is divided up into multiple categories. These categories are: The computational mapping, the ease of use, the performance, and the ecosystem and community.

1. **Computation Graph**
    - **Pytorch:** The computation graph, the connectivity mapping between layers, is constructed as the program is executed allowing for easier debugging.
    - **TensorFlow:** The computation graph is constructed in a static fashion, before execution of the program. This allows for additional optimization of the program compared to Pytorch.
2. **Ease of Use**
    - **Pytorch:** Written in a python friendly manner, Pytorch is popular among researchers for its ease of use and quick adoption.
    - **TensorFlow:** With the introduction of TensorFlow 2.X the ease of use has increased compared to the first versions of TensorFlow, however the learning curve is still steep compared to Pytorch.
3. **Performance**
    - **Pytorch:** The Dynamic nature of Pytorch's computation graph has its downsides in that the performance is less optimal. The performance however is efficient enough for research purposes.
    - **TensorFlow:** The Static nature of TensorFlow computation graph allows for greater optimizations. On top of the higher efficiency TensorFlow also allows for a greater scalability compared to Pytorch.
4. **Ecosystem and Community**
    - **Pytorch:** The community of Pytorch is growing rapidly, especially in academic and research circles. On top of that Pytorch integrates well with other Python libraries.
    - **TensorFlow:** TensorFlow is widely adopted in the industry. TensorFlow has a large ecosystem of tools and a broad user base for the industry.

With these comparisons in mind, it is evident to draw a conclusion that Pytorch is mainly used by academia, whilst the Industry favours TensorFlow.

SNNSim adds additional aspects to the computation graph compared to both Pytorch and TensorFlow, like Pytorch the computation graph is constructed during execution, but unlike Pytorch this computation graph is not limited to predefined frameworks of layers.

3 SNN Simulator

The design goal of this thesis is creating a simulator that allows people to easily extract information from the hidden layers of a network and modify the network freely compared to the more conventionally used simulators. With this in mind, the chapter is divided up in the following sections. Section 3.1 details a detailed description of SNNSim's components. Section 3.2 explains the added observability and modification functions added to SNNSim. Finally Section 3.3 gives a summary of the chapter.

3.1. Components

The components of SNNSim are discussed in the following sections. The first section 3.1.1 describes SNNSim itself. The second section 3.1.2 talks about the event handler and the third section 3.1.3 describes the event reader. In the fourth section 3.1.4 file reader is discussed. The fifth section 3.1.5 introduces the added randomness functions to simulate real world scenarios. The sixth and final section 3.1.6 describes the dropout function of SNNSim.

3.1.1. Simulator

SNNSim is written in an object oriented manner. This way of designing simulator is picked to allow for both easier debugging and for better scalability of SNNSim. On top of the object oriented programming SNNSim also uses an event driven simulation. The decision to use event driven simulation compared to a continues simulation is because of the nature of an SNN. Compared to an ANN an SNN does not always have new information, or events, to process. An event driven simulation entails that not every neuron is processed every time step, only when a spike event occurs and only the neurons that are affected by the spiking event are processed. This reduces the time to run a simulation drastically.
The SNN simulator itself is divided up into three classes. The highest class in SNNSim is the NeuralNetwork class. The class describing the layers is the NeuronCluster class. The deepest class is in SNNSim, describing the neurons, is the Neuron class.

NeuralNetwork The entire combined network is called the NeuralNetwork, the class NeuralNetwork hence contains a vector with all the NeuronClusters. On top of that it also contains the outgoing connections of every neuron, in which cluster they exist and their intrinsic delay. An example of such a list is shown in table 3.1. The first column of the table denotes the assigned number the neuron has. In the second column the assigned cluster is listed. The third column lists every outgoing connection the neuron has. The fourth and final column denotes the intrinsic delay of the neuron. This table is used to determine to which neuron a spike has to be send using the outgoing connectivity list.

Table 3.1: Table of a Neuronlist within the class NeuralNetwork

Neuron	Cluster number	Outputs	Delay
1	1	3,4,5,6	0
2	1	3,4,5,6	0
3	2	7,8,9,...	0
...	...	...	...

NeuronCluster The class Neuroncluster, better known as a layer, keeps track of all the neurons in its layer using a vector of the class Neurons and a list of the ID's of all the neurons in the cluster.

Neuron The information used in the simulation is mostly saved inside the individual Neuron class. Every individual neuron is written as a class that keeps track of the input neuron connections and their weights, its own threshold and its spiking potential function over the entire simulation time.

3.1.2. Event Handler

The event handler is an independent class that has a queue full of the class spike events, as seen in figure 3.1. These events hold the information of what neuron fires and at what instant. The event handler will go through the entire queue one by one and either handle the event if the delay is zero or reduce the delay by one and put it back at the back of the queue. If an event causes a neuron to spike a new event is added into a different queue with the current time and the neuron that spiked, once the queue is empty the new queue is loaded in. This process continues until both queues are empty.

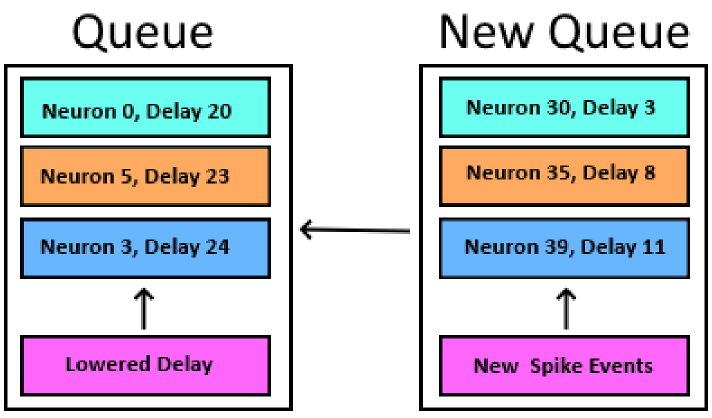


Figure 3.1: Both event Queues

3.1.3. Event Reader

There are two spike event readers, one to read out events from a binary format and one to read out events from a Python Numpy format. The binary event reader reads the binary data from a dataset and converts them to events to be added to the event queue. The events are encoded as 30 bits each as follows:

- X-data 5 bits;
- Y-data 5 bits;
- polarization 1 bits;
- time data 14 bits.

Both the positional data points are encoded with five bits each. This allows for a maximum xy-plane of 32 by 32. The time data is stored in 14 bits, this allows for a total of 16384 possible time data points. These exact amount of bits are picked to minimize the file size of the datasets. To extend the maximum xy-plane the reader has to be adjusted to fit the larger sized events.

The Numpy file format uses the same principle as the binary format, but unlike the binary format the data of one event is not stored in 30 bits total but each data part is stored in its own array index handled by a build in header file. This change removes the limited amount of xy-plane positions and is the preferred reader.

3.1.4. File Reader

SNNSim has two different kind of file readers. The two file readers are: The Python format file reader, and the text format file reader.

Python File Reader The first file reader is in the numpy library format for Python. The first bites are used to describe the format of the network in a header:

- Total amount of bits in header;
- incoming XY dimension;
- incoming Polarities;
- Layer 0 characteristics;
- Layer 1 characteristics.

As each layer of the layers have different kind of characteristics, the different types of layers are saved differently in the file. Every layer is described with five bits. The final bit always denotes the type of layer, where a one means the pooling layer, a two means the convolution layer and a three means the fully connected layer. The functionality of the other four bits are determined by the type of the layer. A full description of the layers is shown in the list below:

- Fully Connected Layer:
  [Neurons in layer, Previous Layer Mask Polarity, window X-axis, window Y-axis, 3];
- Convolution Layer:
  [Output Polarities, Previous Layer Mask Polarity, Window X-Axis, Window Y-Axis, 2];
- Pooling Layer:
  [1, 1, window X-Axis, window Y-Axis, 1].

For the fully connected layer the first bit describes the total amount of neurons in the layer, the second bit the amount of polarities, the amount of masks, the previous layer has, the third and the fourth bit describe the window size of the layer. For the convolution layer the only difference is in the first bit, where it now describes the outgoing amount of polarities. The final layer type is the pooling layer. The first two bits are always one and the third and fourth bit again describe the window size of layer.

This is then followed by the weights for each layer. The Fully connected layer describes each connection between every neuron. The Pooling layer only describes one weight for the entire layer, as it's always the same weight. The Convolution Layer holds the weights of the convolution window for every incoming polarity to every outgoing polarity. As an example, a Convolution layer with a convolution window of $5x5$ with two incoming polarities and 16 outgoing polarities will have a total of $5*5*2*16 = 800$ weights saved. Lastly the Pooling layer, as the pooling layer is simply a summation layer of all the neurons inside the pool only one weight is saved for the entire pooling layer, a weight equal to the threshold of the neurons.

Text File Reader The second File reader is in an ASCII text file format, this file reader allows the user more insight into the network and is only used for small networks. Each file has three components: The Mode, The Clusters and the Neurons. The Mode declares the format of the file, where $Mode : \{1\}$ means a fully declared network with all its neurons and clusters and $Mode : \{2\}$ means a random network. In a random network, the Cluster component is declared as follows $Cluster : \{\{Width, Length\}, Gammacycle\}$. The Width x Length describe the dimensions of the lattice. It then connects all Neurons from one Cluster to all neurons from the next Cluster. In a fully declared network the Cluster component works differently. In this case, the Cluster dimensions do not need to be declared, since the fully declared network also uses the neuron component. The neuron component works as follows: $Neuron : \{\{NeuronNumber, Weight\}, \{NeuronNumber, weight\}, ..., Delay, Threshold\}$ Where the part of $\{NeuronNumber, Weight\}$ declares its outgoing connections with their respective weights. The Delay is the intrinsic delay in the neuron, and the threshold sets the threshold required to fire the Neuron.

3.1.5. Randomness

To simulate real life scenarios SNNSim needs to be able to add a random amount of noise to each connection. SNNSim has three possible ways of introducing randomness.

The first randomness function achieves this by slightly adjusting the weight of a connection within a desired range. To use the function you both declare the layer you want to add randomness to, the one sided amount of randomness you want to add and the percentage of connections you want to give this bias to.

The second function that introduces randomness is a function that instructs a layer to add a specified amount of random delay to each spike event created in said layer. The first inputs specifies the layer you want to introduce the delay to, the second input is the one sided amount of random delay and the third input is the chance an event has a delay.

The final way SNNSim introduces randomness is by adjusting the strength of a spike the neuron creates. This adjustment affects the outgoing potential of all connections.

3.1.6. Dropout

The dropout function is capable of deleting a number of randomly selected connections between neurons. In the function you declare both the amount of connections to be deleted and the layer in which these connections have to be deleted.

3.2. Observability and control

To improve the ability to view and adjust the hidden layers of SNNSim, multiple helper functions are written to extract important information from SNNSim and adjust the connectivity of the neurons.

The helper functions to improve the observability are: the function PrintNeuronList and the function PrintNeuronListOutput 3.2.1, the function PrintNeuronInformation 3.2.2, the function PrintClusterInformation 3.2.3, the function PrintQueuetoFile 3.2.4, the function PrintNeuron-VoltagetoFile 3.2.5.

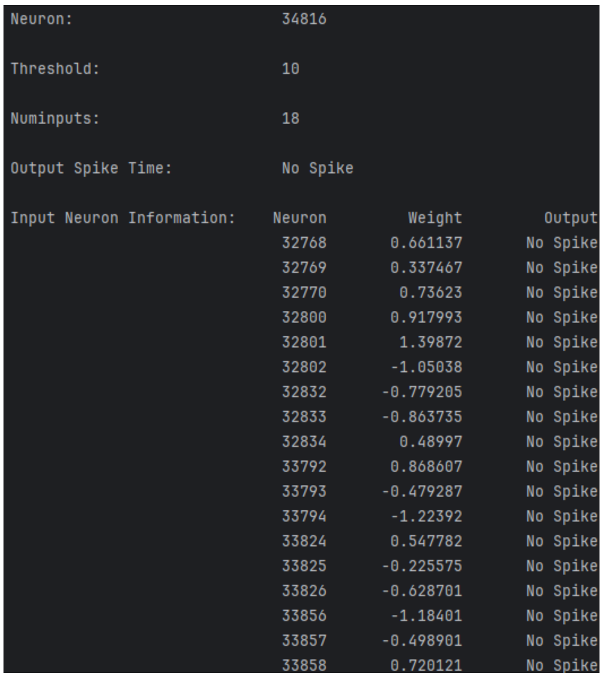
The functions to adjust the connectivity and mapping of the hidden layers are: the function AddConnection and RemoveConnection 3.2.6, the function RemoveNeuron 3.2.7, the function UpdateThreshold 3.2.8 and the function UpdateWeight 3.2.9.

3.2.1. PrintNeuronList

The function PrintNeuronList is used to get the outgoing connections of a neuron. A similar function PrintNeuronListOutput is a function that returns the information of all neurons that have an output to the requested neuron.

3.2.2. PrintNeuronInformation

The function PrintNeuronInformation has multiple possible inputs. You can either enter in the neuron number you want information on or both the layer of the neuron and the neuron's index in said layer. Image 3.2 shows the important information of the neuron 34,816. On top the neurons index of 34,816 is presented. Below the neurons index the threshold of the neuron is displayed. On the third row the amount of inputs the neuron has is shown. The fourth row tells the user if the neuron has spiked during the simulation. The fifth row shows a detailed list of all the incoming neuron connections with their neuron index, their weight and whether or not the incoming neuron has activated. The extracted information can further be used to fine tune a neural network.

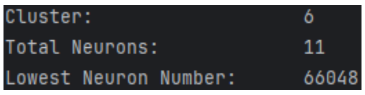


```
Neuron:                      34816

Threshold:                   10

Numinputs:                   18

Output Spike Time:           No Spike

Input Neuron Information:   Neuron        Weight        Output
                             32768       0.661137      No Spike
                             32769       0.337467      No Spike
                             32770        0.73623      No Spike
                             32800       0.917993      No Spike
                             32801        1.39872      No Spike
                             32802       -1.05038      No Spike
                             32832      -0.779205      No Spike
                             32833      -0.863735      No Spike
                             32834        0.48997      No Spike
                             33792       0.868607      No Spike
                             33793      -0.479287      No Spike
                             33794       -1.22392      No Spike
                             33824       0.547782      No Spike
                             33825      -0.225575      No Spike
                             33826      -0.628701      No Spike
                             33856       -1.18401      No Spike
                             33857      -0.498901      No Spike
                             33858       0.720121      No Spike
```

Figure 3.2: Console output from the function Printneuroninformation

3.2.3. PrintClusterInformation

PrintClusterInformation is another such function, as seen in image 3.3 this function prints the requested cluster information to the console. With the first line returning the requested neuron cluster, the second line the amount of neurons inside the requested neuron and the last line the lowest or first index or neuron number inside the cluster. This information can be helpful during fine tuning and pruning neurons with the dropout function.



```
Cluster:                  6
Total Neurons:            11
Lowest Neuron Number:     66048
```

Figure 3.3: Console output from the function PrintClusterInformation

3.2.4. PrintQueuetoFile

The event queue of the last layer is automatically saved into a text file for further analysis. This file contains the neuron number of the spike and the time at which the spike happens. It is also possible to manually call $PrintQueuetoFile$, to manually write all events to a file, or $PrintQueue$ to get the current queue information.

3.2.5. PrintNeuronVoltagetoFile

The function $PrintNeuronVoltagetoFile$ allows the user to print the potentials of all neurons in a layer to a folder to make it easy to read out the post synaptic potential of every neuron.

3.2.6. AddConnection and RemoveConnection

The function AddConnection takes four inputs. The first input is the index of neuron A's the outgoing connection, the second input is the index neuron B's the incoming connection, the third input is the delay on the connection, currently always set to zero, and the final input the weight of the connection. The function RemoveConnection takes two inputs. Neuron A the outgoing connection and neuron B the incoming connection.

3.2.7. RemoveNeuron

The function RemoveNeuron has two possible inputs. The first input type is only the exact index of the neuron that you want to delete, the second possibility is adding the layer number and specifying the neurons number inside the layer.

3.2.8. UpdateThreshold

The function UpdateThreshold takes two inputs. The index of the neuron and the desired new threshold of the neuron.

3.2.9. UpdateWeight

The function UpdateWeight takes in three inputs. The first and second input being both indexes of neuron A and neuron B, and the third input being the desired new weight of the connection.

3.3. Summary

In this chapter the simulator SNNSim has been introduced. The key components of SNNSim are introduced, with a focus on the added observability and control functions that set it apart from other simulators.

4 Functional Validation

To validate that the basic functionality of SNNSim operates as intended, a few initial small scale simulations are run. The first section 4.1 entails the first simulation that tests the connectivity between neurons and the ability for neurons to spike. The second section 4.2 describes the simulation that tests the effect different weights have on the ability for neurons to spike. In the third section 4.3 a simulation is done to test the difference in arrival times. The fourth section 4.4 simulates the ability for SNNSim to have negative weights. The fifth section 4.5 tests the refractory function of SNNSim. Lastly, the final section 4.6 summarizes the functional validation.

To provide more clarity and consistency during the simulations a matching colour coding pattern is used for both the figures and the graphs. The layers and curves are coloured as followed:

- The first layer is coloured cyan;
- The second layer is coloured orange;
- The third layer is coloured blue.

The line arrow between the neurons points to a connection between the two neurons. The weights of the connections are listed on top of the arrow tipped lines, and the thresholds of the neurons are listed underneath the neurons.

The spike response functions used in the simulations to test the functionality of SNNSim are of the form $\epsilon(t) = 1.1\frac{t}{5}e^{1-\frac{t}{5}}$. The refractory response used in the fifth simulation is of the form $v(t) = -e^{1-\frac{t}{5}}$. The formulas used are based on the formula used in the SLAYER [18] for both the spike response function and the refractory function. The variables are adjusted to better fit the graphs.

4.1. Connectivity

The first initial simulation is a simple test where the connectivity between the neurons is validated. The neuron network setup in the first simulation as seen in figure 4.1 is as follows: The zeroth neuron has a connection to the first neuron and the first neuron has a connection to the final second neuron. All neurons in this setup have a threshold to trigger of 1.0 and all connections between the neurons have a weight of 1.0.

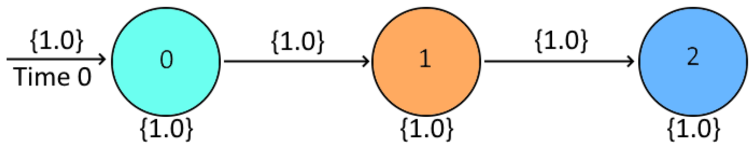


Figure 4.1: Three neurons connected to each other in a line

The graph 4.2 shows that SNNSim is capable of creating new spikes if the potential reaches the threshold. The time step after the cyan line curve crosses the threshold of 1.0 the first neuron, the orange line curve, starts spiking.

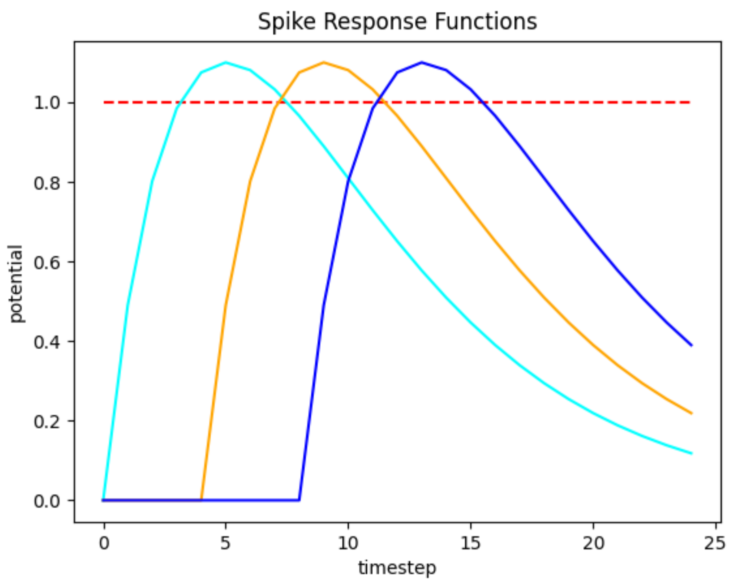


Figure 4.2: The Three potential spike response functions from the first simulation.

4.2. Weights and Thresholds

The second simulation tests the functionality of changing the weights and thresholds of SNNSim. Figure 4.3 shows the neural network setup for the second simulation. The zeroth neuron, the input neuron, has three outputs: The first, second and third neuron. Both the first and second neuron have an outgoing connection with the fourth neuron. The third neuron has an outgoing connection with the fifth neuron.

The simulation is split up in two parts the upper part with neuron zero, one, two and four, and the second part with neuron zero, three and five. To test the functionality of different weights, the weights of neurons are changed.

The zeroth neuron still has a weight of 1.0 with the input and a threshold of 1. The first neuron has a weight of 1.25 with the first neuron and a threshold of 1.0, whereas the second neuron kept its incoming weight of 1 and kept its threshold of 1.0. The fourth neuron has its input connection to the second neuron lowered to 0.4 and its threshold is increased to 1.2.

The third neuron has its threshold lowered to 0.5 and the incoming connection weight from neuron zero is also set to 0.5. The connection between neuron three and neuron five has its weight lowered to 0.4 and the threshold of neuron five remains unchanged.

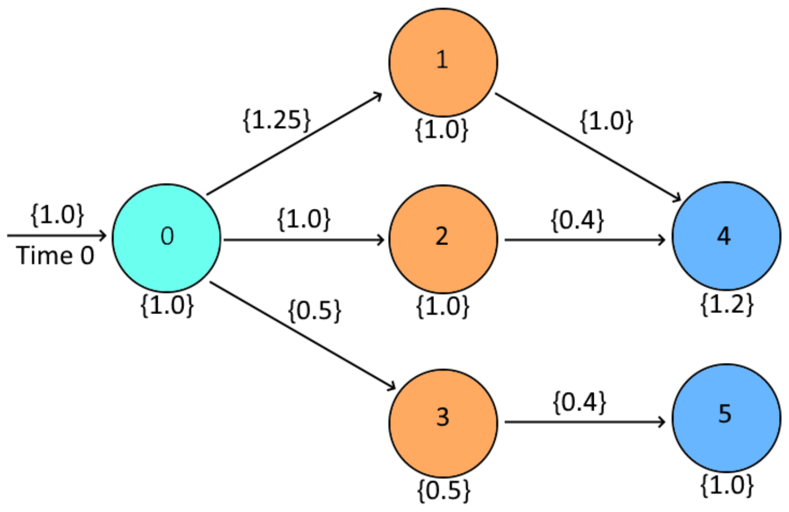


Figure 4.3: The six neurons used in the second simulation

The graph 4.4 shows three subplots from the first part of the second simulation. The first subplot contains the zeroth neuron, the second subplot contains the orange line curve of neuron one and the olive line curve of neuron two, the final subplot contains the light blue line curve coming from neuron two, the navy blue line curve coming from neuron one and the combined blue line curve.

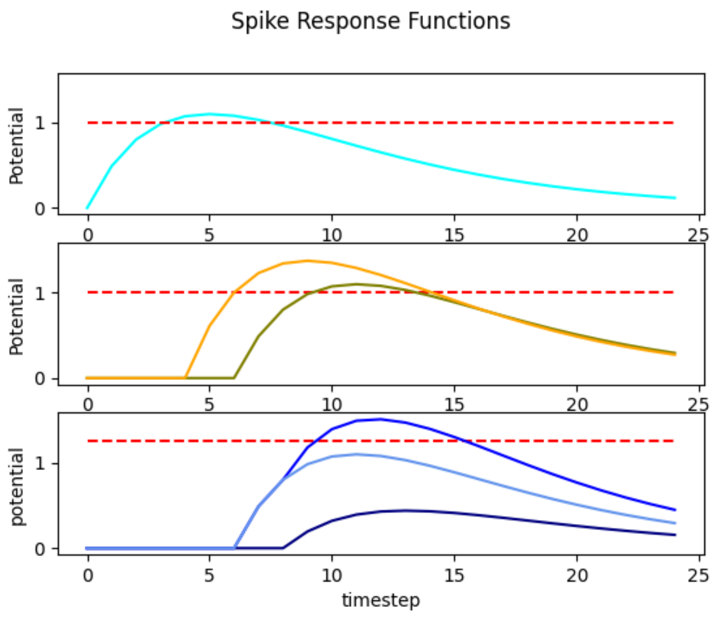


Figure 4.4: Three subplots for the first part of the second simulation

There are two observations to be made from this part of the simulation, namely a higher weight of an incoming connection causes earlier spikes and two combined inputs can overcome an increased weight. The orange curve has a higher peak because of the increased weight compared to the olive curve, because of this it spikes a time step earlier. This causes the light blue curve to rise one time step earlier compared to the navy blue curve. The increased threshold of neuron four would have caused the neuron not to spike with only the dark blue line curve, but because neuron four is also connected to neuron two it still fires.

The second graph 4.5 shows the potentials for the lower half of the neural network. The third neuron barely spikes thanks to its lowered threshold despite the low incoming weight. Finally the fifth neuron does not spike as the weight of the connection with neuron three is too low to reach the threshold.

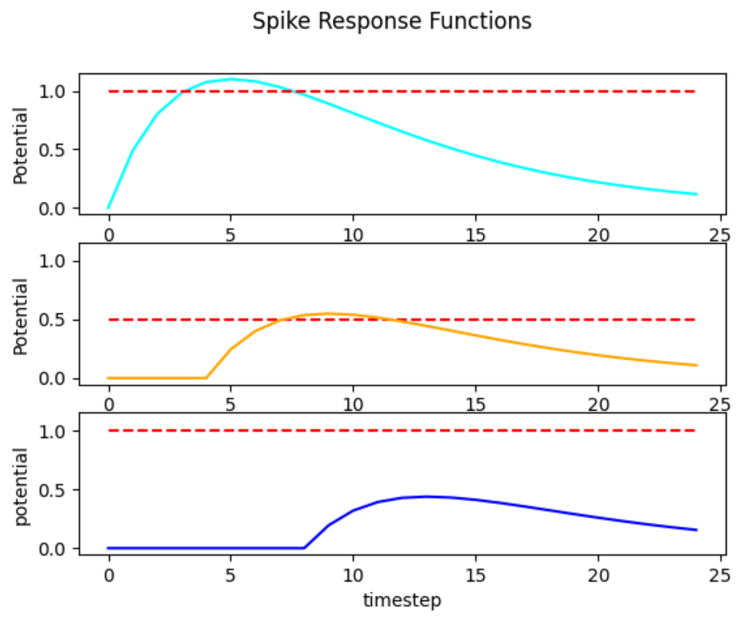


Figure 4.5: Three subplots for the second part of the second simulation

The second simulation validates the functionality of changing the weights and threshold of the neurons in a simulation. On top of that it also shows the effect these changes have on a neurons spike function.

4.3. Spike Arrival Times

In the third simulation, the effect of different spike times is evaluated. The setup as shown in figure 4.6 is as follows: Neuron zero has a single output to neuron three. Neuron one has two outputs to both neuron three and neuron four. Neuron two has a single output to neuron four. All neurons have a threshold and weight of 1.0. The three input neurons receive their input at different time periods. Neuron zero receives its input spike at time step five, neuron one at time step zero and neuron two at time step 10.

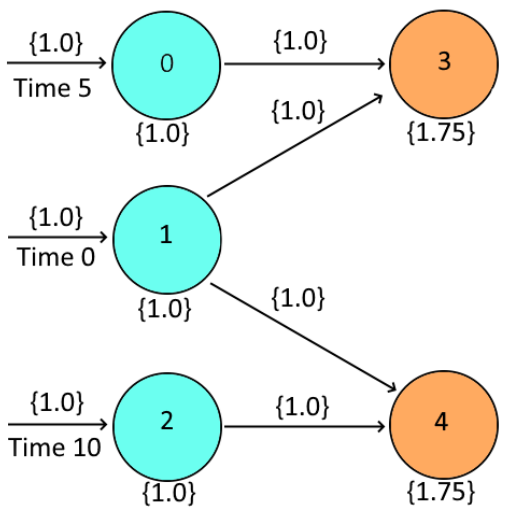


Figure 4.6: Topography of the third simulation testing different input times

As seen in graph 4.7 the first subplot shows the three different times of arrival for the incoming spikes. the blue coloured line curve is neuron zero, the cyan line curve is neuron one, the light blue line curve is of neuron two. The second subplot shows the individual and the combined input spike response functions of neuron three. The orange line curve shows the spike response curve coming from spike of neuron one, the olive line curve shows the spike response curve from neuron zero. The gold curve is the combined line curve of both inputs. The third subplot shows the line curves of the incoming signals for neuron four. The navy blue curve is like the orange curve in subplot two the incoming signal from neuron one. The light blue line curve is the spike response from neuron two. The blue curve is the combined line curve of both inputs.

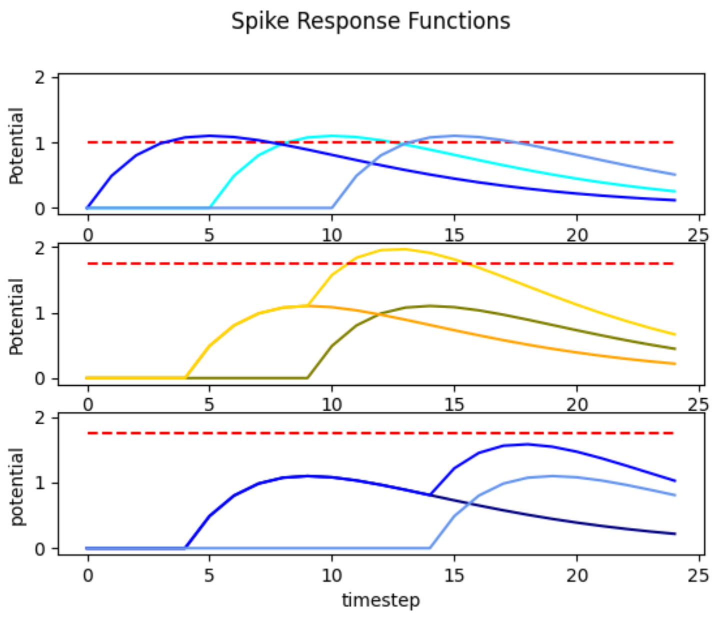


Figure 4.7: Three subplots of the Spike Response functions for the third simulation

As neuron three has its incoming spikes at time step four and six it is able to reach the threshold and create a spike. Neuron four, however, shows clearly that because the incoming spikes are at time step four and nine the spike at time step four has decayed too much to still reach the threshold. The third simulation has shown that SNNSim is capable of having different timed input events and how this affects the simulation.

4.4. Negative Weights

The weights of the connections in SNNSim are not absolute values. This means that a connection between neurons can be an inhibitory, a negative, connection, reducing the value of the spike potential function. To ensure this functionality works in SNNSim the fourth simulation is designed. The layout of the fourth simulation is shown in figure 4.8.

There are two input neurons, neuron zero and neuron one. These neurons both have an output connection to neuron two. The weight of the connection between neuron zero and neuron two is 1.0, the weight of the connection between neuron one and neuron two is lowered to -0.7. The input spike time differs between neuron zero and neuron one, the zeroth neuron receives an input spike at time step zero and the first neuron receives its input spike at time step two.

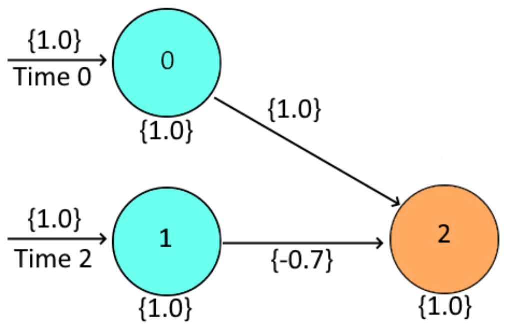


Figure 4.8: Topography of the fourth simulation testing negative weights

The graph 4.9 shows the inhibitory effect a negative weight connection has on a neurons spike response. The yellow line curve shows the spike response function of the zeroth neuron spike. The olive curve shows the negatively weighted response function of the first neuron. The orange curve shows the combined spike response of both connections. The yellow line curve crosses the threshold and the second neuron would have spiked if the inhibitory olive curve was not present. If the spike event of neuron one arrived two time steps later the inhibitory effect of the olive curve would have arrived too late and the orange curve would have still reached the threshold.

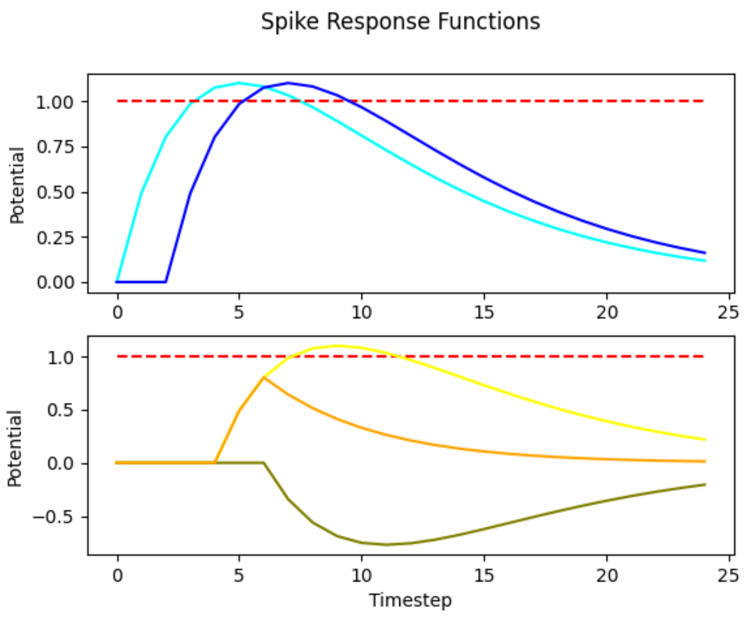


Figure 4.9: Two subplots showing the line curves of the spike responses of simulation four

The fourth simulation shows a more intricate view of what an inhibitory connection can achieve. Not only is the negative weight important to inhibit a spike, but also the time of the inhibitory spike is important.

4.5. Refractory Function

The fifth and final simulation is about the refractory response of a neuron. When a neuron spikes it increases its threshold to spike again by an additive amount equal to the refractory response of the neuron. To test this functionality a new neural network is designed with the layout as shown in figure 4.10. The mapping of the neurons is the same as in the first simulation. The weight the connection between neuron one and neuron two is changed from a weight of 1.0 to a weight of 2.0.

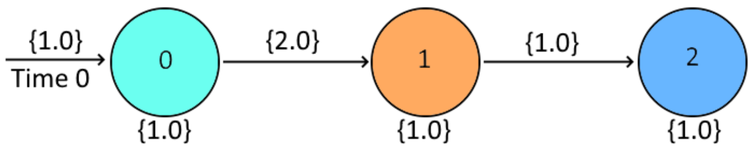


Figure 4.10: Topography of the fifth simulation testing the refractory function

In graph 4.11 the spike responses of the fifth simulation are shown. The first subplot shows the input spike response for neuron zero. The second subplot shows the spike response for neuron one, with the increased threshold of neuron two caused by the refractory function. The third subplot shows the spike response for neuron two with the multiple incoming spikes from neuron one.

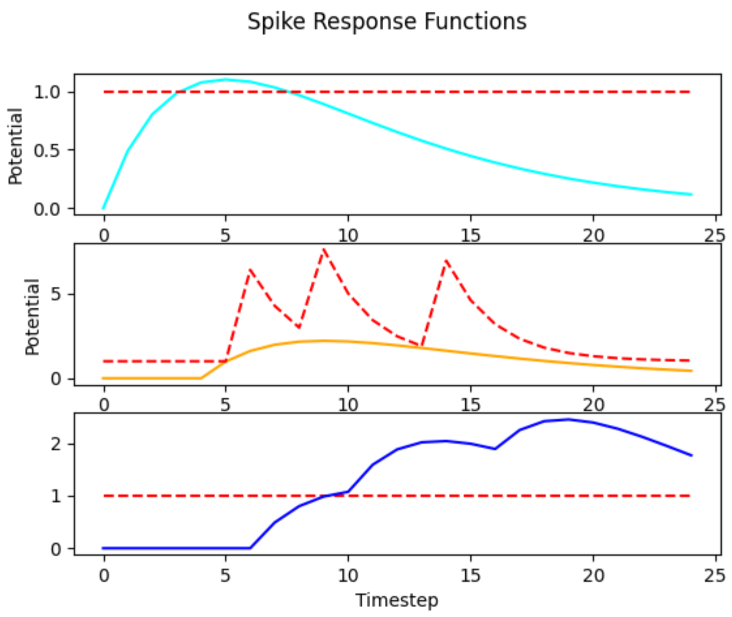


Figure 4.11: Three subplots detailing the line curves of the spike responses used in simulation five

The simulation has shown that having an incoming potential above the threshold for a prolonged time, as is shown in subplot two, causes the inhibitory effect of the neuron wanes off and enables the neuron to fire again. These effects are even clearer in subplot three where the three spikes incoming from neuron one are visible.

4.6. Summary

In this chapter we validated the basic functionality of SNNSim by first validating the connectivity between multiple neurons. Secondly, the ability to change the weight of connections and threshold of neurons were validated. Thirdly, the capability of the simulator to have different timed events was validated. Afterwards, we validated the functionality of inhibitory connections by giving connections a negative weight. Lastly, the refractory function of neurons in SNNSim was validated. These validations have shown that SNNSim has working basic functionalities.

5 Validation

To validate the accuracy of SNNSim two validations were made. Both validations were done by comparing SNNSim with the results from the SLAYER simulator[18]. The first validation 5.1 is done by using the example network given in SLAYER, this network uses the MNIST dataset of handwritten digits [11]. The second validation 5.2 is done using a Network build for the DVSGesture dataset of Gestures [7] also in SLAYER. In the final section 5.3 a conclusion is drawn from both validations.

Both Validations use the same spike response $\epsilon(t)$ together with the same as refractory response $v(t)$ used in the SLAYER simulator.

$$\epsilon(t) = \frac{t}{\tau_s} e^{1-\frac{t}{\tau_s}}$$

$$v(t) = -2\vartheta e^{1-\frac{1}{\tau_r}}$$

5.1. Slayernetwork - Multilayer perception

In this section the first validation is described. The first part of this section 5.1.1 introduces the method of validation. The second part of the section 5.1.2 analyzes and summarizes the first validation of SNNSim.

5.1.1. Validation Details

For the validation, the network was trained with SLAYER and the trained network was afterwards exported to the developed simulator. The MNIST dataset uses images of the size 34 by 34 with two polarities. The network only uses two fully connected layers. The first fully connected layer of 512 neurons is connected to the 2312 input neurons. The second fully connected and output layer is connected to the previous first fully connected layer. For the validation, the first test set is used, namely: $60001.bs2$. The Mean Square Error (MSE) between SLAYER and SNNSim is used validate SNNSim. The formula used to calculate the MSE is as follows:

$$\frac{1}{n}\sum_{n=1}^{n}(TrueGraph - PredictedGraph)^2$$

. As the Mean Square Error heavily depends on peak values of both graphs, the graphs are first normalized to the highest or lowest point in the SLAYERs graph, depending on whether the highest peak is negative or positive. Afterwards, the normalized MSE is calculated.

The potential curve of the first Neuron in layer one is seen in graph 5.1. The figure shows a near match between SNNSim's red curve and the desired SLAYER blue curve.

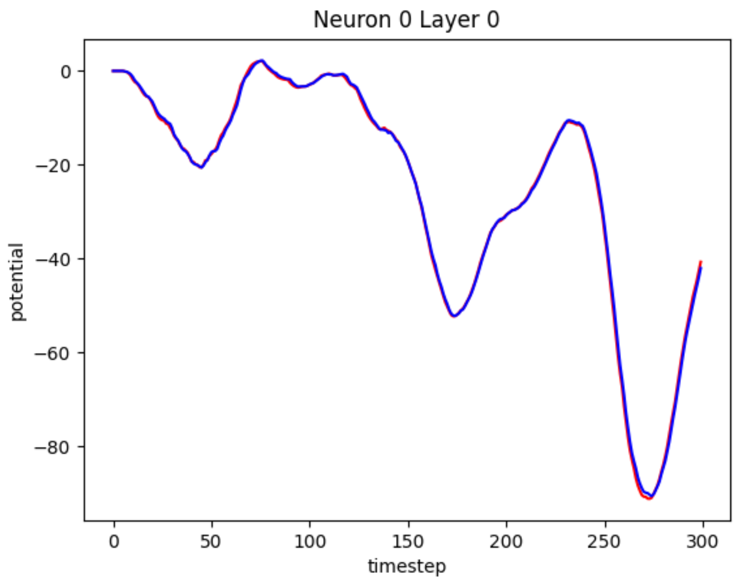


Figure 5.1: The first Neuron in the first layer of the first validation

The normalized MSE of the first neuron is $6.2 * 10^{-5}$ and the average MSE of the entire first layer is $8.7*10^{-5}$. The first layer received has a total of 3323 spike events from the dataset.

In graph 5.2 the eighth neuron of the output layer is shown. As shown in the graph, the red curve of SNNSim closely follows blue SLAYER curve.

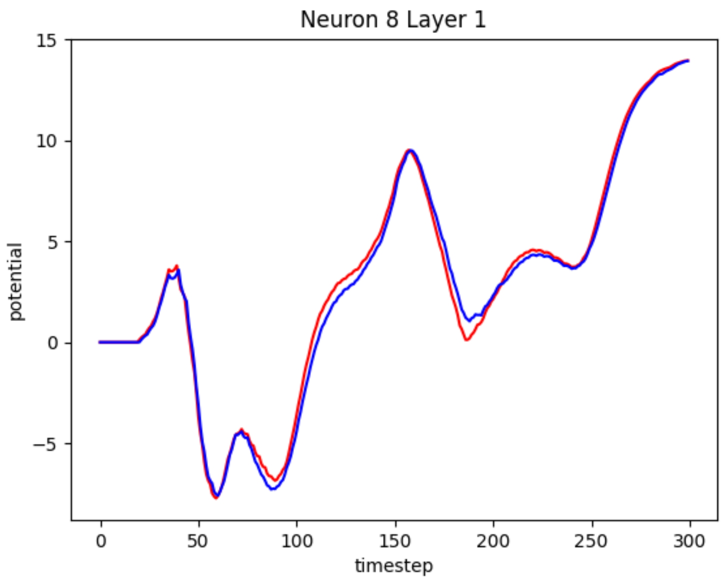


Figure 5.2: The eight Neuron in the second layer of the first validation

The eight neuron has an MSE of $1.1 * 10^{-3}$ and the average MSE of the entire second layer is $6.61 * 10^{-5}$. The output layer received a total of 875 incoming spike events from the first layer.

5.1.2. Validation Analysis

The table shown in 5.1 has all the calculated MSE of every layer combined in one table. The first column describes the layer, the second column describes the average normalized MSE of that layer, the third and fourth column detail the extremes of the layers, with both the highest MSE and the lowest MSE.

Table 5.1: Mean square errors of the first validation

Layer	Average MSE	Highest MSE	Lowest MSE	Total Events
1	$8.7 * 10^{-5}$	$4.36 * 10^{-4}$	$3.81 * 10^{-5}$	3323
2	$6.61 * 10^{-4}$	$1.92 * 10^{-3}$	$1.56 * 10^{-4}$	875

The average MSE of both layers are low enough to conclude that the functionality of SNNSim and the SLAYER SNN implementation are identical. The validation however only consists of two relatively small layers, as is evident from the low amount of spikes both incoming from the dataset and between the two layers. With this in mind another more expansive validation has to be done to ensure the increasing, all be it small, MSE is kept at a minimum.

5.2. Slayernetwork - DVSGesture

In this section the second validation is discussed. The first part of this section 5.2.1 introduces the method of validation. The second part of the section 5.2.2 analyzes and sumarizes the first validation.

5.2.1. Validation Details

The Dynamic Vision Sensor (DVS) Gesture data set is a dataset that includes different kind of gestures. In line with the previous Dataset, the dataset consists of neuromorphic data. The second validation was done with a more extensive network trained in SLAYER. This network uses pool, convolution and fully connected layers. To accommodate for this change, the file reader was extended to be able to also interpret pool and convolution layers.

Compared to the first validation, the new neural network and dataset both contain more neurons, connections and events. When validating SNNSim, this brings multiple complications, both in complexity and in speed. To partly address this problem the sample time is reduced by a factor of 10, this reduction in sample time means that every event that happens between simulation time 15 and 24 will now be set to simulation time 2. These adjustments are both done in SNNSim and in the Slayer script as to not affect the resulting network. The precision of the trained network is heavily reduced by this adjustment, but for the sake of the validation this is not important, since the goal of for SNNSim is to match the network generated in SLAYER, not to have a good result per se.

To simplify the validation the simulation is segmented. Firstly, the entire Data set is read in, and the first layer processes these events. Instead of putting the newly generated events from the first layer into the next queue, they are instead saved into a Numpy file. Once the validation of the first layer is complete, the events from the validated first layer's Numpy files are read in instead of the dataset events. After validating the second layer, the events are once again saved. This process is continued until every layer is validated individually. Afterwards a final full validation is run and the data points from this validation are used.

The network layout used in the second validation consists of the following layers in order is as follows:

- A 32 by 32 pooling layer with 2 polarities;
- A 32 by 32 convolution layer with 16 polarities;
- A 16 by 16 pooling layer with 16 polarities;
- A 16 by 16 convolution layer with 32 polarities;
- An 8 by 8 pooling layer with 32 polarities;
- A fully connected layer with 512 neurons;
- A fully connected layer with 11 neurons.

The dataset events are of the dimension 128 by 128 with two polarities, these neurons are then connected to the first summation pooling layer with a window of four. The first convolution layer has a convolution window of five by five with enough zero padding as to not reduce the xy-plane dimension. The second pooling layer reduces the xy-plane by a factor of two with a window size of two. The second and final convolution layer has a convolution window of three by three again with enough zero padding, increasing the polarities from 16 to 32. The final pooling layer then reduces the xy-plane by a factor two again to eight by eight. The first fully connected layer connects the entire 3rd pooling layer to its 512 neurons. The second fully connected and the last layer in the neural network has all 11 neurons connected to all 512 neurons from the previous layer and is used as the output of the neural network. To keep the graphs consistent with each other the red line curve is always from SNNSim and the blue line curve is always from the SLAYER SNN.

The first layer shows a near perfect match between the SLAYER output and SNNSim's output, as you can see in graph 5.3. The figure shows the Synaptic Potential of both the neuron generated by SLAYER, the turquoise curve and SNNSim, the red curve. The first layer had a total of 148,149 incoming spike events from the Dataset.

The normalized MSE of the Neuron shown in 5.3 is $2.79 * 10^{-6}$. The average Normalized MSE of the first layer $2.36 * 10^{-4}$.

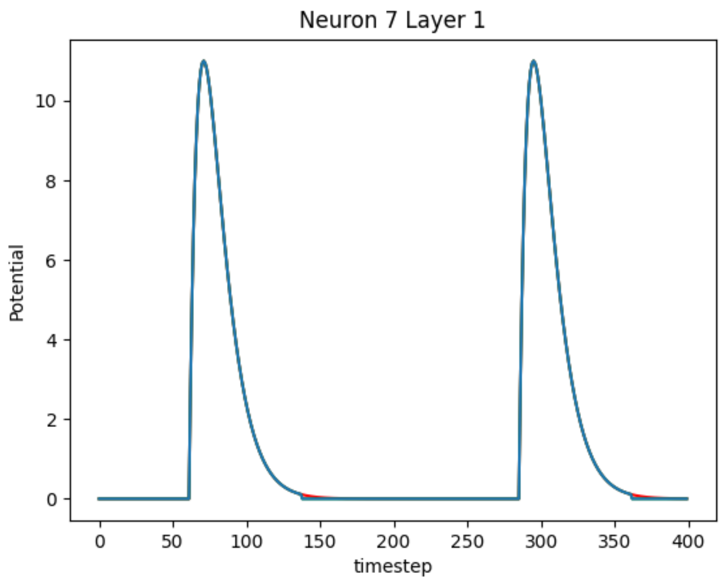


Figure 5.3: The seventh Neuron in the first layer of the second validation

For validating the second layer, the first convolution layer, the seventh neuron is shown in graph 5.4. In this graph the turquoise curve is from SLAYER and the red curve is from SNNSim. The second layer had a total of 208,040 incoming spike events from the first layer. This can be explained by the previous layer being a pooling layer that combined four inputs, creating more time spent above the threshold. The Normalized MSE of the seventh neuron is $7.01 * 10^{-4}$. The average normalized MSE of the entire second layer is $2.35 * 10^{-3}$.

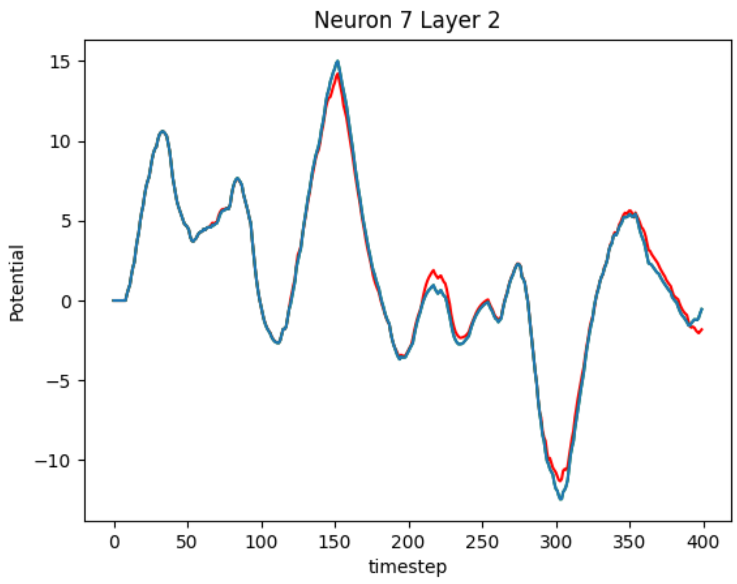


Figure 5.4: The seventh Neuron in the second layer of the second validation

In graph 5.5 the seventh neuron of the third layer, or the second pooling layer, is shown. The red curve follows SNNSim and the turquoise curve follows SLAYER. The third layer had a total amount of incoming events of 3,400,796. This sharp increase in events can easily be explained by the fact that the incoming layer is a convolution layer that increased the masks from two to sixteen. The Normalized MSE of the seventh neuron is $3.02 * 10^-3$. The average MSE of the entire third layer is $4.97 * 10^-3$.

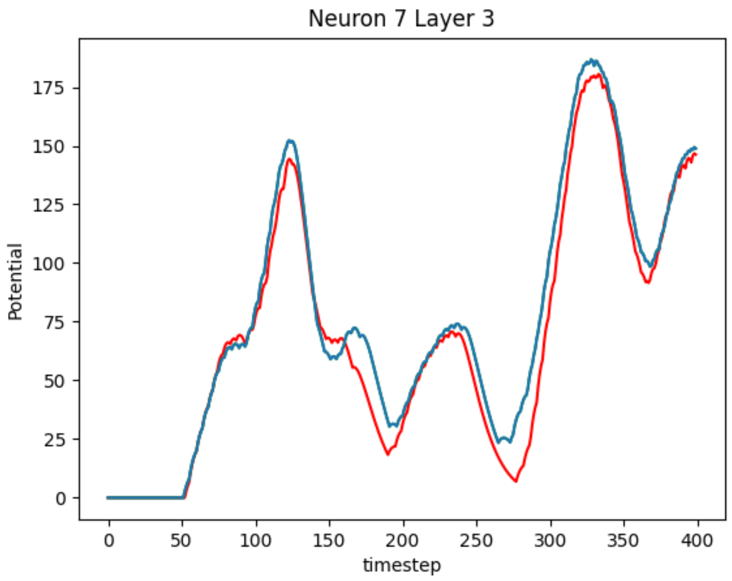


Figure 5.5: The seventh Neuron in the third layer of the second validation

As most neurons in the fourth layer did not have any spikes, the 141st neuron is shown in graph 5.6 instead. The graph shows the spike potential of SNNSim in red and the potential of SLAYER in turquoise. The fourth layer has a reduced amount of incoming events compared to the previous layer, with 1,286,310 spikes. The Normalized MSE of the 141st neuron is $9.32 * 10^{-3}$. The average MSE of the entire fourth layer is $6.37 * 10^{-3}$.

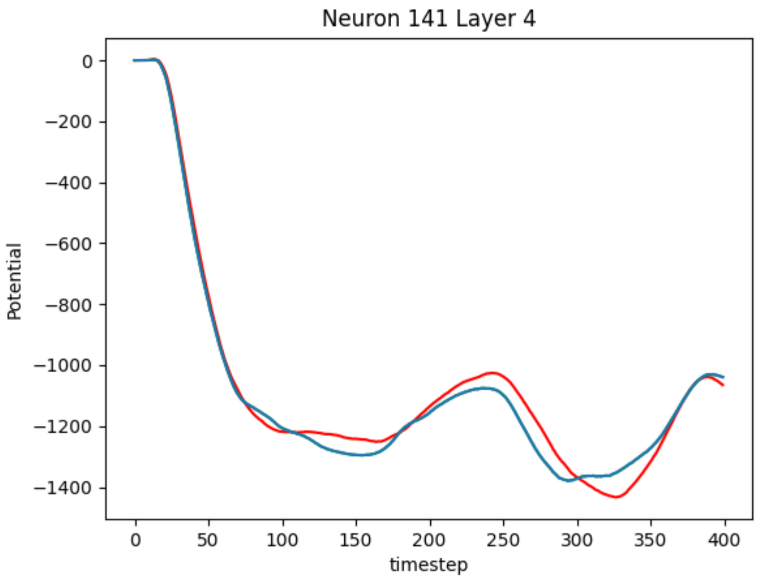


Figure 5.6: The 141th Neuron in the fourth layer of the second validation

In the fifth layer, the third pooling layer, the 141st neuron is again shown in graph 5.7. The spike potential of SNNSim is shown in the graph as red. As the layers are compacting down in size the amount of incoming events are also reduced. The fifth layer had a total of 863,515 incoming spike events. The spike potential of SLAYER is in turquoise. The Normalized MSE of the 141st neuron is $2.02 * 10^{-2}$. The average MSE of the entire fifth layer is $9.11 * 10^{-3}$.

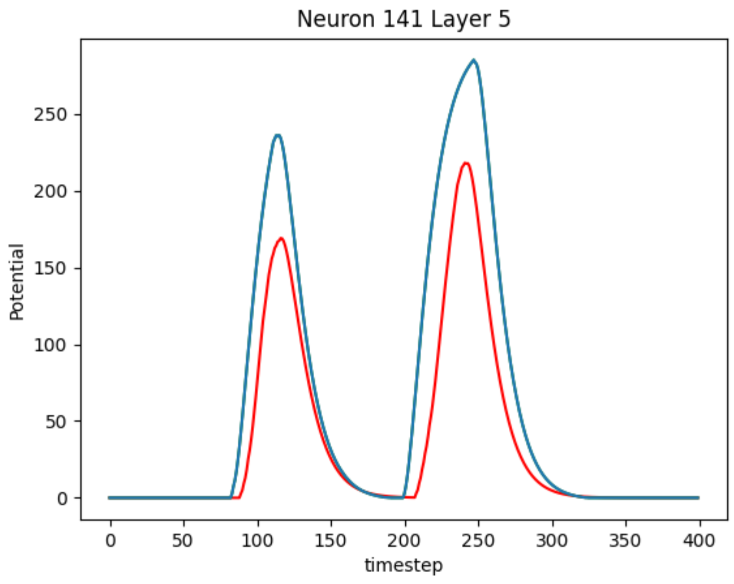


Figure 5.7: The 141th Neuron in the fifth layer of the second validation

With the sixth layer, the first fully connected layer, the 141th neuron is once again picked and shown in graph 5.8. As the amount of neurons is diminishing the amount of incoming is also lowered, the total amount of incoming spike events in the sixth layer is 372,314. The potential of SLAYER is shown in the graph as turquoise and the potential of SNNSim is shown in red. The Normalized MSE is $1.36 * 10^{-3}$. The average MSE of the entire sixth layer is $2.33 * 10^{-3}$.

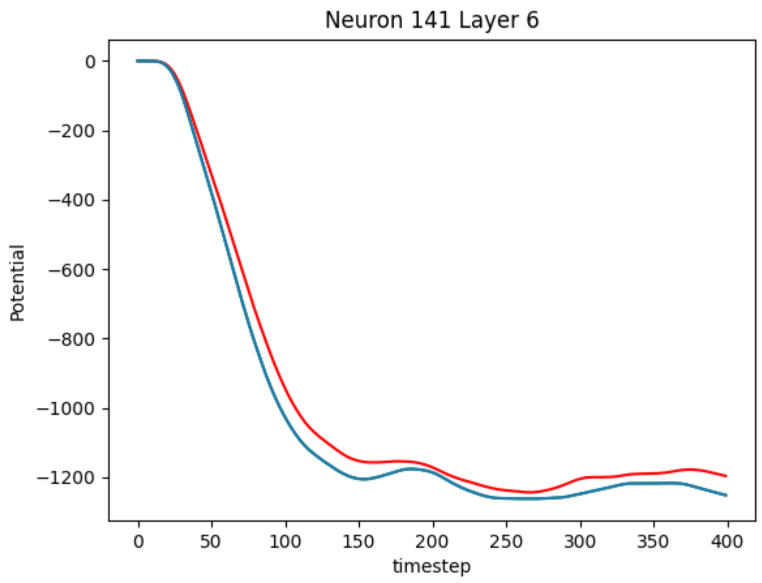


Figure 5.8: The 141th Neuron in the sixth layer of the second validation

In the seventh layer, the final fully connected layer and output layer, the second neuron is shown in graph 5.9. The final graph has the potential of SNNSim in red and of SLAYER in turquoise. The final layer had a total amount of 108,570 incoming events. The Normalized MSE of the second neuron is $5.09 * 10^{-3}$. The average MSE of the entire output layer is $1.03 * 10^{-2}$.

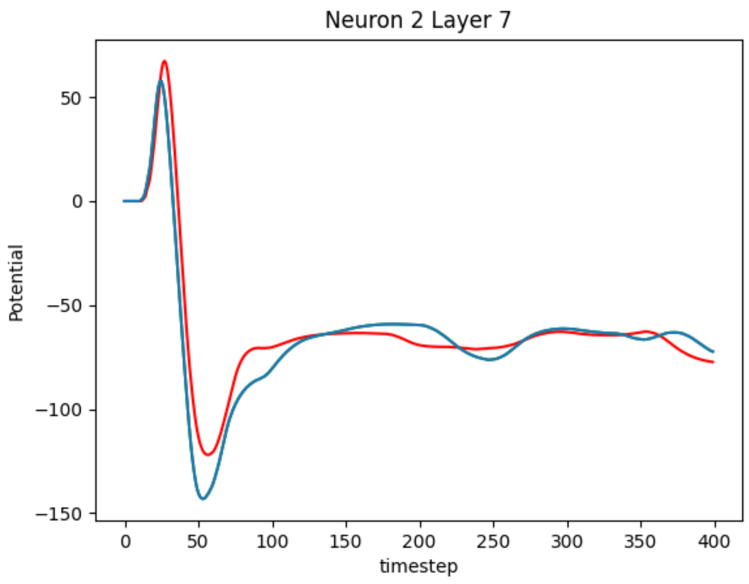


Figure 5.9: The second Neuron in the seventh layer of the second validation

5.2.2. Validation Analysis

The table shown in 5.2 has all MSE combined in one table, with the last column detailing the total amount of spike events that occurred during each layer. The average MSE for each layer is below or at 1%. The high outliers are caused by a few neurons having a low spike count in combination with normalizing every graph, which has the downside of having single spike differences stand out more in low spike numbers compared to high spike count.

Table 5.2: Mean square errors of the second validation

Layer	Average MSE	Highest MSE	Lowest MSE	Total Events
1	0.0002	0.0462	0.0000	148149
2	0.00219	0.1064	0.0000	208040
3	0.00497	0.5933	0.0000	3400796
4	0.0009	0.2091	0.0000	1286310
5	0.0091	1.1046	0.0000	863515
6	0.0023	0.0636	0.0000	372314
7	0.0103	0.0244	0.0030	108570

The average MSE of each layer changes significantly between the first and last layer. As expected the MSE slowly increases between layers as small differences accumulate between layers. The table however also shows that some layers have the effect of lowering the average MSE back down. These layers are the second convolution layer and the first fully connected layer. The high average MSE of the fifth layer can be explained by the high outliers present in the layer. If the outliers are removed the average MSE falls more inline with the surrounding layers. The drop in average MSE between the third layer and the fourth layer is also explainable. Most of the neurons in the fourth layer did not fire a single time during the simulation. This both explains why the average MSE is this low and the relatively high maximum MSE. When a neuron spikes in SNNSim but not in SLAYER or vice versa the MSE is high.

5.3. Validation Conclusion

Both Validations show that the MSE of SNNSim compared to the SLAYER SNN is on average small enough to consider the functionality of SNNSim and the SLAYER SNN near identical. However it also shows that SNNSim starts to diverge from the SLAYER SNN the more layers are added. With this in mind, further improvements have to be made to say with certainty that SNNSim is reliable for simulator larger neural networks.

6 Conclusion

This chapter concludes the thesis with a summary on its achievements, a conclusion is given and highlights possible future works to SNNSim. Section 6.1 provides a summary of all the chapters in the thesis and gives a conclusion to the thesis. Section 6.2 presents possible future works for SNNSim.

6.1. Summary

Chapter 1 introduced the ascent of artificial intelligence to the front page of information technology and how neural networks play a role within this field. The chapter discussed the increased reliance on image recognition in the fields of self driving cars and surveillance. This focus was mostly laid on the specific neural network type of SNN, with the objective to increase the observability of the spiking response functions.

Chapter 2 of the thesis provided an introduction to the concept of neural networks and introduces the inner workings of a neural network. The chapter started with background information on neural networks, which was followed up by the inner workings of an ANN and how an SNN spike function changes the neural network. Finally two neural network libraries were discussed and compared, Pytorch and Tensorflow.

Chapter 3 introduced SNNSim with its components and functionalities. The components of SNNSim are: The simulator classes, the event handler, the event reader, the file reader, the file writer and the dropout function. To improve the observability of SNNSim the ability to view the internal mapping is added. On top of that the control of the SNNSim is improved by adding multiple functions that allow the editing of layers outside of the standardized frameworks.

Chapter 4 details the functional validation done on SNNSim to validate the basic functionalities of SNNSim. The simulations that were tested on SNNSim are as follows: The connectivity of the neurons and the ability to spike. The ability for SNNSim to work with different weights and thresholds. The effect different input arrival times have on a neurons. The inhibitory effect negative weights have on a neuron. The refractory function and what it does to the threshold of a neuron.

Chapter 5 Validated SNNSim by comparing the functionality of a trained network on the developed simulator and on the SLAYER SNN. The first small validation was done using a dataset that contains handwritten letters. The second more expansive validation was done using a dataset that contains hand gestures. Finally a conclusion was made on the validation of SNNSim.

In summary, the thesis has explored the rapid advancement of spiking neural networks. It then presented a potential solution to the standardization caused by this fast-paced growth, by improving the observability and controlability of the hidden layers. We introduced SNNSim, which has been thoroughly tested and validated. Although the validation results indicate SNNSim has a maximum average MSE of less than 2%, further development of SNNSim has to be done to ensure it remains within acceptable ranges of state-of-the-art SNNs in terms of accuracy, when more complex simulations are performed.

6.2. Future Works

There are couple of possibilities left to improve SNNSim. The discussed possibilities are: Adding possible flags to the observability functions, Implementing an internal graphical display, Implementing multithreading and finally introducing the ability to save changed networks.

Flags Currently, the function PrintneuronInformation returns the information of the neuron with exact indexes of the neuron. To improve the readability of this function an improvement would be adding a possible flag to the function, which allows the function to either return the exact indexes of the neuron or the layer the neurons are situated in and their number inside the layer.

Graphs As of right now the only way to get a graph from the Neuron potential is exporting the potential to a Numpy file and importing it to python where Math Plot is able to make a graph. A Wrapper library that does this in C++ would be a great improvement for the goal of this project.

Multi Threading SNNSim currently handles events one by one as of right now. Since the events queue handled events layer by layer, the handling of the events one by one is not necessary as there is no conflict between inputs and outputs between those layers. Therefore, the events could easily be divided up between the available threads, provided by the CPU, to heavily speed up the program.

Saving Changed Networks Currently, the networks can only be saved in a text file format, Although this is advantageous for small size networks to see the inner workings and a great way to learn to understand neural networks, for larger networks this is both unreadable and therefore no longer convenient, On top of that, loading a network saved in text format is incredibly slow compared to a binary format saved network. A great improvement would therefore be a network saved entirely binary. Compared to the input file, this file cannot have the same structure. If for instance connections are removed by using the dropout function the file structure is no longer the same as the input. The moment you drop out a connection in the convolution layer, the saved format of only saving the weights of a window is no longer feasible. One option would be to see every layer as a fully connected layer, where the neurons that don't have a connection get a weight of 0, this would however greatly increase the file size of the network.

Bibliography

[1] URL: https://www.polarismarketresearch.com/industry-analysis/autonomous-vehicle-market.

[2] URL: https://www.grandviewresearch.com/industry-analysis/trade-surveillance-market.

[3] URL: https://www.bouvet.no/bouvet-deler/understanding-convolutional-neural-networks-part-2.

[4] URL: https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html.

[5] URL: https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf.

[6] Martín Abadi et al. *TensorFlow: A system for large-scale machine learning*. 2016. arXiv: 1605.08695 [cs.DC]. URL: https://arxiv.org/abs/1605.08695.

[7] Arnon Amir et al. "A Low Power, Fully Event-Based Gesture Recognition System". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 7388–7397. DOI: 10.1109/CVPR.2017.781.

[8] Keshav Bimbraw. "Autonomous Cars: Past, Present and Future - A Review of the Developments in the Last Century, the Present Scenario and the Expected Future of Autonomous Vehicle Technology". In: *ICINCO 2015 - 12th International Conference on Informatics in Control, Automation and Robotics, Proceedings* 1 (Jan. 2015), pp. 191–198. DOI: 10.5220/0005540501910198.

[9] Oktavia Catsaros. URL: https://www.bloomberg.com/company/press/generative-ai-to-become-a-1-3-trillion-market-by-2032-research-finds/.

[10] Ben Kröse et al. "An introduction to neural networks". In: *J Comput Sci* 48 (Jan. 1993).

[11] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *The mnist database*. 1994. URL: https://yann.lecun.com/exdb/mnist/.

[12] M. Minsky and S. Papert. *Perceptrons; an Introduction to Computational Geometry*. MIT Press. ISBN: 9780262630221.

[13] Gordon E. Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* (1965).

[14] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: https://arxiv.org/abs/1912.01703.

[15] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957. URL: https://websites.umass.edu/brain-wars/files/2016/03/rosenblatt-1957.pdf.

[16] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536. URL: https://api.semanticscholar.org/CorpusID:205001834.

[17] A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229. DOI: `10.1147/rd.33.0210`.

[18] Sumit Bam Shrestha and Garrick Orchard. “SLAYER: Spike Layer Error Reassignment in Time”. In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 1419–1428. URL: `http://papers.nips.cc/paper/7415-slayer-spike-layer-error-reassignment-in-time.pdf`.

[19] A. M. TURING. “I.—COMPUTING MACHINERY AND INTELLIGENCE”. In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. DOI: `10.1093/mind/LIX.236.433`. eprint: `https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf`. URL: `https://doi.org/10.1093/mind/LIX.236.433`.

[20] Jun Wang et al. “Safety of Autonomous Vehicles”. In: *Journal of Advanced Transportation* 2020 (Oct. 2020), pp. 1–13. DOI: `10.1155/2020/8867757`.