ISY Project: Using Spiking Neural Networks as a Sudoku Solver

Balavivek Sivanantham, Ferdinand Schlatt,
Mohith Bhargav Sunkara, Tim Westerhold

August 17, 2024

1 Introduction

The project "Using Spiking Neural Networks as a Sudoku Solver", supervised by Christoph Ostrau and Dr. Michael Thies, revolved around creating a Spiking Neural Network (SNN) to solve Sudokus of differing sizes. The final goal of the project was to integrate an SNN solver into a GUI, to create an easy to use platform to run and test different configurations and to visualize the SNN's output in an elegant way. Intermediately, all group members were to engage with theoretical principles of SNNs through practical application and learn different techniques for coordinating group work and projects.

In the beginning, the focus was set on learning the principles of SNNs, first through introductions to theory by the supervisors and afterwards through testing of small architectures. In group sessions, minimal WTA architectures of two rivaling neuron populations were created to gain insight into the spiking behavior. These were then expanded into a minimal Sudoku like problem. These tests were accompanied by active discussions about parameter fitting and architectural design to obtain the desired behavior.

Following this, three main sub tasks were identified and distributed among the group members. First of all, a Sudoku class was needed to generate and interact with Sudokus of variable sizes. On top of this, a SNN solver class was required, which builds on top of the Sudoku class and provides tools to automatically generate the appropriate solver structure and simulate the SNN. Lastly, a GUI should integrate both classes to provide a clean and intuitive interface to interact with Sudokus and the solver. Ferdinand was tasked with creating the Sudoku class, Tim and Balavivek worked on creating the solver and Mohith created the GUI.

Finally, the solver was evaluated in terms of performance and behavior. The analysis included a speed comparison to a deterministic algorithm, an investigation of the behavior in an ambiguous solution space, finding of a functional dependency between Sudoku size and SNN parameters and a speed analysis between Sudokus of differing difficulties. These analyses were also divided among the group members and results discussed in group meetings.

In the following sections, detailed descriptions of the different sub tasks are given. Next, the results of the analyses are reported and discussed and finally a conclusion about the project summarizes the report.

2 Solver Structure

2.1 Simulation Model

The solver is based on a spiking neural network, which is a technical model of biological neurons. To implement the network the c++ simulation framework cypress, which was developed by University Bielefeld with regards to the Human Brain Project, is used (https://github.com/hbp-unibi/cypress).

The framework employs the leaky integrate and fire model with fixed threshold and exponentially-decaying post-synaptic conductance of the PyNN [DBE+08] package for an efficient simulation of neurons.

Below the simulation model is described: Each neurons membrane voltage of the leaky integrate and fire model is defined by the differential equation

$$C_{\mathrm{m}}\frac{dV}{dt} = -g_{\mathrm{L}}(V(t) - V_{\mathrm{rest}}) + I \tag{1}$$

with the time constant for the leaky integrator is defined by $C_{\mathrm{m}}/g_{\mathrm{L}}$. Here C_{m} describes the capacity of the membrane while g_L is represents the conductance of the ionic channels. Furthermore V_{rest} is the membranepotential the neuron has when not being activated. The stimulus as input current is defined by the conductance based synapse model:

$$I = \sum_i g_{\mathrm{syn,i}}(t)(V(t) - E_{\mathrm{syn,i}}) \tag{2}$$

A synapse with an index i has a conductance $g_{\mathrm{syn,i}}$ and a reversal potential $E_{\mathrm{syn,i}}$. Its conductance has a stochastic dependence of the membranepotential and a time-dependend behavior in real neurons, which is simplified in the model to $-\tau_i\frac{dg_{\mathrm{i}}(t)}{dt} = g_{\mathrm{i}}(t)$ and $g_{\mathrm{i}}(t) \leftarrow g_{\mathrm{i}}(t) + w_{\mathrm{k}}$ if Spike incoming. The reversal potential is responsible for how a neuron reacts on a the output of a synapse. A reversal potential which is below the reset potential leads to an inhibitory reaction while a reversal potential above the threshold to an exitatory. In between the reaction depends on the current membrane voltage. Inhibitory means that the membranepotential is pulled towards hyperpolarisation whereas an exitatory stimulus depolarizes the membrane which facilitates a spike. [GKNP14]

For the implementation of the solver, the tuning of these parameters is crucial. It is explained in chapter 4 in detail.

2.2 Winner-Take-All Architecture

The solver is structured as a winner-take-all architecture (short WTA) as seen in figure 1. This means that the "most frequently firing neuron exerts the strongest inhibition on its competitors and thereby stops them from spiking" [Maa00]. In the following the application specific implementation of the WTA is described:

Every possible number in a field of the Sudoku is represented by a population of neurons. For example in figure 2, the structure of a 2x2 solver is shown, where the marked field represents the population for the number "1" in block, row and column one. These populations should hold themselves spiking and should oppress other populations standing in a logical

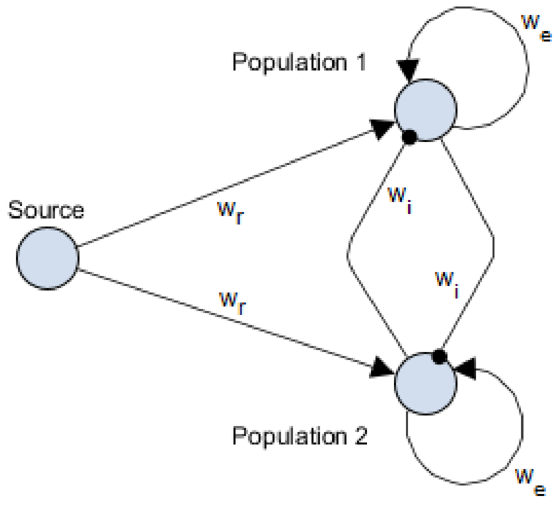


Figure 1: Winner-take-all architecture
w_e: excitatory connection, w_i: inhibitory connection, w_r: noise or trigger connection

conflict with them. A Logical conflict is given when multiple populations which represent different numbers for the same field are spiking. Also when populations representing the same number in the same row, column or finally in the same block of fields are spiking, the solution is wrong. For example if a field has the number one, it cannot have a second number like three at the same time, likewise there cannot be a second one in the same line, row or block.

This behaviour should lead to a stable formation of spiking neuron populations representing

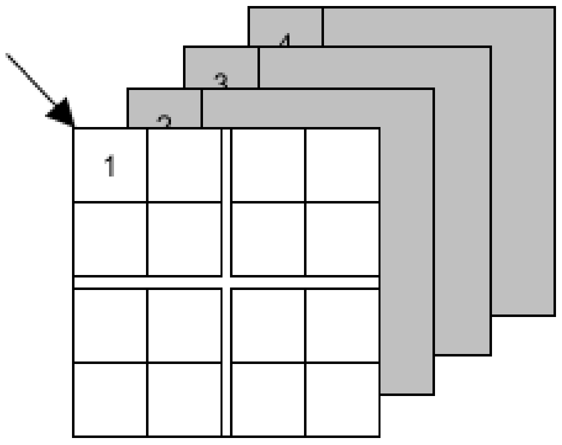


Figure 2: Populations for sudoku

a valid solution for the Sudoku. To achieve the oppression of populations, inhibitory connections with a negative weight are used. For the self-preservation of a neuron, excitatory connections (positive weight) are used. To achieve a quadratic Sudoku with a row size of n and a column size of m in a block, horizontally m and vertically n blocks are needed. This leads to a total amount of $(n * m)^3$ populations.

2.3 Noise

Sudokus are an integer (accepting only discrete values as solution) constraint satisfaction problem relating them to NP-complete complexity class. To solve these spiking neural networks with noise offer two important principles. On the one hand the constraint enforcement which is described int the WTA chapter, and on the other hand a heuristic search: Like shown in [BIP16] and in [FGF17] the deterministic model of a neural network is able to perform a Markov Chain Monte Carlo sampling when the neurons are interfered with noise, so that different possible solutions for a problem are reviewed. The combination forces the network to visit states with much higher probability if they satisfy more constraints, meaning that the spiking neural networks functions as a heuristic optimization. [Maa14] The noise is brought to the network by populations of neurons creating Poisson-noise. Each population in the network is randomly connected to its own noise source to ensure uncorrelated noise. Finally the constraint numbers given by the Sudoku have to be transferred to the network, too. This is done by trigger populations which are also noise sources, having a stronger weighted connection and a higher frequency than the noise sources to enforce the constraints. They are connected to the appropriate populations of the network.

3 Evaluation of spike times

In this evaluation of neural spikes, data are divided into smaller groups(bins) starting at the onset of a stimulus and lasting for complete spiking duration.These bins are evaluated for the number of neurons spiked in a particular bin. The algorithm is described below:

- The Bin size is defined in the parameter file. Number of bins needed for complete simulation time is calculated by dividing the bin size over duration.

  ```
  bin_size = m_config["bin_size"];
  num_of_bins = m_duration / bin_size;
  ```

  A 4Dimensional vector is used to count all the spikes in a sub-square , over complete duration of simulation time. The 1^{st} dimension is the sudoku height, the 2^{nd} the sudoku width, 3^{rd} dimension is the number of possible numbers and 4^{th} dimension is the number of bins.

- A loop that goes through all rows, columns, number and neurons of given sudoku to count the spikes. These spike counts are summed up in the above mentioned 4D vector.

- This step is to find a number, with the maximum amount of spikes for every sudoku block in every bin. Above mentioned count vector is iterated over every bin to calculate result vector with a maximum number of spikes. This result vector is the output of the spike time evaluation function.

4 Parameter Tuning

Parameter tuning is a crucial task, as the performance of an algorithm can be highly dependent on the choice of parameters. Spiking Neural Networks contain a large set of parameters which can be tuned. The specific parameters are explained in chapter 2 in detail. Currently, all the parameters are constant for different sizes of sudokus except for the excitation weight, inhibition weight and weight of random.

We were trying to form an individual function for each weight, that takes in the Sudoku size as input and outputs a weight for the neurons. The standard parameters for two cross two and three cross two Sudokus were found by hand. The data was used to fit a second order polynomial to predict the parameters for larger Sudokus. In this functions variable, x is the size of a Sudoku. For example if it is two cross two, the variable x is equal to 4, accordingly for three cross three Sudokus it is 9. y is then the corresponding weight. In our case, the fine-tuning was done especially on the following parameters:

1. **Weight of Self Excitation:** Weight of excitation was mostly used in between two neurons. w_e in the Figure 1 represent the weight of excitation. Change of this parameter will directly affect the neuron self excitation and connection strength.

$$y = -0.24 + 0.15 * x - 0.01 * x^2 \tag{3}$$

2. **Weight of Inhibition:** This parameter was utilized in all the inhibitory connections in the solver. w_i in the Figure 1 represents the weight of inhibition.

$$y = -0.94 + 0.267 * x - 0.0267 * x^2 \tag{4}$$

3. **Weight of Random Input:** This parameter is used in connections of population neurons and source neurons and is also represented by w_r in the Figure 1. The weight of random is varied to change the weight of this connection, which is used to add noise to the population neurons.

$$y = -0.18 + 0.192 * x - 0.012 * x^2 \tag{5}$$

These functions can then be used for finding the parameters for higher order Sudokus. Since only a small set of parameters was found by hand, it was not possible to find a function which could predict new parameters well. In order to find an optimal function a larger set of parameters is needed. Where as in our case we had only 3 sets of completely solved Sudoku parameters to fit the functions. Therefore these functions at present work well in lower order Sudokus and not for higher order Sudokus.

5 GUI

Sudokus in general have a standard form but can have different sizes by using differing amounts of numbers. This motivates the design of a graphical user interface (GUI), which enables the display and generation of Sudokus of variable size and visualizes the solver behavior. Further basic and advanced features were implemented to help the user to interact with Sudoku and solver which will be described in the following.

5.1 Work-Packages

This following section provides details about the work packages and the tools used to build the GUI and also an gives an overview of the features present in the final **Graphical User Interface (GUI)**.

5.1.1 Tools

The whole Graphical User Interface (GUI) is built using:

- **Tool** : **Qt-Creator 4.8.5**
- **Language** : **C++**
- **Build Tool** : **CMake 3.0**

5.1.2 Features

The GUI's incorporated features can be grouped into the following four categories:

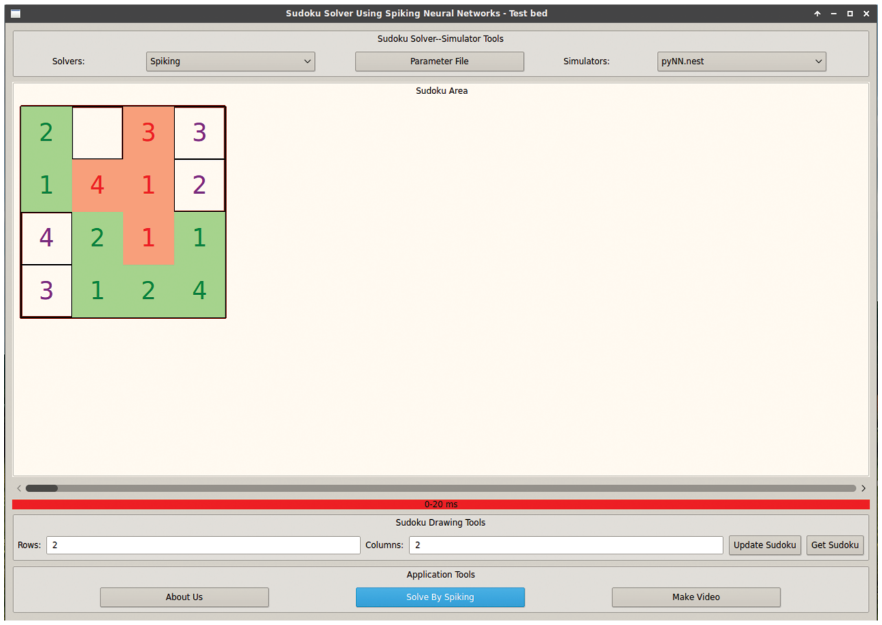
1. **Sudoku Solver & Simulator Tools**
2. **Sudoku Area**
3. **Sudoku Drawing Tools**
4. **Application Tools**

These categories also make up the general structure of the GUI, spanning from top to bottom. This creates a clean and easy to understand interface. The individual features contained within each category are explained in detail in the following section.

5.2 Structure And Functionality

Using figure 3, the features and their positions within the GUI are explained in detail.

1. **Sudoku Solver & Simulator Tools:** This slot holds the options for switching between different solver architectures, different parameter files for the chosen solver and different simulator environments in which the spiking neural network will be run.
    - Solvers: This feature provides the option to select different versions of solvers based on his/her preference for solving the Sudoku using spiking neural networks.
    - Parameter File: This feature provides the option to select the hyper-parameter files which are needed for solvers to perform the computation and solve the given Sudoku.



Solvers: Spiking
Simulators: pyNN.nest
Rows: 2
Columns: 2

Figure 3: Graphical User Interface

- Simulators: This feature provides the option to select the simulators on which the spiking solvers are needed to be simulated and solve the given Sudoku. This currently allows the switching between a simulated environment or dedicated hardware environment.

2. **Sudoku Area:** This slot holds the features related to the graphical display, observation of solver steps to solve the given Sudoku over-time and a status indicator.
    - Sudoku Display: This feature could be referred to as the face of the Graphical User Interface because of its functionality of drawing the Sudokus as specified by the user.
    - Scrub Bar: This feature provides two functionalities to the user namely:
        - Scrub Through Solution: This function provides the user to scroll through the solver time steps. Every number is checked for validity and color coded appropriately. This can be seen in figure 3 where a light-green color is used if the number is correct or a light-red color if not. A dark dark-magenta color is used for input numbers and dark-green if the Sudoku is solved.
        - Save As Png: This function is performed in the back-end for saving images (.Png) of the solver steps as the user scrubs over the solution and later used by another feature of the GUI to make a video from the saved images. The functionality is displayed in figure 4.
    - Color Indicator: This feature provides the user with the status of the Sudoku whether it is solved or not solved for each time-bin when the user scrubs over the solution. The figure 3 shows the color indicator for the first time-bin.

3. **Sudoku Drawing Tools:** This slot holds the features related to the drawing of the specified Sudoku by the user and also an extra function of drawing a Sudoku from a file.

   - Rows, Columns And Update Sudoku: This set of features combined together provide the user with the functionality of inputting his/her desired Sudoku size which is then directly scraped from a website.
   - Get Sudoku: This feature allows the user to input the Sudoku which is needed to be solved from a file. The file format which is supported is JSON.

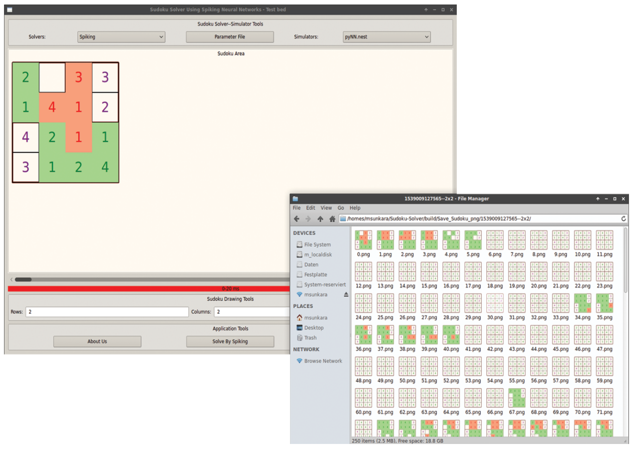


Figure 4: Graphical User Interface With Saving Images

4. **Application Tools:** This slot holds all the features related to application functionality such as basic information about why this GUI is built, and a button to initiate the solver and video creator.

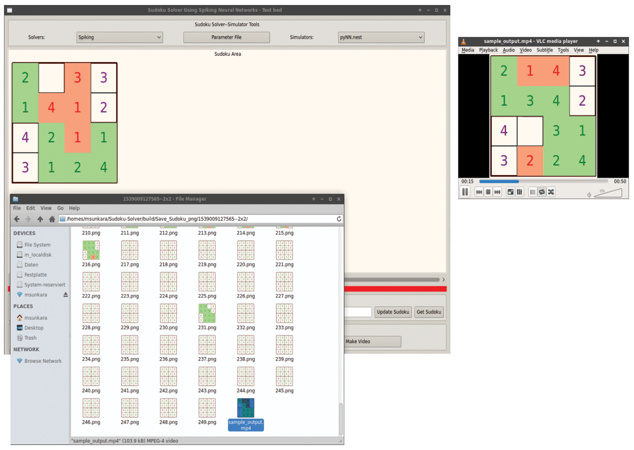

Sudoku Solver Using Spiking Neural Networks - Test bed
Sudoku Solver-Simulator Tools
Solvers:
Spiking
Parameter File
Simulators:
pyNN.nest
Sudoku Area
Update Sudoku
Get Sudoku
Make Video
sample_output.mp4 - VLC media player
File Manager

Figure 5: Graphical User Interface With Video Sample

   - About Us: This feature provides a basic information related to the project and GUI.

- Solve By Spiking: This feature holds the functionality of initiating solvers to solve the given Sudoku by the user using above features. Once initiated the spiking neural network starts the back-end computation to solve the Sudoku. Deep understanding of this solving can be seen in the above sections.
- Make Video: This feature holds the functionality of making a sample output video from the saved images, which can later be used to analyze the output in more detail as seen in figure 5.

6 Brute Force Solver

To be able to gauge the performance of the spiking Sudoku solver, a generic solver algorithm was implemented as a comparison. Different types of algorithms exist to solve Sudokus ranging from backtracking algorithms, to constraint programming and logic deduction. In this case, a simple greedy backtracking algorithm was implemented and is described below:

```
for row in Sudoku do
  for column in Sudoku do
    skip preset cells
    for number in possible numbers do
      if Sudoku[row][column] $\leftarrow$ number is valid then
        mark number as checked
        continue
      end if
    end for
    if no valid number found then
      reset all checked numbers for current cell
      backtrack to previous decision
    end if
  end for
end for
```

This algorithm has a run time $O(n^{n-k})$ in the worst case, with k equal to the number of fixed cells. Through improvements, like choosing the cell with the least number of possibilities first, it is possible to achieve a linear run time in many cases [Wel08].

7 Results

7.1 Brute Force vs Spiking Neural Network

For the evaluation of solve times of both the brute force solver and the SNN solver, 100 of each 2x2, 2x3, 2x4, 3x3 and 3x4 Sudokus were randomly generated. Both solvers were run on all Sudokus and the average solve time computed. The brute force solver was run until completion while the SNN was simulated for 5000 ms biological time. Since the brute force solver is a deterministic algorithm, it was able to solve every Sudoku. Average solve times for the brute force solver are in ms runtime (Table 1). The SNN solver on the other hand, was not able to solve all puzzles and failed to solve any of the 3x4 Sudokus. Solve percentages and solve times for the SNN solver are reported in ms biological time (Table 1).

Size	Solve Time		Solve Percentage	
	Brute Force	Spiking NN	Brute Force	Spiking NN
2x2	0.74	214.8	100%	100%
2x3	13.11	1252.14	100%	84%
2x4	804.83	2997.33	100%	15%
3x3	8963.48	2477.95	100%	39%
3x4	580634.03	$\emptyset$	100%	0%

Table 1: Brute force and SNN solver time comparison

The brute force algorithm shows an exponential increase in solve time. Compared to this, the SNN solver shows a nigh linear increase with increasing Sudoku size. While the brute force solver is able to solve 2x2, 2x3 and 2x4 Sudokus considerably faster than the SNN, at a size of 3x3, the SNN shows a more than 3 times better speed than the brute force solver. For increasing sizes, the advantage of the SNN would therefore increase exponentially.

A peculiar result is the increased solve time and decreased solve percentage of 2x4 Sudokus compared to 3x3 Sudokus for the SNN solver, even though the total size of the Sudoku is smaller. Further testing is needed to find the cause of this effect. Multiple reasons could be, for example, inadequate parameters for the 2x4 Sudoku size, increased difficulty for the solver for asymmetric Sudokus or increased difficulty through an exponential relationship between the size of single dimensions and difficulty.

Finally, this time analysis shows the potential of SNNs for NP problems. Through parallelization, the solve time of an, in the worst case, exponential problem was reduced to an almost linear solve time in biological time. This relationship does not carry over to simulation time though, as simulation times also increase exponentially with the Sudoku size (8.55, 21.55, 77.60, 148.36 and 164.38 for 2x2, 2x3, 2x4, 3x3 and 3x4 Sudokus respectively). To alleviate this problem, the possibility of running SNNs on dedicated neuromorphic hardware exists [PGJ+13] [SBG+10]. This can yield a consistent speed up of 10^4 times biological speed.

7.2 Ambiguous Sudokus

As described in chapter 2 the spiking neural network performs a heuristic optimization of the Sudoku problem given the rules of Sudoku and the initial numbers as constraints. In theory it should be capable of finding more than one solution for the Sudoku if it offers multiple possibilities. All costs of distributions of numbers in the Sudoku can be imagined as a landscape. Caused by the constraints of preset numbers correct solutions build a valley towards which the solver trends and between it should be able to change. This is caused by the noise input giving a non steady factor to enable the MCMC sampling. To review the hypothesis and to test whether the simulated network can find diverse solutions, a simple 2x2 Sudoku is used which is shown in figure 6. The initialization provides 4^{12} possible arrangements from which 12 are valid solutions. This is easy to derive:

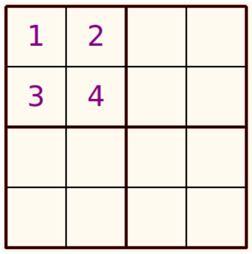


Figure 6: Initial Sudoku

In figure 7 every field is named by its index and a the letter c for a field with an initial constraint, so with a fix number, or f for a still free field. The block in the upper right and in the lower left are only depending on the initial block. As an example for F13 and F14 exists 2! possibilities, namely the numbers 3 and 4 which were not used in the same row (C11 and C12). The same principle can be applied to the second row, the first and second column. Which makes 2^4 possibilities. The block in the bottom right corner is then depending on these two blocks.

C11	C12	F13	F14
C21	C22	F23	F24
F31	F32	F33	F34
F41	F42	F43	F44

Figure 7: Sudoku with initial fields (c) and free fields (f)

In experiments using the standard parameters for two cross two Sudokus the solver finds different solutions in the first place needing different durations, but doesn't change between them, so the parameters need to be changed.

In the following experiments the simulation time was increased to 10,000 ms to observe how many solutions are found in one simulation. In a first parameter tuning experiment to find a feasible parameter set the firing rate of the noise sources was increased in steps of 40 while the weight for the noise populations was hold on 0.5. In figure 8 the results can be seen. Each number was tested three times. During the test, the number of found solutions varied for every try, but the solver didn't find all solutions independent of the noise rate. For higher noise rates the solver found up to 10 solutions on average. It can be assumed that with the noise rate the number of found solutions increases. Whether the solver reaches a plateau at $r = 160$ and $r = 200$ can not be said by this experiment, but it makes the impression to have a plateau at this point. As a second parameter, the weight of the noise connections was tested while holding the noise rate on 80. Here the weight was increased in 0.2 steps from 0.5 to 0.9. The results in figure 9 show that for higher weights the solver is more likely to find all solutions (for $w = 0.9$ nearly as good as $w = 0.7$. The experiment has no huge quantity of measurements, but gives a hint that the number of found solutions depends more on the weight than the rate. Although in the fist experiment the rate of 80 has the worst outcome, when the weight in the second experiment is increased for a rate of 80 it comes near finding all solutions. Following this idea, in a third experiment the rate is

r	80	120	160	200
n1	8	9	10	10
n2	9	6	11	9
n3	5	11	9	11
m	7.3	8.7	10.0	10.0

Figure 8: Number n of found solutions and mean m when increasing noise rate r (noise weight w = 0.5)

w	0.5	0.7	0.9
n1	9	12	12
n2	7	11	10
n3	6	12	12
m	7.3	11.7	11.3

Figure 9: Number n of found solutions and mean m when increasing noise connection weight w (noise rate r = 80)

increased, but with the weight of 0.7. In figure 10 can be seen that for every noise rate all solutions are found.

r	80	100	120
n	12	12	12

Figure 10: Number S of solutions when increasing noise rate R (weight W = 0.7)

To conclude not the noise rate is crucial for the network to find many solutions, but how strong the noise is, which can be influenced by the weight. A possible explanation is, that by increasing the noise rate, the hole spiking rate in the network increases, following the solutions are not stable anymore. Whereas raising the noise weight in comparison to the weights of the other connections to a population enhance the importance of the noise. Therefor the network is able to test new solutions more efficient although being inhibited by a current solution.

With a rough tuning of the parameters a feasible set is found for two cross two Sudokus is found, so it can be shown that the solver is able to find multiple solutions for ambigous Sudokus, although it is still unresolved whether the found parameters also fit for larger Sudokus.

8 Conclusion

Concluding, the project was successfully completed by creating an easy to use framework for generating and testing Sudokus on a spiking neural network Sudoku solver. Different solver configurations were tested for performance and evaluated in comparison to a typical deterministic algorithm. Finally, properties of the solver under non trivial circumstances were tested, for example for ambiguous and non symmetric Sudokus.

Next to the executed analyses, further open questions and improvements were identified which were not possible to complete in the scope of the project. These include integration and testing of the spiking neural network solver on dedicated hardware, identification of a functional dependency between network parameters and Sudoku size and analysis of behavior for Sudokus with differing numbers of open squares.

Finally, working as a group, it was possible to show the potential of spiking neural networks and at the same time learn important management, communication and development skills. It will be exciting to see a continuation of the project and the general development of spiking neural networks in the future.

References

[BIP16] Jonathan Binas, Giacomo Indiveri, and Michael Pfeiffer. Spiking analog vlsi neuron assemblies as constraint satisfaction problem solvers. *IEEE International Symposium on Circuits and Systems*, 2016.

[DBE+08] Andrew P. Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. Pynn: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2008.

[FGF17] Gabriel A. Fonseca Guerra and Steve B. Furber. Using stochastic spiking neural networks on spinnaker to solve constraint satisfaction problems. *Frontiers in Neuroscience*, 2017.

[GKNP14] Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. Neural dynamics - from single neurons to networks and models of cognition. website, 2014. online available under https://neuronaldynamics.epfl.ch/index.html; retrieved at 25th of October 2018.

[Maa00] Wolfgang Maass. On the computational power of winner-take-all. *Neural Computation, Volume 12*, 2000.

[Maa14] Wolfgang Maass. Noise as a resource for computation and learning in networks of spiking neurons. *Proceedings of the IEEE 102.5*, pages 860–880, 2014.

[PGJ+13] Thomas Pfeil, Andreas Grübl, Sebastian Jeltsch, Eric Müller, Paul Müller, Mihai A Petrovici, Michael Schmuker, Daniel Brüderle, Johannes Schemmel, and Karlheinz Meier. Six networks on a universal neuromorphic computing substrate. *Frontiers in neuroscience*, 7:11, 2013.

[SBG+10] Johannes Schemmel, Daniel Briiderle, Andreas Griibl, Matthias Hock, Karlheinz Meier, and Sebastian Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Circuits and systems (ISCAS), proceedings of 2010 IEEE international symposium on*, pages 1947–1950. IEEE, 2010.

[Wel08] Mathias Weller. Counting, generating, and solving sudoku. *Ph. D. Dissertation*, 2008.