ML2R Coding Nuggets
Hopfield Nets for Sudoku

Christian Bauckhage
Machine Learning Rhine-Ruhr
Fraunhofer IAIS
St. Augustin, Germany

Fabrice Beaumont
Computer Science
University of Bonn
Bonn, Germany

Sebastian Müller
Machine Learning Rhine-Ruhr
University of Bonn
Bonn, Germany

ABSTRACT

This note demonstrates that Hopfield nets can solve Sudoku puzzles. We discuss how to represent Sudokus in terms of binary vectors and how to express their rules and hints in terms of matrix-vector equations. This allows us to set up energy functions whose global minima encode the solution to a given puzzle. However, as these energy functions typically have numerous local minima, Hopfield nets with random selection or steepest descent updates rarely find the correct solution. We therefore consider stochastic Hopfield nets or Boltzmann machines whose neurons update according to a stochastic process called simulated annealing. Our corresponding *NumPy* code is comparatively simple and efficient and consistently yields good results.

1 INTRODUCTION

Sudoku is a combinatorial puzzle game in which players have to fill the cells of an $N \times N$ grid with the digits $1, 2, \ldots, N$. In the most common version of the game $N = 9$, but Sudokus can be designed for any square number $N = M^2 \geq 4$ (Fig. 1 shows examples where $N = 2^2$ and $N = 3^2$). Each Sudoku provides a few hints, i.e. a partially completed grid, and the catch is that players have to complete the grid such that each row, each column, and each $M \times M$ block of the grid contains all the digits 1 through N.

Human players typically resort to logical reasoning or guesswork when solving a Sudoku. While there exist AI solutions that mimic such strategies [26, 28], most computational Sudoku solvers rely on heuristic search techniques [19–21]. And of course, modern neural networks have been unleashed on the problem as well [25]. However, we are interested in the fact that Sudokus can also be solved using calculus-based optimization [2, 3, 13]. We have already seen that corresponding problem formulations can (sometimes) be translated into Hopfield energy minimization problems [8, 9], and indeed, Hopfield himself has written extensively on Hopfield nets for Sudoku [15]. This is of course of relevance for our current line of study.

In this note, we therefore discuss one possible way of setting up Sudoku as a Hopfield energy minimization problem. This will be a bit tedious but, given a well posed Sudoku, the energy function we derive will have a unique global minimizer that encodes the solution of the puzzle. While this seems to promise an easy approach towards solving Sudokus, it turns out that our energy function "suffers" from numerous local minima. Hence, the simple algorithms for running Hopfield nets which we considered earlier tend to fail in finding the sought after solution.

We therefore discuss the idea of *stochastic Hopfield* nets or *Boltzmann machines* and how to run them using a mechanism called *simulated annealing*. Computationally, this new framework is much more demanding than the methods we have worked with so far but empirical results indicate it to be well suited for Sudoku.

		3	
3			
			1
	2		

2	1	3	4
3	4	1	2
4	3	2	1
1	2	4	3

(a) 4×4 **Sudoku**

	4	9				1		6
		1		8				
	3				4		2	7
2		5	4	7	8	3		1
3		4	1		6	9	7	5
6		7		9				
	7					2		
		3	2		1			
			8	3			1	9

8	4	9	7	5	2	1	3	6
7	2	1	6	8	3	5	9	4
5	3	6	9	1	4	8	2	7
2	9	5	4	7	8	3	6	1
3	8	4	1	2	6	9	7	5
6	1	7	3	9	5	4	8	2
1	7	8	5	6	9	2	4	3
9	6	3	2	4	1	7	5	8
4	5	2	8	3	7	6	1	9

(b) 9×9 **Sudoku**

Figure 1: Examples of Sudoku puzzles and their solutions.

Given these remarks, it is no surprise that the following theory section will be longer and more elaborate than those in our previous notes on Hopfield nets. The upshot is that the practice section will be shorter than usual. Although the theory in section 2 may seem intricate, it is easily implemented in plain vanilla *NumPy* code. Readers who would like to experiment with the practical implementations we discuss in sectin 3 should be familiar with *NumPy*/*SciPy* [24] and only need to

```
import numpy as np
import numpy.random as rnd
```

2 THEORY

To ease into our theoretical discussion, we recall that a Hopfield net is a recurrent neural network of n interconnected neurons and that each of these neurons is a bipolar threshold unit. That is, if the vector $\boldsymbol{w}_i \in \mathbb{R}^n$ and scalar $\theta_i \in \mathbb{R}$ denote the input weights and threshold value of neuron i and if the vector $\boldsymbol{s} = [s_1, \ldots, s_n]^\top$

z_1	z_2	z_3	z_4
z_5	z_6	z_7	z_8
z_9	z_{10}	z_{11}	z_{12}
z_{13}	z_{14}	z_{15}	z_{16}

Figure 2: The content of a cell c of an $N \times N$ Sudoku grid can be encoded using a binary vector $z_c \in \{0,1\}^N$. Here, $N = 4$.

$\mathbf{0}$	$\mathbf{0}$	e_3	$\mathbf{0}$
e_3	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	e_1
$\mathbf{0}$	e_2	$\mathbf{0}$	$\mathbf{0}$

e_2	e_1	e_3	e_4
e_3	e_4	e_1	e_2
e_4	e_3	e_2	e_1
e_1	e_2	e_4	e_3

Figure 3: Using cell codes $z_c \in \{\mathbf{0}, e_1, \dots e_4\}$, the 4×4 Sudoku and its solution in Fig. 1(a) can be encoded as shown here.

gathers the current activation values of all the neurons of a Hopfield net, then neuron i computes its activation value as

$$s_i = \begin{cases} +1 & \text{if } \boldsymbol{w}_i^\top \boldsymbol{s} - \theta_i \geq 0 \\ -1 & \text{otherwise} \end{cases} \\ = \text{sign}\Big(\boldsymbol{w}_i^\top \boldsymbol{s} - \theta_i\Big) \tag{1}$$

Consequently, the collection of all activation values of a Hopfield net corresponds to a bipolar vector $\boldsymbol{s} \in \{-1,+1\}^n$.

For reasons that will become apparent soon, we will henceforth refer to an individual neuron's activation $s_i \in \{\pm 1\}$ as a *microscopic state* and to an activation vector $\boldsymbol{s} \in \{\pm 1\}^n$ as a *macroscopic state* of a Hopfield net.

Next, we recall that we may gather all weight vectors and threshold values in an $n \times n$ matrix and an n vector

$$\boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1^\top \\ \vdots \\ \boldsymbol{w}_n^\top \end{bmatrix} \quad \text{and} \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \tag{2}$$

and that the energy of a Hopfield network in state $\boldsymbol{s}$ is defined as

$$E(\boldsymbol{s}) = -\tfrac{1}{2}\, \boldsymbol{s}^\top \boldsymbol{W} \boldsymbol{s} + \boldsymbol{\theta}^\top \boldsymbol{s} \tag{3}$$

Finally, we recall the following important result: *If* the weight matrix $\boldsymbol{W}$ of a Hopfield net is symmetric and hollow (i.e. if $\boldsymbol{W} = \boldsymbol{W}^\top$ and $\text{diag}[\boldsymbol{W}] = \mathbf{0}$) and *if* its neurons update one by one using (1), *then* the energy of a Hopfield net can never increase. As there are "only" 2^n distinct states such a Hopfield net can be in, it will therefore settle in a (local) energy minimum after finitely many update steps [14].

All of this is to say that Hopfield nets allow for (approximately) solving **quadratic unconstrained binary optimization (QUBO) problems** of the following form

$$\boldsymbol{s}^* = \underset{\boldsymbol{s} \in \{\pm 1\}^n}{\text{argmin}} \; -\tfrac{1}{2}\, \boldsymbol{s}^\top \boldsymbol{W} \boldsymbol{s} + \boldsymbol{\theta}^\top \boldsymbol{s} \tag{4}$$

Given our topic in this note, the obvious question is now how to encode or represent Sudokus such that the task of solving them becomes the task of solving QUBOs as in (27)?

2.1 Hopfield Nets for Sudoku

In what follows, we will adhere to a strategy we have used before (see, for instance, [8]). That is, we will first demonstrate how to encode a Sudoku in terms of a QUBO over *binary* decision variables $z_i \in \{0,1\}$ and then rewrite it as a QUBO over *bipolar* decision variables $s_i \in \{-1,+1\}$.

Our method for encoding a Sudoku in terms of binary variables adopts earlier ideas [2, 3, 13] and may look tedious or complicated. But bear with us! It will pay off!

Dealing with an $N \times N$ Sudoku, we encode the content of each cell $c \in \{1,2,\dots,N^2\}$ of the grid as a binary vector $\boldsymbol{z}_c \in \{0,1\}^N$ (see Fig. 2 for a 4×4 example). These binary vectors are supposed to represent the numbers 1 through N as well as the number 0 which we use to mark initially empty cells. A natural code is thus

$$0 \Leftrightarrow \mathbf{0} \;\; = \big[0,0,\dots,0\big]^\top \tag{5}$$
$$1 \Leftrightarrow \boldsymbol{e}_1 \;\; = \big[1,0,\dots,0\big]^\top \tag{6}$$
$$2 \Leftrightarrow \boldsymbol{e}_2 \;\; = \big[0,1,\dots,0\big]^\top \tag{7}$$
$$\vdots$$
$$N \Leftrightarrow \boldsymbol{e}_N = \big[0,0,\dots,1\big]^\top \tag{8}$$

This way, the Sudoku in, say, Fig. 1(a) can be expressed as in Fig. 3.

Using cell codes $\boldsymbol{z}_c \in \{\mathbf{0}, \boldsymbol{e}_1, \dots \boldsymbol{e}_N\} \subset \{0,1\}^N$, a whole Sudoku grid can be now represented in terms of a vector

$$\boldsymbol{z} = \begin{bmatrix} \boldsymbol{z}_1 \\ \boldsymbol{z}_2 \\ \vdots \\ \boldsymbol{z}_{N^2} \end{bmatrix} \tag{9}$$

such that $\boldsymbol{z} \in \{0,1\}^{N^3}$. For example, for a 4×4 Sudoku, $\boldsymbol{z}$ will be a 64-dimensional binary vector, for a 9×9 Sudoku, it will be 729-dimensional, and so on and so forth.

Next, we need to think of how to encode the constraints a valid Sudoku solution must fulfill.

- No cell of a completed Sudoku grid can be empty; it rather has to contain one of the numbers 1 through N. With respect to the binary vector $\boldsymbol{z}_c$ that represents cell c, we must therefore insist on

  $$\mathbf{1}_N^\top \boldsymbol{z}_c = 1 \tag{10}$$

  where $\mathbf{1}_N$ denotes the N-dimensional vector of all ones. To express this constraint for all the N^2 grid cells in a single equation, we introduce an $N^2 \times N^3$ matrix

  $$\boldsymbol{G} = \boldsymbol{I}_{N^2} \otimes \mathbf{1}_N^\top \tag{11}$$

where $\boldsymbol{I}_{N^2}$ denotes the $N^2 \times N^2$ identity matrix and $\otimes$ is the Kronecker product. Using $\boldsymbol{G}$, we can write the individual constraints in (10) collectively as

$$\boldsymbol{G}\,\boldsymbol{z} = \boldsymbol{1}_{N^2} \tag{12}$$

where $\boldsymbol{1}_{N^2}$ denotes the N^2-dimensional vector of all ones.

- Each row of a completed Sudoku grid must contain each of the numbers 1 through N exactly once. To enforce this for row $i \in \{1, \ldots, N\}$, we need to insist on

$$\begin{bmatrix}\boldsymbol{I}_N \cdots \boldsymbol{I}_N\end{bmatrix} \begin{bmatrix} z_{(i-1)\cdot N+1} \\ \vdots \\ z_{(i-1)\cdot N+N} \end{bmatrix} = \boldsymbol{1}_N \tag{13}$$

where $\boldsymbol{I}_N$ denotes the $N \times N$ identity matrix. To express these N row constraints in a single equation, we introduce an $N^2 \times N^3$ matrix

$$\boldsymbol{R} = \boldsymbol{I}_N \otimes \left(\boldsymbol{1}_N^\intercal \otimes \boldsymbol{I}_N\right) \tag{14}$$

This way, we can write the individual constraints in (13) collectively as

$$\boldsymbol{R}\,\boldsymbol{z} = \boldsymbol{1}_{N^2} \tag{15}$$

- Each column of a completed Sudoku grid must contain each of the numbers 1 through N exactly once. To enforce this for column $j \in \{1, \ldots, N\}$, we need to insist on

$$\begin{bmatrix}\boldsymbol{I}_N \cdots \boldsymbol{I}_N\end{bmatrix} \begin{bmatrix} z_{(1-1)\cdot N+j} \\ \vdots \\ z_{(N-1)\cdot N+j} \end{bmatrix} = \boldsymbol{1}_N \tag{16}$$

To express these N column constraints in a single equation, we introduce an $N^2 \times N^3$ matrix

$$\boldsymbol{C} = \boldsymbol{1}_N^\intercal \otimes \boldsymbol{I}_{N^2} \tag{17}$$

This way, we can write the individual constraints in (16) collectively as

$$\boldsymbol{C}\,\boldsymbol{z} = \boldsymbol{1}_{N^2} \tag{18}$$

- Letting $M = \sqrt{N}$, each $M \times M$ block of a completed Sudoku grid must contain each of the numbers 1 through N exactly once. To make a long story short, we may introduce an $N^2 \times N^3$ matrix

$$\boldsymbol{B} = \boldsymbol{I}_M \otimes \left(\boldsymbol{1}_M^\intercal \otimes \left(\boldsymbol{I}_M \otimes \left(\boldsymbol{1}_M^\intercal \otimes \boldsymbol{I}_M\right)\right)\right) \tag{19}$$

where $\boldsymbol{I}_M$ denotes the $M \times M$ identity matrix and $\boldsymbol{1}_M$ the M-dimensional vector of all ones. Using $\boldsymbol{B}$, we can then write the individual block constraints collectively as

$$\boldsymbol{B}\,\boldsymbol{z} = \boldsymbol{1}_{N^2} \tag{20}$$

- Finally, we must pay attention to the fact that a valid completed Sudoku must comply with the initial hints. If there are H hints, we may consider an $H \times N^3$ matrix $\boldsymbol{H}$ which has to be constructed in a case by case manner. That is, in contrast to the structural constraints we considered so far, there is no "simple" mathematical expression for this matrix. Yet,we will see below that it is easy to implement code for constructing it. Assuming matrix $\boldsymbol{H}$ has been constructed, the hint constraints can be collectively expressed as

$$\boldsymbol{H}\,\boldsymbol{z} = \boldsymbol{1}_H \tag{21}$$

where $\boldsymbol{1}_H$ denotes the H-dimensional vector of all ones.

In order to turn the Sudoku constraints in (12), (15), (18), (20), and (21) into the ingredients of a QUBO, we now note that

$$\boldsymbol{G}\,\boldsymbol{z} = \boldsymbol{1}_{N^2} \Leftrightarrow \left(\boldsymbol{G}\,\boldsymbol{z} - \boldsymbol{1}_{N^2}\right)^2 = 0 \tag{22}$$

and that

$$\left(\boldsymbol{G}\,\boldsymbol{z} - \boldsymbol{1}_{N^2}\right)^2 = \boldsymbol{z}^\intercal \boldsymbol{G}^\intercal \boldsymbol{G}\,\boldsymbol{z} - 2\,\boldsymbol{1}_{N^2}^\intercal \boldsymbol{G}\,\boldsymbol{z} + const \tag{23}$$

Similar considerations do of course apply to the other constraints as well. Hence, if we introduce *importance factors*

$$\lambda_G, \lambda_R, \lambda_C, \lambda_B, \lambda_H \in \mathbb{R}_+ \tag{24}$$

which we may use to weigh the influence of the corresponding constraints, we can compute a symmetric $N^3 \times N^3$ matrix

$$\boldsymbol{P} = \lambda_G \boldsymbol{G}^\intercal \boldsymbol{G} + \lambda_R \boldsymbol{R}^\intercal \boldsymbol{R} + \lambda_C \boldsymbol{C}^\intercal \boldsymbol{C} + \lambda_B \boldsymbol{B}^\intercal \boldsymbol{B} + \lambda_H \boldsymbol{H}^\intercal \boldsymbol{H} \tag{25}$$

and an N^3-dimensional vector

$$\boldsymbol{p} = 2\left[\boldsymbol{1}_{N^2}^\intercal \left[\lambda_G \boldsymbol{G} + \lambda_R \boldsymbol{R} + \lambda_C \boldsymbol{C} + \lambda_B \boldsymbol{B}\right] + \boldsymbol{1}_H^\intercal \lambda_H \boldsymbol{H}\right]^\intercal \tag{26}$$

and find that to solve a Sudoku is to solve the following binary QUBO

$$\boldsymbol{z}^* = \underset{\boldsymbol{s} \in \{0,1\}^{N^3}}{\operatorname{argmin}}\ \boldsymbol{z}^\intercal \boldsymbol{P} \boldsymbol{z} - \boldsymbol{p}^\intercal \boldsymbol{z} \tag{27}$$

In order to turn this binary QUBO into a bipolar QUBO of a form that can be solved by a Hopfield net, we proceed as we have done several times before and introduce

$$\boldsymbol{Q} = \tfrac{1}{4}\,\boldsymbol{P} \tag{28}$$

$$\boldsymbol{q} = \tfrac{1}{2}\left[\boldsymbol{P}\,\boldsymbol{1} - \boldsymbol{p}\right] \tag{29}$$

$$\boldsymbol{W} = -2\left[\boldsymbol{Q} - \operatorname{diag}\left[\operatorname{diag}\left[\boldsymbol{Q}\right]\right]\right] \tag{30}$$

$$\boldsymbol{\theta} = \boldsymbol{q} \tag{31}$$

to finally obtain the Sudoku problem as

$$\boldsymbol{s}^* = \underset{\boldsymbol{s} \in \{\pm 1\}^{N^3}}{\operatorname{argmin}}\ -\tfrac{1}{2}\,\boldsymbol{s}^\intercal \boldsymbol{W}\,\boldsymbol{s} + \boldsymbol{\theta}^\intercal \boldsymbol{s} \tag{32}$$

Given all our previous notes on Hopfield nets for problem solving, it seems as if we could conclude our theoretical discussion at this point. This is because we might now use one of the previously discussed update mechanisms based on (1) in order to run a Hopfield net to (try to) solve (32). However, it turns out that Sudokus are difficult. By that we mean that we did of course experiment with the update mechanisms in our earlier notes but found them to be generally incapable of finding the solution to (32). The problem is that the energy function in (32) generally has numerous local minima which almost correspond to the optimal solution but only almost. Therefore, since almost solving a Sudoku puzzle is not the same as solving a Sudoku puzzle, we need to continue our discussion and introduce a novel, alternative idea for how to (possibly) find the global minimizer of the Sudoku problem in (32). Again, for those who have never seen what is to come next, the following approach may look rather complicated. But bear with us, it can easily be implemented and (usually) solves Sudokus correctly.

2.2 Boltzmann Machines

The methods we discuss next can be traced back to work by Ackley, Hinton, and Sejnowski in the 1980s [1]. The basic idea is to understand the neurons of a Hopfield net as a large statistical ensemble of microscopic entities and to apply techniques from **statistical mechanics** to reason about the collective behavior of these entities or, equivalently, about the macroscopic state of the network.

Drawing a physical analogy to, say, a chunk of ferromagnetic material whose macroscopic magnetic moment arises from the interactions of numerous microscopic magnetic dipoles, we say that, at *thermodynamic temperature* T, the *probability* of observing a Hopfield net in a specific macroscopic state $\boldsymbol{s}$ is given by the **Boltzmann distribution**

$$p(\boldsymbol{s} \mid T) = \frac{1}{Z} \cdot e^{-\frac{1}{k_B T} E(\boldsymbol{s})} \tag{33}$$

Here, the normalization constant

$$Z = \sum_{\boldsymbol{s}'} e^{-\frac{1}{k_B T} E(\boldsymbol{s}')} \tag{34}$$

is called the **partition function**. To compute it for a Hopfield net of n neurons, we would have to sum over all 2^n possible macroscopic states of the network. For large n, say $n \geq 100$, this is practically infeasible. However, for the result we derive below, this is no reason for concern as we only work with expressions where occurrences of Z cancel out.

The parameter $k_B = 1.380649 \times 10^{-23}$ is called the **Boltzmann constant** (expressed in Joule per Kelvin). As data scientists, we need not worry about its physical significance. For simplicity, we thus henceforth assume that

$$k_B \equiv 1 \tag{35}$$

The notion of the **thermodynamic temperature** T can be understood as follows: *thermal fluctuations* of a system are random deviations from the system's *average state at equilibrium*. At high temperatures T, such thermal fluctuations are larger and more likely. In the context of Hopfield nets, this means that, at high temperatures, neurons may likely and spontaneously switch their microscopic state so that the macroscopic state of the network is less deterministic than at low temperatures. In order to reduce notational clutter, we henceforth work with the reciprocal of the thermodynamic temperature, namely the **thermodynamic beta**

$$\beta \equiv \frac{1}{T} \tag{36}$$

In order to reduce notational clutter even further, we henceforth simply write

$$p(\boldsymbol{s}) \equiv p(\boldsymbol{s} \mid T) \tag{37}$$

Finally, we point out that this joint probability can generally be factored as

$$p(\boldsymbol{s}) = p(s_1, \ldots, s_i, \ldots, s_n) = p(s_i \mid s_{j \neq i}) \cdot p(s_{j \neq i}) \tag{38}$$

Given these preparatory remarks, we now consider a Hopfield net that has been running for a while and thus has reached a certain macroscopic state. With respect to this state, we are interested in the conditional probability $p(s_i = +1 \mid s_{j \neq i})$.

To determine this probability, we observe that, with respect to s_i, we can think of either one of the following two macroscopic states

$$\boldsymbol{s}_1 = \big[s_1, \ldots, s_i = +1, \ldots s_n\big]^\top \tag{39}$$

$$\boldsymbol{s}_2 = \big[s_1, \ldots, s_i = -1, \ldots s_n\big]^\top \tag{40}$$

so that we can compute the *log-odds* for s_i having a value of +1 as

$$\ln \frac{p(\boldsymbol{s}_1)}{p(\boldsymbol{s}_2)} = \ln \frac{1}{Z} - \beta E(\boldsymbol{s}_1) - \ln \frac{1}{Z} + \beta E(\boldsymbol{s}_2) \tag{41}$$

$$= \beta \Big(E(\boldsymbol{s}_2) - E(\boldsymbol{s}_1) \Big) \tag{42}$$

$$= \beta \, \Delta E \tag{43}$$

Using the factorization (38), we can also write these odds as follows

$$\ln \frac{p(\boldsymbol{s}_1)}{p(\boldsymbol{s}_2)} = \ln \frac{p(s_i = +1 \mid s_{j \neq i}) \cdot p(s_{j \neq i})}{p(s_i = -1 \mid s_{j \neq i}) \cdot p(s_{j \neq i})} \tag{44}$$

$$= \ln \frac{p(s_i = +1 \mid s_{j \neq i})}{p(s_i = -1 \mid s_{j \neq i})} \tag{45}$$

$$= \ln p(s_i = +1 \mid s_{j \neq i}) - \ln p(s_i = -1 \mid s_{j \neq i}) \tag{46}$$

$$= \ln p(s_i = +1 \mid s_{j \neq i}) - \ln \Big(1 - p(s_i = +1 \mid s_{j \neq i})\Big) \tag{47}$$

By equating the expressions in (43) and (47), we therefore find that

$$\ln p(s_i = +1 \mid s_{j \neq i}) - \ln \Big(1 - p(s_i = +1 \mid s_{j \neq i})\Big) = \beta \, \Delta E \tag{48}$$

and an exponentiation of this equation provides us with

$$\frac{p(s_i = +1 \mid s_{j \neq i})}{1 - p(s_i = +1 \mid s_{j \neq i})} = e^{\beta \, \Delta E} \tag{49}$$

In order to solve this expression for the sought after probability $p(s_i = +1 \mid s_{j \neq i})$, we note that (49) is equivalent to

$$p(s_i = +1 \mid s_{j \neq i}) = e^{\beta \, \Delta E} - e^{\beta \, \Delta E} \, p(s_i = +1 \mid s_{j \neq i}) \tag{50}$$

This, in turn, is equivalent to

$$p(s_i = +1 \mid s_{j \neq i}) + e^{\beta \, \Delta E} \, p(s_i = +1 \mid s_{j \neq i}) = e^{\beta \, \Delta E} \tag{51}$$

from which we find

$$p(s_i = +1 \mid s_{j \neq i}) \cdot \big(1 + e^{\beta \, \Delta E}\big) = e^{\beta \, \Delta E} \tag{52}$$

and therefore

$$p(s_i = +1 \mid s_{j \neq i}) = \frac{e^{\beta \, \Delta E}}{1 + e^{\beta \, \Delta E}} \tag{53}$$

To further consolidate this result, we multiply the probability in (53) with a rather intricate version of the number 1 which leads to

$$\frac{e^{\beta \, \Delta E}}{1 + e^{\beta \, \Delta E}} \cdot \frac{e^{-\beta \, \Delta E}}{e^{-\beta \, \Delta E}} = \frac{1}{e^{-\beta \, \Delta E} + 1} = \frac{1}{1 + e^{-\beta \, \Delta E}} \tag{54}$$

All in all, we therefore have the remarkable and crucial result that, at thermodynamic temperature $T = 1/\beta$, the probability

$$p(s_i = +1 \mid s_{j \neq i}) = \frac{1}{1 + e^{-\beta \, \Delta E}} \tag{55}$$

of a Hopfield neuron i being active is a **logistic function** of the scaled energy difference ΔE between its inactive and active state.

From an earlier discussion [4] we also know that the particular energy difference

$$\Delta E = E(\boldsymbol{s}_2) - E(\boldsymbol{s}_1) \tag{56}$$

can also be written as

$$\Delta E = 2 \cdot \left(\boldsymbol{w}_i^\top \boldsymbol{s} - \theta_i\right) \tag{57}$$

It is therefore surprisingly easy to practically compute the probability in (55). Given a thermodynamic temperature $T = 1/\beta$, the weights $\boldsymbol{W}$, biases $\boldsymbol{\theta}$, and current macroscopic state $\boldsymbol{s}$ of a Hopfield net, the right hand side of (55) can be readily evaluated.

Based on this observation, we now introduce the crucial idea of *stochastic* updates of a neuron i of a Hopfield net. In this framework, we first compute the probability $p(s_i = +1 \mid s_{j\neq i})$ which we henceforth abbreviate as q_i. That is, we compute

$$q_i = \left(1 + e^{-\frac{2}{T}(\boldsymbol{w}_i^\top \boldsymbol{s} - \theta_i)}\right)^{-1} \tag{58}$$

for which we are guaranteed that it is a number between 0 and 1. We may thus treat q_i as the success parameter of a Bernoulli distribution over a binary random variable and sample this distribution

$$z_i \sim Bernoulli(q_i) \tag{59}$$

Finally, we can turn this binary number z_i into a bipolar number

$$s_i = 2 \cdot z_i - 1 \tag{60}$$

and thus obtain a state update of the neuron under consideration.

A Hopfield net whose neurons update like this is a called a *stochastic Hopfield net* or **Boltzmann machine** and we note the following:

- In contrast to the asynchronous updates of a deterministic Hopfield net, asynchronous updates of a stochastic Hopfield net may cause its energy to increase. However, this may be beneficial as it can allow the network to escape from local energy minima …
- At (very) high thermodynamic temperatures $T \gg 0$, we have $q_i \approx 1/2$ regardless of the value of the energy difference $\Delta E = 2\left(\boldsymbol{w}_i^\top \boldsymbol{s} - \theta_i\right)$. At high temperatures, stochastic updates of a neuron will therefore equally likely increase or decrease the network's energy.
- At lower temperatures, on the other hand, the value of ΔE will matter. Recall that we understand it as the decrease or increase in energy due to switching micro state s_i from +1 to −1. If such a switch would decrease the overall energy, then $\Delta E \leq 0$. However, if deactivating s_i is energetically favorable, then the probability $q_i = p(s_i = +1 \mid s_{j\neq i})$ should be small. And indeed, for cases where $\Delta E \leq 0$, the exponent of e in (58) becomes positive so that q_i becomes small. For lower and lower temperatures, it is thus less and less likely that stochastic updates increase the network's energy.
- In fact, "one can show" that, in the limit $T \to 0$, stochastic Hopfield nets become deterministic Hopfield nets whose neurons update according to (1).
- Moreover, "one can show" that, after many asynchronous updates, the probability for a stochastic Hopfield net to be in a specific macroscopic state $\boldsymbol{s}$ follows the Boltzmann distribution in (33), irrespective of the value of T and the macroscopic state the network started in. The corresponding likelihood

$$\ln p(\boldsymbol{s}) \propto -\tfrac{1}{T} E(\boldsymbol{s}) \tag{61}$$

  is thus proportional to the energy of the macroscopic state.
- This is interesting, because, for two different states $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ with $E(\boldsymbol{s}_1) \leq E(\boldsymbol{s}_2)$, this implies $\ln p(\boldsymbol{s}_1) \geq \ln p(\boldsymbol{s}_2)$ which is to say that low energy states are more likely.
- Note that the likelihood in (61) also depends on T. If we thus consider a state of low energy $E(\boldsymbol{s}) < 0$ and two temperatures $T_1 < T2$, we find $\ln p(\boldsymbol{s} \mid T_1) > \ln p(\boldsymbol{s} \mid T_2)$. For low temperatures, it thus appears to be even more likely to find the network in a low energy state. Recall, however, that, at low temperatures, stochastic updates mainly tend to decrease energies. For a stochastic Hopfield net running at low temperatures, there is thus a certain risk that it will get trapped in local energy minima so that low temperatures per se are not necessarily advantageous.

All these observations have led to the idea of running Hopfield nets under a dynamic called **simulated annealing** which we will discuss next.

Algorithm 1 Simulated annealing of a Hopfield net

randomly initialize a macroscopic state $\boldsymbol{s}$

for $T = T_h$ **downto** T_l **do**

  for $r = 1$ **to** $r_{\max}$ **do**

    for $i = 1$ **to** n **do**

      $q_i = \left(1 + e^{-\frac{2}{T}(\boldsymbol{w}_i^\top \boldsymbol{s} - \theta_i)}\right)^{-1}$

      $z_i \sim Bernoulli(q_i)$

      $s_i = 2 \cdot z_i - 1$

2.3 Simulated Annealing

The term simulated annealing alludes to the process of annealing in metallurgy, a technique that involves repeated heating and controlled cooling of a material to increase the size of its constituent crystals and reduce potential defects. For stochastic Hopfield nets or Boltzmann machines, the process is as follows:

Consider an initial state $\boldsymbol{s}$ and a fairly high temperature T. At this temperature, consider a fairly large number of rounds (e.g. at least twice as large as the number of neurons of the network). In each round, asynchronously (re)set the states s_i of all neurons according to the stochastic update mechanism in (58)–(60). Then, lower the temperature and repeat.

A pseudo-code implementation of this procedure is shown in Algorithm 1. An interpretation of the behavior of this algorithm is as follows: At the initial high temperatures, the neurons of the network flip more or less randomly so that the network "explores" the whole state space. If this exploration runs long enough, the network will likely assume a macroscopic state in the vicinity of

a (global) energy minimum. Gradually reducing the temperature will cause the network to explore less and less but to begin to descend down the energy landscape. This way, it may converge to a Boltzmann distribution where the network's energy fluctuates around the global minimum.

3 PRACTICE

In this section, we discuss how to practically implement Hopfield nets for Sudoku. As a simple practical example, we consider the Sudoku in Fig. 1(a) which we may implement as a 2D *NumPy* array

```
sudoku = np.array([[0,0,3,0],
                   [3,0,0,0],
                   [0,0,0,1],
                   [0,2,0,0]])
```

Based on this representation, we can initialize the weight matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{\theta}$ of a corresponding Sudoku Hopfield net. To accomplish this, we use function `hnetInitParameters` in Listing 1 and call

```
matW, vecT = hnetInitParameters(sudoku)
```

to obtain corresponding *NumPy* arrays `matW` and `vecT`. In light of our discussion in section 2.1, the inner workings of this function are rather self explanatory. Note that we could have implemented the required computations in fewer lines of code but opted for better readability instead. Also note that the parameters `lH`, `lG`, `lR`, `lC`, and `lB` of `hnetInitParameters` represent the influence factors λ_H, λ_G, λ_R, λ_C, and λ_B we introduced above and that these factors may need manual tuning to accommodate Sudokus of different sizes and with different numbers of hints.

As the next step, we randomly initialize a macroscopic state $\boldsymbol{s}$ for our Hopfield net to start in. To this end, we use

```
n3 = sudoku.shape[0]**3
vecS = rnd.binomial(n=1,p=0.05,size=n3) * 2 - 1
```

which produces an array `vecS` of size N^3 where about 5% of the entries will have a value of $+1$ and the remaining ones a value of -1. Such a random initialization of the macroscopic state of our network may seem curious for it ignores the hints contained in array `sudoku`. However, the way we did set up the weights and biases of our network encodes and enforces these hints.

After these preparations, we can now run the simulated annealing procedure discussed in section 2.3 in order to (try to) have our Hopfield converge towards a low energy state. Ideally, this would be the state of lowest possible energy which correspondingly would encode the solution to our Sudoku. We therefore use

```
vecS = hnetSimAnn(vecS, matW, vecT)
```

which calls function `hnetSimAnn` in Listing 2. In addition to the parameters `vecS`, `matW`, and `vecT` whose role is obvious, this function has further parameters `Th`, `Tl`, `numT`, and `rmax`. Parameters `Th` and `Tl` represent the highest and lowest thermodynamic temperature we wish to work with and `numT` indicates how many temperature levels are to be considered. Parameter `rmax` indicates the number of rounds that ought to be run at each temperature level. Given these explanations, the inner workings of `hnetSimAnn` are again rather self explanatory as the function realizes a straightforward *NumPy* implementation of the pseudo-code in Alg. 1. However, we should probably point out that we are considering an annealing or cooling schedule where temperatures are linearly decreased from `Th` down to `Tl`. Other schedules such as exponential or power-law decay are

Listing 1: initializing parameters W and θ of a Hopfield net

```
def hnetInitParameters(sudoku, lH=4., lG=2., lR=1., lC=1., lB=1.):
    n1 = sudoku.shape[0]
    n2 = n1**2
    n3 = n1**3
    m  = np.sqrt(n1).astype(int)
    nh = np.sum(sudoku>0) # number of hints

    vec1_m  = np.ones(m )
    vec1_n1 = np.ones(n1)
    vec1_n2 = np.ones(n2)
    vec1_n3 = np.ones(n3)
    vec1_nh = np.ones(nh)

    matI_m  = np.eye(m )
    matI_n1 = np.eye(n1)
    matI_n2 = np.eye(n2)

    # matrix for hint constraints
    h    = 0
    matH = np.zeros((nh, n3))
    for i in range(n1):
        for j in range(n1):
            v = sudoku[i,j]
            if v>0:
                matH[h,i*n2+j*n1:i*n2+j*n1+n1] = matI_n1[v-1]
                h = h+1

    # matrices for cell, row, column, and block constraints
    matG = np.kron(matI_n2, vec1_n1)

    matR = np.kron(vec1_n1, matI_n1)
    matR = np.kron(matI_n1, matR   )

    matC = np.kron(vec1_n1, matI_n2)

    matB = np.kron(vec1_m, matI_n1)
    matB = np.kron(matI_m, matB   )
    matB = np.kron(vec1_m, matB   )
    matB = np.kron(matI_m, matB   )

    # matrix and vector for binary QUBO
    matP  = lH * matH.T @ matH
    matP += lG * matG.T @ matG
    matP += lR * matR.T @ matR
    matP += lC * matC.T @ matC
    matP += lB * matB.T @ matB

    vecP  = lH * 2 * vec1_nh @ matH
    vecP += lG * 2 * vec1_n2 @ matG
    vecP += lR * 2 * vec1_n2 @ matR
    vecP += lC * 2 * vec1_n2 @ matC
    vecP += lB * 2 * vec1_n2 @ matB

    # matrix and vector for bipolar QUBO
    matQ = 0.25 *  matP
    vecQ = 0.50 * (matP @ vec1_n3 - vecP)

    # weight matrix and bias vector for Hopfield net
    matW = -2 * matQ; np.fill_diagonal(matW, 0)
    vecT = vecQ

    return matW, vecT
```

Listing 2: simulated annealing of a Hopfield net

```
def hnetSimAnn(vecS, matW, vecT, Th=10, Tl=0.5, numT=21, rmax=200):
    n = len(vecS)

    for T in np.linspace(Th, Tl, numT):
        for r in range(rmax):
            for i in range(n):
                q = 1 / (1 + np.exp(-2/T * (matW[i]@vecS-vecT[i])))
                z = rnd.binomial(n=1, p=q)
                vecS[i] = 2 * z - 1

    return vecS
```

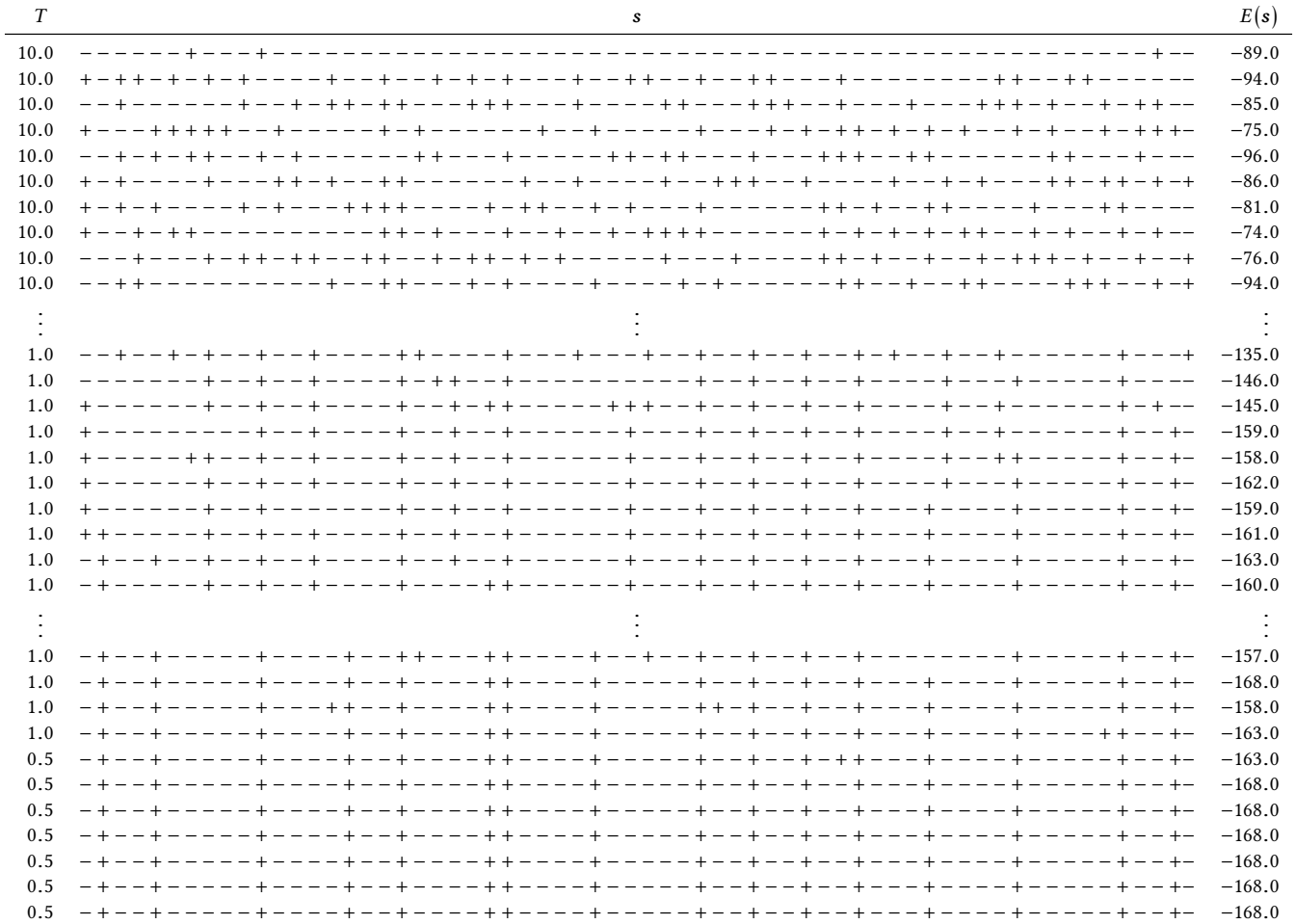

Figure 4: Visualization of an annealing process of a Hopfield net of 64 neurons that solves the 4×4 Sudoku in Fig. 1(a). In this example, only 3 neurons were initially active. At the initially high temperature level ($T = 10.0$), energies $E(s)$ were fairly high and fluctuated up and down. Later in the process, at lower temperature levels ($T = 1.0$), the network reached a state space region of noticeably lower energies and did not fluctuate as widely anymore. At the lowest temperature level considered here ($T = 0.5$), the network eventually reached a state of globally minimum energy and did not leave it anymore.

possible as well and it might be an interesting exercise to experiment with those. We also remark that it may be prudent to keep track of which macroscopic state achieved the overall lowest energy value during the annealing process and to return this state from `hnetSimAnn`. However, we did not implement this straightforward idea but leave it as another exercise for the reader.

Figure 4 visualizes (parts of) the evolution of the macroscopic state of a Hopfield net for solving our specific Sudoku. In the exemplary run, the simulated annealing process caused the network to converge to the state of globally minimal energy and thus to find the sought after solution. Indeed, if we rearrange the state $\boldsymbol{s}$ in the last row of Fig. 4 on a 4×4 grid, it reads

```
-+-- +--- --+- ---+
--+- ---+ +--- -+--
---+ --+- -+-- +---
+--- -+-- ---+ --+-
```

which, when decoded into the numbers 1 to 4, corresponds to

```
2 1 3 4
3 4 1 2
4 3 2 1
1 2 4 3
```

and thus represents the solution on the right of Fig. 1(a).

In general, i.e. for arbitrary Sudokus, we observe the following: Using the parameters provided in the above implementations (and investing no efforts into manual tweaking), Boltzmann machines that update based on simulated annealing are very well able to solve 4×4 Sudokus. In about 90% of all runs, our networks converged to the correct solution. While this sounds less than perfect, we emphasize that our method is an inherently stochastic approach towards constraint satisfaction. Taking this into account, one might argue that a success rate of 9 out of 10 actually constitutes a surprisingly good performance.

The approach also works well for 9×9 Sudokus. Here, however, the default parameters of `hnetInitParameters` and the parameter `rmax` of `hnetSimAnn` require more careful (manual) tuning.

In either case, i.e. for 4×4 or 9×9 Sudokus, the runtime behavior of the approach is less than stellar and we remark that there are much faster algorithms for Sudokus which we will discuss in upcoming notes.

4 CONCLUSION

This note demonstrated that stochastic Hopfield nets which apply simulated annealing to update their neurons can solve Sudoku puzzles. While the theory behind this result might have been a bit involved, its translation into *NumPy* code was not.

To some, it may seem miraculous that a comparatively simple neurocomputing paradigm can successfully tackle a problem that seemingly requires higher cognitive capabilities. Others may now wonder whether Hopfield nets are a strong contender for Sudoku? Here, the sobering answer is: No, not really, mainly because they are too slow. While there exist much more efficient annealing methods for Hopfield nets [10], there also exist better kinds of algorithms. These consider Sudoku as an instance of the exact cover problem and use corresponding techniques. That is, although exact cover is an instance of Karp's 21 NP-complete problems [17], Knuth's algorithm X or dancing links algorithm [18] solves it very efficiently and is considered to be the fastest Sudoku algorithm to date.

However, the fact that Hopfield nets can solve Sudoku is still interesting because, if a problem can be (re)written as a problem that can be solved by Hopfield nets, it can also be solved using special purpose digital annealers [11, 22, 23] or (adiabatic) quantum computing [6, 16]. Indeed, one of the first practical problems ever to be solved on a D-Wave quantum annealer was a 4×4 Sudoku [12]. Funnily enough, the blogosphere at that time took this as reason for ridicule because Sudoku was not seen as a problem that required new computing paradigms. Yet, (computer) games have traditionally been one of the major drivers of progress in computer science and many of the techniques we take for granted nowadays were developed in this context.

Speaking of which, our topic in this note was a topic rooted in the area of game AI. In addition to Sudoku and similar combinatorial puzzles, there many other problems in this arena where Hopfield nets can play a role [5, 7, 27, 29]. Some of these, too, will be addressed in upcoming notes in our coding nuggets series.

ACKNOWLEDGMENTS

This material was produced within the Competence Center for Machine Learning Rhine-Ruhr (**ML2R**) which is funded by the Federal Ministry of Education and Research of Germany (grant no. 01IS18038C). The authors gratefully acknowledge this support.

REFERENCES

[1] D.H. Ackley, G.E. Hinton, and T.J. Sejnowski. 1985. A Learning Algorithm for Boltzmann Machine. *Cognitive Science* 9, 1 (1985).
[2] P. Babu, K. Pelckmans, P. Stoica, and J. Li. 2010. Linear Systems, Sparse Solutions, and Sudoku. *IEEE Signal Processing Letters* 17, 1 (2010).
[3] A. Bartlett, T.P. Chartier, A.N. Langville, and T.D. Rankin. 2008. An Integer Programming Model for the Sudoku Problem. *J. of Online Mathematics and Its Applications* 8, May (2008).
[4] C. Bauckhage, F. Beaumont, and S. Müller. 2021. *ML2R Coding Nuggets: Hopfield Nets for Max-Sum Diversification.* Technical Report. MLAI, University of Bonn.
[5] C. Bauckhage, A. Drachen, and R. Sifa. 2015. Clustering Game Behavior Data. *IEEE Trans. on Computational Intelligence and AI in Games* 7, 3 (2015).
[6] C. Bauckhage, R. Sanchez, and R. Sifa. 2020. Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In *Proc. IJCNN.* IEEE.
[7] C. Bauckhage, R. Sifa, A. Drachen, C. Thurau, and F. Hadiji. 2014. Beyond Heatmaps: Spatio-Temporal Clustering using Behavior-Based Partitioning of Game Levels. In *Proc. CIG.* IEEE.
[8] C. Bauckhage and P. Welke. 2021. *ML2R Coding Nuggets: Hopfield Nets for Sorting.* Technical Report. MLAI, University of Bonn.
[9] C. Bauckhage and P. Welke. 2021. *ML2R Coding Nuggets: Sorting as a QUBO.* Technical Report. MLAI, University of Bonn.
[10] G. Bilbro, R. Mann, T.K. Miller, W.E. Snyder, D.E. Van den Bout, and M. White. 1988. Optimization by Mean Field Annealing. In *Proc. NIPS.*
[11] J. Boyd. 2018. Silicon Chip Delivers Quantum Speeds. *IEEE Spectrum* 55, 7 (2018).
[12] G. Brumfiel. 2007. Quantum Computing at 16 qubits. *Nature* (2007).
[13] J. Gunther and T. Moon. 2012. Entropy Minimization for Solving Sudoku. *IEEE Trans. on Signal Processing* 60, 1 (2012).
[14] J.J. Hopfield. 1982. Neural Networks and Physical Systems with Collective Computational Abilities. *PNAS* 79, 8 (1982).
[15] J.J. Hopfield. 2008. Searching for Memories, Sudoku, Implicit Check Bits, and the Iterative Use of Not-Always-Correct Rapid Neural Computation. *Neural Computation* 20, 5 (2008).
[16] M. Johnson and et al. 2011. Quantum Annealing with Manufactured Spins. *Nature* 473, 7346 (2011).
[17] R.M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computation*, R.E. Miller, J.W. Thatcher, and J.D. Bohlinger (Eds.). Springer.
[18] D.E. Knuth. 2000. Dancing Links. In *Millennial Perspectives in Computer Science*, J. Davies, B. Roscoe, and J. Woodcock (Eds.). Red Globe Press.
[19] R. Lewis. 2007. Metaheuristics Can Solve Sudoku Puzzles. *J. of Heuristics* 13, 4 (2007).
[20] T. Mantere and J. Koljonen. 2007. Solving, Rating and Generating Sudoku Puzzles with GA. In *Proc. Int. Conf. on Evolutionary Computation.* IEEE.
[21] A. Moraglio, J. Togelius, and S. Lucas. 2018. Product Geometric Crossover for the Sudoku Puzzle. In *Proc. Int. Conf. on Evolutionary Computation.* IEEE.
[22] S. Mücke, N. Piatkowski, and K. Morik. 2019. Hardware Acceleration of Machine Learning Beyond Linear Algebra. In *Proc. ECML/PKDD.*
[23] S. Mücke, N. Piatkowski, and K. Morik. 2019. Learning Bit by Bit: Extracting the Essence of Machine Learning. In *Proc. KDML-LWDA.*
[24] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).
[25] R.B. Palm, U. Paquet, and O. Winther. 2018. Recurrent Relational Networks. In *Proc. NeurIPS.*
[26] G. Santos-Garcia and M. Palomino. 2007. Solving Sudoku Puzzles with Rewriting Rules. *Electronic Notes in Theoretical Computer Science* 176, 4 (2007).
[27] R. Sifa, A. Drachen, and C. Bauckhage. 2015. Large-scale Cross-game Player Behavior Analysis on Steam. In *Proc. AIIDE.* AAAI.
[28] H. Simonis. 2005. Sudoku as a Constraint Problem. In *Int. Workshop on Modelling and Reformulating Constraint Satisfaction Problems.*
[29] C. Thurau, T. Paczian, and C. Bauckhage. 2005. Is Bayesian Imitation Learning the Route to Believable Gamebots?. In *Proc. GAME-ON North America.*