ML2R Coding Nuggets
Hopfield Nets for Sorting

Christian Bauckhage*
Machine Learning Rhine-Ruhr
University of Bonn
Bonn, Germany

Nico Piatkowski†
Machine Learning Rhine-Ruhr
Fraunhofer IAIS
St. Augustin, Germany

ABSTRACT

We show how to use Hopfield networks for sorting. We first derive a corresponding energy function, then present an efficient algorithm for its minimization, and finally implement our ideas in *NumPy*.

1 INTRODUCTION

We continue our study of Hopfield networks and show that they can sort the elements $x_l \in \mathbb{R}$ of an n-dimensional vector

$$\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^\mathsf{T} \tag{1}$$

In [2], we already saw that sorting can be cast as a **quadratic unconstrained binary optimization (QUBO) problem**. Crucial steps and ingredients of this insight are summarized in appendix A. Here, we simply recall that we considered binary decision variables $\boldsymbol{z} \in \{0,1\}^{n^2}$ and derived a matrix $\boldsymbol{R} \in \mathbb{R}^{n^2 \times n^2}$ and vector $\boldsymbol{r} \in \mathbb{R}^{n^2}$ which allowed us to think of sorting as the task of solving

$$\boldsymbol{z}^* = \underset{\boldsymbol{z} \in \{0,1\}^{n^2}}{\operatorname{argmin}} \; \boldsymbol{z}^\mathsf{T} \boldsymbol{R} \boldsymbol{z} - \boldsymbol{r}^\mathsf{T} \boldsymbol{z} \tag{2}$$

This is because, if we could solve this QUBO for $\boldsymbol{z}^*$, we could turn that solution into an $n \times n$ permutation matrix $\boldsymbol{P}^\mathsf{T} = \operatorname{mat}(\boldsymbol{z}^*)$ which would then allow us to compute a sorted version $\boldsymbol{y} = \boldsymbol{P}\boldsymbol{x}$ of $\boldsymbol{x}$.

Given this QUBO formulation of the sorting problem, we also presented a greedy algorithm for its solution [2]. This algorithm works well but is computationally inefficient. However, we promised that the QUBO in (2) can be further rewritten and then be solved by means of Hopfield nets which update in an informed and thus efficient manner [13].

In this note, we fulfill these promises. Based on the QUBO in (2), we first derive expressions for the weights and bias parameters of a sorting Hopfield network (section 2.1). Since sorting is not a difficult problem, the problem of minimizing the energy of a sorting Hopfield net is not difficult either. Indeed, we will see that there is a simple steepest descent algorithm for this purpose (section 2.2). Finally, we implement all our ideas in plain vanilla *NumPy* (section 3).

Throughout, we assume that our readers know about the theory behind Hopfield networks and are familiar with the corresponding terminology. Those who would like to experiment with our code should also be familiar with *NumPy* [12] and need to

```
import numpy as np
```

* 0000-0001-6615-2128
† 0000-0002-6334-8042

2 THEORY

A Hopfield network is a recurrent neural net of m interconnected neurons $s_1, s_2, \ldots, s_m$ each of which is a bipolar threshold unit

$$s_i = \begin{cases} +1 & \text{if } \boldsymbol{w}_i^\mathsf{T} \boldsymbol{s} - \theta_i \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

$$= \operatorname{sign}\left(\boldsymbol{w}_i^\mathsf{T} \boldsymbol{s} - \theta_i\right) \tag{3}$$

where the bipolar vector $\boldsymbol{s} = [s_1, \ldots, s_m]^\mathsf{T} \in \{-1, +1\}^m$ denotes the overall state of the network.

If the weight matrix $\boldsymbol{W}$ of a Hopfield net is symmetric and has a diagonal of all zeros and *if* the neurons of the network update their individual states one at a time, *then* the Hopfield energy

$$E(\boldsymbol{s}) = -\tfrac{1}{2} \boldsymbol{s}^\mathsf{T} \boldsymbol{W} \boldsymbol{s} + \boldsymbol{\theta}^\mathsf{T} \boldsymbol{s} \tag{4}$$

can never increase. Indeed, since

$$\nabla E(\boldsymbol{s}) = -\boldsymbol{W} \boldsymbol{s} + \boldsymbol{\theta} \tag{5}$$

we realize that the updates in (3) amount to $s_i = \operatorname{sign}\left(-\nabla E\right)_i$ and thus perform subgradient descent on (4). As there are "only" 2^m states the network can be in, it will reach a (local) energy minimum after finitely many updates [6].

This behavior of Hopfield nets can be used for problem solving if the problem at hand can be written as a QUBO. The idea is to run a Hopfield net whose weight- and bias parameters $\boldsymbol{W}$ and $\boldsymbol{\theta}$ are chosen such that minimum energy states(s)

$$\boldsymbol{s}^* = \underset{\boldsymbol{s} \in \{-1, +1\}^m}{\operatorname{argmin}} \; -\tfrac{1}{2} \boldsymbol{s}^\mathsf{T} \boldsymbol{W} \boldsymbol{s} + \boldsymbol{\theta}^\mathsf{T} \boldsymbol{s} \tag{6}$$

of the network encode the sought after solution(s).

The major challenge in this regard is to (re)write problems in terms of QUBOs. If this is possible, it is usually easy to determine a corresponding Hopfield net. We will see this in the following where we turn the objective function of the sorting QUBO in (2) into an energy function that can be minimized by a Hopfield network.

2.1 Hopfield Nets for Sorting

Note that the minimization problems in (2) and (6) are both QUBOs. However, while the former minimizes over *binary* vectors $\boldsymbol{z}$ with entries $z_i \in \{0,1\}$, the latter minimizes over *bipolar* vectors $\boldsymbol{s}$ whose entries are $s_i \in \{-1, +1\}$.

To turn the sorting QUBO in (2) into a QUBO that can be solved by a Hopfield net, we therefore recall that binary and bipolar vectors are isomormic in the sense that

$$\boldsymbol{s} = 2\boldsymbol{z} - \boldsymbol{1} \;\Leftrightarrow\; \boldsymbol{z} = \tfrac{1}{2}\left(\boldsymbol{s} + \boldsymbol{1}\right) \tag{7}$$

If we thus plug $z = (s+1)/2$ into the objective function of our sorting QUBO, we obtain

$$\begin{aligned} z^\mathsf{T} R\, z - r^\mathsf{T} z &= \tfrac{1}{4}\,(s+1)^\mathsf{T} R\,(s+1) - \tfrac{1}{2}\, r^\mathsf{T}(s+1) \\ &= \tfrac{1}{4}\, s^\mathsf{T} R\, s + \tfrac{1}{2}\,(R\,\mathbf{1} - r)^\mathsf{T} s + const \\ &\equiv s^\mathsf{T} Q\, s + q^\mathsf{T} s + const \end{aligned} \tag{8}$$

In other words, we find that the sorting problem can also be cast as the problem of solving

$$s^* = \underset{s \in \{-1,+1\}^{n^2}}{\operatorname{argmin}} \; s^\mathsf{T} Q s + q^\mathsf{T} s \tag{9}$$

where

$$Q = \tfrac{1}{4}\, R \tag{10}$$

$$q = \tfrac{1}{2}\,(R\,\mathbf{1} - r) \tag{11}$$

This new version of the sorting QUBO already resembles the Hopfield energy minimization problem in (6). However, there are two differences we need to pay attention to: First of all, the coupling matrix Q in (9) is *not* hollow, i.e. it has diagonal entries different from 0. Second of all, the sign patterns and scaling of the objectives in (6) and (9) disagree.

To address the former, we refer to appendix A and note that the all the diagonal elements of $Q = \frac{1}{4}\, R$ are given by $\lambda = \frac{1}{4}\,(\lambda_r + \lambda_c)$. This allows to write the quadratic part of the objective in (9) as

$$\begin{aligned} s^\mathsf{T} Q\, s &= s^\mathsf{T}(Q - \lambda\, I)\, s + \lambda\, s^\mathsf{T} I\, s \\ &= s^\mathsf{T}(Q - \lambda\, I)\, s + \lambda\, s^\mathsf{T} s \\ &= s^\mathsf{T}(Q - \lambda\, I)\, s + \lambda\, n^2 \\ &\equiv s^\mathsf{T} Q' s + const \end{aligned} \tag{12}$$

where we used the fact that $s^\mathsf{T} s = \dim(s)$ for any bipolar vector s. In other words, the QUBO in (9) can alternatively be written as

$$s^* = \underset{s \in \{-1,+1\}^{n^2}}{\operatorname{argmin}} \; s^\mathsf{T} Q' s + q^\mathsf{T} s \tag{13}$$

where

$$Q' = \tfrac{1}{4}\left(R - (\lambda_r + \lambda_c)\, I\right) \tag{14}$$

is a hollow matrix whose diagonal entries are all zero.

Yet, the sign patterns and scaling of (13) still deviate from (6). However, we may simply define

$$W = -2\, Q' = -\tfrac{1}{2}\left(R - (\lambda_r + \lambda_c)\, I\right) \tag{15}$$

$$\theta = q = \tfrac{1}{2}\,(R\,\mathbf{1} - r) \tag{16}$$

in order to (re)write the QUBO in (13) as

$$s^* = \underset{s \in \{-1,+1\}^{n^2}}{\operatorname{argmin}} \; -\tfrac{1}{2}\, s^\mathsf{T} W s + \theta^\mathsf{T} s \tag{17}$$

At this point, we have succeeded and turned the problem of sorting the entries of a vector $x \in \mathbb{R}^n$ into an energy minimization problem that can be tackled by a Hopfield net of $m = n^2$ neurons.

From our discussion in [2] we know that, once a minimum energy state s^* has been determined, we can compute $z^* = (s^* + \mathbf{1})/2$ to obtain a permutation matrix $P^\mathsf{T} = \mathrm{mat}(z^*)$ and therefore the sorted version $y = Px$ of x.

The remaining open question is thus if Hopfield nets can sort (more or less) efficiently? Next, we will see that, yes, they can!

2.2 Running Hopfield Nets for Sorting

Every first year computer science student can attest that sorting is not a difficult problem. This translates to sorting with Hopfield networks in the sense that minima of their energy functions are easy to find. Indeed, there is a simple steepest descent algorithm that causes sorting Hopfield nets to quickly converge to a stable state of minimum energy.

Let s_t and s_{t+1} be two consecutive states during the temporal evolution of a Hopfield net and assume that neuron s_i has updated its activation at time t. It is well known that the energy difference $\Delta E = E(s_{t+1}) - E(s_t)$ in this situation amounts to

$$\Delta E = 2 \cdot (s_t)_i \cdot \left(w_i^\mathsf{T} s_t - \theta_i\right) \leq 0 \tag{18}$$

This allows us to determine which neuron should update its activation in iteration t so as to maximally decrease the network's current energy $E(s_t)$. Letting $\odot$ denote the Hadamard product, we simply could compute a vector

$$\Delta e = 2 \cdot s_t \odot \left(W s_t - \theta\right) \tag{19}$$

of expected energy decrements, determine the index u of its most negative entry

$$u = \underset{i}{\operatorname{argmin}}\; \Delta e_i \tag{20}$$

and then update neuron s_u to obtain a new global state

$$(s_{t+1})_i = \begin{cases} (s_t)_u \cdot \operatorname{sign}(\Delta e_u) & \text{if } i = u \\ (s_t)_i & \text{otherwise} \end{cases} \tag{21}$$

When using this mechanism to guide the evolution of sorting Hopfield net, we would initialize its global state to $s_0 = -\mathbf{1}$ where all neurons are inactive. Then, the update rule in (21) would only cause selected neurons to switch from -1 to $+1$.

This is essentially the same idea as in [2]. There, however, we had to evaluate energies $E(z)$ of binary vectors z which required efforts of $O\left(n^{2^3}\right)$. Now that we consider Hopfield nets and bipolar state vectors s, it is sufficient to evaluate gradients $-\nabla E(s) = Ws - \theta$ which only requires $O\left(n^{2^2}\right)$.

3 PRACTICE

In this section, we discuss how to practically implement Hopfield nets that sort the entries of a vector $x \in \mathbb{R}^n$.

To work with a specific example, we first create a vector of, say, $n = 5$ entries which we represent as a one-dimensional *NumPy* array `vecX`. To tie in with our earlier notes [2, 3], we once again consider

```
vecX = np.array([46, 52, 12, 10, 51])
```

Given x, we initialize the weight matrix W and bias vector θ of a corresponding sorting Hopfield network. To this end, we use function `hnetInitParameters` in Listing 1 and call

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

NOTE: Lines 1–18 of `hnetInitParameters` merely repeat the computations of function `initializeSortQUBO` which we discussed in [2] so that we will not discuss them again. The additional parts in lines 20–24 simply implement equations (10), (11), (15), and (16).

Listing 1: setting up the parameters of a sorting Hopfield net

```
1  def hnetInitParameters(vecX, lr=None, lc=None):
2      n = len(vecX)
3
4      vecN = np.arange(n) + 1
5
6      matI = np.eye(n)
7      vec1 = np.ones(n)
8
9      matN  = np.kron(matI, vecN)
10     matCc = np.kron(matI, vec1)
11     matCr = np.kron(vec1, matI)
12
13     if lr is None or lc is None:
14         vecX = vecX / np.sum(vecX)
15         lr = lc = n
16
17     matR = lr * matCr.T @ matCr + lc * matCc.T @ matCc
18     vecR = vecX @ matN + 2 * vec1 @ (lr * matCr + lc * matCc)
19
20     matQ = 0.25 * matR
21     vecQ = 0.50 * (matR @ np.ones(n**2) - vecR)
22
23     matW = -2 * matQ; np.fill_diagonal(matW, 0)
24     vecT = vecQ
25
26     return matW, vecT
```

Having initialized the parameters `matW` and `vecT` of our sorting Hopfield net, we next need to initialize its state vector. As discussed in section 2.2 we choose $\boldsymbol{s} = -\mathbf{1}$ and thus use

```
n    = len(vecX)
vecS = -np.ones(n**2)
```

Now, after all these preparations, we are good to go and may call

```
vecS = hnetRunGreedy(vecS, matW, vecT, tmax=10)
```

to solve our sorting problem. This applies unction `hnetRunGreedy` in Listing 2 which implements the greedy update mechanism we discussed above.

The first three parameters of `hnetRunGreedy` are self-explanatory. Parameter `tmax` indicates the number of update iterations our Hopfield should perform and `verbose` is a flag variable which, when `True`, causes printing of status information in each iteration. To be specific, the printed information consists of the current update cycle t, the corresponding global state $\boldsymbol{s}_t$, and the corresponding energy $E(\boldsymbol{s}_t)$ of the network. Printing happens in lines 7–10 and does not merit discussion. An exemplary output is shown in Fig. 1

Lines 12–15 of `hnetRunGreedy` are *NumPy* implementations of equations (19)–(21). Here, it is worth mentioning that line 15 involves a custom made sign function `signum` which is defined in lines 1 and 2 of Listing 2. We use `signum` instead of the *NumPy* function `np.sign` because the latter implements

$$\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{if } x > 0 \end{cases}$$

However, for Hopfield nets to work properly, we must insists on

$$\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ +1 & \text{if } x \geq 0 \end{cases}$$

as computed by `signum`.

Finally, `hnetRunGreedy` returns a state vector $\boldsymbol{s}$ which represents the activation of the neurons of the network after `tmax` updates.

Listing 2: greedily running a sorting Hopfield net

```
1  def signum(x):
2      return np.where(x >= 0, +1, -1)
3
4
5  def hnetRunGreedy(vecS, matW, vecT, tmax=1000, verbose=True):
6      for t in range(tmax):
7          if verbose:
8              s = ''.join(['+' if x >= 0 else '-' for x in vecS])
9              E = -0.5 * vecS @ matW @ vecS + vecT @ vecS
10             print ('{:4d}  {}  {:+.1f}'.format(t, s, E))
11
12         dltE = vecS * (matW @ vecS - vecT)
13         updt = np.argmin(dltE)
14
15         vecS[updt] = vecS[updt] * signum(dltE[updt])
16
17     return vecS
```

t	$\boldsymbol{s}_t$	$E(\boldsymbol{s}_t)$
0	−−−−−−−−−−−−−−−−−−−−−−−−−	−117.5
1	−−−−−−−−−+−−−−−−−−−−−−−−−	−129.0
2	−−−−−−−−−+−−−−−−−−−−−−−+−	−140.2
3	−−+−−−−−−+−−−−−−−−−−−−−+−	−151.0
4	−−+−−−−−−+−+−−−−−−−−−−−+−	−161.2
5	−−+−−−−−−+−+−−−+−−−−−−−+−	−171.2
6	−−+−−−−−−+−+−−−+−−−−−−−+−	−171.2
7	−−+−−−−−−+−+−−−+−−−−−−−+−	−171.2
8	−−+−−−−−−+−+−−−+−−−−−−−+−	−171.2
9	−−+−−−−−−+−+−−−+−−−−−−−+−	−171.2

Figure 1: Visualization of the temporal evolution of the state and energy of a Hopfield network of $m = n^2$ neurons that solves the problem of sorting the entries of a vector $\boldsymbol{x} \in \mathbb{R}^n$. In this example, $n = 5$ and all $m = 25$ neurons are initially inactive. Over time t, the state $\boldsymbol{s}_t$ of the network changes, i.e. several of its neurons greedily switch from inactive (−) to active (+), and the energy $E(\boldsymbol{s}_t)$ decreases. This process quickly converges to a stable state which does indeed encode the solution to the sorting problem at hand.

All that is left to truly solve our sorting problem is to turn this vector $\boldsymbol{s}$ into the required permutation matrix $\boldsymbol{P}$. This can, for instance, be accomplished using

```
matP = np.where(vecS>0, 1, 0).reshape(n,n).T
```

For our running example, the resulting array `matP` turns out to be

```
[[0. 0. 0. 1. 0.]
 [0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1.]
 [0. 1. 0. 0. 0.]]
```

And, to verify that it indeed solves our exemplary problem, we use

```
print ('vecX = ', vecX)
print ('vecY = ', matP @ vecX)
```

which yields

```
vecX =  [46 52 12 10 51]
vecY =  [10 12 46 51 52]
```

Success! We did run a Hopfield network to obtain a sorted version $\boldsymbol{y} = \boldsymbol{P}\boldsymbol{x}$ of an unordered vector $\boldsymbol{x}$.

4 CONCLUSION

This note demonstrated that Hopfield networks can sort. Based on earlier abstract formulations of the sorting problem [2, 3], we derived expressions for the weight matrix and bias vector of a sorting Hopfield net and discussed a greedy energy minimization procedure for fast convergence to a stable state.

A practical example showed our ideas to work well. Indeed, our Hopfield networks are much more efficient than the related approaches in [2, 3]. Readers are encouraged to verify this for themselves and "Hopfield sort" a vector $\boldsymbol{x}$ of size, say, $n = 100$. Moreover, our sorting Hopfield nets also perform more reliable than (much older) previous solutions [1] which are known to struggle with larger problem sizes.

Of course, Hopfield networks are still no match for specialized sorting algorithms such as *quicksort*. Nevertheless, it is interesting to see that the simplest possible neurocomputing techniques can sort at all. After all, a Hopfield network consists of mere threshold units which hardly anybody ever associates with tasks other than binary pattern recognition.

Still, our Hopfield network point of view on sorting may soon become more relevant. This is because it suggests how to estimate permutation matrices on adiabatic quantum computers [7] or on digital annealers [4, 9, 10]. Once hardware platforms such as these reach the technical maturity to deal with massive problem sizes, the approach considered in this note could be applied to practically more demanding permutation problems [5, 8, 11, 14].

A APPENDIX

This appendix briefly summarizes our earlier discussions which led to the sorting QUBO in (2).

In [2, 3], we saw that to sort the entries of $\boldsymbol{x} \in \mathbb{R}^n$ is to search for an $n \times n$ permutation matrix $\boldsymbol{P}$ such that that the entries of $\boldsymbol{y} = \boldsymbol{P}\boldsymbol{x}$ obey $y_i \leq y_{i+1}$. Letting $\mathbf{1} \in \mathbb{R}^n$ be the vector of all ones and assuming a specific auxiliary vector

$$\boldsymbol{n} = [1, 2, \ldots, n]^\mathsf{T} \tag{22}$$

the problem of finding such a permutation matrix can be cast as an equality constrained integer programming problem, namely

$$\begin{aligned} \boldsymbol{P} = \underset{Z \in \{0,1\}^{n \times n}}{\text{argmin}} \quad & -\boldsymbol{x}^\mathsf{T} Z^\mathsf{T} \boldsymbol{n} \\ \text{s.t.} \quad & Z\mathbf{1} = \mathbf{1} \\ & Z^\mathsf{T}\mathbf{1} = \mathbf{1} \end{aligned} \tag{23}$$

This problem over matrices $Z \in \{0,1\}^{n \times n}$ can also be written as a problem over vectors $\boldsymbol{z} = \text{vec}(Z) \in \{0,1\}^{n^2}$ because one can show that $Z^\mathsf{T}\boldsymbol{n} = \boldsymbol{N}\boldsymbol{z}$, $Z\mathbf{1} = C_r\,\boldsymbol{z}$, and $Z^\mathsf{T}\mathbf{1} = C_c\,\boldsymbol{z}$ which requires the following $n \times n^2$ matrices

$$\boldsymbol{N} = \boldsymbol{I} \otimes \boldsymbol{n}^\mathsf{T} \tag{24}$$

$$C_r = \mathbf{1}^\mathsf{T} \otimes \boldsymbol{I} \tag{25}$$

$$C_c = \boldsymbol{I} \otimes \mathbf{1}^\mathsf{T} \tag{26}$$

where $\boldsymbol{I}$ and $\otimes$ are the $n \times n$ identity matrix and the Kronecker product. The Lagrangian of the correspondingly rewritten problem amounts to

$$L(\boldsymbol{z}, \lambda_r, \lambda_c) = \boldsymbol{z}^\mathsf{T}\boldsymbol{R}\,\boldsymbol{z} - \boldsymbol{r}^\mathsf{T}\boldsymbol{z} \tag{27}$$

where the matrix $\boldsymbol{R} \in \mathbb{R}^{n^2 \times n^2}$ and vector $\boldsymbol{r} \in \mathbb{R}^{n^2}$ are given by

$$\boldsymbol{R} = \lambda_r\, C_r^\mathsf{T} C_r + \lambda_c\, C_c^\mathsf{T} C_c \tag{28}$$

$$\boldsymbol{r} = \boldsymbol{N}^\mathsf{T}\boldsymbol{x} + 2\left(\lambda_r\, C_r + \lambda_c\, C_c\right)^\mathsf{T}\mathbf{1} \tag{29}$$

The two scalars λ_r and λ_c are Lagrange multipliers which we treat as parameters that have to be set manually.

All of this establishes that the problem in (23) can also be cast as a quadratic unconstrained binary optimization problem, namely

$$\boldsymbol{z}^* = \underset{\boldsymbol{z} \in \{0,1\}^{n^2}}{\text{argmin}} \quad \boldsymbol{z}^\mathsf{T}\boldsymbol{R}\,\boldsymbol{z} - \boldsymbol{r}^\mathsf{T}\boldsymbol{z} \tag{30}$$

which we recognize as the QUBO in (2). If we could solve it for $\boldsymbol{z}^*$, the actually sought after permutation matrix could then be computed as $\boldsymbol{P}^\mathsf{T} = \text{mat}(\boldsymbol{z}^*)$ and would allow us to obtain the sorted version $\boldsymbol{y} = \boldsymbol{P}\boldsymbol{x}$ of $\boldsymbol{x}$.

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. 01|S18038C). The authors gratefully acknowledge this support.

REFERENCES

[1] M.A. Atkins. 1989. Sorting by Hopfield Net. In *Proc. IJCNN.*
[2] C. Bauckhage and P. Welke. 2021. *ML2R Coding Nuggets: Sorting as a QUBO.* Technical Report. MLAI, University of Bonn.
[3] C. Bauckhage and P. Welke. 2021. *ML2R Coding Nuggets: Sorting as Linear Programming.* Technical Report. MLAI, University of Bonn.
[4] J. Boyd. 2018. Silicon Chip Delivers Quantum Speeds. *IEEE Spectrum* 55, 7 (2018).
[5] G.D. Evangelidis and C. Bauckhage. 2013. Efficient Subframe Video Alignment Using Short Descriptors. *IEEE Trans. Pattern Analysis and Machine Intelligence* 35, 10 (2013).
[6] J. Hopfield. 1982. Neural Networks and Physical Systems with Collective Computational Abilities. *PNAS* 79, 8 (1982).
[7] M. Johnson and et al. 2011. Quantum Annealing with Manufactured Spins. *Nature* 473, 7346 (2011).
[8] J. Kunegis, D. Fay, and C. Bauckhage. 2010. Network Growth and the Spectral Evolution Model. In *Proc. CIKM.* ACM.
[9] S. Mücke, N. Piatkowski, and K. Morik. 2019. Hardware Acceleration of Machine Learning Beyond Linear Algebra. In *Proc. ECML/PKDD.*
[10] S. Mücke, N. Piatkowski, and K. Morik. 2019. Learning Bit by Bit: Extracting the Essence of Machine Learning. In *Proc. LWDA.*
[11] A. Nowak, S. Villar, A.S. Bandeira, and J. Bruna. 2017. Revised Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks. *arXiv:1706.07450 [stat.ML]* (2017).
[12] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).
[13] L. von Rueden, S. Mayer, K. Beckh, B. Georgiev, S. Giesselbach, R. Heese, B. Kirsch, J. Pfrommer, A. Pick, R. Ramamurthy, M. Walczak, J. Garcke, C. Bauckhage, and J. Schuecker. 2019. Informed Machine Learning – A Taxonomy and Survey of Integrating Knowledge into Learning Systems. *arXiv:1903.12394 [stat.ML]* (2019).
[14] M.M. Zavlanos and G. J. Pappas. 2008. A Dynamical Systems Approach to Weighted Graph Matching. *Automatica* 44, 11 (2008).