# ML2R Coding Nuggets
# Hopfield Nets for Subset Sum

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

## ABSTRACT

This note discusses a QUBO formulation of the subset sum problem. Cast as a QUBO, this combinatorial problem can be tackled using Hopfield nets. However, as subset sum is NP-complete, we cannot guarantee our Hopfield nets to always discover an optimal solution. Nevertheless, using multiple restarts, our simple *NumPy* implementation quickly and consistently finds valid solutions.

## 1 INTRODUCTION

If you are bored and don't know how to spend your time, then here is a task that should keep you busy for a while: The following is a **multiset** $\mathcal{X}$ of $n = 100$ numbers $x_i \in \mathbb{Z}$

$$\mathcal{X} = \{\ 265,\ 453,\ 311,\ 876,\ 158,\ 344,\ \ 65,\ 411,\ 366,\ 314,$$
$$200,\ 816,\ 305,\ 716,\ 892,\ 787,\ \ 45,\ 258,\ 967,\ 669,$$
$$525,\ 638,\ 351,\ 438,\ 839,\ 438,\ 732,\ 675,\ 429,\ 278,$$
$$175,\ 192,\ 408,\ 243,\ 733,\ 176,\ 111,\ 570,\ 332,\ 766,$$
$$925,\ 680,\ 305,\ 805,\ 167,\ 162,\ 442,\ 404,\ \ 14,\ 603,$$
$$279,\ 636,\ 927,\ 757,\ 780,\ 892,\ 178,\ 148,\ \ 11,\ 766,$$
$$272,\ 520,\ 317,\ 573,\ \ 89,\ 776,\ 389,\ 444,\ 429,\ 984,$$
$$296,\ \ 68,\ 415,\ 257,\ 205,\ 409,\ 664,\ 472,\ 888,\ \ 25,$$
$$641,\ 367,\ 791,\ 687,\ 344,\ 320,\ 936,\ 520,\ \ 36,\ 494,$$
$$719,\ 362,\ \ 78,\ 437,\ 841,\ 163,\ 869,\ 945,\ 918,\ 840\ \}$$

At least one subset of all these numbers sums to a *target value* of

$$t = 8364$$

Go ahead and find it …

Why are we so sure that this seemingly simple task will be neigh impossible for non-savants to solve without computer assistance? Because it is an instance of the **subset sum problem (SSP)** where we are given a multiset of integers and a target value and have to decide whether any subset of our integers sums to the target.

At first glance, this seems to be a contrived problem of little relevance. But in fact the SSP is of substantial practical importance. It arises, say, in areas such as financial accounting and regulatory reporting where analysts must figure out what may have gone wrong when numbers don't add up [5, 6, 19, 20]. The issue is that subset sum is known to be NP-complete [12] and thus difficult to solve even for only moderately large problem sizes.

In order to see why this is the case, we first recall that the power set $2^{\mathcal{X}} = \{\mathcal{Y} \mid \mathcal{Y} \subseteq \mathcal{X}\}$ of a finite (multi-)set $\mathcal{X}$ is the set of all subsets of $\mathcal{X}$ (including the empty set $\emptyset$ and $\mathcal{X}$ itself). We also remark that the notation $2^{\mathcal{X}}$ may seem strange but reflects an observation regarding finite sets: For any set $\mathcal{X}$ of finite size $|\mathcal{X}|$, the size of its power set $2^{\mathcal{X}}$ amounts to $\left|2^{\mathcal{X}}\right| = 2^{|\mathcal{X}|}$.

Next, we recall that NP-completeness of the SSP roughly means the following: Given enough time, a brute force algorithm could check all subsets and their sums and thus find a solution if one exists; the validity of this solution is easily verified.

The second part of this statement can be made clear immediately. For instance, if we claim that

$$\mathcal{Y} = \{\ 265,\ 411,\ 200,\ 438,\ 839,\ 438,\ 675,\ 766,$$
$$167,\ 404,\ 603,\ 296,\ 641,\ 367,\ 936,\ \ 78,\ 840\ \}$$

is a subset of our exemplary multiset $\mathcal{X}$ and that the $y_j \in \mathcal{Y}$ sum to $t = 8364$, then both these claims are easily verified to be true.[1]

To illustrate that the first part of this statement may be problematic, we next perform a little calculation: Our exemplary set $\mathcal{X}$ contains $n = 100$ numbers and thus has

$$2^{100} \approx 1.27 \cdot 10^{30}$$

subsets. Most of them will obviously not solve our problem. Yet, to be absolutely sure not to miss a potential solution, we would have to check them all. Let us therefore assume, we had a formidable computer that evaluates a staggering $10^{12}$ (a trillion) subsets per second. Running 24/7, this machine could check

$$10^{12} \cdot 60 \cdot 60 \cdot 24 \cdot 365 \approx 3.15 \cdot 10^{19}$$

subsets per year. Checking all $2^{100}$ possible subsets would then take

$$\frac{1.27 \cdot 10^{30}\ \left[check\right]}{3.15 \cdot 10^{19}\ \left[\frac{check}{year}\right]} \approx 4 \cdot 10^{10}\ \left[year\right] = 40\ \text{ billion years}$$

The age of the universe, on the other hand, is currently estimated to be about 14 billion years.

This sobering calculation reveals that it is generally hopeless to try to brute force SSPs. Even if we had a supercomputer orders of magnitude faster than our hypothetical machine,[2] we must keep in mind that our calculation only considered a measly $n = 100$ numbers. Just doubling this to $n = 200$ would require a brute force search over $2^{200} \approx 1.6 \cdot 10^{60}$ subsets which is beyond hopeless.

So, what gives? Well, there exist quite sophisticated and highly specialized algorithms,[3] but we may also try our luck with simple Hopfield nets. Hence, in section 2, we derive a **QUBO** formulation of the SSP and discuss how to solve it using Hopfield nets. In section 3, we put this theory to work and present corresponding *NumPy* code. Throughout, we assume that readers already know about Hopfield nets. Those who want to experiment with our code should be familiar with *NumPy* [16] as well and need to

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

[1] Simply use the search function of your PDF reader to verify our first claim and add the numbers in $\mathcal{Y}$ to verify our second claim. This should only take a couple of minutes.
[2] Current supercomputers can perform (slightly) more than an exaFLOP or $10^{18}$ floating point operations per second …
[3] https://en.wikipedia.org/wiki/Subset_sum_problem

## 2 THEORY

Our first goal in this section is to show that the subset sum problem can be understood as a QUBO of the following form

$$\boldsymbol{s}^* = \underset{\boldsymbol{s} \in \{\pm 1\}^n}{\operatorname{argmin}} \; \boldsymbol{s}^\top \boldsymbol{Q}\, \boldsymbol{s} + \boldsymbol{q}^\top \boldsymbol{s} \tag{1}$$

where the matrix $\boldsymbol{Q} \in \mathbb{R}^{n \times n}$ and vector $\boldsymbol{q} \in \mathbb{R}^n$ contain problem specific parameters. As we shall see, this will be rather easy.

Our second goal is to tie in the QUBO formulation of set cover with Hopfield nets for combinatorial problem solving [10, 11]. This part of our discussion will largely recap points we made in earlier notes (see, for instance, [1–3]).

### 2.1 Subset Sum as a QUBO

Expressing the subset sum problem as a QUBO is surprisingly easy, because we may think of it in terms ow two steps.

*Step 1:* solve

$$\mathcal{Y}^* = \underset{\mathcal{Y} \in 2^{\mathcal{X}}}{\operatorname{argmin}} \left( t - \sum_{x \in \mathcal{Y}} x \right)^2 \tag{2}$$

*Step 2:* verify if

$$t = \sum_{x \in \mathcal{Y}^*} x \tag{3}$$

The idea behind the (quadratic) objective function in (2) is rather obvious, isn't it? If we could find a subset $\mathcal{Y} \subseteq \mathcal{X}$ or, equivalently, an element $\mathcal{Y} \in 2^{\mathcal{X}}$ such that the squared difference between the sum of its elements $x \in \mathcal{Y}$ and the target value $t$ is minimal, i.e. 0, we would have solved our problem.

Then why do we need the second step? Well, the problem in (2) still poses a combinatorial optimization problem because its feasible set is a discrete **lattice**, namely the power set $2^{\mathcal{X}}$ of $\mathcal{X}$. This complicates things. Whereas quadratic minimization problems over continuous sets are guaranteed to have a unique global minimum, the discrete nature of power set lattices generally causes our objective function to have numerous sub-optimal local minima. Hence, any presently known optimization algorithm for solving (2) may get stuck in one of these local minima and we therefore need to verify (3) in order to be sure that we have found a valid solution. This, however, should not be reason for concern since this second step does not pose any significant computational challenge.

An upshot of the problem formulation in (2) is that it can also be expressed in terms of (indicator-)vectors. We may simply represent the given set $\mathcal{X}$ of $n$ numbers $x_i \in \mathbb{Z}$ as a vector

$$\boldsymbol{x} = \big[x_1, x_2, \ldots, x_n\big]^\top \tag{4}$$

and any subset $\mathcal{Y}$ of $\mathcal{X}$ as a binary indicator vector $\boldsymbol{z} \in \{0,1\}^n$ where

$$z_i = \begin{cases} 1 & \text{if } x_i \in \mathcal{Y} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Using these representations of the objects (sets and subsets) under consideration, the problem in (2) can be written as

$$\boldsymbol{z}^* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} \left( t - \sum_{i=1}^{n} x_i \cdot z_i \right)^2 \tag{6}$$

$$= \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} \; \big(t - \boldsymbol{x}^\top \boldsymbol{z}\big)^2 \tag{7}$$

Now, looking at this expression, we realize that we have already succeeded in our quest of finding a QUBO formulation of the SSP. To make this more explicit, we note that

$$\big(t - \boldsymbol{x}^\top \boldsymbol{z}\big)^2 = \boldsymbol{z}^\top \boldsymbol{x}\boldsymbol{x}^\top \boldsymbol{z} - 2\, t\, \boldsymbol{x}^\top \boldsymbol{z} + t^2 \tag{8}$$

where $t^2$ is a constant independent of $\boldsymbol{z}$.

But this means that we may equivalently write our optimization problem in (7) as

$$\boldsymbol{z}^* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} \; \boldsymbol{z}^\top \boldsymbol{x}\boldsymbol{x}^\top \boldsymbol{z} - 2\, t\, \boldsymbol{x}^\top \boldsymbol{z} \tag{9}$$

$$\equiv \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} \; \boldsymbol{z}^\top \boldsymbol{P}\, \boldsymbol{z} + \boldsymbol{p}^\top \boldsymbol{z} \tag{10}$$

where

$$\boldsymbol{P} = \boldsymbol{x}\boldsymbol{x}^\top \tag{11}$$

$$\boldsymbol{p} = -2\, t\, \boldsymbol{x} \tag{12}$$

Finally, to turn this QUBO over binary variables $\boldsymbol{z} \in \{0,1\}^n$ into a QUBO over bipolar variables $\boldsymbol{s} \in \{-1,+1\}^n$, we recall that binary and bipolar vectors are isomorphic in the sense that

$$\boldsymbol{s} = 2\,\boldsymbol{z} - \boldsymbol{1} \;\Leftrightarrow\; \boldsymbol{z} = \tfrac{1}{2}\big[\boldsymbol{s} + \boldsymbol{1}\big] \tag{13}$$

where $\boldsymbol{1} \in \mathbb{R}^n$ is the vector of all ones. Hence, plugging $\boldsymbol{z} = \frac{1}{2}[\boldsymbol{s}+\boldsymbol{1}]$ into the objective of our QUBO in (10), we obtain

$$\begin{aligned} \boldsymbol{z}^\top \boldsymbol{P}\, \boldsymbol{z} + \boldsymbol{p}^\top \boldsymbol{z} &= \tfrac{1}{4}\big[\boldsymbol{s} + \boldsymbol{1}\big]^\top \boldsymbol{P}\big[\boldsymbol{s} + \boldsymbol{1}\big] + \tfrac{1}{2}\boldsymbol{p}^\top\big[\boldsymbol{s} + \boldsymbol{1}\big] \\ &= \tfrac{1}{4}\boldsymbol{s}^\top \boldsymbol{P}\, \boldsymbol{s} + \tfrac{1}{2}\big[\boldsymbol{P}\,\boldsymbol{1} + \boldsymbol{p}\big]^\top \boldsymbol{s} + const \\ &\equiv \boldsymbol{s}^\top \boldsymbol{Q}\, \boldsymbol{s} + \boldsymbol{q}^\top \boldsymbol{s} + const \end{aligned} \tag{14}$$

We therefore find that the subset sum problem can also be understood as the problem of solving

$$\boldsymbol{s}^* = \underset{\boldsymbol{s} \in \{\pm 1\}^n}{\operatorname{argmin}} \; \boldsymbol{s}^\top \boldsymbol{Q}\, \boldsymbol{s} + \boldsymbol{q}^\top \boldsymbol{s} \tag{15}$$

where

$$\boldsymbol{Q} = \tfrac{1}{4}\boldsymbol{P} \tag{16}$$

$$\boldsymbol{q} = \tfrac{1}{2}\big[\boldsymbol{P}\,\boldsymbol{1} + \boldsymbol{p}\big] \tag{17}$$

That is it! We now have a QUBO formulation of the SSP and may run Hopfield nets to try to solve it. If we do this, we will obtain a bipolar vector $\boldsymbol{s}^* \in \{-1,+1\}^n$ which we turn into a binary vector $\boldsymbol{z}^* = \frac{1}{2}\big[\boldsymbol{s} + \boldsymbol{1}\big]$ which the allows us to test if $\boldsymbol{x}^\top \boldsymbol{z}^* = t$.

## 2.2 Hopfield Nets for Subset Sum

Recall that a Hopfield net is a recurrent neural network of $n$ neurons $s_1, \ldots, s_n$ which are bipolar threshold units. In other words, letting $\boldsymbol{w}_i \in \mathbb{R}^n$ and $\theta_i \in \mathbb{R}$ denote input weights and threshold value of neuron $s_i$ and $\boldsymbol{s} = [s_1, \ldots, s_n]^\top$ denote the current global state of a Hopfield net, neuron $s_i$ computes its activation or local state as

$$\begin{aligned} s_i &= \begin{cases} +1 & \text{if } \boldsymbol{w}_i^\top \boldsymbol{s} - \theta_i \geq 0 \\ -1 & \text{otherwise} \end{cases} \\ &= \operatorname{sign}\left(\boldsymbol{w}_i^\top \boldsymbol{s} - \theta_i\right) \end{aligned} \tag{18}$$

so that the global state of a Hopfield network is a bipolar vector $\boldsymbol{s} \in \{-1, +1\}^n$.

Recall further that we may gather all the parameters of a Hopfield net in weight matrix and a threshold vector

$$\boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1^\top \\ \vdots \\ \boldsymbol{w}_n^\top \end{bmatrix} \qquad \text{and} \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \tag{19}$$

and that the energy of a Hopfield net in state $\boldsymbol{s}$ is defined to be

$$E(\boldsymbol{s}) = -\tfrac{1}{2}\, \boldsymbol{s}^\top \boldsymbol{W}\, \boldsymbol{s} + \boldsymbol{\theta}^\top \boldsymbol{s} \tag{20}$$

Finally, recall the very crucial fact: *If* the weight matrix of a Hopfield net is symmetric ($\boldsymbol{W} = \boldsymbol{W}^\top$) and hollow ($\operatorname{diag}[\boldsymbol{W}] = \boldsymbol{0}$) and *if* its neurons update asynchronously (one at a time), *then* the energy of the network never increases. As there are "only" $2^n$ distinct global states it can be in, it necessarily converges to a state of (locally) minimum energy within finitely many updates [10].

This then suggests that Hopfield nets can (approximately) solve QUBOs. All we seemingly need to do is to let $\boldsymbol{W} = -2\,\boldsymbol{Q}$ and $\boldsymbol{\theta} = \boldsymbol{q}$ to obtain a Hopfield net whose energy function corresponds to the QUBO's minimization objective.

However, there remains subtlety we must attend to: Matrix $\boldsymbol{Q}$ in (15) may not be hollow but hollowness of the weight matrix $\boldsymbol{W}$ of a Hopfield net is required to be able to guarantee that its global state eventually converges to a (local) minimum of its energy function.

We therefore note that we can write matrix $\boldsymbol{Q}$ in (15) as a sum of a diagonal matrix $\boldsymbol{D}$ and an off-diagonal matrix $\boldsymbol{O}$. The former represents the diagonal entries of $\boldsymbol{Q}$ and the latter represents the off-diagonal entries of $\boldsymbol{Q}$. The quadratic term in (15) then reads

$$\boldsymbol{s}^\top \boldsymbol{Q}\, \boldsymbol{s} = \boldsymbol{s}^\top \left[\boldsymbol{D} + \boldsymbol{O}\right] \boldsymbol{s} = \boldsymbol{s}^\top \boldsymbol{D}\, \boldsymbol{s} + \boldsymbol{s}^\top \boldsymbol{O}\, \boldsymbol{s} \tag{21}$$

Next, we point out that the expression $\boldsymbol{s}^\top \boldsymbol{D}\, \boldsymbol{s}$ always amounts to

$$\boldsymbol{s}^\top \boldsymbol{D}\, \boldsymbol{s} = \sum_i \sum_i s_i\, D_{ij}\, s_i = \sum_i D_{jj}\, s_i^2 \tag{22}$$

because $\boldsymbol{D}$ is a diagonal matrix. Moreover, we will necessarily have $s_i^2 = 1$, because $s_i \in \{-1, +1\}$. Since this implies that the term

$$\boldsymbol{s}^\top \boldsymbol{D}\, \boldsymbol{s} = \sum_i D_{jj} = const \tag{23}$$

does not impact the solution of (15), we may let

$$\boldsymbol{W} = -2\left[\boldsymbol{Q} - \boldsymbol{D}\right] \tag{24}$$

$$\boldsymbol{\theta} = \boldsymbol{q} \tag{25}$$

to obtain a (valid) Hopfield net with hollow weight matrix.

**Listing 1: *NumPy* implementations of a target integer $t \in \mathbb{Z}$ and an $n = 100$ dimensional vector $\boldsymbol{x} \in \mathbb{Z}^n$**

```
trgt = 8364
vecX = np.array([265, 453, 311, 876, 158, 344,  65, 411, 366, 314,
                 200, 816, 305, 716, 892, 787,  45, 258, 967, 669,
                 525, 638, 351, 438, 839, 438, 732, 675, 429, 278,
                 175, 192, 408, 243, 733, 176, 111, 570, 332, 766,
                 925, 680, 305, 805, 167, 162, 442, 404,  14, 603,
                 279, 636, 927, 757, 780, 892, 178, 148,  11, 766,
                 272, 520, 317, 573,  89, 776, 389, 444, 429, 984,
                 296,  68, 415, 257, 205, 409, 664, 472, 888,  25,
                 641, 367, 791, 687, 344, 320, 936, 520,  36, 494,
                 719, 362,  78, 437, 841, 163, 869, 945, 918, 840])
```

Regarding the (asynchronous) update mechanism of Hopfield nets, we note that neuron $s_u$ which updates its state in round $t$ is typically chosen at random. While this mechanism will converge to a network state of (locally) minimum energy, it typically converges very slowly. We can do better than this.

Letting $\boldsymbol{s}_t$ and $\boldsymbol{s}_{t+1}$ denote two consecutive states during the evolution of a Hopfield net and assuming that neuron $s_u$ updated at time $t$, we know that the energy difference in this situation amounts to $\Delta E = E(\boldsymbol{s}_{t+1}) - E(\boldsymbol{s}_t) = 2 \cdot \left[\boldsymbol{s}_t\right]_u \cdot \left(\boldsymbol{w}_u^\top \boldsymbol{s}_t - \theta_u\right) \leq 0$.

We can exploit this to determine which neuron should update its activation in iteration $t$ so as to maximally decrease the network's current energy $E(\boldsymbol{s}_t)$. Letting $\odot$ denote the Hadamard product, we may compute a vector

$$\Delta \boldsymbol{e} = 2 \cdot \boldsymbol{s}_t \odot \left[\boldsymbol{W}\, \boldsymbol{s}_t - \boldsymbol{\theta}\right] \tag{26}$$

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

$$u = \operatorname*{argmin}_j \left[\Delta \boldsymbol{e}\right]_j \tag{27}$$

and then update neuron $s_u$ to obtain a new global state where

$$\left[\boldsymbol{s}_{t+1}\right]_i = \begin{cases} \left[\boldsymbol{s}_t\right]_u \cdot \operatorname{sign}\left[\Delta \boldsymbol{e}\right]_u & \text{if } j = u \\ \left[\boldsymbol{s}_t\right]_i & \text{otherwise} \end{cases} \tag{28}$$

This greedy selection of updating neurons obviously minimizes the network's energy but does so faster than a random selection scheme. Alas, greedy updates may get trapped more easily in local minima of the energy landscape. However, subset sum problems come with very rugged energy landscapes anyway so that random updates, too, are very likely to miss global minima. A valid coping strategy is thus to run a Hopfield net many times, each time from a different randomly chosen initial state and to keep track of the best solution found so far. Used within this meta-algorithm, greedy updates of the states of the network are again superior as they considerably reduce the duration of each run.

# 3 PRACTICE

In this section, we discuss how to implement the above ideas for subset sum Hopfield nets using *NumPy*. As a practical example to work with, we consider the SSP instance from the introduction.

Listing 1 shows our corresponding implementations of the target value $t$ as an integer `trgt` and the data vector $\boldsymbol{x}$ as a *NumPy* array `vecX`.

**Listing 2: setting parameters of a subset sum Hopfield net**

```
def hnet_init_params(vecX, trgt):
    vec1 = np.ones_like(vecX)
    matP = np.outer(vecX, vecX)
    vecP = 2 * trgt * vecX

    matQ = 0.25 *  matP
    vecQ = 0.50 * (matP @ vec1 - vecP)

    matW = -2 * matQ; np.fill_diagonal(matW, 0)
    vecT =      vecQ

    return matW, vecT
```

**Listing 3: greedy state updates for Hopfield nets**

```
def signum(x):
    return np.where(x >= 0, +1, -1)



def hnet_run_greedy(vecS, matW, vecT, tmax=100):
    for t in range(tmax):
        grad = matW @ vecS - vecT
        updt = np.argmin(vecS * grad)
        vecS[updt] = signum(grad[updt])

    return vecS
```

**Listing 4: (re-)running Hopfield nets for subset sum**

```
1  rmax = 100
2  solutions = []
3
4  for r in range(rmax):
5      vecS = 2 * rnd.binomial(n=1, p=0.5, size=len(vecX)) - 1
6      vecS = hnet_run_greedy(vecS, matW, vecT, tmax=len(vecX)//2)
7
8      vecZ = np.where(vecS>0, True, False)
9      ssum = np.sum(vecX[vecZ])
10
11     if ssum == trgt:
12         vecY = vecX[vecZ]
13         solutions.append(vecY)
```

Given these ingredients, we can compute arrays `matW` and `vecT` which represent the weight matrix $\boldsymbol{W}$ and the threshold vector $\boldsymbol{\theta}$ of a subset sum Hopfield net. To this end we call

```
matW, vecT = hnet_init_params(vecX, trgt)
```

using function `hnet_init_params` in Listing 2. Its parameters are self explanatory; its body is an almost verbatim *NumPy* implementation of equations (11)–(12), (16)–(17), and (24)–(25). However, we point out the use of `np.fill_diagonal` which is a simple (and correct!) way of ensuring that network's weight matrix is hollow.

Function `hnet_run_greedy` in Listing 3 povides code for greedy activation updates while running a Hopfield net. Its parameters are again self explanatory; its body is an almost verbatim *NumPy* implementation of equations (26), (27), and (28). As always, we note our use of the custom made sign function `signum` which we worked with several times before [1–3].

Having all required ingredients and mechanisms in place, we can now run the meta-algorithm for solving subset sum problems which we sketched above. Corresponding code is shown in Listing 4. Here, we opt to only perform $r_{\max} = 100$ runs (line 1). Line 2 initializes an empty list `solutions` in which we store valid results that may be found in these runs.

In each run, we first randomly initialize an array `vecS` which represents the initial bipolar global state vector of our Hopfield net (line 5). We then run `hnet_run_greedy` for a total of $n/2$ greedy update steps to obtain a final state $\boldsymbol{s}^*$ (line 6).

Lines 8 and 9 turn this bipolar vector $\boldsymbol{s}^*$ into a binary vector $\boldsymbol{z}^*$ and compute the sum $t^* = \boldsymbol{x}^\intercal \boldsymbol{z}^*$ which is stored in `ssum`.

The remaining lines check if `ssum` equal our target value `trgt`. Should this not be the case, the current run ended in a spurious local minimum of the network's energy function. Should it be the case, we have found a valied solution and append it to the list `solutions`.

Table 1 shows 14 distinct solutions we found by running the code in Listing 4 for 100 rounds. On a single Intel i5 CPU core (3Ghz), this only took fractions of a second.

For the fun of it, we also ran our code for 500, 000 rounds. This took just a few minutes and produced 45, 951 solution to our SSP. While this sounds like a lot, we put it into perspective by using our empirical findings to estimate how likely a random $\mathcal{Y} \subseteq \mathcal{X}$ where $|\mathcal{X}| = 100$ will solve our problem. Computing

$$\frac{4.5 \cdot 10^5}{1.27 \cdot 10^{30}} \approx 3.5 \cdot 10^{-25}$$

we find that it is exceedingly unlikely that random sampling will produce valid SSP solutions.

On the other hand, regarding our idea of repeatedly running greedily updated Hopfield nets from random initial states, our empirical results are fairly encouraging as they suggest that about 10% of all runs yield a solution. While this may not sound like much, it is actually a useful success rate given the difficulty of the problem.

## 4 SUMMARY AND OUTLOOK

In this note, we demonstrated that the combinatorial subset sum problem can be tackled using Hopfield nets.

To this end, we first derived a QUBO formulation of the problem which we then turned into a Hopfield energy minimization task. This is of practical interest, because subset sum problems frequently arise in, say, financial document analysis [5, 6, 8, 9, 19, 20].

We saw that, despite the difficulty of the problem, simple Hopfield nets can quickly and consistently identify solutions. In real world applications, our simple meta-algorithm of repeatedly running Hopfield nets can easily be distributed over several CPUs or GPUs [5] and also allows for implementations on very energy efficient platforms [7, 14, 15]. Moreover, since we saw that subset sum problems can be solved by running a Hopfield nets, we know that the can also be solved on (adiabatic) quantum computers [4]. While these have not yet reached the level of technological maturity required for industrial applications at scale, practical applications of quantum computing for computational intelligence are currently progressing quickly [13, 17, 18]. Should this development continue, we might see quantum subset set solvers deployed in practice soon.

However, the simple practical example we worked with in this note also showed that subset sum problems can have surprisingly many solutions. For analysts in industry, this is challenging because even an automated downstream analysis of tens of thousands of solutions may be difficult to conduct.

In an upcoming note, we will therefore consider constrained subset sum problems where we impose additional conditions on the solutions of interest.

**Table 1: 14 subset sum solutions found by running the code in Listing 4.**

| round | solution |
|---|---|
| 10 | $\mathcal{Y}$ = {311, 158, 344, 65, 366, 314, 45, 258, 351, 438, 429, 175, 192, 332, 162, 404, 178, 148, 11, 272, 389, 444, 429, 68, 415, 409, 344, 36, 362, 78, 437} |
| 27 | $\mathcal{Y}$ = {265, 366, 314, 258, 351, 438, 278, 192, 408, 176, 332, 305, 162, 442, 404, 148, 272, 389, 444, 429, 296, 257, 205, 409, 25, 362, 437} |
| 37 | $\mathcal{Y}$ = {453, 344, 200, 305, 45, 258, 525, 438, 438, 175, 192, 176, 111, 332, 167, 442, 404, 14, 279, 148, 272, 89, 444, 68, 205, 409, 472, 25, 494, 362, 78} |
| 57 | $\mathcal{Y}$ = {158, 411, 305, 45, 258, 438, 278, 175, 176, 332, 167, 162, 442, 14, 603, 279, 636, 148, 11, 573, 89, 389, 296, 68, 415, 205, 520, 36, 494, 78, 163} |
| 63 | $\mathcal{Y}$ = {265, 344, 65, 45, 258, 351, 429, 278, 175, 408, 243, 167, 279, 11, 272, 317, 389, 444, 429, 296, 257, 205, 409, 472, 367, 344, 320, 362, 163} |
| 68 | $\mathcal{Y}$ = {453, 65, 411, 366, 314, 200, 45, 258, 351, 438, 438, 429, 175, 408, 176, 167, 404, 279, 272, 389, 444, 296, 68, 257, 205, 25, 367, 344, 320} |
| 79 | $\mathcal{Y}$ = {265, 314, 200, 305, 258, 438, 438, 429, 278, 175, 243, 111, 332, 305, 162, 442, 14, 178, 272, 89, 389, 429, 296, 68, 25, 344, 520, 36, 494, 78, 437} |
| 84 | $\mathcal{Y}$ = {453, 158, 65, 411, 305, 45, 258, 351, 438, 438, 429, 408, 243, 111, 162, 404, 148, 11, 89, 389, 444, 429, 296, 68, 205, 409, 472, 25, 344, 320, 36} |
| 87 | $\mathcal{Y}$ = {265, 344, 411, 314, 45, 438, 438, 278, 408, 111, 167, 442, 404, 279, 272, 89, 429, 68, 205, 472, 367, 344, 320, 520, 494, 362, 78} |
| 88 | $\mathcal{Y}$ = {265, 453, 65, 411, 200, 45, 258, 525, 638, 192, 408, 243, 176, 111, 570, 162, 442, 404, 14, 11, 272, 444, 296, 415, 409, 25, 367, 344, 36, 163} |
| 94 | $\mathcal{Y}$ = {453, 311, 158, 65, 200, 45, 258, 525, 351, 429, 278, 192, 408, 111, 305, 162, 404, 14, 148, 11, 429, 296, 257, 205, 25, 367, 344, 320, 494, 362, 437} |
| 96 | $\mathcal{Y}$ = {453, 311, 158, 314, 258, 351, 438, 278, 175, 192, 408, 111, 332, 162, 404, 603, 178, 148, 573, 89, 444, 68, 257, 205, 409, 36, 494, 78, 437} |
| 97 | $\mathcal{Y}$ = {158, 366, 314, 200, 305, 45, 438, 429, 278, 175, 192, 408, 111, 305, 162, 442, 404, 14, 178, 11, 272, 317, 389, 444, 429, 296, 68, 205, 409, 437, 163} |
| 99 | $\mathcal{Y}$ = {311, 158, 65, 366, 314, 45, 258, 525, 438, 438, 429, 278, 175, 192, 408, 243, 111, 167, 162, 442, 14, 178, 272, 317, 444, 520, 494, 437, 163} |

Our correspondingly modified problem formulations will involve tunable parameters which allow for significantly reducing the number of solutions and thus for simplifying further analysis. Nevertheless, all these variants of the subset sum problem can again be solved using Hopfield nets and, given what we discussed here, it will be easy to see why and how this is.

## 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] C. Bauckhage. 2022. *ML2R Coding Nuggets: Hopfield Nets for Set Cover.* Technical Report. MLAI, University of Bonn.
[2] C. Bauckhage, F. Beaumont, and S. Müller. 2021. *ML2R Coding Nuggets: Hopfield Nets for Bipartition Clustering.* Technical Report. MLAI, University of Bonn.
[3] C. Bauckhage, F. Beaumont, and S. Müller. 2021. *ML2R Coding Nuggets: Hopfield Nets for Max-Sum Diversification.* Technical Report. MLAI, University of Bonn.
[4] C. Bauckhage, R. Sanchez, and R. Sifa. 2020. Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In *Proc. IJCNN.* IEEE.
[5] D. Biesner, R. Sifa, and C. Bauckhage. 2022. Solving Subset Sum Problems using Binary Optimization with Applications in Auditing and Financial Data Analysis. *TechRxiv* (2022).
[6] E. Brito, R. Sifa, C. Bauckhage, R. Loitz, U. Lohmeier, and C. Pünt. 2019. A Hybrid AI Tool to Extract Key Performance Indicators from Financial Reports for Benchmarking. In *Proc. Symposium on Document Engineering.* ACM.
[7] S. Buschjäger, L. Pfahler, J. Buss, K. Morik, and W. Rhode. 2021. On-Site Gamma-Hadron Separation with Deep Learning on FPGAs. In *Proc. ECML PKDD.*
[8] C.L. Chapman, L.P. Hillebrand, M.R. Stenzel, T. Deusser, D. Biesner, C. Bauckhage, and R. Sifa. 2022. Towards Generating Financial Reports from Tabular Data Using Transformers. In *Proc. Cross Domain Conf. for Machine Learning and Knowledge Extraction (CD-MAKE).*
[9] L.P. Hillebrand, T. Deußer, T. Dilmaghani Khameneh, B. Kliem, R. Loitz, C. Bauckhage, and R. Sifa. 2022. KPI-BERT: A Joint Named Entity Recognition and Relation Extraction Model for Financial Reports. *arXiv:2208.02140 [cs.CL]* (2022).
[10] J.J. Hopfield. 1982. Neural Networks and Physical Systems with Collective Computational Abilities. *PNAS* 79, 8 (1982).
[11] J.J. Hopfield and D.W. Tank. 1985. "Neural" Computation of Decisions in Optimization Problems. *Biological Cybernetics* 52 (1985), 141–152.
[12] 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.
[13] S. Mücke, R. Heese, S. Müller, M. Wolter, and N. Piatkowski. 2022. Quantum feature Selection. *arXiv:2203.13261 [quant-ph]* (2022).
[14] S. Mücke, N. Piatkowski, and K. Morik. 2019. Hardware Acceleration of Machine Learning Beyond Linear Algebra. In *Proc. ECML/PKDD.*
[15] S. Mücke, N. Piatkowski, and K. Morik. 2019. Learning Bit by Bit: Extracting the Essence of Machine Learning. In *Proc. KDML-LWDA.*
[16] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).
[17] N. Piatkowski, T. Gerlach, R. Hugues, R. Sifa, C. Bauckhage, and F. Barbaresco. 2022. Towards Bundle Adjustment for Satellite Imaging via Quantum Machine Learning. In *Proc. FUSION.*
[18] N. Piatkowski and C. Zoufal. 2022. On Quantum Circuits for Discrete Graphical Models. *arXiv:2206.00398 [quant-ph]* (2022).
[19] R. Ramamurthy, M. Pielka, R. Stenzel, C. Bauckhage, R. Sifa, T. Khameneh, U. Warning, B. Kliem, and R. Loitz. 2021. ALiBERT: Improved Automated List Inspection (ALI) with BERT. In *Proc. Symposium on Document Engineering.* ACM.
[20] R. Sifa, A. Ladi, M. Pielka, R. Ramamurthy, L. Hillebrand, B. Kirsch, D. Biesner, R. Stenzel, T. Bell, M. Lübbering, U. Nütten, C. Bauckhage, U. Warning, B. Fürst, T. Khameneh, D. Thom, I. Huseynov, R. Kahlert, J. Schlums, H. Ismail, B. Kliem, and R. Loitz. 2019. Towards Automated Auditing with Machine Learning. In *Proc. Symposium on Document Engineering.* ACM.