Spherical Decoder via Neural Networks

Nícolas Morazotti

February 3, 2022

1 Introduction

Following [1], we wish to treat the Closest Vector in a lattice Problem (CVP) by the means of Spherical decoding. Let me explain.

Suppose there is a lattice Λ (a discrete additive subgroup of, say, $\mathbb{R}^n$), and a point $\mathbf{x}$ such that $\mathbf{x} \notin \Lambda$. We wish to obtain the lattice vector $\boldsymbol{\ell} \in \Lambda$ closest to $\mathbf{x}$, i.e. minimizes the squared Euclidean distance[1]. Notice that any point $\boldsymbol{\ell} \in \Lambda$ can be written as $\boldsymbol{\ell} = G\mathbf{s}$, where $\mathbf{s}$ is a vector of its component in each direction given by the basis matrix G. Thus, any point $\mathbf{x} = G\mathbf{s} + \mathbf{w}$, where $\mathbf{w}$ is some noise.

Spherical decoding tells us to search for a hypersphere of radius r_i around $\mathbf{x}$ such that there is only one lattice point inside it. It follows that this point must be the closest vector in the lattice.

However, the problem is not as easy as it sounds. We may choose some r_i s.t. there is no point inside the hypersphere, but as we iterate, r_{i+1} has more than one. To find the optimal r_i, we make use of the neural networks framework. We will implement this mainly in `python`.

The DNN aims to provide q possible r_i s.t. we may be able to find the closest vector in a lattice for y. At first, we will implement this by considering mean squared error as our loss function, only to generalize it later by the complexity metric.

2 The lattice in question

In our case, the lattice we wish to use is not a usual $\mathbb{Z}^n$ lattice with some basis G, but it is a lattice generated by unitary transformations diagonal in the computational basis, which requires us to sweat a bit.

As [2] points out, a Hamiltonian containing terms only proportional to $\mathbb{I}$ and σ_z, called *diagonal in the computational basis*, generates a unitary transformation that is able to be approximated by a lattice, and this approximation gives us the least quantum complexity of the transformation. Our lattice basis can be chosen to be $\{J_z\}_{z=1,\ldots,2n-1}$, $J_z = 2\pi\left(|z\rangle\langle z| - |0\rangle\langle 0|\right)$, where n is the number of qubits of our system.

In our first approach, we wish to attack two qubits. Therefore, $n = 2 \implies \{J_1, J_2, J_3\}$. We have four possible states: $|0\rangle|0\rangle, |0\rangle|1\rangle, |1\rangle|0\rangle, |1\rangle|1\rangle$, which we can translate as

$$|0\rangle \equiv |0\rangle|0\rangle \tag{1}$$
$$|1\rangle \equiv |0\rangle|1\rangle \tag{2}$$
$$|2\rangle \equiv |1\rangle|0\rangle \tag{3}$$
$$|3\rangle \equiv |1\rangle|1\rangle. \tag{4}$$

Clearly, we can define

$$|0\rangle = \begin{pmatrix}1 & 0 & 0 & 0\end{pmatrix}, \tag{5}$$
$$|1\rangle = \begin{pmatrix}0 & 1 & 0 & 0\end{pmatrix}, \tag{6}$$
$$|2\rangle = \begin{pmatrix}0 & 0 & 1 & 0\end{pmatrix}, \tag{7}$$
$$|3\rangle = \begin{pmatrix}0 & 0 & 0 & 1\end{pmatrix}, \tag{8}$$

[1] We may minimize other distances as well.

with $\mathbb{I} = \sum_{i=0}^{3} |i\rangle\langle i|$.

Once an all-identity term $\mathbb{I} \otimes \mathbb{I}$ in the Hamiltonian gives us no evolution in time, there is three possible terms to consider: $\mathbb{I} \otimes \sigma_z$, $\sigma_z \otimes \mathbb{I}$ and $\sigma_z \otimes \sigma_z$. Thus, our lattice is three-dimensional, and since z can not be zero, we may define the basis of our system in terms of the non-zero-index diagonal term of J_z, which is a set of three three-dimensional vectors. We may write then

$$\mathbf{J}_1 = \begin{pmatrix} 2\pi & 0 & 0 \end{pmatrix}, \tag{9}$$

$$\mathbf{J}_2 = \begin{pmatrix} 0 & 2\pi & 0 \end{pmatrix}, \tag{10}$$

$$\mathbf{J}_3 = \begin{pmatrix} 0 & 0 & 2\pi \end{pmatrix}, \tag{11}$$

as the elements of our basis.

Now, to translate each possible element of our Hamiltonian, we must analyze all tensor products:

$$\begin{aligned} \mathbb{I} \otimes \sigma_z &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}, \end{aligned} \tag{12}$$

$$\begin{aligned} \sigma_z \otimes \mathbb{I} &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}, \end{aligned} \tag{13}$$

$$\begin{aligned} \sigma_z \otimes \sigma_z &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \end{aligned} \tag{14}$$

By analogy, we may infer them as the following vectors:

$$\mathbb{I} \otimes \sigma_z = \frac{1}{2\pi}(-\mathbf{J}_1 + \mathbf{J}_2 - \mathbf{J}_3) \tag{15}$$

$$\sigma_z \otimes \mathbb{I} = \frac{1}{2\pi}(\mathbf{J}_1 - \mathbf{J}_2 - \mathbf{J}_3) \tag{16}$$

$$\sigma_z \otimes \sigma_z = \frac{1}{2\pi}(-\mathbf{J}_1 - \mathbf{J}_2 + \mathbf{J}_3). \tag{17}$$

Any time-dependent diagonal Hamiltonian for two qubits becomes

$$\begin{aligned} \mathbf{H}(t) &= \frac{1}{2\pi}\big[H_{IZ}(t)(-\mathbf{J}_1 + \mathbf{J}_2 - \mathbf{J}_3) + H_{ZI}(t)(\mathbf{J}_1 - \mathbf{J}_2 - \mathbf{J}_3) + H_{ZZ}(t)(-\mathbf{J}_1 - \mathbf{J}_2 + \mathbf{J}_3)\big] \\ &= \frac{1}{2\pi}\{\mathbf{J}_1[H_{ZI}(t) - H_{IZ}(t) - H_{ZZ}(t)] \qquad (18) \\ &\quad + \mathbf{J}_2[H_{IZ}(t) - H_{ZI}(t) - H_{ZZ}(t)] \qquad (19) \\ &\quad + \mathbf{J}_3[H_{ZI}(t) - H_{IZ}(t) - H_{ZI}(t)]\} \qquad (20) \end{aligned}$$

in this lattice basis.

3 Data generation

To feed our neural network, we need data. As such, we will generate a lot of points and compute the distance to each lattice node in a separate python script.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sys
```

```
J = np.eye(1)
```

```
lattice_range = range(-10, 11)
global lattice
lattice = np.array([[i]@J
                    for i in lattice_range])
```

```
def points_within(r, y):
    tof = np.linalg.norm(lattice-y, axis = 1) <= r
    return lattice[tof]

def get_radius(lp, y):
    return np.linalg.norm(lp-y, axis = 1)
```

```
data = 20*np.random.random([100_000, 1])@J - 10
```

```
radii = []
point = []
for k,v in enumerate(data[:50_000]):
    point.append(v)
    radii.append(np.sort(get_radius(lattice, v)))
    if k%5 == 0: print(f"{100*k/data.shape[0]}%")
np.savez("/home/nicolas/points_radii_data_1.npz", points=point, radii=radii)
point = []
radii = []
for k,v in enumerate(data[50_000:]):
    point.append(v)
    radii.append(np.sort(get_radius(lattice, v)))
    if k%5 == 0: print(f"{100*k/data.shape[0]}%")
np.savez("/home/nicolas/points_radii_data_2.npz", points=point, radii=radii)
```

```
time python data.py
```

References

[1] Mostafa Mohammadkarimi, Mehrtash Mehrabi, Masoud Ardakani, and Yindi Jing. Deep Learning-Based Sphere Decoding. *IEEE Transactions on Wireless Communications*, 18(9):4368–4378, September 2019.

[2] Michael A. Nielsen. A geometric approach to quantum circuit lower bounds. *arXiv:quant-ph/0502070*, February 2005.