IMPLEMENTATION OF A QUANTUM PERCEPTRON IN INTEL-QS

Sep. 2019

AUTHOR:
Mohammad Reza Noormandipour
(Shahid Beheshti University, Iran)

IT-DI-OPL

SUPERVISORS:

Federico Carminati (CERN)
Fabio Fracas (CERN)
Sofia Vallecorsa (CERN)

CERN openlab

ABSTRACT

With the pervasiveness of high-speed computers and processors, computer companies are looking for new technologies to incorporate into their products and use as a competitive advantage in the market. Two modern and rapidly growing techniques are quantum computing and the use of artificial neural networks (ANN) in computations. In this article, the possibility of integration of these two technologies is investigated and based on recently published research outcomes, an approach for the implementation of a recent model of a quantum perceptron is proposed. The implementation is conducted in the Intel quantum simulator (Intel-QS), which is a simulator written in C++ format using MPI to take advantage of multi-core and multi-node architectures for speed-up [1].

Keywords: Artificial Neural Networks (ANN) . Quantum Computing . Intel-QS . MPI

TABLE OF CONTENTS

Introduction 04

The Model 04

Unitary Transformations 06

Sign-Flip Block Approach

HSGS Algorithm

Results and Discussion 08

Conclusion 11

Acknowledgement 11

References 11

1. INTRODUCTION

Artificial neural networks are a set of algorithms, that are inspired by but not identical to, biological neural networks of brain and they have proved to be able to learn different tasks such as pattern recognition, image classification, and decision making by looking at a set of examples [5]. It is also worth mentioning that as complex as the architecture of the ANN becomes, it requires more computer resources (e.g. memory and CPU) to be trained and used properly. Hence, considerable effort has been devoted to research on finding a practical implementation of a neural network in inexpensive and fast hardware devices. *Field Programmable Gate Arrays (FPGAs) [2], Neuromorphic Chips [3], and Optical Networks [4]* are among the successful approaches to realize a neural network in micro-controllers, chemical materials with neural behaviours, and optical systems, respectively. These hardware based neural networks have many applications in science and technology. For instance, FPGAs are being tested in real-time triggering systems at the European organization for nuclear research (CERN) as an efficient way to perform various tasks such as particle identification and track reconstruction with much less overall latency [7].

Although these implementations are faster in some cases and less expensive than conventional memory and CPU used in laptops, they still need lots of resources. A possible solution for this issue is to find a mechanism or an architecture which is intrinsically capable of storing and accessing a large amount of information at a given clock cycle. Qubits have such properties. For this reason, researchers have attempted to build a model for a Quantum Perceptron which uses qubits to perform its job. The perceptron is the simplest possible architecture of an artificial neural network which essentially has one layer of nodes with the corresponding mutual connection between them. F. Tacchino et al. have recently proposed a model to implement an artificial neuron on an actual quantum processor [5]. This new model combines the mathematical modeling features offered by neural networks and the increased speed and access to more storage and information offered by qubits. The model is briefly explained in the next section and a successful implementation of a simple case with N = 2 qubits in the quantum register is performed in the Intel quantum simulator (Intel-QS [1]) and the outcomes are given in the results section.

2. The Model

R. Rosenblatt in 1957 [8] was the first person to build a simple model for a perceptron; which is represented in Fig.1a [5]. In this model, two binary valued vectors are given as input and weight vectors and the inner product of these two vectors is then passed to an activation function with a particular threshold value based on which it is decided whether the perceptron is activated or not. This type of binary valued perceptron is known as a McCulloch-Pitts neuron in literature and facilitates the definition of a quantum perceptron, because the input and weight vectors can be encoded as the sign of amplitudes in front of the Hilbert space states [9].

F. Tacchino et al. have employed a particular class of quantum hypergraph states to find an algorithm for quantum modeling of a perceptron [10]. The corresponding quantum version of the classical perceptron defined in Fig.1a, is depicted in Fig.1b. As generally accepted, any perceptron should have a main feature, which is the non-linearity of its outcome.This feature has been achieved in this proposal by introducing an Ancilla qubit in qubit register and then exploiting the non-linearity of the quantum measurement process of Ancilla qubit in order to implement the threshold function [5]. The input to the perceptron now comprises a qubit register with a set of qubits initialized in $|0\rangle$ state. Then, two unitary transformations, namely U_i and U_w are defined in such a way that they transform the initial qubit register to a state that if projected along the Ancilla qubit, is equal to the inner product of the classical input and weight vectors, up to a normalization factor [5].

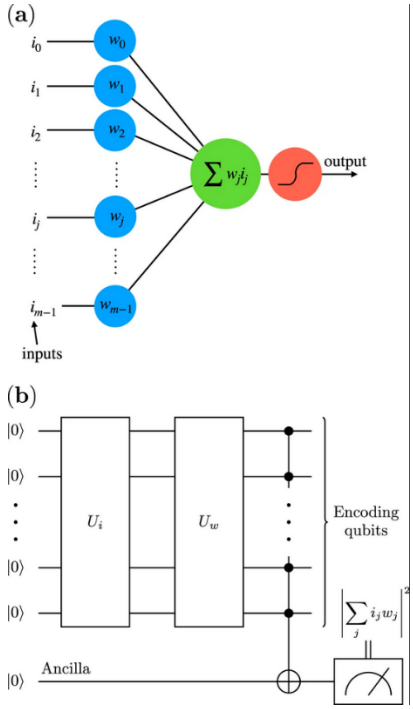

Figure 1. (a) Schematic representation of a classical versus (b) a quantum version of a perceptron (obtained from [5]).

In order to demonstrate how the unitary transformations perform this task, the input and weight vectors are considered as m-dimensional vectors with binary valued components $i_j \,\&\, w_j \in \{-1, +1\}$ as written in Equ.1. Furthermore, two quantum states $|\psi_i\rangle$ and $|\psi_w\rangle$ are defined using the corresponding vector components as amplitudes for the eigenstates in the m-dimensional Hilbert space of N qubits (see Equ.2), where $m = 2^N$. These two quantum states are equally weighted superpositions of every possible state in the Hilbert space.

$$\vec{i} = \begin{pmatrix} i_0 \\ i_1 \\ \vdots \\ i_{m-1} \end{pmatrix} \quad \& \quad \vec{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{m-1} \end{pmatrix} \tag{1}$$

$$|\psi_i\rangle = \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} i_j |j\rangle \quad \& \quad |\psi_w\rangle = \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} w_j |j\rangle \tag{2}$$

The states $|j\rangle \in \{|00\ldots00\rangle, |00\ldots01\rangle, \ldots, |11\ldots11\rangle\}$ are the members of the Hilbert space for *N* qubits in the qubit register - this space is the so-called computational basis of the quantum register. The index *j* is the decimal representation of the corresponding binary string for each member of the computational basis.

U_i is defined in such a way that when applied on the initial qubit register, it produces the $|\psi_i\rangle$ quantum state and similarly, U_w is defined in a way that when it acts on the quantum state $|\psi_w\rangle$, the computational basis member $|1\rangle^{\otimes N}$ is obtained.

$$U_i|0\rangle^{\otimes N} = |\psi_i\rangle \quad \& \quad U_w|\psi_w\rangle = |1\rangle^{\otimes N} = |m-1\rangle \tag{3}$$

As stated by F. Tacchino et al., any unitary matrix with the input vector in the first column can be used as U_i and any unitary matrix with the weight vector in its last row can be substituted in the Equ.3 for U_w. The action of the U_w operator on $|\psi_i\rangle$ is also defined as $|\phi_{i,w}\rangle$ and since $[U_i, U_w] = 0$, we can expand the result as in Equ.4.

$$U_w|\psi_i\rangle = \sum_{j=0}^{m-1} c_j|j\rangle \equiv |\phi_{i,w}\rangle \tag{4}$$

From Equ.3 it is easy to deduce that the inner product of quantum states in Equ.2 can be written as

$$\langle\psi_w|\psi_i\rangle = \langle\psi_w|U_w^\dagger U_w|\psi_i\rangle = \langle m-1|\phi_{i,w}\rangle = c_{m-1} \tag{5}$$

Moreover, using Equ.2 it is straightforward to demonstrate that $\vec{\imath}.\vec{w} = m\,\langle\psi_w|\psi_i\rangle$. Therefore, the inner product of the input and weight vector which is the most important element for a perceptron, is related to the coefficient c_{m-1} (see Equ.5) up to a normalization factor. In order to obtain this coefficient, an Ancilla qubit is used in the qubit register which is initially in state $|0\rangle$ [5]. A N-controlled NOT gate between the usual qubits in the quantum register and the target Ancilla qubit, will change the term $|\phi_{i,w}\rangle|0\rangle_a$ as below [5].

$$|\phi_{i,w}\rangle|0\rangle_a \longmapsto \sum_{j=0}^{m-2} c_j|j\rangle|0\rangle_a + c_{m-1}|m-1\rangle|1\rangle_a \tag{6}$$

Therefore, one can obtain the coefficient c_{m-1} by performing a measurement on the probability of the Ancilla qubit being in state $|1\rangle_a$ [5]. This quantum measurement introduces a non-linearity in the outcome of the threshold function which is desired for an effective modeling of a quantum perceptron [5].

3. Unitary Transformations

In this section the practical implementation of the unitary transformations is explained. These transformations consist of a set of operators which are unitary by themselves and will prepare the quantum states $|\psi_i\rangle$ and $|\psi_w\rangle$ as defined in Equ.2. There are two approaches for this purpose, namely the sign-flip blocks and the Hypergraph States Generation Subroutine (HSGS) algorithm [5].

a. Sign-Flip Block Approach

This approach is a brute-force method and utilizes a successive application of sign-flip blocks to manipulate the sign in front of the amplitudes associated with each member of the computational basis within $|\psi_i\rangle$ expansion [5]. A definition of the sign-flip operator is given in the Equ.7. The $SF_{N,j}$ can be seen as a controlled Z gate, which is a useful quantum gate [6] and can be employed to change the signs in front of computational basis vectors based on a given input or weight vector to obtain the desired representation for $|\psi_i\rangle$ and $|\psi_w\rangle$. For example $Z \equiv SF_{1,1}$ and $C^NZ \equiv SF_{N,m-1}$ [5]. Therefore, a generic sign-flip block can be decomposed into a product of NOT and $\mathrm{C^NZ}$ quantum gates as illustrated in Equ.8.

$$SF_{N,j}|j'\rangle = \begin{cases} |j'\rangle & if \;\; j \neq j' \\ -|j'\rangle & if \;\; j = j' \end{cases} \tag{7}$$

$$SF_{N,j} = O_j(C^N Z)O_j \qquad \& \qquad O_j = \otimes_{l=0}^{m-1}(NOT_l)^{1-j_l} \tag{8}$$

The index l in the Equ.8 means that the particular quantum gate should be applied on the l-th qubit in the qubit register and $j_l = 0(1)$ if the l-th qubit is in state $|0\rangle$ ($|1\rangle$) [5].

This approach starts with a qubit register initialized in $|0\rangle^{\otimes N}$ state and then N number of parallel Hadamard gates are applied to obtain an equally weighted and positively signed superposition of all possible computational basis vectors. Then, based on the input or weight vector components, sign-flip blocks are applied to the qubits to produce the quantum states of Equ.2 [5]. The set of Hadamard, NOT and $\mathrm{C}^N\mathrm{Z}$ gates are then packed as the U_i or U_w unitary transformation introduced above.

b. HSGS Algorithm

The HSGS algorithm is based on the fact that there is a mapping between a mathematical hypergraph and a quantum circuit consisting of controlled Z gates [10]. A simple example of such a mapping is shown in Fig.2. This algorithm also starts with the qubit register in the state $|0\rangle^{\otimes N}$ and after applying the parallel Hadamard gates, based on a given input or weight vector, it follows the steps outlined below [5]:

- First, Z gates are applied on the computational basis vectors within the quantum state with only one qubit being in state $|0\rangle$ which require a (-1) sign in front of it. It should be noted that extra sign flips might happen and they must be considered at the end of the algorithm implementation.
- Second, $\mathrm{C}^p\mathrm{Z}$ gates are applied on the computational basis vectors with exactly p qubits in state $|1\rangle$, where $\mathrm{p} \in \{2, 3, \ldots, \mathrm{N}\}$. This leaves the computational basis members with less than p qubits in state $|1\rangle$ untouched.

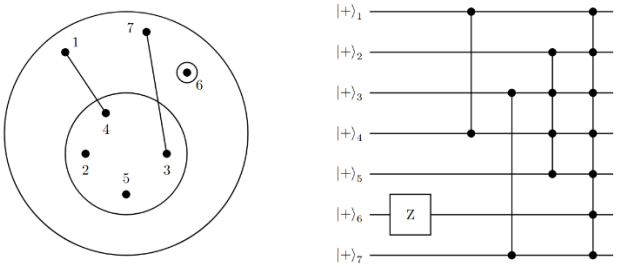

Figure 2. A mathematical heypergraph (on the left) and the corresponding quantum circuit (obtained from [5]). Circles represent a controlled Z gate.

As previously stated, after applying the unitary transformations, the state of the perceptron should be projected along done using an N-controlled NOT gate (Toffoli gate), where N is the number of qubits in the qubit register (excluding the Ancilla qubit) [5]. To further clarify the HSGS algorithm and also the role of Toffoli quantum gate, the quantum version of a classical perceptron with the below input and weight vectors is depicted in Fig.3, which is based on HSGS algorithm.

$$\vec{i} = \begin{pmatrix} -1 \\ -1 \\ +1 \\ -1 \end{pmatrix} \quad \& \quad \vec{w} = \begin{pmatrix} -1 \\ -1 \\ -1 \\ +1 \end{pmatrix} \tag{9}$$

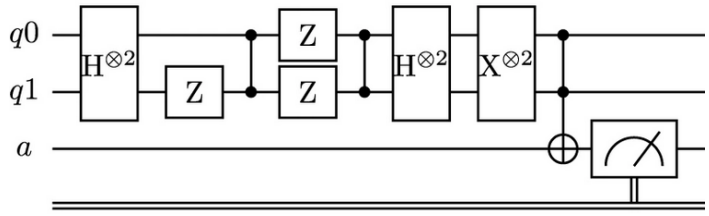

Figure 3. Quantum circuit for N = 2 case Q-Perceptron [5].

4. Results and Discussion

The N = 2 case quantum perceptron is implemented in Intel-QS. Intel-QS is a distributed high-performance quantum simulator on classical supercomputers, which is written in C++ format using a message passing interface (MPI). It can perform general qubit gates in multi-core architectures for obtaining high performance and high hardware efficiency [1]. As an example, the script for the quantum circuit of Fig.3 for the particular combination of input and weight vectors in equation 2 is given in the code snippet below.

```
#include "../qureg/qureg.hpp"
using namespace std;
#include <iostream> // to use: std::cout, std::cin and std::endl
#include <iomanip> // to use: setw() in making tables
#include <complex>
////////////////////////////////////////////////////////////////////////////////
//
////////////////////////////////////////////////////////////////////////////////
//
////////////////////////////////////////////////////////////////////////////////
//
int main(int argc, char **argv)
{
unsigned myrank=0, nprocs=1;
#ifdef INTELQS_HAS_MPI
openqu::mpi::Environment env(argc, argv);
myrank = openqu::mpi::Environment::rank();
nprocs = openqu::mpi::Environment::size();
// MPI_rank(MPI_COMM_WORLD, &myrank);
#endif
double sum = 0.;
std::cout << "------------------\n"
<< " Single qubit \n"
<< "------------------\n";
QubitRegister<ComplexDP> psi(3,"base",0);
psi.EnableStatistics();
psi.Print(" initial state |psi>=|-> : ");
psi.ApplyHadamard(0);
psi.ApplyHadamard(1);
psi.Print(" initial state |psi>=|-> : ");
psi.ApplyPauliZ(1);
psi.Print(" initial state |psi>=|-> : ");
```

```
psi.ApplyCPauliZ(0,1);
psi.Print(" initial state |psi>=|-> : ");
psi.ApplyPauliZ(0);
psi.ApplyPauliZ(1);
psi.Print(" initial state |psi>=|-> : ");
psi.ApplyCPauliZ(0,1);
psi.Print(" initial state |psi>=|-> : ");
psi.ApplyHadamard(0);
psi.ApplyHadamard(1);
psi.Print(" initial state |psi>=|-> : ");
psi.ApplyPauliX(0);
psi.ApplyPauliX(1);
psi.Print(" initial state |psi>=|-> : ");
psi.ApplyToffoli(2,0,1);
psi.Print(" initial state |psi>=|-> : ");
double prob = 0.;
prob = psi.GetProbability(2);
std::cout<< " Ancilla Measurement: " << prob << std::endl;
return 0;
}
```

In Fig.3 as we proceed from left to right, after applying each gate or set of parallel gates, the state of the qubit register is printed out with the command line "psi.Print(" initial state |psi>=|-> : ");" to make it possible for the reader to follow the changes in each step. The outcome of the above code is shown below.

```
 initial state |psi>=|-> : =[
       1.00000000    + i * 0.00000000      % |000> p=1.000000
       0.00000000    + i * 0.00000000      % |100> p=0.000000
       0.00000000    + i * 0.00000000      % |010> p=0.000000
       0.00000000    + i * 0.00000000      % |110> p=0.000000
       0.00000000    + i * 0.00000000      % |001> p=0.000000
       0.00000000    + i * 0.00000000      % |101> p=0.000000
       0.00000000    + i * 0.00000000      % |011> p=0.000000
       0.00000000    + i * 0.00000000      % |111> p=0.000000
]; % cumulative probability = 1.000000
permutation:  0 1 2
 initial state |psi>=|-> : =[
       0.50000000    + i * 0.00000000      % |000> p=0.250000
       0.50000000    + i * 0.00000000      % |100> p=0.250000
       0.50000000    + i * 0.00000000      % |010> p=0.250000
       0.50000000    + i * 0.00000000      % |110> p=0.250000
       0.00000000    + i * 0.00000000      % |001> p=0.000000
       0.00000000    + i * 0.00000000      % |101> p=0.000000
       0.00000000    + i * 0.00000000      % |011> p=0.000000
       0.00000000    + i * 0.00000000      % |111> p=0.000000
]; % cumulative probability = 1.000000
permutation:  0 1 2
 initial state |psi>=|-> : =[
       0.50000000    + i * 0.00000000      % |000> p=0.250000
       0.50000000    + i * 0.00000000      % |100> p=0.250000
      -0.50000000    + i * 0.00000000      % |010> p=0.250000
      -0.50000000    + i * 0.00000000      % |110> p=0.250000
       0.00000000    + i * 0.00000000      % |001> p=0.000000
       0.00000000    + i * 0.00000000      % |101> p=0.000000
       0.00000000    + i * 0.00000000      % |011> p=0.000000
       0.00000000    + i * 0.00000000      % |111> p=0.000000
]; % cumulative probability = 1.000000
permutation:  0 1 2
 initial state |psi>=|-> : =[
       0.50000000    + i * 0.00000000      % |000> p=0.250000
       0.50000000    + i * 0.00000000      % |100> p=0.250000
      -0.50000000    + i * 0.00000000      % |010> p=0.250000
       0.50000000    + i * 0.00000000      % |110> p=0.250000
       0.00000000    + i * 0.00000000      % |001> p=0.000000
       0.00000000    + i * 0.00000000      % |101> p=0.000000
       0.00000000    + i * 0.00000000      % |011> p=0.000000
       0.00000000    + i * 0.00000000      % |111> p=0.000000
]; % cumulative probability = 1.000000
permutation:  0 1 2
```

```
 initial state |psi>=|-> : =[
        0.50000000    + i * 0.00000000      % |000> p=0.250000
        -0.50000000   + i * 0.00000000      % |100> p=0.250000
        0.50000000    + i * 0.00000000      % |010> p=0.250000
        0.50000000    + i * 0.00000000      % |110> p=0.250000
        0.00000000    + i * 0.00000000      % |001> p=0.000000
        0.00000000    + i * 0.00000000      % |101> p=0.000000
        0.00000000    + i * 0.00000000      % |011> p=0.000000
        0.00000000    + i * 0.00000000      % |111> p=0.000000
]; % cumulative probability = 1.000000
permutation:  0 1 2
 initial state |psi>=|-> : =[
        0.50000000    + i * 0.00000000      % |000> p=0.250000
        -0.50000000   + i * 0.00000000      % |100> p=0.250000
        0.50000000    + i * 0.00000000      % |010> p=0.250000
        -0.50000000   + i * 0.00000000      % |110> p=0.250000
        0.00000000    + i * 0.00000000      % |001> p=0.000000
        0.00000000    + i * 0.00000000      % |101> p=0.000000
        0.00000000    + i * 0.00000000      % |011> p=0.000000
        0.00000000    + i * 0.00000000      % |111> p=0.000000
]; % cumulative probability = 1.000000
permutation:  0 1 2
 initial state |psi>=|-> : =[
        0.00000000    + i * 0.00000000      % |000> p=0.000000
        1.00000000    + i * 0.00000000      % |100> p=1.000000
        0.00000000    + i * 0.00000000      % |010> p=0.000000
        0.00000000    + i * 0.00000000      % |110> p=0.000000
        0.00000000    + i * 0.00000000      % |001> p=0.000000
        0.00000000    + i * 0.00000000      % |101> p=0.000000
        0.00000000    + i * 0.00000000      % |011> p=0.000000
        0.00000000    + i * 0.00000000      % |111> p=0.000000
]; % cumulative probability = 1.000000

 permutation:  0 1 2
  initial state |psi>=|-> : =[
        0.00000000    + i * 0.00000000      % |000> p=0.000000
        0.00000000    + i * 0.00000000      % |100> p=0.000000
        1.00000000    + i * 0.00000000      % |010> p=1.000000
        0.00000000    + i * 0.00000000      % |110> p=0.000000
        0.00000000    + i * 0.00000000      % |001> p=0.000000
        0.00000000    + i * 0.00000000      % |101> p=0.000000
        0.00000000    + i * 0.00000000      % |011> p=0.000000
        0.00000000    + i * 0.00000000      % |111> p=0.000000
 ]; % cumulative probability = 1.000000
 permutation:  0 1 2
  initial state |psi>=|-> : =[
        0.00000000    + i * 0.00000000      % |000> p=0.000000
        0.00000000    + i * 0.00000000      % |100> p=0.000000
        1.00000000    + i * 0.00000000      % |010> p=1.000000
        0.00000000    + i * 0.00000000      % |110> p=0.000000
        0.00000000    + i * 0.00000000      % |001> p=0.000000
        0.00000000    + i * 0.00000000      % |101> p=0.000000
        0.00000000    + i * 0.00000000      % |011> p=0.000000
        0.00000000    + i * 0.00000000      % |111> p=0.000000
 ]; % cumulative probability = 1.000000
  Ancilla Measurement: 0
```

As shown in the above results, the amplitudes and probabilities corresponding to each of the computational basis vectors is given as a complex valued and a real number, respectively. Furthermore, the outcome of the Ancilla qubit measurement is 0:0, which means that the inner product of the input and weight vectors must also be zero and this can easily be verified from equation 1.

It is worth mentioning that, despite the achieved speed-up because of multi-core parallelization and also high hardware efficiency through efficient memory allocation, there are limitations coming from sustainable memory and network bandwidth of the machine [1]. Simulation of qubits is so costly and needs a huge amount of memory (of the order of Petabytes for a few tens of qubits, as reported in [1]) and network bandwidth. This is because of the fact that by increasing the number of qubits, there is an exponential increase in the resources requests to store the quantum information. With the current state of the available supercomputers, it is only possible to simulate a quantum system with at most 50 qubits in it [1].

From the algorithmic point of view, it is straightforward to show that there is a symmetry under a global sign flip for input and weight vectors and the outcome of the perceptron remains unchanged [5]. Therefore, in

the worst case, the sign-flip approach requires at most $\frac{m}{2} = 2^{N-1}$ sign-flip blocks for a total of $\frac{m}{2}$ independent (-1) factors in the expansion of quantum states in Equ.2 [5]. Since each sign-flip block consists of an $\mathrm{C}^N\mathrm{Z}$ gate, this approach is exponentially expensive in terms of number of qubits involved in each gate and also the number of gates [5]. On the other hand, the HSGS algorithm uses at most one $\mathrm{C}^N\mathrm{Z}$ gate and a number of $\mathrm{C}^p\mathrm{Z}$ gates with $p < \mathrm{N}$. Therefore, the circuit built with this algorithm needs less resources and also proves to be more efficient than sign-flip approach in classification tasks as demonstrated by F. Tacchino et al. The HSGS is based on hypergraph states which have been studied and employed in quantum algorithms that have manifested successful results in practical applications [10][11].

5. Conclusion

In conclusion, the simplest case of a perceptron composed of two qubits was implemented in Intel-QS. The obtained results were consistent with the results by F. Tacchino et al. In principle, this type of perceptron can offer more storage capacity for information which is in general of great importance for machine learning algorithms [5]. On the other hand, if the implementation is on classical hardware, the resource requests also increase exponentially. One suggestion for an efficient implementation, is using near-term quantum processing devices such as cloud-based quantum hardware called IBM-Q [5]. However, since the Intel-QS uses multi-core and multi-node architectures for implementation, the classical supercomputers are likely to be the main feasible simulation devices for the next one or two decades.

Moreover, although the HSGS algorithm is more efficient and accurate than sign-flip approach and optimizes the number of multi-qubit operations [5], it still needs a considerable amount of resources.

As a future direction, author aims to completely implement the $N = 2$ case qubit register for arbitrary combinations of input and weight vectors and then perform classification tasks with that. Furthermore, generalization of the binary valued quantum perceptron to one with any possible value for input and weight vectors is being researched by the author. If this attempt proves to be successful, the perceptron can be trained since it would be possible to perform back-propagation algorithms. Moreover, one can look at a deep quantum neural network with many layers and interconnection built with a trainable quantum perceptron.

6. Acknowledgement

The author acknowledges the European organization for nuclear research to provide the required computing facilities for this research. The author also thanks F. Fracas, S. Vallecorsa, and F. Carminati for the valuable discussions and all the support provided at CERN OpenLab. The author thanks F. Tacchino et al. for giving the permission to use their papers figures and diagrams in this work. The views expressed are those of the author and do not reflect the official policy or position of Intel company.

7. References

[1] M. Smelyanskiy, N. P. D. Sawaya, and A. Aspuru-Guzik. qHiPSTER: The Quantum High Performance Software Testing Environment. arXiv:1601.07195v2 [quant-ph], 12 May 2016.

[2] S. Sahin, Y. Becerikli, and S. Yazici. Neural Network Implementation in Hardware Using FPGAs. Springer Berlin Heidelberg, 2006.

[3] F. Walter, F. Röhrbein, and A. Knoll. Neuromorphic implementations of neurobiological learning algorithms for spiking neural networks. Neural Networks. 72. 10.1016/j.neunet.2015.07.004.
[4] G. R. Steinbrecher, J. P. Olson, D. Englund, and J. Carolan. Quantum optical neural networks. npj Quantum Information. 5, Article number: 60, 2019.
[5] F. Tacchino, C. Macchiavello, D. Gerace, and D. Bajoni. An artificial neuron implemented on an actual quantum processor. npj Quantum Information 5, Article number 26 , 2019.
[6] M. A. Nielsen and I. L. Chuang. Quantum Computation and Qunautm Information. Cambridge Series on Information and the Natural Sciences, Cambridge University Press, Cambridge, United Kingdom, 2004.
[7] G. Hadash, E. Kermany, B. Carmeli, O. Lavi, G. Kour, and A. Jacovi. FPGA-accelerated machine learning inference as a service for particle physics computing. arXiv:1904.08986v1 [physics.data-an] , 18 April 2019.
[8] F. Rosenblatt. The Perceptron: A perceiving and recognizing automaton. Tech. Rep. Inc. Report No. 85-460-1 (Cornell Aeronautical Laboratory, 1957).
[9] W.S. McCulloch, W. Pitts. A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. 5,115-133m, 1943.
[10] M. Rossi, M. Huber, D. Brub, and C. Macchiavello. Quantum hypergraph states. New J. Phys. 15, 113022, 2013.
[11] M. Ghio, D. Malpetti, M. Rossi, D. Bruß, and C. Macchiavello. Multipartite entanglement detection for hypergraph states. J. Phys. A: Math. Theor. 51, 045302, 2018.