COMP 4560 : Undergraduate Industrial Project
Final Report

Implementation of the Quantum Perceptron Algorithm

Abhishek .
Department of Computer Science, University of Manitoba
Email : abhishek@myumanitoba.ca

Supervisor : Dr. Parimala Thulasiraman
Department of Computer Science, University of Manitoba
Email : thulasir@cs.umanitoba.ca

April 26, 2018

Abstract

The classical perceptron algorithm is one of the foundational algorithms in machine learning and has served as a basis for many modern day machine learning models. Quantum computing and quantum information processing techniques have been shown to provide speedup to many classical machine learning algorithms. The purpose of this project was to explore one of the recently proposed models of the quantum perceptron in detail and gain hands-on experience in programming a simulated quantum computer through implementation of the quantum perceptron algorithm using the circuit model of quantum computation. The complete implementation is available at `https://github.com/abhishekabhishek/COMP-4560-Industrial-Project`

Acknowledgements

I would like to thank my supervisor, Dr. Parimala Thulasiraman for her guidance, comments and expert knowledge throughout the project. I would also like to thank the Undergraduate Industrial Project Coordinator, Dr. Ruppa K. Thulasiram for allowing me to have this amazing opportunity to learn the important concepts and explore the recent advancements in the field of Quantum Computing and Machine Learning.

Contents

1 Introduction

Quantum computing refers to the usage of quantum information processing devices and techniques in order to perform computational tasks. Machine Learning is the field of Computer Science in which data-driven and statistical-based learning algorithms are used to build a model to perform a given artificial intelligence task such as regression, classification, recognition etc.
One of the major challenges in the development and training of machine learning models today is that even the best machine learning models suffer from limitations in terms of training times, resources and dataset size requirements. Quantum parallelism is an inherent property of quantum information processing devices. The ability and ingenuity to exploit the massive parallelism inherent in the the quantum information processing devices forms the basis of many well-known and well studied quantum algorithms such as Quantum Fourier Transform [1] and Quantum Search [2]. Since, many algorithms in machine learning have an operational parallelism due to the architecture and structure of the models, applications of quantum computing techniques in order to provide speed-up to existing classical machine learning algorithms have become a topic of widespread interest.
The original perceptron algorithm was published by Frank Rosenblatt in 1958 [3] and is an algorithm for learning a binary classifier that takes a real-valued vector as an input and outputs a single binary value (interpreted as a class label). The perceptron can also be described as an artificial neuron with the unit step function as the activation function in the context of Artificial Neural Networks (ANNs). The McCulloch-Pitts Neuron [4] is a specialization of the Rosenblatt perceptron where the input vector is binary-valued $\{-1, +1\}$ instead of real values.
Due to the simplicity and the importance of the perceptron model, many methods have been proposed in order to implement the perceptron algorithm on a quantum computer [5] [6]. In this industrial project, we considered the model proposed by Tacchino et al. [5] in detail and implemented it on a simulated quantum computer. We received the expected results and compared the results of the classical and quantum models of the perceptron.

2 Theoretical Background

In this section, we will discuss the theoretical background necessary to either implement or understand the implementation of the classical and the quantum perceptron.

2.1 Classical Perceptron

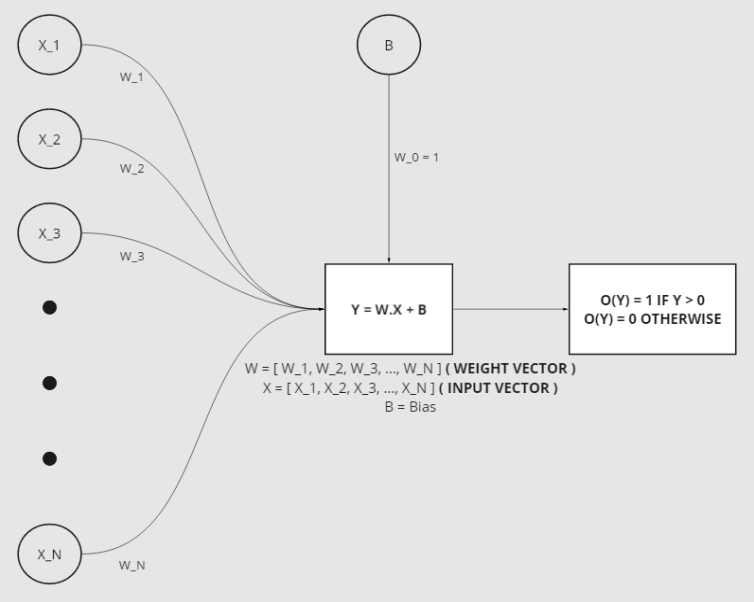


Figure 1: The architecture of the classical perceptron algorithm introduced by Frank Rosenblatt in 1958.

The original classical perceptron algorithm defines a classifier which outputs a binary value which can be interpreted as a class label. The output of the classical perceptron is defined as the following :

$$f(X) = \begin{cases} 1 & , \mathbf{W}.\mathbf{X} + B > 0 \\ 0 & otherwise \end{cases} \tag{1}$$

$\mathbf{X}$ is the n-dimensional real-valued input vector, $\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix}$, $\mathbf{W}$ is the n-dimensional real-valued weight vector, $\mathbf{W} = \begin{bmatrix} w_1 \\ w_2 \\ . \\ . \\ w_n \end{bmatrix}$ and B is the bias value for the perceptron. $\mathbf{W}.\mathbf{X}$ is the dot product of the two vectors i.e. $\sum_{i=1}^{n} x_i.w_i$.
If we consider the input vector $\mathbf{X}$ and the weight vector $\mathbf{W}$ to be binary valued i.e. $\forall i : w_i, x_i \in \{-1, +1\}$, we get the McCulloch-Pitts Neuron model.

2.2 Quantum Circuits

In the circuit model of universal quantum computation, quantum circuits are used in order to represent operations on quantum systems. The simplest of these operations are known as single qubit operations and are defined as single qubit gates in the circuit model. Controlled operations are another important set of operations. **A deep understanding and knowledge of these operations is necessary in order to implement or understand the implementation of many quantum algorithms.**

2.2.1 Single Qubit Operations

A single qubit is defined as the vector, $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ and is parameterized by two complex numbers which satisfy the condition, $|\alpha|^2 + |\beta|^2 = 1$. Single qubit operations act on this vector and are described as 2×2 unitary matrices.
For example, consider the Hadamard operation H for which the unitary matrix is given as :

$$H \equiv \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \tag{2}$$

The application of this operations acting on an arbitrary qubit state, $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ is defines as :

$$H|\psi\rangle \equiv \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \begin{bmatrix}\frac{\alpha+\beta}{\sqrt{2}}\\ \frac{\alpha-\beta}{\sqrt{2}}\end{bmatrix} \tag{3}$$

Some of the most common and important single qubits operations are given below :

- Hadamard gate, $H \equiv \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$
- Pauli-X gate, $X \equiv \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$
- Pauli-Y gate, $Y \equiv \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix}$
- Pauli-Z gate, $Z \equiv \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$
- Phase gate, $S \equiv \begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix}$
- $\pi/8$ gate, $T \equiv \begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix}$

In this project, we will use the Hadamard gate and the Pauli-Z gate extensively in order to implement the models.

2.2.2 Controlled Operations

The Controlled Operations form another set of important operations in quantum computing and computing in general. A Controlled operation is a two or more qubit operation in which two sets of qubits are used : control and target. The state of the control qubits is used to determine whether to apply a given unitary operation on the target qubits. More precisely, if all the control qubits are in the computational state, $|1\rangle$, then the unitary operation is applied to the target qubits. Note that this may not be trivial in the case where the control and target qubits are in a superposition or entangled. The most commonly known controlled operation is the controlled-NOT which is defined as the two input gate with one control and one target qubit. The action of the CNOT gate on an input state $|c\rangle |t\rangle$ is given as the following :

$$CNOT |c\rangle |t\rangle = |c\rangle |t \oplus c\rangle \tag{4}$$

The CNOT operation is defined by the unitary matrix :

$$CNOT \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In a general case, consider U to be an unitary operation. Then, the controlled-U operation, CU is a two qubit operation in which the unitary operation U is applied to the target qubit if the control qubit is set. Otherwise, the target qubit is left alone i.e.
$CU |c\rangle |t\rangle = |c\rangle U^C |t\rangle$
Another well known controlled operation which is often used in the quantum algorithms and is also used in the quantum perceptron algorithm is the controlled-Z operation, CZ. The operation is define by the unitary matrix :

$$CZ \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

Note that the controlled operations can be extended to more than two qubits to multiple qubits used as control and target. In fact, three qubit gates such as the Toffoli gate and the Fredkin gate are used commonly in practice.
In this project, we will use the (multi-)controlled-NOT gate and the (multi-)controlled-Z gate extensively in order to implement the models.

2.3 Quantum Perceptron

In this section, we will discuss the models in detail to build an intuitive understanding of the algorithm and its implementation.
In the recent years, several models have been presented forward for the algorithm and implementation of the quantum version of the perceptron. We used the model proposed by Tacchino et al. [5] since it is a pure quantum version of the algorithm rather than some of the other methods which use quantum algorithm techniques such as Grover's Search to develop an hybrid algorithm for the quantum perceptron and also since it is based in the

circuit model of quantum computation which is used for implementation by IBM Q and the python library QisKit.
In order to develop the quantum version of the perceptron, the first problem to solve is encoding the input and weight vectors in quantum states. The advantage of using quantum computation for the model particularly the exponential advantage of quantum information storage is clearly visible since only n qubits are used in order to encode m-dimensional input and weight vectors. We consider the case only where m is an exponential power of 2 i.e. $m = 2^n$. The encoding problem is solved in [5] as following : Since the goal of the quantum perceptron algorithm is to compute the dot product of the input and weight vector, a gate sequence is used in order to construct unitary transformation, U_i and U_w such that $U_i \left|0^{\otimes N}\right\rangle = |\psi_i\rangle$ and $U_w |\psi_w\rangle = |m-1\rangle$ where the quantum states $|\psi_i\rangle$ and $|\psi_w\rangle$ are defined as the following :

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

when the input and weight vectors, $\vec{i}$ and $\vec{w}$ are given as :

$$\vec{i} = \begin{bmatrix} i_0 \\ i_1 \\ . \\ . \\ . \\ i_{m-1} \end{bmatrix}, \vec{w} = \begin{bmatrix} w_0 \\ w_1 \\ . \\ . \\ . \\ w_{m-1} \end{bmatrix}$$

We note that any $m \times m$ unitary matrix either having $\vec{i}$ as the first column (with the initial state as $|0\rangle^n$) or having $\vec{i}$ as the diagonal can be used as U_i (with the initial state in a linear superposition). Similarly, any $m \times m$ unitary matrix having $\vec{w}^T$ as the last row can be used as U_w.
We apply the unitary operation for the weights U_w after the unitary operation for the inputs U_i, the result quantum state is given by :

$$|\phi_{i,w}\rangle = U_w |\psi_i\rangle = \sum_{j=0}^{m-1} c_j |j\rangle \tag{5}$$

The dot product of the two quantum states $|\psi_i\rangle$ and $|\psi_w\rangle$ is computed as the following :

$$\langle\psi_w|\psi_i\rangle = \langle\psi_w| U_w^\dagger U_w |\psi_i\rangle = \langle m-1|\phi_{i,w}\rangle = \langle m-1| \sum_{j=0}^{m-1} c_j |j\rangle = c_{m-1}$$

The result of the dot product of the two quantum states $|\psi_i\rangle$ and $|\psi_w\rangle$ corresponding to the input and the weight vectors is contained in the coefficient c_{m-1} upto a constant normalization factor.
In order to extract this result from the end state $|\phi_{i,w}\rangle$, we use an ancilla qubit and a multi-controlled NOT gate which uses all the n qubits as the control qubits and the ancilla qubit as the target. The operation is defined as the following :

$$C^n NOT |\phi_{i,w}\rangle |0\rangle = \sum_{j=0}^{m-2} c_j |j\rangle |0\rangle + c_{m-1} |m-1\rangle |1\rangle \tag{6}$$

The non-linearity of the measurement function is used in order to apply the threshold function and the ancilla qubit is measured in the state $|1\rangle$ with probability $|c_{m-1}|^2$ and in the state $|0\rangle$ with probability $1 - |c_{m-1}|^2$.
The next problem to solve is the efficient implementation of the unitary operations U_i and U_w. In [5], the authors proposed two methods in order to implement the unitary operations :

- Brute-force application of successive sign-flip blocks
- Using (HSGS) Hypergraph State Generation Subroutine

Although both the methods have a exponential worst-case gate complexity, the second method on average requires a lower number of gates to be applied to the qubits in order to implement the quantum perceptron.
A Sign Flip Block $SF_{N,j}$ acting on a computation basis of N qubits is defined as the following :

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

In other words, a Sign Flip Block $SF_{N,j}$ flips the phase of the j^{th} computational state. This is achieved through the use of single qubit NOT gates and (multi)controlled-Z gates. The Sign Flip Block $SF_{N,j}$ is defined as :

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

where NOT_l is the NOT gate applied to the l^{th} qubit and

$$j_l = \begin{cases} 0, & \text{if the } l^{th} \text{ qubit is in the state } |0\rangle \text{ in the } j^{th} \text{ computational basis state} \\ 1, & \text{if the } l^{th} \text{ qubit is in the state } |1\rangle \text{ in the } j^{th} \text{ computational basis state} \end{cases} \tag{9}$$

Note that the Sign-Flip Blocks commute with each other and therefore can be applied in any order in the quantum circuit. However, the gate complexity of the algorithm would still be exponential since we need to apply a Sign-Flip block to each of the computational basis state every where the -1 phase is desired in the input and weight vectors.
The unitary transformation U_i corresponding to the input vector $\vec{i}$ can then be implemented by first apply the Hadamard transformation, $H^{\otimes N}$ to the initial state $|0\rangle^{\otimes N}$ in order to place the qubits in a linear superposition i.e.

$$H^{\otimes N}|0\rangle^{\otimes N} = \frac{1}{\sqrt{2^N}}\sum_{j=0}^{2^N-1}|j\rangle$$

The Sign-Flip Blocks can then be consecutively applied to every j^{th} computational basis state where there is a factor of -1 i.e. the j^{th} input in the input vector $\vec{i}$ is -1. The gate complexity of this would be $2^{N-1} = \frac{2^N}{2}$ since the problem is symmetric under a global phase factor of -1.
Similarly, the unitary transformation U_w corresponding to the weight vector $\vec{w}$ can be constructed using the Sign-Flip blocks. This leads to the balanced superposition $|\psi_0\rangle$. In order to extract the dot product from the coefficient of this state, we first apply the

Hadamard operation on the N qubits, $H^{\otimes N}$ and then apply the NOT operation on the N qubits, $NOT^{\otimes N}$ which results in the qubits being in the state $|m-1\rangle$. Applying the multi-controlled NOT gate on the n qubits with the ancilla qubit as the target leads to the dot product being stored in the $|1\rangle$ state of the ancilla qubit as seen in Equation (6). Finally, the measurement operation can be used to simulate the non-linearity of the threshold function. The basis with the higher probability is denoted as the output of the quantum perceptron algorithm.

Another method that the authors of [5] propose is the use of the HSGS (Hypergraph State Generation Subroutine) which is described as the following :

Starting from $p = 1$, we determine whether there is a computational basis state with p qubits in the state $|1\rangle$ requiring a -1 factor. This can be achieved by simply applying the single qubit Pauli-Z gate to the qubit which corresponds to the computational basis state requiring a -1 factor. This procedure is continued for $p = 2, ..., N$ where N is the number of qubits being used to encode the input and the weight vectors. This is achieved by applying the operation C^pZ between the p qubits in the state $|1\rangle$. For each of the computational basis states with p qubits in the state $|1\rangle$, a phase factor of -1 is introduced if needed through the given operation if it was not introduced before, otherwise it can be removed if not needed.

Another thing to note here is that the operations C^pZ only flips the phase of computational basis states with p or more qubits in the state $|1\rangle$. This is important since that means the method terminates when $p = N$ and all the signs are as desired.

Similar to the method that uses the Sign-Flip Blocks, the unitary transformation U_w can be implemented in a similar way followed by the application of the Hadamard operation $H^{\otimes N}$ on the N qubits and the NOT operations $NOT^{\otimes N}$ on the N qubits in order to extract the result of the dot product of the quantum states corresponding to the input and the weight vectors.

3 Implementation details

In this section, we will discuss in detail how the classical algorithm and the two different methods of the quantum perceptron algorithm can be implemented. We considered the single and double qubit case in order to implement the algorithms since even the low number of qubits are sufficient to show the exponential storage advantage of the quantum model as well as the comparison of the two approaches : Sign-Flip Blocks and Hypergraph State Generation Subroutine.

3.1 Overview

The implementation for both the classical and quantum perceptron model was done in Python 3.6.6. Only the default python library calls were used in order to implement the classical perceptron model. For the quantum perceptron model, Qiskit [7] is an open-source quantum computing framework which allows to develop and execute quantum programs locally as well as remotely on the IBM Q machines. Due to the limitations of access to the actual quantum hardware from IBM, we used only the local simulators in order to execute the quantum circuits. We used Anaconda as a package management tool in order to maintain the package dependencies. Additionally, the implementation is done in Jupyter Notebooks which are commonly used to develop quantum programs due to the ease of explaining the implementation within the notebooks.
We considered 4 cases for each of the single and double qubit implementation :

1. Single Qubit Case
   (a) Case 1 : $\vec{i} = \begin{bmatrix}1\\1\end{bmatrix}, \vec{w} = \begin{bmatrix}1\\1\end{bmatrix}$
   (b) Case 2 : $\vec{i} = \begin{bmatrix}1\\-1\end{bmatrix}, \vec{w} = \begin{bmatrix}1\\1\end{bmatrix}$
   (c) Case 3 : $\vec{i} = \begin{bmatrix}1\\1\end{bmatrix}, \vec{w} = \begin{bmatrix}1\\-1\end{bmatrix}$
   (d) Case 4 : $\vec{i} = \begin{bmatrix}1\\-1\end{bmatrix}, \vec{w} = \begin{bmatrix}1\\-1\end{bmatrix}$
2. Double Qubit Case
   (a) Case 1 : $\vec{i} = \begin{bmatrix}1\\1\\1\\1\end{bmatrix}, \vec{w} = \begin{bmatrix}1\\1\\1\\1\end{bmatrix}$
   (b) Case 2 : $\vec{i} = \begin{bmatrix}1\\-1\\-1\\-1\end{bmatrix}, \vec{w} = \begin{bmatrix}1\\1\\1\\1\end{bmatrix}$
   (c) Case 3 : $\vec{i} = \begin{bmatrix}1\\1\\1\\1\end{bmatrix}, \vec{w} = \begin{bmatrix}1\\-1\\1\\-1\end{bmatrix}$

(d) Case 4 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$

3.2 Classical Perceptron

For both the single and double qubit case, the classical perceptron can be easily implemented by first calculating the dot product for each pair of input-weight, storing it in an array and then computing the sum of the array.
In order to implement the threshold function, a simple conditional if statement can be used to determine whether the output of the perceptron should be 1 ($\vec{w}.\vec{x} + B > 0$) or 0 ($\vec{w}.\vec{x} + B \leq 0$).
This was implemented in Python 3.6.6 by first using the $zip()$ function to compute the element-wise product and then using the $sum()$ function to compute the dot product of the two vectors.

3.3 Quantum Perceptron

3.3.1 API Access Setup

In order to build and execute programs on the IBM Q machine or local simulators, we need to have authorized access to the machine. The access can be requested through the IBM Q website by creating an IBMid. Once the access is gained, the python library Qiskit can be used to develop and execute programs locally on simulators or remotely on the IBM Q backends.

3.3.2 Single Qubit Case

For the single qubit case, both the methods i.e. using the Sign-Flip Blocks and using the Hypergraph State Generation Subroutine (HSGS) return the same quantum circuit configuration as seen in Appendix B (Quantum Perceptron).

3.3.3 Double Qubit Case

For the double qubit case, the quantum circuit generated using the Sign-Flip Blocks method is much deeper than the quantum circuit generated using the Hypergraph State Generation Subroutine (HSGS).

3.3.4 Constructing Quantum Circuits

The problem to be solved during the implementation section is the efficient translation of the methods discussed in the Section 2.3 to quantum circuits. Although some of the trivial gates such as the single qubit Z gate used to implement the methods are readily available in QisKit, for many other gates we need to find an accurate decomposition into the set of gates available. For example, consider the single qubit NOT gate which is widely used in the first method of the Sign-Flip Blocks. The NOT gate is not available in QisKit although can be constructed as following :
Consider applying NOT gate to a single qubit in the state $|\psi\rangle$, the qubit state $|\psi\rangle$ can either be $|0\rangle$, $|1\rangle$ or a superposition. The NOT gate can then be decomposed into a single

qubit Hadamard gate H, a single qubit Pauli-Z gate Z and a single qubit Hadamard gate H i.e.

$$NOT \equiv HZH$$

A similar decomposition has to be constructed for multi-controlled unitary gates since most of them are not directly available in QisKit.
Another problem to be solved is the construction of the sub-routines which support the encoding of the input and weight vectors as well as the computation of the dot product. Most of the solutions to this problem has been discussed in Section 2.3.

4 Discussion

4.1 Results

For all the cases described in Section 3.1, both the classical perceptron algorithm and the quantum perceptron algorithm generated similar outputs which shows the correctness of the methods used to implement the quantum perceptron. A detailed summary of the outputs for each of the cases is provided in Appendix A (Classical Perceptron) and Appendix B (Quantum Perceptron).
Another thing to note in the results is that the classical perceptron algorithm generates deterministic output whereas the quantum perceptron algorithms generates a probabilistic output due to the qubit states being in a superposition.
We also observed and as is clear in Appendix B, the method using the Hypergraph State Generation Subroutine (HSGS) resulted in more efficient quantum circuits than the method using the Brute-force application of the Sign-Flip Blocks.

4.2 Limitations

4.2.1 Exponential gate complexity

In the worst-case, both the methods (Sign-Flip Blocks and Hypergraph State Generation Subroutine) have an exponential gate complexity. This is not viable for actual quantum processors since in the current implementations, the qubits are not able to maintain coherence for a long amount of time and therefore only shallow circuits can be implemented on real quantum processors.

4.2.2 Real-valued inputs and weights

The models discussed in this industrial project are limited to having only binary-valued inputs and weights. This significantly restricts the applications of the models as many problems have real-valued inputs and weights. Although for some problems, the inputs and weights may be scaled up or down to binary values, for many other problems a model which can deal with real-valued inputs and weights is required.

4.2.3 Perceptron Learning Problem

The model presented in [5] and implemented above only considers the forward procedure of the classical perceptron algorithm i.e. calculating the output of the perceptron given the inputs $\vec{i}$ and weights $\vec{w}$. Another crucial part of the perceptron model is the learning of the weight so that the model learns for the data and can therefore adjusts its weights to accurately classify the inputs. Classically, the gradient descent algorithm is used to learn the weights of the perceptron. However, the learning of the weights of the perceptron from the data through error propagation is still an open problem to be solved.

4.3 Future work

The current limitations of the methods discussed in the section 5.1 can be considered as potential problems to be explored in future works. These are three problems which are fundamental to the development of deep learning models in the quantum model of computation since as we increase the complexity of the models : the resources required

scale exponentially, the limitations of the model in terms of inputs and weights affect the performance and the models require efficient training algorithms to learn the weights. Other than the model limitations discussed above, future work may also explore using the perceptrons as the building blocks of multi-layered neural networks which are currently one of the most popular and widely used models in machine learning.

5 Conclusion

In this industrial project, we explored in detail one of the recently proposed models for the quantum perceptron. We then implemented the proposed models on a local quantum simulator and compared the classical and the quantum perceptron algorithm as well as the two different methods to implement the quantum perceptron using the circuit model of quantum computation : one using Sign-Flip Blocks and the other using the Hypergraph State Generation Subroutine (HSGS). Through experimentation using single and double qubit circuits, we determined that although both the methods have a similar worst-case exponential gate complexity, the method using the Hypergraph State Generation Subroutine (HSGS) in general performs better and greatly reduces the gate complexity of the algorithm. We also discussed the limitations of the current approach and possible extensions of the model to be considered in future works.

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Review*, vol. 41, no. 2, pp. 303–332, 1999.

[2] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, (New York, NY, USA), pp. 212–219, ACM, 1996.

[3] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, pp. 65–386, 1958.

[4] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, Dec 1943.

[5] D. G. D. B. Francesco Tacchino, Chiara Macchiavello, "An artificial neuron implemented on an actual quantum processor," 2018.

[6] K. M. S. Nathan Wiebe, Ashish Kapoor, "Quantum perceptron models," 2016.

[7] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. J. Cabrera-Hernádez, J. Carballo-Franquis, A. Chen, C.-F. Chen, J. M. Chow, A. D. Córcoles-Gonzales, A. J. Cross, A. Cross, J. Cruz-Benito, C. Culver, S. D. L. P. González, E. D. L. Torre, D. Ding, E. Dumitrescu, I. Duran, P. Eendebak, M. Everitt, I. F. Sertage, A. Frisch, A. Fuhrer, J. Gambetta, B. G. Gago, J. Gomez-Mosquera, D. Greenberg, I. Hamamura, V. Havlicek, J. Hellmers, Ł. Herok, H. Horii, S. Hu, T. Imamichi, T. Itoko, A. Javadi-Abhari, N. Kanazawa, A. Karazeev, K. Krsulich, P. Liu, Y. Luh, Y. Maeng, M. Marques, F. J. Martín-Fernández, D. T. McClure, D. McKay, S. Meesala, A. Mezzacapo, N. Moll, D. M. Rodríguez, G. Nannicini, P. Nation, P. Ollitrault, L. J. O'Riordan, H. Paik, J. Pérez, A. Phan, M. Pistoia, V. Prutyanov, M. Reuter, J. Rice, A. R. Davila, R. H. P. Rudy, M. Ryu, N. Sathaye, C. Schnabel, E. Schoute, K. Setia, Y. Shi, A. Silva, Y. Siraichi, S. Sivarajah, J. A. Smolin, M. Soeken, H. Takahashi, I. Tavernelli, C. Taylor, P. Taylour, K. Trabing, M. Treinish, W. Turner, D. Vogt-Lee, C. Vuillot, J. A. Wildstrom, J. Wilson, E. Winston, C. Wood, S. Wood, S. Wörner, I. Y. Akhalwaya, and C. Zoufal, "Qiskit: An open-source framework for quantum computing," 2019.

A Classical Perceptron

The Jupyter Notebook with the complete implementation can be found at :
`https://github.com/abhishekabhishek/COMP-4560-Industrial-Project/blob/master/Classical%20Perceptron.ipynb`

A.1 Single Qubit Case

1. Case 1 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

   (a) Python implementation :

```
# Case 1
i_vector = [1, 1]
w_vector = [1, 1]

out = 1 if sum([x*y for x,y in zip(i_vector, w_vector)]) > 0 else 0
print('Case 1 :\n Input i_0 = {0}, Input i_1 = {1}\n Weight w_0 = {2}, Weight w_1 = {3}\n Output = {4}\n'.format(
    i_vector[0], i_vector[1],w_vector[0],w_vector[1], out))
```

   (b) Output :

```
Case 1 :
 Input i_0 = 1, Input i_1 = 1
 Weight w_0 = 1, Weight w_1 = 1
 Output = 1
```

2. Case 2 : $\vec{i} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

   (a) Python implementation :

```
# Case 2
i_vector = [1, -1]
w_vector = [1, 1]

out = 1 if sum([x*y for x,y in zip(i_vector, w_vector)]) > 0 else 0
print('Case 2 :\n Input i_0 = {0}, Input i_1 = {1}\n Weight w_0 = {2}, Weight w_1 = {3}\n Output = {4}\n'.format(
    i_vector[0], i_vector[1],w_vector[0],w_vector[1], out))
```

   (b) Output :

```
Case 2 :
 Input i_0 = 1, Input i_1 = -1
 Weight w_0 = 1, Weight w_1 = 1
 Output = 0
```

3. Case 3 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

   (a) Python implementation :

```
# Case 3
i_vector = [1, 1]
w_vector = [1, -1]

out = 1 if sum([x*y for x,y in zip(i_vector, w_vector)]) > 0 else 0
print('Case 3 :\n Input i_0 = {0}, Input i_1 = {1}\n Weight w_0 = {2}, Weight w_1 = {3}\n Output = {4}\n'.format(
    i_vector[0], i_vector[1],w_vector[0],w_vector[1], out))
```

(b) Output :

```
Case 3 :
 Input i_0 = 1, Input i_1 = 1
 Weight w_0 = 1, Weight w_1 = -1
 Output = 0
```

4. Case 4 : $\vec{i} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

(a) Python implementation :

```
# Case 4
i_vector = [1, -1]
w_vector = [1, -1]

out = 1 if sum([x*y for x,y in zip(i_vector, w_vector)]) > 0 else 0
print('Case 4 :\n Input i_0 = {0}, Input i_1 = {1}\n Weight w_0 = {2}, Weight w_1 = {3}\n Output = {4}\n'.format(
    i_vector[0], i_vector[1],w_vector[0],w_vector[1], out))
```

(b) Output :

```
Case 4 :
 Input i_0 = 1, Input i_1 = -1
 Weight w_0 = 1, Weight w_1 = -1
 Output = 1
```

A.2 Double Qubit Case

1. Case 1 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$

(a) Python implementation :

```
# Case 1
i_vector = [1, 1, 1, 1]
w_vector = [1, 1, 1, 1]

out = 1 if sum([x*y for x,y in zip(i_vector, w_vector)]) > 0 else 0
print('Case 1 :\n Input i_0 = {0}, Input i_1 = {1}, Input i_2 = {2}, Input i_3 = {3}\n Weight w_0 = {4},'.format(
    i_vector[0], i_vector[1], i_vector[2], i_vector[3], w_vector[0])+
      ' Weight w_1 = {0}, Weight w_1 = {1}, Weight w_1 = {2}\n Output = {3}\n'.format(
      w_vector[1], w_vector[2], w_vector[3], out))
```

(b) Output :

```
Case 1 :
 Input i_0 = 1, Input i_1 = 1, Input i_2 = 1, Input i_3 = 1
 Weight w_0 = 1, Weight w_1 = 1, Weight w_1 = 1, Weight w_1 = 1
 Output = 1
```

2. Case 2 : $\vec{i} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$

(a) Python implementation :

```
# Case 2
i_vector = [1,-1, -1, -1]
w_vector = [1, 1, 1, 1]

out = 1 if sum([x*y for x,y in zip(i_vector, w_vector)]) > 0 else 0
print('Case 2 :\n Input i_0 = {0}, Input i_1 = {1}, Input i_2 = {2}, Input i_3 = {3}\n Weight w_0 = {4},'.format(
    i_vector[0], i_vector[1], i_vector[2], i_vector[3], w_vector[0])+
      ' Weight w_1 = {0}, Weight w_1 = {1}, Weight w_1 = {2}\n Output = {3}\n'.format(
      w_vector[1], w_vector[2], w_vector[3], out))
```

(b) Output :

```
Case 2 :
 Input i_0 = 1, Input i_1 = -1, Input i_2 = -1, Input i_3 = -1
 Weight w_0 = 1, Weight w_1 = 1, Weight w_1 = 1, Weight w_1 = 1
 Output = 0
```

3. Case 3 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$

(a) Python implementation :

```
# Case 3
i_vector = [1, 1, 1, 1]
w_vector = [1, -1, 1, -1]

out = 1 if sum([x*y for x,y in zip(i_vector, w_vector)]) > 0 else 0
print('Case 3 :\n Input i_0 = {0}, Input i_1 = {1}, Input i_2 = {2}, Input i_3 = {3}\n Weight w_0 = {4},'.format(
    i_vector[0], i_vector[1], i_vector[2], i_vector[3], w_vector[0])+
      ' Weight w_1 = {0}, Weight w_1 = {1}, Weight w_1 = {2}\n Output = {3}\n'.format(
      w_vector[1], w_vector[2], w_vector[3], out))
```

(b) Output :

```
Case 3 :
 Input i_0 = 1, Input i_1 = 1, Input i_2 = 1, Input i_3 = 1
 Weight w_0 = 1, Weight w_1 = -1, Weight w_1 = 1, Weight w_1 = -1
 Output = 0
```

4. Case 4 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$

   (a) Python implementation :

```
# Case 4
i_vector = [1, 1, -1, -1]
w_vector = [1, -1, -1, -1]

out = 1 if sum([x*y for x,y in zip(i_vector, w_vector)]) > 0 else 0
print('Case 4 :\n Input i_0 = {0}, Input i_1 = {1}, Input i_2 = {2}, Input i_3 = {3}\n Weight w_0 = {4},'.format(
    i_vector[0], i_vector[1], i_vector[2], i_vector[3], w_vector[0])+
      ' Weight w_1 = {0}, Weight w_1 = {1}, Weight w_1 = {2}\n Output = {3}\n'.format(
      w_vector[1], w_vector[2], w_vector[3], out))
```

   (b) Output :

```
Case 4 :
 Input i_0 = 1, Input i_1 = 1, Input i_2 = -1, Input i_3 = -1
 Weight w_0 = 1, Weight w_1 = -1, Weight w_1 = -1, Weight w_1 = -1
 Output = 1
```

B Quantum Perceptron

The Jupyter Notebook with the complete implementation can be found at : `https://github.com/abhishekabhishek/COMP-4560-Industrial-Project/blob/master/Quantum%20Perceptron.ipynb`

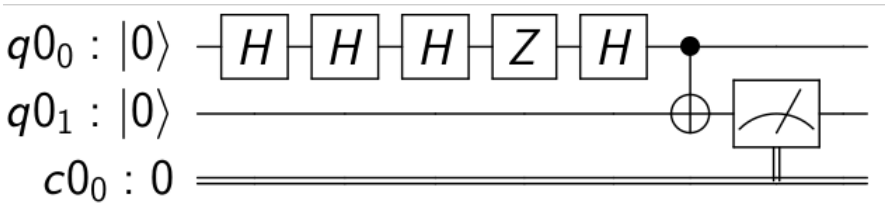
B.1 Single Qubit Case

1. Case 1 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

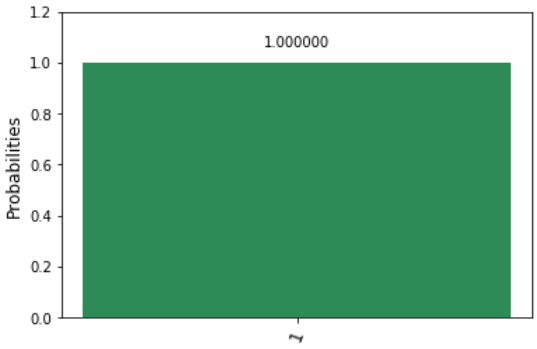
   (a) Python Implementation :

```
# Initialize the Quantum Circuit

# Number of quantum registers
qreg = 2

# Number of classical registers
creg = 1

# Create a quantum register "qr" with qreg qubits
qr = QuantumRegister(qreg)

# Create a classical register called "cr" with creg bits
cr = ClassicalRegister(creg)

# Create a Quantum Circuit involving "qr" and "cr"
circuit_1 = QuantumCircuit(qr, cr)

# Add the operations to the Quantum circuit

# Add a Hadamard gate to put the single qubit in a superposition
circuit_1.h(qr[0])

# Apply a Hadamard gate to neutralize the application of the weight unitary matrix
circuit_1.h(qr[0])

# Apply a NOT to put the qubits in the state |m-1> which stores the result of the dot product
circuit_1.h(qr[0])
circuit_1.z(qr[0])
circuit_1.h(qr[0])

# Apply the controlled not on the original qubit and the ancilla
circuit_1.cx(qr[0], qr[1])

# Measure the last qubit in the sequence
circuit_1.measure(qr[1], cr[0])

# Draw the given circuit
circuit_drawer(circuit_1)
```

   (b) Quantum Circuit :



   (c) Output :

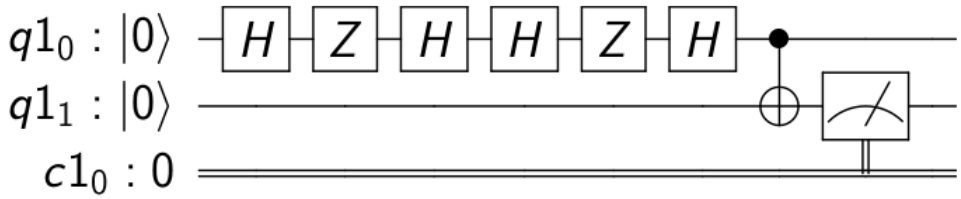


2. Case 2 : $\vec{i} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

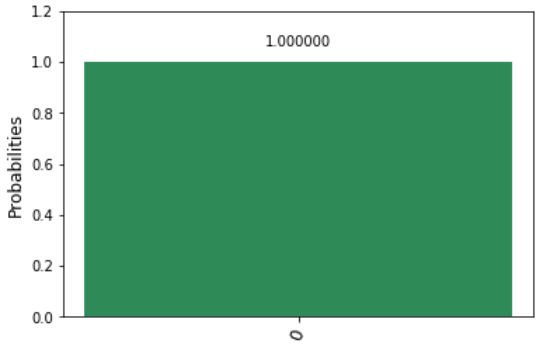
   (a) Python Implementation :

```
# Initialize the Quantum Circuit

# Number of quantum registers
qreg = 2

# Number of classical registers
creg = 1

# Create a quantum register "qr" with qreg qubits
qr = QuantumRegister(qreg)

# Create a classical register called "cr" with creg bits
cr = ClassicalRegister(creg)

# Create a Quantum Circuit involving "qr" and "cr"
circuit_1 = QuantumCircuit(qr, cr)

# Add the operations to the Quantum circuit

# Add a Hadamard gate to put the single qubit in a superposition
circuit_1.h(qr[0])

# Apply the unitary matrix for the input ( Apply a -1 phase to the second qubit )
circuit_1.z(qr[0])

# Apply a Hadamard gate to neutralize the application of the weight unitary matrix
circuit_1.h(qr[0])

# Apply a NOT to put the qubits in the state |m-1>
circuit_1.h(qr[0])
circuit_1.z(qr[0])
circuit_1.h(qr[0])

# Apply the controlled not on the original qubit and the ancilla
circuit_1.cx(qr[0], qr[1])

# Measure the last qubit in the sequence
circuit_1.measure(qr[1], cr[0])

# Draw the given circuit
circuit_drawer(circuit_1)
```

   (b) Quantum Circuit :



(c) Output :

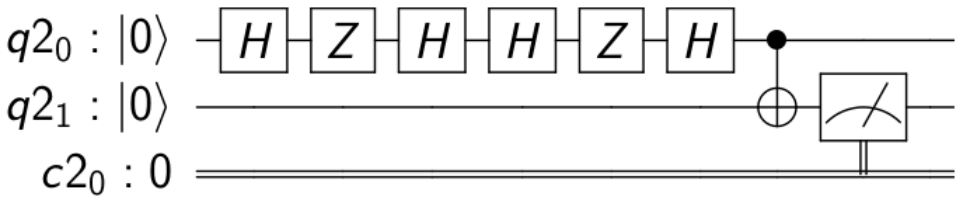


3. Case 3 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

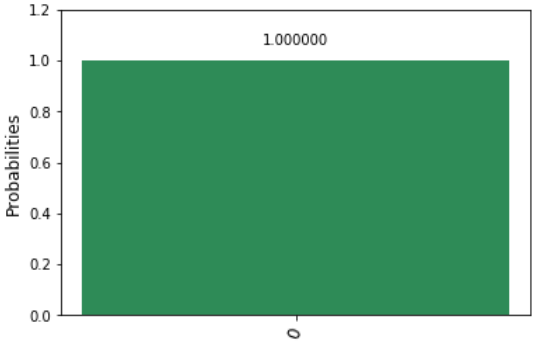
(a) Python Implementation :

```
# Initialize the Quantum Circuit

# Number of quantum registers
qreg = 2

# Number of classical registers
creg = 1

# Create a quantum register "qr" with qreg qubits
qr = QuantumRegister(qreg)

# Create a classical register called "cr" with creg bits
cr = ClassicalRegister(creg)

# Create a Quantum Circuit involving "qr" and "cr"
circuit_1 = QuantumCircuit(qr, cr)

# Add the operations to the Quantum circuit

# Add a Hadamard gate to put the single qubit in a superposition
circuit_1.h(qr[0])

# Apply the unitary matrix for the weight ( Apply a -1 phase to the second qubit )
circuit_1.z(qr[0])

# Apply a Hadamard gate to neutralize the application of the weight unitary matrix
circuit_1.h(qr[0])

# Apply a NOT to put the qubits in the state |m-1>
circuit_1.h(qr[0])
circuit_1.z(qr[0])
circuit_1.h(qr[0])

# Apply the controlled not on the original qubit and the ancilla
circuit_1.cx(qr[0], qr[1])

# Measure the last qubit in the sequence
circuit_1.measure(qr[1], cr[0])

# Draw the given circuit
circuit_drawer(circuit_1)
```

(b) Quantum Circuit :



(c) Output :

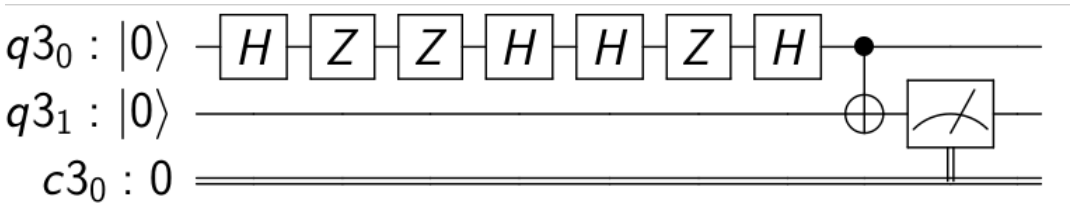


4. Case 4 : $\vec{i} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

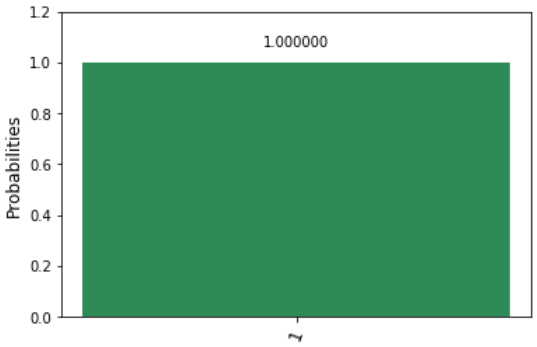
   (a) Python Implementation :

```
# Initialize the Quantum Circuit

# Number of quantum registers
qreg = 2

# Number of classical registers
creg = 1

# Create a quantum register "qr" with qreg qubits
qr = QuantumRegister(qreg)

# Create a classical register called "cr" with creg bits
cr = ClassicalRegister(creg)

# Create a Quantum Circuit involving "qr" and "cr"
circuit_1 = QuantumCircuit(qr, cr)

# Add the operations to the Quantum circuit

# Add a Hadamard gate to put the single qubit in a superposition
circuit_1.h(qr[0])

# Apply the unitary matrix for the input ( Apply a -1 phase to the second qubit )
circuit_1.z(qr[0])

# Apply the unitary matrix for the weight ( Apply a -1 phase to the second qubit )
circuit_1.z(qr[0])

# Apply a Hadamard gate to neutralize the application of the weight unitary matrix
circuit_1.h(qr[0])

# Apply a NOT to put the qubits in the state |m-1>
circuit_1.h(qr[0])
circuit_1.z(qr[0])
circuit_1.h(qr[0])

# Apply the controlled not on the original qubit and the ancilla
circuit_1.cx(qr[0], qr[1])

# Measure the last qubit in the sequence
circuit_1.measure(qr[1], cr[0])

# Draw the given circuit
circuit_drawer(circuit_1)
```

   (b) Quantum Circuit :



(c) Output :

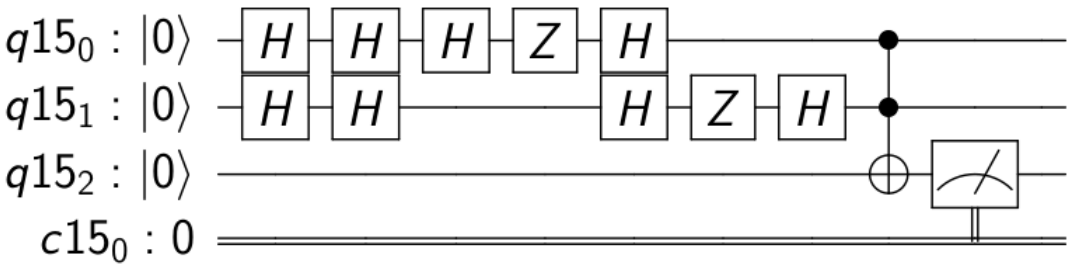


Due to the limitation of space, we will not include the python implementation here. However, the complete implementation is Python can be found in the link given above.

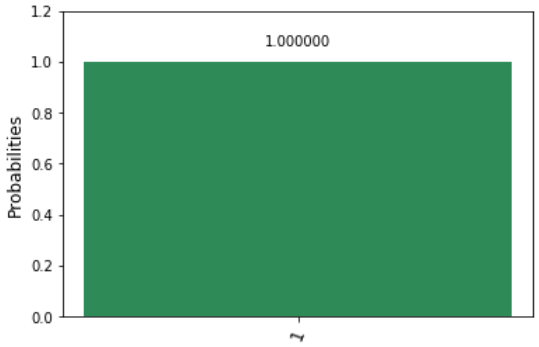
B.2 Double Qubit Case - Using Sign Flip Blocks

1. Case 1 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$

(a) Quantum Circuit :

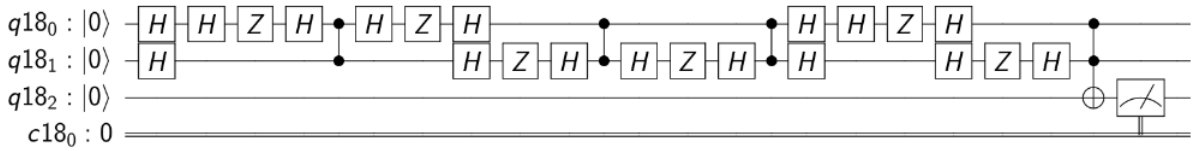


(b) Output :

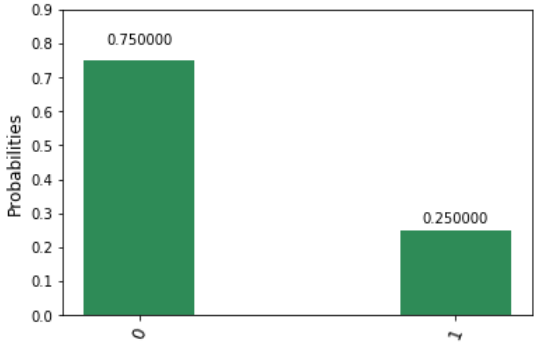


2. Case 2 : $\vec{i} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$

   (a) Quantum Circuit :

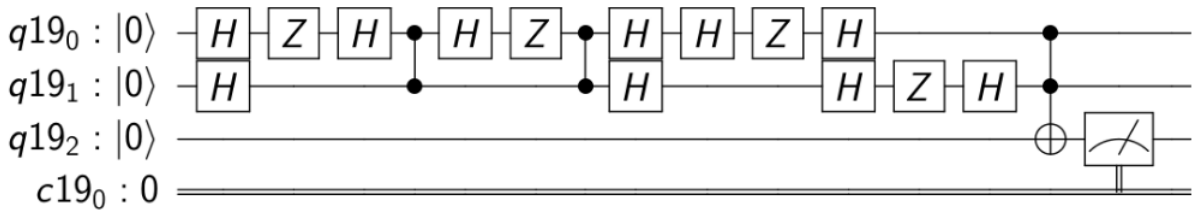


   (b) Output :

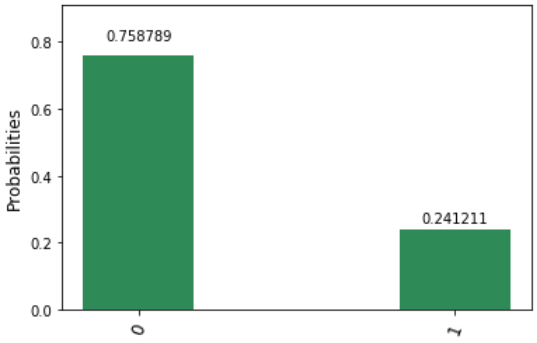


3. Case 3 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$

   (a) Quantum Circuit :

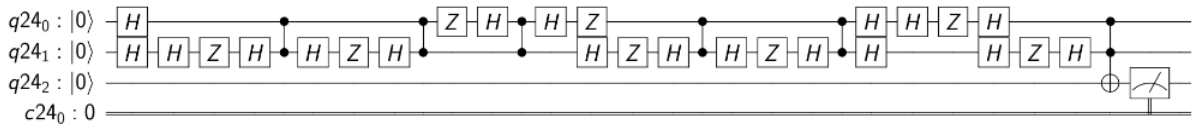


(b) Output :

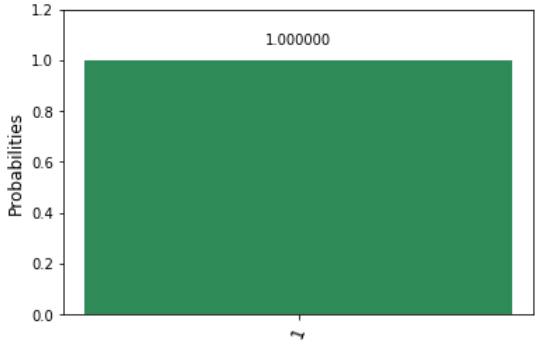


4. Case 4 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$

(a) Quantum Circuit :



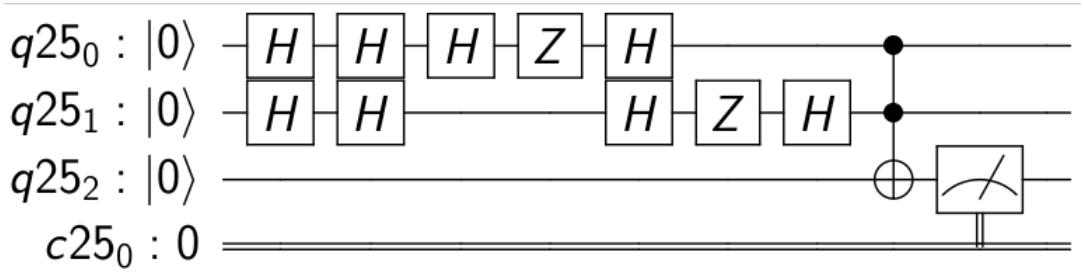
(b) Output :



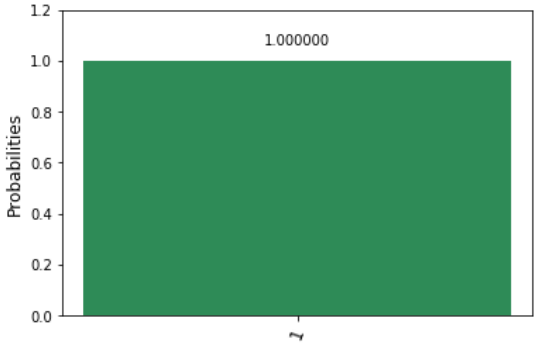
B.3 Double Qubit Case - Using Hypergraph State Generation Subroutine (HSGS)

1. Case 1 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$

   (a) Quantum Circuit :

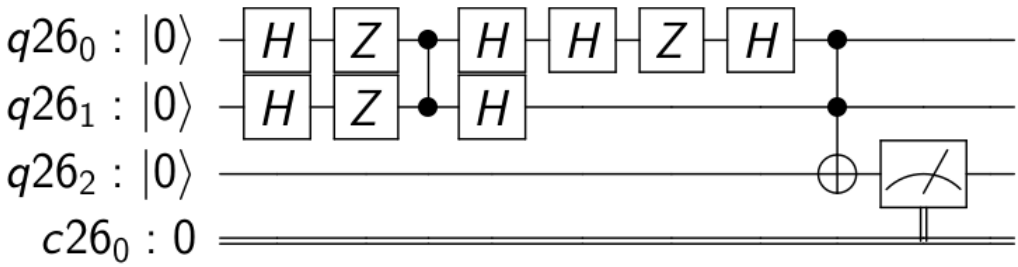


   (b) Output :

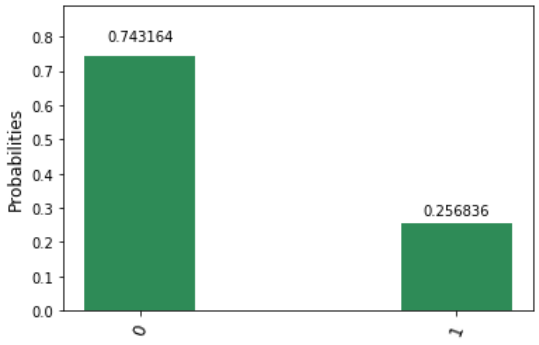


2. Case 2 : $\vec{i} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$

   (a) Quantum Circuit :

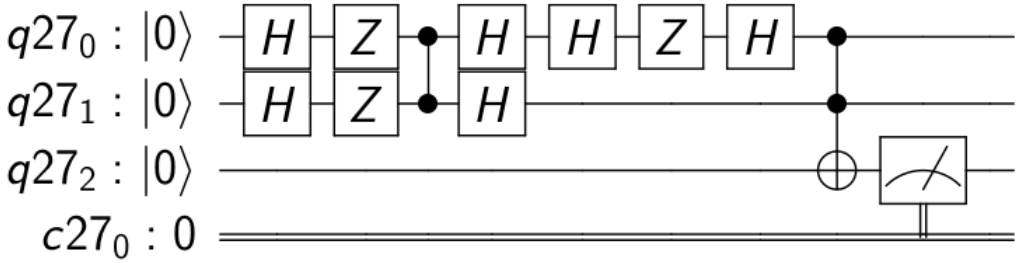


   (b) Output :

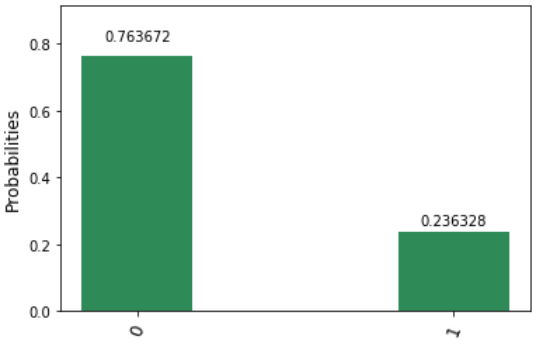


3. Case 3 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$

   (a) Quantum Circuit :

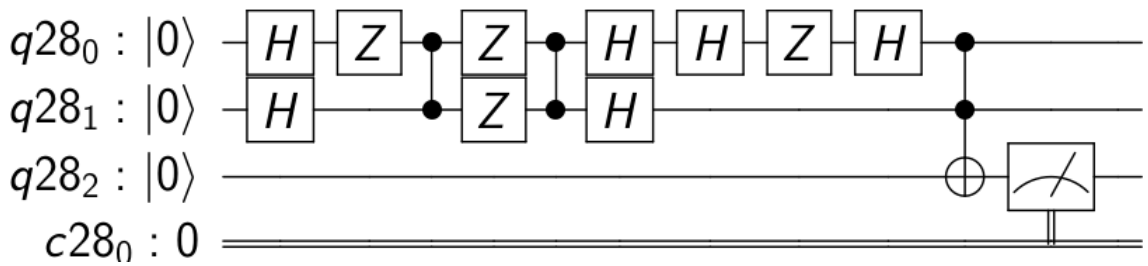


   (b) Output :

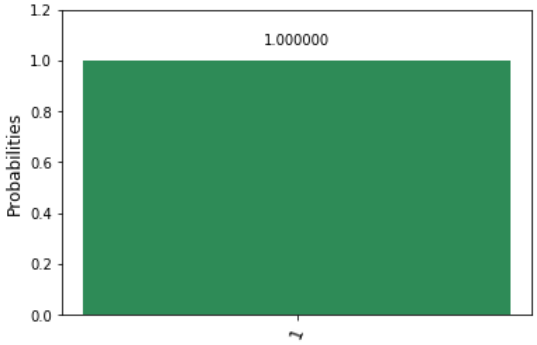


4. Case 4 : $\vec{i} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}, \vec{w} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$

   (a) Quantum Circuit :


$q28_0 : |0\rangle$
$q28_1 : |0\rangle$
$q28_2 : |0\rangle$
$c28_0 : 0$
H
Z
Z
H
H
Z
H
H
Z
H
H

(b) Output :


1.000000
Probabilities
1.2
1.0
0.8
0.6
0.4
0.2
0.0
1