

Simulating a Quantum Perceptron

Collin Farquhar

M.Eng. Project, Applied and Engineering Physics

Cornell University

Advised by Dr. Bart Selman

My M.Eng. project is an extension of a CS 6700 project I did with Nicholas Kaye, who helped author the first section “Quantum Perceptron”, which recreates and open-sources the code outlined in the paper *An Artificial Neuron Implemented on an Actual Quantum Processor* by Tacchino et al.

The M.Eng. project is an entirely original extension of this work that goes beyond the paper that I solely worked on and authored, and is the content of “A One-Qubit Perceptron”. Because this work is an extension, it would be incomprehensible without the context and groundwork of the original, which is why the first section is included in this report.

All code discussed can be found at

<https://github.com/Farquhar13/QuantumComputing/tree/master/QuantumPerceptron>

Part 1-3 focus on recreating and explaining elements of the Tacchino et al paper. *1-qubit perceptron* is my original work creating a 1-qubit version of the perceptron algorithm and examining its performance on boolean functions.

Quantum Perceptron

Description

In classical computing, the origin of neural networks can be traced to the perceptron algorithm, introduced by Frank Rosenblatt in 1957. The perceptron is a linear classifier that can sort an input vector, \vec{i} , to a binary output, by calculating the inner product between the input vector, \vec{i} , and some weight vector, \vec{w} , and applying activation and threshold functions. While such an algorithm appears very limited in its abilities, they are the fundamental building blocks of much more complex neural networks, which are made of many layers of this perceptron construction. However, as a neural network grows more complex, the computational cost increases drastically. Concerns of computational complexity of various modern neural networks will no doubt shape the future of the field.

Equal amounts of data can be encoded in exponentially fewer quantum bits (qubits) than classical bits. This may present an advantage in the implementation of the quantum perceptron, and potentially a quantum implementation of a neural network at a later stage. Quantum encoding of the data for such a mechanism may allow for a drastic reduction in required storage for the data and network. Furthermore, the language of quantum mechanics represents quantum states as complex valued vectors and matrices as operators acting on such objects. Due to a shared basis in linear algebra representation, it is possible to transform similar operations used in classical networks to equivalent operations for a quantum network.

Additionally, due to the nature of data encoding and measurement in quantum mechanics, the perceptron acts as a pattern classifier, rather than a linear classifier as in classical algorithm. Thus, more complex patterns such as XOR, that are not solvable by a single classical perceptron, are solvable by a single quantum perceptron. This may allow for much more complex problems to be solved by relatively simpler quantum networks in comparison to classical networks. However, with our particular encoding scheme, this advantage is data dependent as there are limitations of the quantum perceptron for which the classical algorithm performs better.

There are two quantum properties, superpositions and entanglement, that give promise to the advantage of quantum computation. While a classical bit must be in a state of either 0 or 1, a quantum bit may be in a superposition of 0 and 1. Upon measurement this bit will collapse to one of these states, but prior to that it can “*be*” in both simultaneously.

A qubit, as described below in vector representation, will collapse to state 0 with probability $|\alpha|^2$ and to state 1 with probability $|\beta|^2$, where α and β are elements of the set of complex numbers, for which the absolute squares sum to 1.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \qquad |\alpha|^2 + |\beta|^2 = 1$$

This means that, when raised to a series of n classical bits, there are 2^n possible states those classical bits may be in. Classically, these bits can only be in one of those states at a time. Quantumly, however, a qubit over these bits can *be* in all 2^n states simultaneously and collapse to a single state upon measurement. Ergo, if we can control the probability of collapse to certain states with clever quantum algorithms, we may see that quantum computing offers stark increases in computational efficiency for modeling complex networks.

Quantum mechanics also allows for entanglement between qubits. Essentially, this entanglement is a stronger form of correlation between qubits. With two qubits Ψ_0 and Ψ_1 , defined below, we get the following definition for entanglement (where, more technically, the final state cannot be factored as a product of two composite states).

$$|\Psi_0\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle, \quad |\Psi_1\rangle = \beta_0|0\rangle + \beta_1|1\rangle$$

$$|\Psi_0\rangle|\Psi_1\rangle = \alpha_0\beta_0|00\rangle + \alpha_0\beta_1|01\rangle + \alpha_1\beta_0|10\rangle + \alpha_1\beta_1|11\rangle$$

In this paper, we shall investigate the potential for a 2-qubit quantum perceptron and how these advantages of quantum computing may be used in practice on some simple problems.

Structure and encoding information

The first step in the creation of a quantum implementation of the perceptron algorithm is to encode all of the classical data into a quantum representation. We will be creating a 2-qubit

quantum perceptron over binary inputs $\{-1,1\}$. With $N=2$ qubits, we are able to model a classical vector of dimension $m=2^N=4$. We will also be using binary inputs to our vectors, so these 4 dimensions span a possible $2^m=16$ distinct states.

For each of these distinct states, we can define a quantum state vector $|\Psi_i\rangle$ such that:

$$|\Psi_i\rangle = \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} i_j |j\rangle \quad [5]$$

In this notation, m is the dimensionality of the classical input, i_j is an element of the classical input vector, and $|j\rangle$ is a computational basis state.

For example, in our 4 dimensional case, we may have a classical input vector $[1,1,-1,1]$. In our encoding of our data to a quantum vector, this will be transformed to a quantum state vector with probability amplitudes $[0.5,0.5,-0.5,0.5]$.

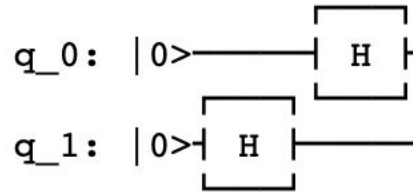
A drawback of quantum computing is that there is no access to reading in or reading out values in a program. Therefore, we must adjust the program to run for different inputs. This is similar to a program where all variables are assigned at the start of the program, as opposed to variables that can be defined and changed throughout the program^[5]. This idea makes intuitive sense if these quantum programs are seen as their equivalent quantum circuits. So, in order to run the circuit on different inputs, there must be a method to adjust the circuit to our inputs.

Before any further circuit is defined, our quantum state vector must be encoded at the beginning of the circuit. We will implement a *Brute Force Sign-Flip*^[5] function to encode quantum state vectors, which will be able to encode all of 2^m states corresponding to the vectors indexed by $k=0$ to $k=2^m-1$. We want a function such that for every -1 value in the classical input vector, a -1 factor will be applied to the corresponding index in the quantum state vector.

We start by first initializing the state for k_0 , by creating parallel Hadamard gates. This creates an equal superposition $[0.5,0.5,0.5,0.5]$:

$$|00\rangle \rightarrow |+\rangle^{\otimes 2}$$

with circuit representation:

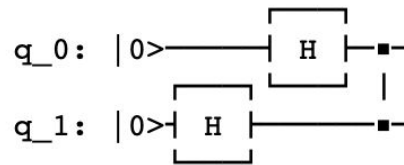


We will start every circuit with this initial superposition and create the states k_i , $i > 0$, by building upon this circuit.

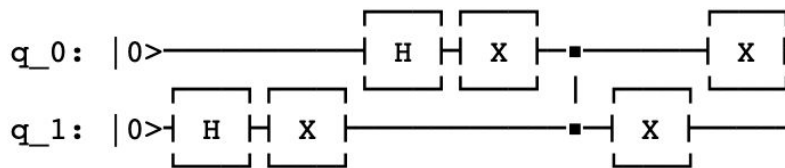
This algorithm uses X gates, which takes $|0> \rightarrow |1>$ and $|1> \rightarrow |0>$, and Controlled Z (CZ) gates, which switches the sign of $|11>$.

For example, below is the circuit representation of k_1 , k_8 , and k_9 . Observe that the circuit $k_9 = k_8 + k_1$. This gives us a way to inductively construct the more complex circuits.

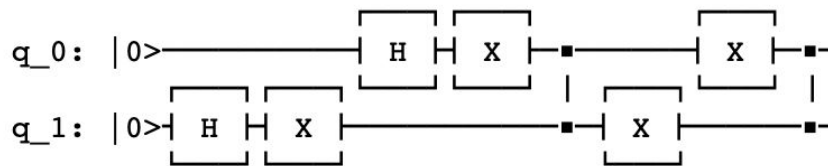
$$k_1 = [0.5, 0.5, 0.5, -0.5]$$



$$k_8 = [-0.5, 0.5, 0.5, 0.5]$$



$$k_9 = [-0.5, 0.5, 0.5, -0.5]$$



The *Brute Force Sign-Flip* algorithm takes in a classical binary, $\{-1, 1\}$, input vector \vec{i} and will output the circuit encoding that vector \vec{i} .

Wherever there is a -1 element in the input vector \vec{i} , use that index to identify the corresponding computational basis vector. The algorithm then sandwiches the two qubits in the 0 state with X gates. In between these gates, a Controlled Z gate must be applied. This algorithm

will return the circuit corresponding to the given binary input vector \vec{i} . That the source code faithfully performs this function is demonstrated explicitly in the source code of the jupyter notebook *Part 1*.

Through analysis of the algorithm and looking at the specific circuits that must be created, one can see that the worst cases – high values of k – require circuits that scale exponentially in depth.

This algorithm will return the circuit for k_i in isolation. However, we want to be able to combine these circuits with others in series. For that reason, when we use this encoding algorithm in later stages, the initialization of k_0 with parallel Hadamard gates will be excluded. This will allow these circuits to be used as parts of more complex composite operations.

The function *SF_encoder* is able to generate and combine these circuits to algorithmically generate a quantum classifier. For such a classifier an additional ancillary qubit is added to the circuits, which is only used in the final step for measurement.

Perceptron

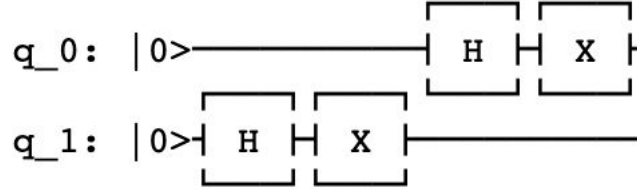
Now that we have a method to encode an input vector in a quantum state vector, and accompanying circuit, we can create the basic structure of the perceptron algorithm. The classical perceptron algorithm consists of three essential steps:

1. Take the inner product of the input vector, \vec{i} , and weight vector, \vec{w} .
2. Apply an activation function to the inner product.
3. Compare against some threshold for perceptron activation to create the label.

If we are given a binary input vector, \vec{i} , and weight vector, \vec{w} , then we can encode both of these states and use them in the perceptron algorithm. We need to add an operator, a circuit, U_w , that will take the inner product between the input state vector and the weight state vector.

We can accomplish by implementing a unitary matrix U_w that has the effect of rotating $|\Psi_w\rangle$ to the $|1\rangle^{\otimes N}$ computational basis state and rotate $|\Psi_i\rangle$ by the same angle. This projects the rotated $|\Psi_i\rangle$ onto the rotated $|\Psi_w\rangle$. Therefore, the inner product will be stored as the amplitude of the $|1\rangle^{\otimes N}$ computational basis state.

The circuit for such an operation is shown below:



The following example, where $\vec{i} = \vec{w}$ and therefore $|\Psi_i\rangle = |\Psi_w\rangle$, shows how this rotation will work. This case of $\vec{i} = \vec{w}$, will give a perfect activation of the perceptron.

$$\begin{aligned}
 H^{\otimes N} |0\rangle &\rightarrow |\Psi_0\rangle \\
 U_i |\Psi_0\rangle &\rightarrow |\Psi_i\rangle \\
 U_w |\Psi_i\rangle &\rightarrow |\Psi_0\rangle \\
 H^{\otimes N} |\Psi_0\rangle &\rightarrow |0\rangle^{\otimes N} \\
 X^{\otimes N} |0\rangle^{\otimes N} &\rightarrow |1\rangle^{\otimes N}
 \end{aligned}$$

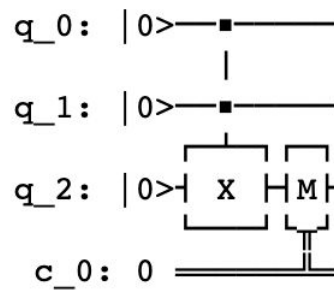
The first step of this explanation takes simply initializes the circuit to the superposition state by using parallel Hadamard gates (line 1), like previous methods. By applying the operator U_i to $|\Psi_0\rangle$, the superposition state $|\Psi_0\rangle$ is taken to the encoded input state $|\Psi_i\rangle$ (line 2).

Since $|\Psi_i\rangle = |\Psi_w\rangle$, then $U_w = U_i$ must also hold since both operations must take the superposition state $|\Psi_0\rangle$ to the same encoded input state $|\Psi_i\rangle = |\Psi_w\rangle$. Additionally, since all quantum operations are reversible, applying U_w to $|\Psi_i\rangle = |\Psi_w\rangle$ will result in the superposition state (line 3).

By the same logic applying the parallel Hadamard gate operation to this superposition state $|\Psi_0\rangle$ will return to $|0\rangle^{\otimes N}$ (line 4). Finally, by applying X gates to this penultimate state, the state $|0\rangle^{\otimes N}$ will be transformed into $|1\rangle^{\otimes N}$ (line 5). This state corresponds to perfect activation of the perceptron since $\vec{i} = \vec{w}$. In cases where of $\vec{i} \neq \vec{w}$, the amplitude of this state will correspond to the inner product of the two vectors in the quantum states.

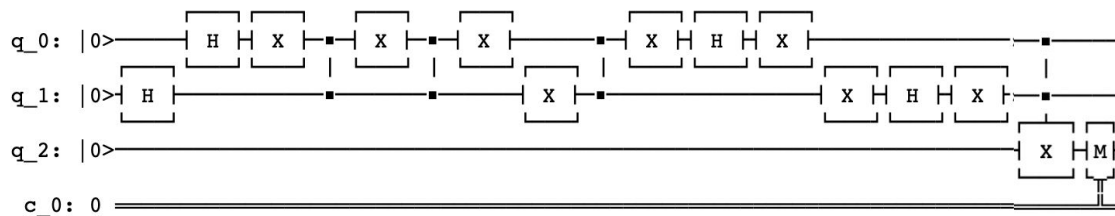
The key difference between the quantum perceptron algorithm and the classical perceptron algorithm is the activation function. The original classical perceptron algorithm is limited by a linear activation function, which limits its scope to linearly-separable data. The classical perceptron can be implemented with nonlinear activation functions, but these can become complex to compute, especially across many different layers in a neural network.

In a quantum perceptron, however, we can use measurement as our activation function, since measurement is an observation of a quadratic probabilistic function. Thus, a simple measurement of a perceptron in the $|1\rangle^{\otimes N}$ state can detect nonlinear patterns. To measure the amplitude of the $|1\rangle^{\otimes N}$ state we can use a multi-controlled-X gate using the two qubits upon which we have done the previous operations and output to the third (ancillary) qubit. We can then measure the third qubit and observe its state – this observation is returned as the output of the circuit. The measurement circuit is created as below:



Now that we have the circuits for all of the necessary steps (1. Initialize, 2. Encode input vector, 3. Encode weight vector, 4. Inner product, 5. Measurement), we can combine these 5 circuits to create our overall perceptron circuit/program.

Below is an example of this measurement of the inner product between vectors k_3 and k_8 . The perceptron will be activated if the two vectors are equivalent and will not be activated if the two vectors are different. Thus, for k_3 and k_8 , we would expect our program and circuit to output 0 and remain inactivated.

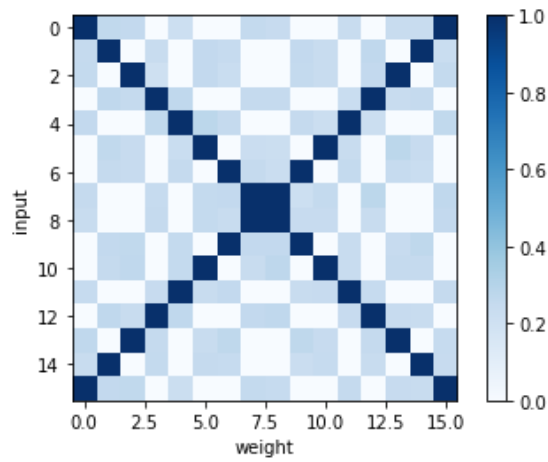


Since these outputs are over a probability distribution, we will sample the circuit and our final output will be the most common output, between 0 and 1. When this circuit is sampled 1024 times, we find that the outputs follow the following distribution: $\{0:738, 1:286\}$.

Assuming a reasonable threshold of 0.5, the node would not be activated and 0 would be returned.

Evaluation

We have observed that this program and accompanying circuit are successful for the single example shown above. However, we will perform a more rigorous test over a wider array of values. We will test the method over all combinations of k_i and k_j for $i, j = 0, \dots, 15$. This results in the following table:



From this graph, we can see that the quantum perceptron implementation is not limited to a linear separator as the classical perceptron implementation is. This nonlinearity would allow such a perceptron to correctly classify the XOR dataset. This dataset poses problems to the classical implementation, which is limited to linear problems.

This nonlinearity is induced by the measurement of the quantum system, which is over a quadratic probability distribution, and as a consequence of the quantum treatment of global phase. We observe that a vector \vec{i} and its negative $-\vec{i}$ are equivalent in this representation. This equality comes from the fact that in quantum mechanics, global phase is physically insignificant for these quantum systems, so $\Psi = -\Psi$.

Thus, we have a successful perceptron algorithm for any input vector, \vec{i} , and a precalculated weight vector, \vec{w} . The next step is to create a method to actually learn the weight vector for the algorithm, to not be reliant on a precomputed vector.

Weights

The final step in the development of a quantum perceptron is a method to train on labeled input data and calculate the weight vector, \vec{w} . The method to train and calculate the weight vector is extremely similar in structure to the classical perceptron algorithm.

In the classical perceptron algorithm, the weight vector is calculated in the following manner:

1. Randomly initialize the weight vector, \vec{w} .
2. Loop through training data (\vec{x}, y) :

If a data point is misclassified ($y_i \neq f(x(\vec{i}))$):

$$\vec{w} = \vec{w} + y_i \cdot x(\vec{i})$$

3. Repeat until all training data is correctly labeled

For the quantum perceptron, the weight vector is calculated in the following manner:

1. Randomly initialize the weight vector, \vec{w} with elements $\in \{-1, 1\}$.
2. Loop through the training data (\vec{x}, y) :

If a data point is misclassified:

If training example classified as positive, not negative:

Move \vec{w} closer to \vec{i}

Randomly flip a fraction, l_n , of the signs where \vec{w} and \vec{i} are equal.

If training example classified as negative, not positive:

Move \vec{w} closer to $-\vec{i}$

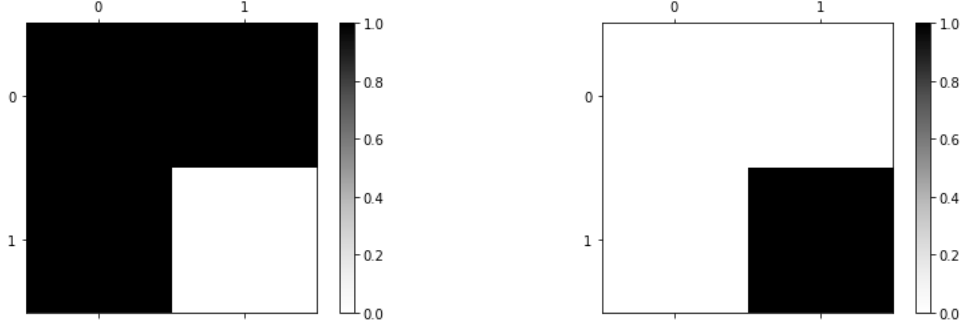
Randomly flip a fraction, l_p , of the signs where \vec{w} and \vec{i} differ.

3. Repeat until all training data is correctly

A key point to remember, is that due to global phase, for a given pattern the perceptron is learning, the negative of the pattern is treated equally. This will be evident in the evaluation of the weight vector below.

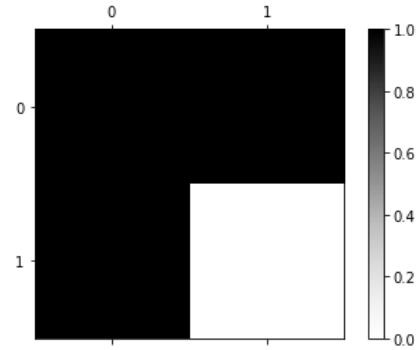
To evaluate the accuracy of this method, we will attempt to learn the pattern corresponding to $\vec{i} = \mathbf{k}_1 = [1, 1, 1, -1]$. Such a pattern corresponds to the plot below, on the left.

Additionally, due to global phase, the negative of \vec{i} , to $-\vec{i} = k_{14} = [-1, -1, -1, 1]$, will also be labeled positively. The plot on the right refers to this vector k_{14} , which must be adjusted in the vector of correct labels.



When we train the weight vector for the above pattern, the matrix of inputs is $[k_0, \dots, k_{15}]$, with label 1 for the above vectors k_1, k_{15} , and -1 for all other vectors. When our method is used on this training data, we get the following output and corresponding graph:

```
starting w: [ 1 -1 -1 -1]
w: [ 1  1  1 -1]
converged in 2 epochs
w: [ 1  1  1 -1]
```



Clearly our method has quickly converged to the correct pattern after being initialized to a random weight vector. This is the final step necessary to creating a 2-qubit quantum perceptron. This perceptron is effective over classical four-dimensional binary data.

Conclusion

In this paper we have outlined a method by which to implement a 2-qubit quantum perceptron. The ability to take advantage of the quantum features of superposition and entanglement, may yield benefits over the classical implementation. The first of these benefits is the potential to drastically decrease the amount of storage required to hold the data. Encoding the data in qubits rather than classical bits can exponentially decrease the total necessary memory.

More important for the functionality of the perceptron, is that the quantum implementation can take advantage of non-linear properties of measurement and global phase over quantum states. While the classical perceptron is limited by its linear constraint, measurement for this new implementation is over a quadratic probability distribution. Thus, our implementation of the 2-qubit quantum perceptron can correctly classify any single pattern and it's negative, which is not possible for a single classical perceptron.

Further extensions of this study could examine how this implementation scales with noise. Through our evaluation, we did not include any noise in our experiments. Another extension could be to design a rudimentary neural net by using combinations of these quantum perceptrons. Quantum neural networks are still quite cutting edge and it is not yet clear how an optimal quantum neural network could improve efficiency in terms of data storage, training cycles, and convergence.

This work is inspired by the theoretical and algorithmic work in the paper *An artificial neuron implemented on an actual quantum processor*, by Tacchino et al.

Code

The source code for our implementation can be found in parts 1-3 at
<https://github.com/Farquhar13/QuantumComputing/tree/master/QuantumPerceptro>

A One-Qubit Perceptron

Motivation

In recreating and open-sourcing the code for the perceptron algorithm proposed in *An artificial neuron implemented on an actual quantum processor*, by Tacchino et al, I had the idea to examine if it was possible to implement the quantum perceptron algorithm using only a single qubit. With some modifications, this proved to be possible. Implementing the algorithm on a single qubit is the clearest possible demonstration of the differences between the capabilities and limitations of the quantum algorithm in comparison to the classical algorithm.

Furthermore, a 1-qubit perceptron takes as input 2-dimensional classical data and returns a binary output $\{-1, 1\}$. This allows one to easily test and analyze the performance of the quantum perceptron on a set of boolean functions that take 2-bit input. This is convenient and elucidating as the performance of boolean functions are well tested in the classical algorithm. Notably, the inability of the classical algorithm to solve nonlinear problems, such as an XOR data set, profoundly changed the way these algorithms viewed by both the public and academics, leading to what is known as the AI winter. The one-qubit perceptron allows us to most fundamentally explore the nonlinearity of the quantum algorithm to see if the data sets such as boolean XOR are learnable, and what other limitations there may be.

One-qubit structure and modifications

The formulation of quantum state vector for 1-qubit:

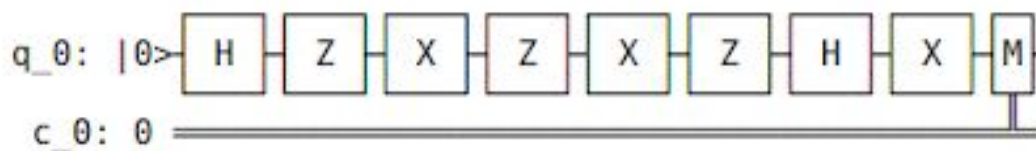
$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \qquad |\alpha|^2 + |\beta|^2 = 1$$

In the original perceptron algorithm proposed by Tacchino et al, the *Brute force sign flip* algorithm uses a combination of X, and CZ gates (gates / unitary transformations / operators / matrices) to encode classical information in quantum state vectors. The algorithm was designed so that by applying the correct sequence of gates the information about the inner product between two vectors would be contained in the amplitude of the $|1\rangle^{\otimes N}$ state. Thus, a multi-controlled-X gate could be applied using the encoding qubits as “control” and an

additional ancillary qubit as the “target”. Finally, the ancillary qubit can be measured as the output of the circuit and that information is ultimately used for classification.

Using only a single qubit, one can limit the set of gates needed to just X and Z. We may replace the (multi-)controlled-gates with single qubit gates. Furthermore, because the entire state is represented by the coefficient of the $|1\rangle$ computational basis state (β as in the above formulation), there is no longer a need for the (multi-)controlled-X and the use of an extra ancillary qubit. Indeed, all of the quantum perceptron algorithm for 2-dimensional classical input can be done on a single qubit using single-qubit-gates.

The design of the quantum circuits need to encode all $m=2$ -dimensional classical input ($2^m=4$) vectors is demonstrated in the jupyter notebook *1-qubit perceptron* where the circuits construction is stored in a dictionary with can be thought of as a database look-up. It can then be seen, by using these encoding circuits in composition, a circuit for all combinations of input and weight vectors can be algorithmically constructed. Below an example of the circuit for a 1-qubit classifier encoding the input vector indexed by k_1 and weight vector indexed by k_3 .

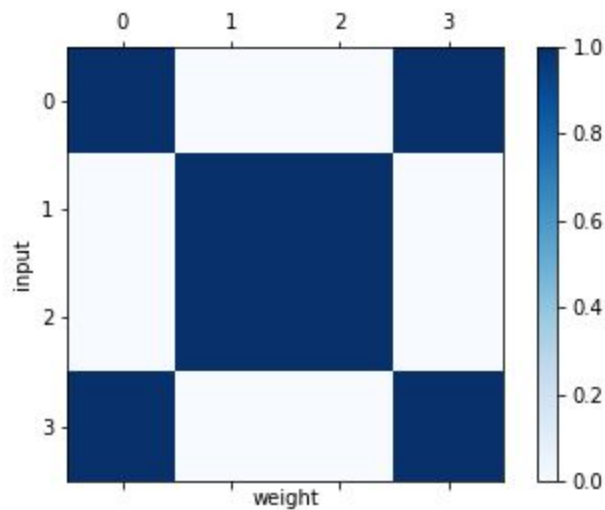


There are 16 possible combinations of two 2-dimensional classical vectors (input and weight combinations). As predicted by the general quantum algorithm, the 1-qubit quantum classifier should only activate when the quantum state vectors that encode the input and weights are equal or negatives of each other.

$$|\Psi_i\rangle = |\Psi_w\rangle$$

$$|\Psi\rangle = -|\Psi\rangle$$

The proper function of the 1-qubit classifier is demonstrated by testing all possible combinations of input and weight vector and is shown below.



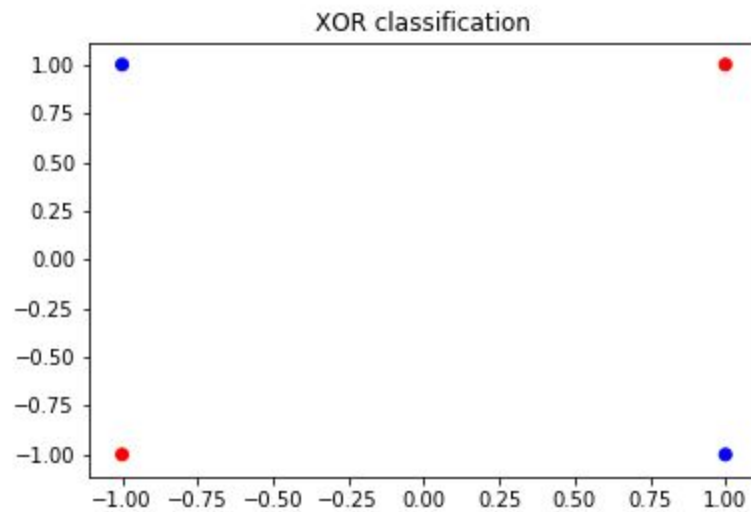
Learning weights

The nonlinearity of quantum measurement and the consequence of global phase produce some surprising results as to what functions can and cannot be learned by a 1-qubit quantum classifier with this binary $\{-1, 1\}$ encoding scheme. We will begin by using the training algorithm originally defined by Tacchino et al.

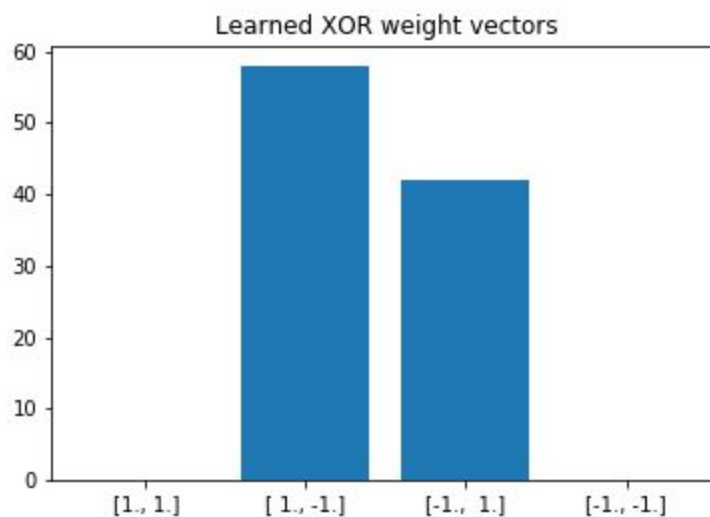
First we try, XOR data, a function that is not learnable for the classical algorithm.

input	XOR
00	0
01	1
10	1
11	0

Running the training algorithm, it is shown in the jupyter notebook *1-qubit-perceptron* that the algorithm does in fact converge to a learned weight vector. Using this weight vector, we can show the classification results for all inputs. We comparing coordinates from the table above to graph below, 0 is mapped to 1 and 1 is mapped to -1. Red points are classified positively, and blue points are classified negatively.



We see that the perceptron accurately learned and classified an XOR data set. While the nonlinearity of quantum measurement does allow the classifier to detect complex patterns, the particular ability to classify the XOR data is due to the treatment of global phase in quantum systems (a quantum state vector is indistinguishable from its negative). For the XOR data this means that two possible weight vectors may be returned.



The classification abilities and unavoidable effects of global phase for our encoding scheme does limit the effectiveness of the 1-qubit quantum perceptron on other data sets. Take for example the boolean AND function.

input	AND
00	0
01	0
10	0
11	1

This data set is in fact not classifiable on the quantum perceptron with this encoding scheme. This is due to global phase treating the (1,1) input and the (0,0) input equally. Thus the classifier cannot return a positive classification for an input of (1,1) while simultaneously returning a negative classification for (0,0). By trying to train on this data one can see that the training algorithm does not converge.

```
In [24]: AND_weight = one_qubit_train(data, AND_labels)
Not converging
w: [-1. -1.]
```

Below I propose a custom selective training algorithm that will converge to the weights one would expect a classical perceptron to converge to for boolean AND.

A new quantum training algorithm: selective training

In this binary encoding scheme, a quantum classifier cannot distinguish between a vector and its negative due to global phase. When training on a dataset with a vector and its negative assigned different labels, the algorithm will not converge. Below I propose an algorithm that allows for convergence to the weight vector one would expect a classical perceptron to converge to. In other words, providing the ability to single out a specific weight vector in distinction from its negative. Below I describe the modifications one must make to the training algorithm proposed by Tachhino et al.

Selective training algorithm:

Pre-training:

1. For every positively labeled data identify the label of the negative data.
2. If the negative data is not positively labeled add it to a list of opposite labels.

Initialization

- Randomly initialize the weight vector, \vec{w} with elements $\in \{-1, 1\}$
- Initialize a misclassification counter $m=0$ and a boolean variable $\text{full_loop}=\text{False}$. This serves as a second misclassification check.

Training:

- Loop through all data, getting classification prediction with current weight
- If an example is misclassified and does not belong to the list of opposite labels, adjust the weights and increment the misclassification counter and set the boolean variable $\text{full_loop}=\text{False}$
 - If the example is classified as positive when it should be negative, move the input vector further from the weights by flipping a fraction of the signs (l_n) where the input and weight vectors coincide.
 - If the example is classified as negative when it should be positive, move the input vector closer to the weights by flipping a fraction of the signs (l_p) where the input and weight vectors differ.
- if an example is misclassified and DOES belong to the list of opposite labels, adjust the weights but do NOT increment a misclassification counter

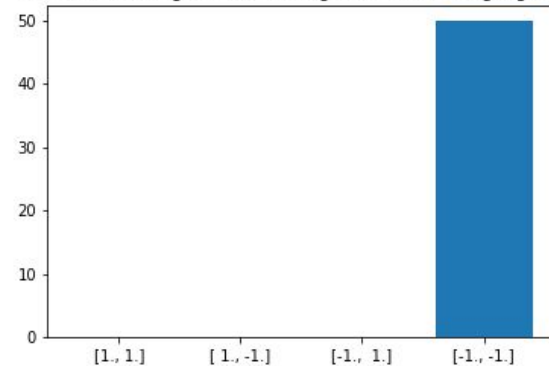
At the end of an epoch (training loop)

- if $m>0$ repeat Training
- if $m=0$ and $\text{full_loop}=\text{False}$ set $\text{full_loop}=\text{True}$ and repeat Training
- if $m=0$ and $\text{full_loop}=\text{True}$ return the learned weights

Below we can see that this algorithm does accurately learn the weights for the boolean AND data set whereas the original training algorithm does not converge.

```
data
[[ 1.  1.]
 [ 1. -1.]
 [-1.  1.]
 [-1. -1.]]
labels [-1 -1 -1  1]
```

Learned AND weight vectors using modified training algorithm



Verification of the selective training algorithm

To examine why this algorithm works. Let's work through an example.

Example:

Data ordering:

$[-1, 1]$

$[1, -1]$

$[-1, -1]$

$[1, 1]$

Labels: $[-1 -1 -1 1]$

- Weights initialized to $[1, 1]$
- Misclassification counter $m=0$. Boolean variable *full_loop=False*
- Input data $[-1, 1]$ is classified as negative. This is correct. There is no weight change.
- Input data $[1, -1]$ is classified as negative. This is again correct.
- Input data $[-1, -1]$ is classified as positive. This is again correct as this is our one positive label.
- Input data $[1, 1]$ is classified as positive. This is incorrect as it should be classified as negative. The weights are adjusted to either $[-1, 1]$ or $[1, -1]$. The misclassification counter m is not incremented, but the boolean variable *full_loop=False* so the algorithm does not terminate at the end of this epoch.

One may continue to think through this example to see that the algorithm will only terminate when the learned weight is exactly the classical target weight for the AND data set and not its negative.

From the above example is shown that training may occur on all data, but data that correspond to the list of `opposite_labels` (i.e. their negative data is positively labeled) do not increment the misclassification counter.

When data corresponding to the list of `opposite_labels` is classified positively and the currently weights are the negative of the target weights, we see that the weights are adjusted so that the negative version of the target weight is not returned.

When data corresponding to the list of `opposite_labels` is classified positively and the current weights are the target weights, we see that the weights are not adjusted for the 1-qubit example. This is because the training algorithm specifies that when an example is classified positively when it's label is negative, the weights are adjusted by changing the sign of elements where the weight vector and the input coincide. For the 1-qubit example no elements of the input vector and weight vector would coincide. Thus the training on the `opposite_label` data does not move the weight vector away from the target weight vector.

It is demonstrated in the jupyter notebook *1 qubit perceptron* that the selective training algorithm is invariant to the order of the data. Furthermore it is shown that the algorithm correctly learns the target weight vector for all boolean functions with one positive label, regardless of data ordering and weight initialization.

Conclusions

The 1-qubit implementation of a quantum perceptron most clearly shows the differences between the quantum and classical perceptron. The quantum perceptron can capture nonlinear patterns such as XOR, but the encoding scheme and the consequence global phase implies that it cannot learn certain linear decision boundaries that the classical perceptron could.

Furthermore, I have proposed a new selective training algorithm for the quantum perceptron that is able to return the exact target weight and not its negative. Such an algorithm could prove useful for future implementations when one might want to use the quantum algorithm for training and learning a weight vector and then use a classical classifier for future predictions.

Code

All code can be found in the jupyter notebook *1-qubit perceptron at*
<https://github.com/Farquhar13/QuantumComputing/tree/master/QuantumPerceptron>

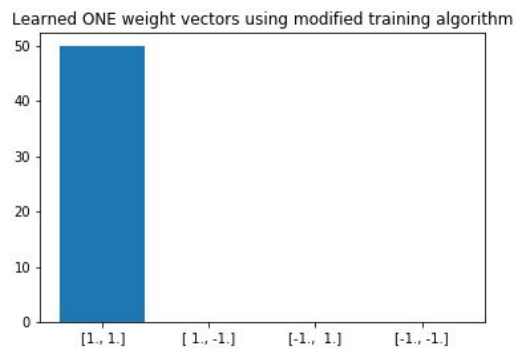
Data addendum

Performance of selective training algorithm for boolean functions with one positive label (would not converge in original algorithm).

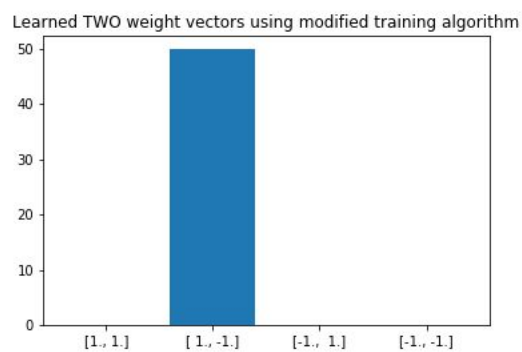
Data:

```
[[ 1.  1.]  
 [ 1. -1.]  
 [-1.  1.]  
 [-1. -1.]]
```

ONE labels: [1 -1 -1 -1]

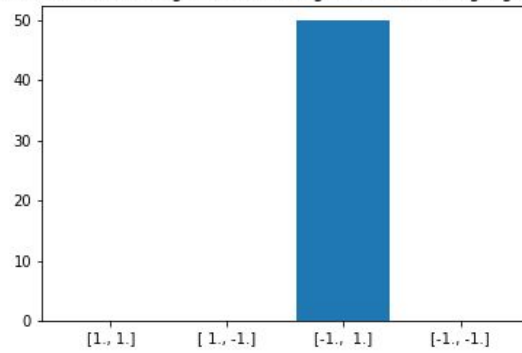


TWO labels: [-1 1 -1 -1]



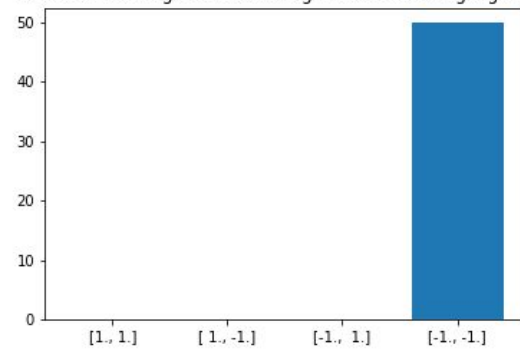
THREE labels: $[-1 \ -1 \ 1 \ -1]$

Learned THREE weight vectors using modified training algorithm



FOUR (AND) labels: $[-1 \ -1 \ -1 \ 1]$

Learned AND weight vectors using modified training algorithm



References

- [1] Ahire, Jayesh Bapu. "The Artificial Neural Networks Handbook: Part 1." *Medium*, Coinmonks, 24 Aug. 2018, medium.com/coinmonks/the-artificial-neural-networks-handbook-part-1-f9ceb0e376b4.
- [2] Hui, Jonathan. "QC - What Are Qubits in Quantum Computing?" *Medium*, Medium, 20 Nov. 2018, medium.com/@jonathan_hui/qc-what-are-qubits-in-quantum-computing-cdb3cb566595.
- [3] Kaye, Phillip, et al. *An Introduction to Quantum Computing*. Oxford University Press, 2010.
- [4] Sharma, Sagar. "What the Hell Is Perceptron?" *Towards Data Science*, Towards Data Science, 9 Sept. 2017, towardsdatascience.com/what-the-hell-is-perceptron-626217814f53.
- [5] Tacchino, Francesco, et al. "An Artificial Neuron Implemented on an Actual Quantum Processor." *Npj Quantum Information*, vol. 5, no. 1, 2019, doi:10.1038/s41534-019-0140-4.

Part 1

The brute-force sign flip algorithm for encoding classical inputs in quantum state vectors

Inspired by work by Francesco Tacchino, Chiara Macchiavello, Dario Gerace & Daniele Bajoni

<https://www.nature.com/articles/s41534-019-0140-4?amp%3Bcode=4cf1b507-7e23-4df0-a2a1-e82a3fe2bc4b>

In [1]:

```
import math
import numpy as np
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute
from qiskit import BasicAer
```

Generate all possible binary-valued vectors in 4 dimensions (2 qubits)

A quantum state vector of $N=2$ qubits is described by a vector of dimension $m=2^N=4$. With binary amplitudes, this implies $2^{2^N}=16$ distinct states. We will use $\{-1,1\}$ as components of our binary vector.

A way to think about generating the binary vectors is to assign an integer k to each vector. You can convert the integer k to a four-digit binary string $\{0,1\}$ $n_0 n_1 n_2 n_3$ and generate the vectors as $\vec{v} = [(-1)^{n_0}, (-1)^{n_1}, (-1)^{n_2}, (-1)^{n_3}]$

In [2]:

```
def k2vec(k, m):
    """
    Parameters:
        - k (integer) the integer number for the corresponding binary vector
        - m (integer) the length of binary string (needed to distinguish all possibilities)
    Returns:
        - v (list) a binary vector corresponding to integer k

    Description: By taking a fixed total ordering of binary strings of fixed length we
    can associate each binary vector with an integer. The integer is converted to a
    binary string which can be used to generate unique vectors.
    """
    v = -1*np.ones(m)
    binary_string = ("{:0%db}" % m).format(k) # convert k to an m-digit binary number
    #print(binary_string)
    v = list(map(lambda v, b : v**int(b), v, binary_string))

    return np.array(v)
```

In [3]:

```
m = 4 # for two qubits
vectors = [k2vec(k,m) for k in range(16)]
for i, v in enumerate(vectors):
    print("k=", i, " ", v)
```

```
k= 0    [1.  1.  1.  1.]
k= 1    [ 1.   1.   1. -1.]
k= 2    [ 1.   1. -1.   1.]
k= 3    [ 1.   1. -1. -1.]
k= 4    [ 1. -1.   1.   1.]
k= 5    [ 1. -1.   1. -1.]
k= 6    [ 1. -1. -1.   1.]
k= 7    [ 1. -1. -1. -1.]
k= 8    [-1.   1.   1.   1.]
k= 9    [-1.   1.   1. -1.]
k= 10   [-1.   1. -1.   1.]
k= 11   [-1.   1. -1. -1.]
k= 12   [-1. -1.   1.   1.]
k= 13   [-1. -1.   1. -1.]
k= 14   [-1. -1. -1.   1.]
k= 15   [-1. -1. -1. -1.]
```

```

k= 10    [-1.  1. -1.  1.]
k= 11    [-1.  1. -1. -1.]
k= 12    [-1. -1.  1.  1.]
k= 13    [-1. -1.  1. -1.]
k= 14    [-1. -1. -1.  1.]
k= 15    [-1. -1. -1. -1.]

```

We can encode this input vector as the quantum state vector $|\Psi_i\rangle$ such that $|\Psi_i\rangle = \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |i_j\rangle$ Where m is the dimensionality of the input, i_j is an element of the classical input vector, and $|j\rangle$ is a computational basis state.

In [4]:

```

state_vectors = [v/np.sqrt(len(v)) for v in vectors]
for i, v in enumerate(state_vectors):
    print("k=", i, " ", v)

```

```

k= 0    [0.5 0.5 0.5 0.5]
k= 1    [ 0.5  0.5  0.5 -0.5]
k= 2    [ 0.5  0.5 -0.5  0.5]
k= 3    [ 0.5  0.5 -0.5 -0.5]
k= 4    [ 0.5 -0.5  0.5  0.5]
k= 5    [ 0.5 -0.5  0.5 -0.5]
k= 6    [ 0.5 -0.5 -0.5  0.5]
k= 7    [ 0.5 -0.5 -0.5 -0.5]
k= 8    [-0.5  0.5  0.5  0.5]
k= 9    [-0.5  0.5  0.5 -0.5]
k= 10   [-0.5  0.5 -0.5  0.5]
k= 11   [-0.5  0.5 -0.5 -0.5]
k= 12   [-0.5 -0.5  0.5  0.5]
k= 13   [-0.5 -0.5  0.5 -0.5]
k= 14   [-0.5 -0.5 -0.5  0.5]
k= 15   [-0.5 -0.5 -0.5 -0.5]

```

What quantum circuits will encode these vectors?

Think of a quantum circuit as the program. With a quantum computer we don't have easy access to read in/out so we have to change our quantum circuits (change our program) to run with different inputs. It's like assigning values to variables at the top of a program.

We'll use a "brute-force Sign-Flip" algorithm. The idea is that for entry of -1 in the input vector you apply a -1 factor to the corresponding entry in the quantum state vector.

The only gates the brute-force Sign-Flip algorithm uses only X and CZ gates. The X gate takes $|0\rangle \rightarrow |1\rangle$ and $|1\rangle \rightarrow |0\rangle$. The CZ has the effect of switching the sign of $|11\rangle$.

In [5]:

```

def draw_state_vector(circ):
    """
    Input:
    - circ (qiskit.circuit.quantumcircuit.QuantumCircuit) quantum circuit
      to be executed
    Description:
    - Runs the quantum circuit and prints the resulting quantum state vector
      and draws the circuit.
    """

    backend = BasicAer.get_backend('statevector_simulator')
    job = execute(circ, backend)
    result = job.result()
    output_state = result.get_statevector(circ, decimals=3)
    print("state vector after circuit:", output_state)
    print(circ.draw())

```

In [6]:

```

# create quantum register and circuit
q = QuantumRegister(2, 'q')
circ = QuantumCircuit(q)

```

Parallel hadamard gates will produce an equal super position, $|00\rangle \rightarrow |+\rangle^{\otimes 2}$

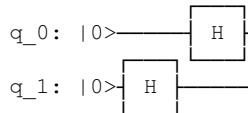
We will need this at the start of every circuit. It also produces the corresponding state vector for $k=0$!

In [7]:

```
k=0
print("target state vector:", state_vectors[k])
circ.h(q[0])
circ.h(q[1])

draw_state_vector(circ)
```

target state vector: [0.5 0.5 0.5 0.5]
state vector after circuit: [0.5+0.j 0.5+0.j 0.5+0.j 0.5+0.j]

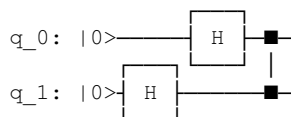


In [8]:

```
k=1
print("target state vector:", state_vectors[k])
circ1 = QuantumCircuit(q)
circ1.cz(q[0], q[1])

draw_state_vector(circ + circ1)
```

target state vector: [0.5 0.5 0.5 -0.5]
state vector after circuit: [0.5+0.j 0.5+0.j 0.5+0.j -0.5+0.j]



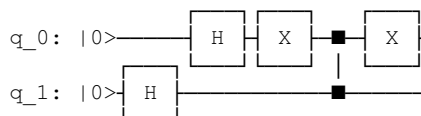
In [9]:

```
k=2
print("target state vector:", state_vectors[k])

circ2 = QuantumCircuit(q)
circ2.x(q[0])
circ2.cz(q[0], q[1])
circ2.x(q[0])

draw_state_vector(circ + circ2)
```

target state vector: [0.5 0.5 -0.5 0.5]
state vector after circuit: [0.5+0.j 0.5+0.j -0.5+0.j 0.5+0.j]



In [10]:

```
k=3
print("target state vector:", state_vectors[k])
circ3 = QuantumCircuit(q)

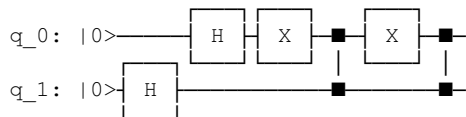
circ3.x(q[0])
circ3.cz(q[0], q[1])
circ3.x(q[0])

circ3.cz(q[0], q[1])
```

```
draw_state_vector(circ + circ3)
```

target state vector: [0.5 0.5 -0.5 -0.5]

state vector after circuit: [0.5+0.j 0.5+0.j -0.5+0.j -0.5+0.j]



In [11]:

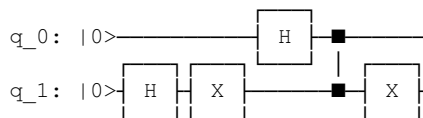
```
k=4
print("target state vector:", state_vectors[k])
```

```
circ4 = QuantumCircuit(q)
circ4.x(q[1])
circ4.cz(q[0], q[1])
circ4.x(q[1])
```

```
draw_state_vector(circ + circ4)
```

target state vector: [0.5 -0.5 0.5 0.5]

state vector after circuit: [0.5+0.j -0.5+0.j 0.5+0.j 0.5+0.j]



In [12]:

```
k=5
print("target state vector:", state_vectors[k])
circ5 = QuantumCircuit(q)
```

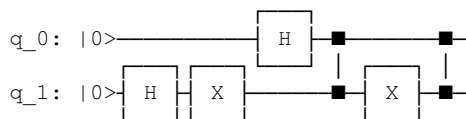
```
circ5.x(q[1])
circ5.cz(q[0], q[1])
circ5.x(q[1])
```

```
circ5.cz(q[0], q[1])
```

```
draw_state_vector(circ + circ5)
```

target state vector: [0.5 -0.5 0.5 -0.5]

state vector after circuit: [0.5+0.j -0.5+0.j 0.5+0.j -0.5+0.j]



In [13]:

```
k=6
print("target state vector:", state_vectors[k])
```

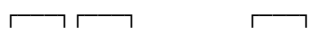
```
circ6 = QuantumCircuit(q)
circ6.x(q[0])
circ6.cz(q[0], q[1])
circ6.x(q[0])
```

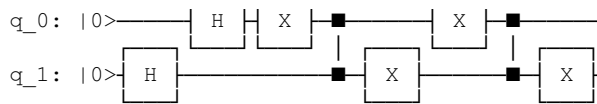
```
circ6.x(q[1])
circ6.cz(q[0], q[1])
circ6.x(q[1])
```

```
draw_state_vector(circ + circ6)
```

target state vector: [0.5 -0.5 -0.5 0.5]

state vector after circuit: [0.5+0.j -0.5+0.j -0.5+0.j 0.5+0.j]





In [14]:

```
k=7
print("target state vector:", state_vectors[k])

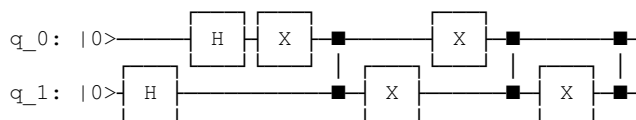
circ7 = QuantumCircuit(q)
circ7.x(q[0])
circ7.cz(q[0], q[1])
circ7.x(q[0])

circ7.x(q[1])
circ7.cz(q[0], q[1])
circ7.x(q[1])

circ7.cz(q[0], q[1])

draw_state_vector(circ + circ7)
```

target state vector: [0.5 -0.5 -0.5 -0.5]
state vector after circuit: [0.5+0.j -0.5+0.j -0.5+0.j -0.5+0.j]



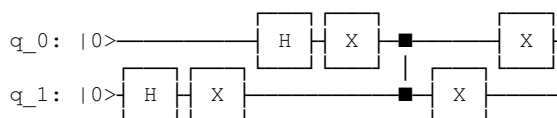
In [15]:

```
k=8
print("target state vector:", state_vectors[k])

circ8 = QuantumCircuit(q)
circ8.x(q[0])
circ8.x(q[1])
circ8.cz(q[0], q[1])
circ8.x(q[0])
circ8.x(q[1])

draw_state_vector(circ + circ8)
```

target state vector: [-0.5 0.5 0.5 0.5]
state vector after circuit: [-0.5+0.j 0.5+0.j 0.5+0.j 0.5+0.j]



In [16]:

```
k=9
print("target state vector:", state_vectors[k])

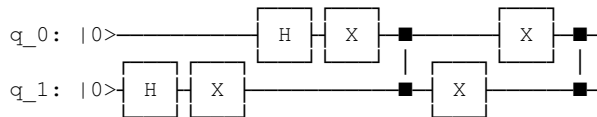
circ9 = QuantumCircuit(q)

# this becomes the "encode -|00> block" or k_8
circ9.x(q[0])
circ9.x(q[1])
circ9.cz(q[0], q[1])
circ9.x(q[0])
circ9.x(q[1])

# This is k_1
circ9.cz(q[0], q[1])

draw_state_vector(circ + circ9)
```

target state vector: [-0.5 0.5 0.5 -0.5]
state vector after circuit: [-0.5+0.j 0.5+0.j 0.5+0.j -0.5+0.j]



A pattern emerges

Here we begin to repeat circuit combinations. E.g. $k_9 = k_8 + k_1$

As noted in the block above, the sign flip block for k_8 will now be needed for every $k \leq 16$.

So we may as well just incorporate circ8 in combination with some circ(<8) into our future circuits.

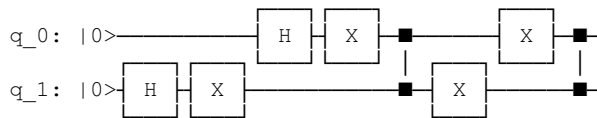
In [17]:

```
# alternatively for k=9
print("target state vector:", state_vectors[k])

circ9a = QuantumCircuit(q)
circ9a = circ + circ8 + circ1

draw_state_vector(circ9a)
```

target state vector: [-0.5 0.5 0.5 -0.5]
state vector after circuit: [-0.5+0.j 0.5+0.j 0.5+0.j -0.5+0.j]



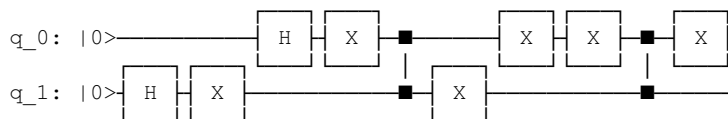
In [18]:

```
k=10
print("target state vector:", state_vectors[k])

circ10 = QuantumCircuit(q)
circ10 = circ + circ8 + circ2

draw_state_vector(circ10)
```

target state vector: [-0.5 0.5 -0.5 0.5]
state vector after circuit: [-0.5+0.j 0.5+0.j -0.5+0.j 0.5+0.j]



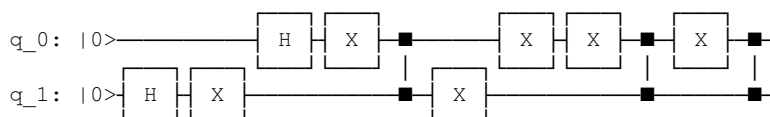
In [19]:

```
k=11
print("target state vector:", state_vectors[k])

circ11 = QuantumCircuit(q)
circ11 = QuantumCircuit(q)
circ11 = circ + circ8 + circ3

draw_state_vector(circ11)
```

target state vector: [-0.5 0.5 -0.5 -0.5]
state vector after circuit: [-0.5+0.j 0.5+0.j -0.5+0.j -0.5+0.j]



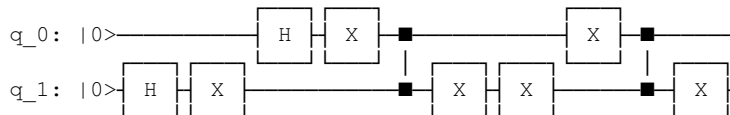
In [20]:

```
k=12
print("target state vector:", state_vectors[k])
```

```
circl2 = QuantumCircuit(q)
circl2 = QuantumCircuit(q)
circl2 = circ + circ8 + circ4
```

```
draw_state_vector(circl2)
```

target state vector: [-0.5 -0.5 0.5 0.5]
state vector after circuit: [-0.5+0.j -0.5+0.j 0.5+0.j 0.5+0.j]



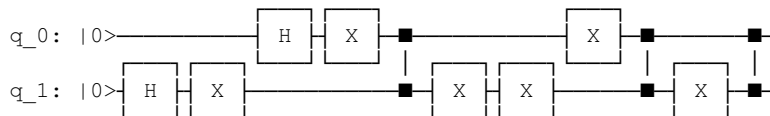
In [21]:

```
k=13
print("target state vector:", state_vectors[k])
```

```
circl3 = QuantumCircuit(q)
circl3 = QuantumCircuit(q)
circl3 = circ + circ8 + circ5
```

```
draw_state_vector(circl3)
```

target state vector: [-0.5 -0.5 0.5 -0.5]
state vector after circuit: [-0.5+0.j -0.5+0.j 0.5+0.j -0.5+0.j]



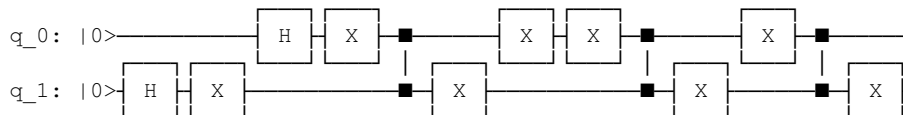
In [22]:

```
k=14
print("target state vector:", state_vectors[k])
```

```
circl4 = QuantumCircuit(q)
circl4 = QuantumCircuit(q)
circl4 = circ + circ8 + circ6
```

```
draw_state_vector(circl4)
```

target state vector: [-0.5 -0.5 -0.5 0.5]
state vector after circuit: [-0.5+0.j -0.5+0.j -0.5+0.j 0.5+0.j]



In [23]:

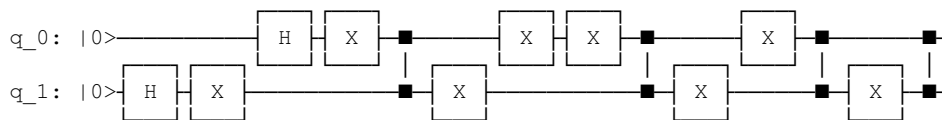
```
k=15
print("target state vector:", state_vectors[k])
```

```
circl5 = QuantumCircuit(q)
circl5 = QuantumCircuit(q)
circl5 = circ + circ8 + circ7
```

```
draw_state_vector(circl5)
```

target state vector: [-0.5 -0.5 -0.5 -0.5]
state vector after circuit: [-0.5+0.j -0.5+0.j -0.5+0.j -0.5+0.j]

```
state vector after circuit: [-0.5+0.j -0.5+0.j -0.5+0.j -0.5+0.j]
```



Big O

Just by looking at the circuits, we start to get a sense of how the brute-force sign flip algorithm scales.

In the worst cases, which correspond to high values of k , the corresponding state vectors require circuits that scale

exponentially in depth (left to right time slices).

A general Sign-Flip function to encode a quantum state vector

This brute-force Sign-Flip algorithm uses only X and CZ gates. It effectively acts on one component of the state vector at a time.

Right now this will only work for $N=2$ qubits. But it gives you the right idea!

In [24]:

```
def SF_encoder(i):
    """
    Parameters:
        - i (list) A classical binary valued {-1, 1} input vector
    Returns:
        - circ (qiskit.circuit.quantumcircuit.QuantumCircuit) A quantum circuit that
          properly encodes the input vector i
    Description:
        - convert vector to binary string, or somehow generate computational basis vectors
        - Apply SF algorithm whenever there is a negative -1 in i
        - Use -1 position to indentify corresponding computational basis, sandwich qubits
          in the 0 state with X gates, and in-between apply a CZ
    """

    i = np.array(i)
    d = len(i) # dimesionality of input vector

    # Validate input
    assert math.log(d, 2)%1 == 0, "Invalid vector length. Must be 2^N"

    N = int(math.log(d, 2)) # number of qubits

    # Generate computational basis (standard basis) vectors with Dirac labeling
    Dirac_vectors = [("{:0%db}"%N).format(k) for k in range(d)]

    '''
    for b in Dirac_vectors:
        print(b)
    '''

    # create quantum register and circuit
    q = QuantumRegister(N, 'q')
    circ = QuantumCircuit(q)

    # always start in equal superposition
    circ.h(q[0])
    circ.h(q[1])

    # Find all components with a -1 factor in i (and thus our target state vector)
    indices = np.where(i == -1)[0]
    if indices.size > 0:
        for idx in indices:
            # Need to switch qubits in the 0 state so CZ will take effect
            for i, b in enumerate(Dirac_vectors[idx]):
                if b == '0':
                    circ.x(q[(N-1)-i]) # (N-1)-i is to match the qubit ordering Qiskit uses
                                     (reversed)

            circ.cz(q[0], q[1]) # this is the only part that doesn't generalize for N!=2

    # And switch the flipped qubits back
    for i, b in enumerate(Dirac_vectors[idx]):
```

```

    for i, b in enumerate(dirac_vectors[idx]):
        if b == '0':
            circ.x(q[(N-1)-i])

    return circ

```

In [25]:

```

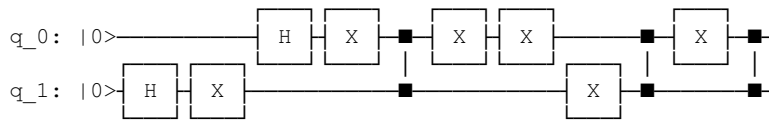
# You can test the algorithm here!
k = 11
print(state_vectors[k])
cc = SF_encoder(vectors[k])
draw_state_vector(cc)

```

```

[-0.5  0.5 -0.5 -0.5]
state vector after circuit: [-0.5+0.j  0.5+0.j -0.5+0.j -0.5+0.j]

```



In []:

Part 2

Encoding weights, finding $\$U_w\$, inner products, and classification$

Inspired by work by Francesco Tacchino, Chiara Macchiavello, Dario Gerace & Daniele Bajoni

<https://www.nature.com/articles/s41534-019-0140-4?amp%3Bcode=4cf1b507-7e23-4df0-a2a1-e82a3fe2bc4b>

In [1]:

```
import math
import numpy as np
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute, BasicAer, Aer
from qiskit.tools.visualization import plot_histogram, plot_state_city, plot_bloch_multivector
```

We'll again make use of the sign-flip algorithm from last time, with slight modifications. This time, it will not begin each circuit with parallel Hadamard gates and will output 3 qubit circuits with 2 qubits plus an ancillary qubit.

In [4]:

```
def SF_encoder(i):
    """
    Parameters:
        - i (list) A classical binary valued {-1, 1} input vector
    Returns:
        - circ (qiskit.circuit.quantumcircuit.QuantumCircuit) A quantum 3 qubit circuit that
          properly encodes the input vector i
    Description:
        - convert vector to binary string, or somehow generate computational basis vectors
        - Apply SF algorithm whenever there is a negative -1 in i
        - Use -1 position to indentify corresponding computational basis, sandwich qubits
          in the 0 state with X gates, and in-between apply a CZ
        - Tested for 2 qubits
    """

    i = np.array(i)
    d = len(i) # dimesionalitiy of input vector

    # Validate input
    assert math.log(d, 2)%1 == 0, "Invalid vector length. Must be 2^N"

    N = int(math.log(d, 2)) # number of qubits

    # Generate computational basis (standard basis) vectors with Dirac labeling
    Dirac_vectors = [{"{:0%db}"%N}.format(k) for k in range(d)]

    '''
    for b in Dirac_vectors:
        print(b)
    '''

    # create quantum register and circuit (N+1 for the ancilla qubit)
    q = QuantumRegister(N+1, 'q')
    c = ClassicalRegister(1, 'c')
    circ = QuantumCircuit(q, c)

    # removed Hadamard gates

    # Find all components with a -1 factor in i (and thus our target state vector)
    indices = np.where(i == -1)[0]
    if indices.size > 0:
        for idx in indices:
            # Need to switch qubits in the 0 state so CZ will take effect
            for i, b in enumerate(Dirac_vectors[idx]):
                if b == '0':
                    circ.x(q[(N-1)-i]) # (N-1)-i is to match the qubit ordering Qiskit uses
(reversed)

                    circ.cz(q[0], q[1]) # this is the only part that doesn't generalize for N!=2
```

```

circ.x(q[0], q[1]) # this is the only gate that doesn't generalize for N > 2

# And switch the flipped qubits back
for i, b in enumerate(Dirac_vectors[idx]):
    if b == '0':
        circ.x(q[(N-1)-i])

return circ

def k2vec(k, m=4):
    """
    Parameters:
        - k (integer) the integer number for the corresponding binary vector
        - m (integer) the length of binary string (needed to distinguish all possibilities)
    Returns:
        - v (list) a binary vector corresponding to integer k

    Description: By taking a fixed total ordering of binary strings of fixed length we
    can associate each binary vector with an integer. The integer is converted to a
    binary string which can be used to generate unique vectors.
    """

    v = -1*np.ones(m)
    binary_string = ("{:0%db}"%m).format(k) # convert k to an m-digit binary number
    #print(binary_string)
    v = list(map(lambda v, b : v**int(b), v, binary_string))

    return v

def draw_state_vector(circ):
    """
    Input:
        - circ (qiskit.circuit.quantumcircuit.QuantumCircuit) quantum circuit
        to be executed
    Description:
        - Runs the quantum circuit and prints the resulting quantum state vector
        and draws the circuit.
    """

    backend = BasicAer.get_backend('statevector_simulator')
    job = execute(circ, backend)
    result = job.result()
    output_state = result.get_statevector(circ, decimals=3)
    print("state vector after circuit:", output_state)
    print(circ.draw())

```

A perceptron classification circuit

The perceptron algorithm was invented by former Cornellian Frank Rosenblatt in 1960. The perceptron is the fundamental unit of a neural network and I'll assume that you're familiar with the algorithm. Essentially, it involves:

1. taking an inner product between input and weight vectors
2. applying an activation function
3. setting a threshold for perceptron activation

Using the SF_encoder function (brute-force sign-flip) from "Explicit_Input-Encoding" I'll create a full perceptron classification circuit with $N=2$ qubits with an extra ancillary qubit to extract the inner product information. Then I'll work backwards to describe what's going on.

Circuit for $\vec{k} = k_3$ and $\vec{w} = k_8$

In [3]:

```

# perhaps change this to match 3/8 example
# create quantum register and circuit
q = QuantumRegister(3, 'q')
c = ClassicalRegister(1, 'c')
circ = QuantumCircuit(q, c)

circ.h(q[0])
circ.h(q[1])
circ += SF_encoder(k2vec(3, 4)) + SF_encoder(k2vec(8, 4))
circ.h(q[0])
circ.h(q[1])

```

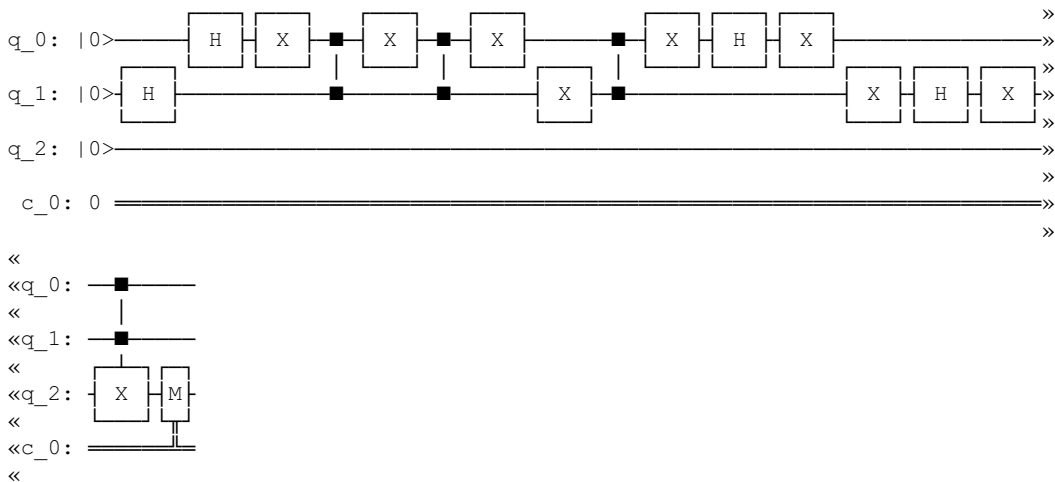
```

circ.x(q[0])
circ.x(q[1])
circ.ccx(q[0], q[1], q[2])
circ.measure(q[2], c[0])

draw_state_vector(circ)

```

state vector after circuit: $[-0.5+0.j \quad 0.5+0.j \quad -0.5+0.j \quad 0. +0.j \quad 0. +0.j \quad 0. +0.j \quad 0. +0.j \quad -0.5+0.j]$



Step 0 - Start the qubits in an equal superposition

It's ability of qubits to be in a superposition of states that allows us to use N qubits to encode a $m = 2^N$ dimensional classical vector.

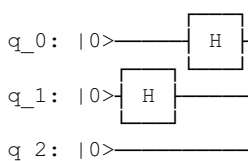
In [4]:

```

# create a fresh quantum register and circuit
q = QuantumRegister(3, 'q')
superposition_circ = QuantumCircuit(q)

# put the qubits in an equal superposition (not the ancilla qubit - i.e. qubit 2)
superposition_circ.h(q[0])
superposition_circ.h(q[1])
print(superposition_circ)

```



Step 1 - Encode the input

Using the Sign-Flip function from last time we will encode our input $\vec{\psi} = k_3$ into the first two qubits as a quantum state vector $|\Psi_i\rangle$. 11 in binary is (1011) which, based the procedure defined last time, results in vector input vector $[-1, 1, -1, -1]$

In [5]:

```

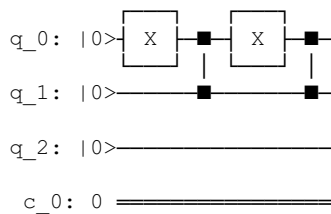
k = 3
m = 4 # 2 qubits -> 4 dimensional classical vector

input_vector = k2vec(k, m)
print("Input vector:", input_vector)

input_circ = SF_encoder(input_vector)
print(input_circ)

```

Input vector: $[1.0, 1.0, -1.0, -1.0]$



Step 2 - Encode the weight vector

After our circuit to encode the input vector. The next step is encode the weight vector. We'll use the Sign-Flip algorithm again to encode the weight vector $\vec{w} = k_8$ into a quantum vector $|\Psi_w\rangle$.

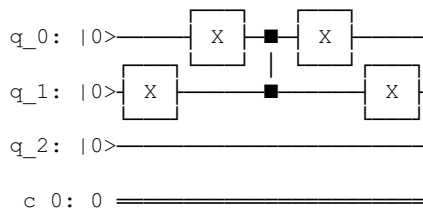
In [6]:

```
k = 8
m = 4

weight_vector = k2vec(k, m)
print("Weight vector:", weight_vector)

weight_circ = SF_encoder(weight_vector)
print(weight_circ)
```

Weight vector: [-1.0, 1.0, 1.0, 1.0]



Step 3 - Rotate the vectors

We want an operator, U_w , that will operate on $|\Psi_i\rangle$ and that will take the inner product between the inputs and weights. Furthermore, we would hope that the result of $U_w|\Psi_i\rangle$ will allow us to actually extract the inner product information.

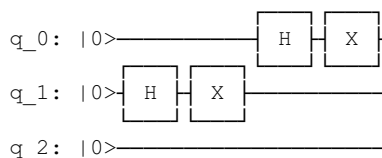
One way to do this is to rotate $|\Psi_w\rangle$ to the $|1\rangle^{\otimes N}$ computational basis state and rotate $|\Psi_i\rangle$ by the same angle. By doing this, the inner product will project the rotated $|\Psi_i\rangle$ onto the rotated $|\Psi_w\rangle$ and therefore the inner product result will be stored as the amplitude of the $|1\rangle^{\otimes N}$ computational basis state.

The first part of creating the U_w block was to encode $|\Psi_w\rangle$, the second part is to rotate our vectors.

In [7]:

```
#q1 = QuantumRegister(3, 'q')
rotation_circ = QuantumCircuit(q)

rotation_circ.h(q[0])
rotation_circ.h(q[1])
rotation_circ.x(q[0])
rotation_circ.x(q[1])
print(rotation_circ)
```



Why does the above circuit give the correct rotation? Here's the idea.

Why does the above circuit give the correct result? Here's the reason:

Consider when $\vec{i} = \vec{w}$ and therefore $|\Psi_i\rangle = |\Psi_w\rangle$. This will give a perfect activation of the perceptron. We start in the equal superposition state $|\Psi_0\rangle$ and take it to the encoded input state $|\Psi_i\rangle$ with the operator U_i . $U_i|\Psi_0\rangle \rightarrow |\Psi_i\rangle$ The the operation to encode $|\Psi_w\rangle$, which is the same as $|\Psi_i\rangle$, so $U_i = U_w$. Because quantum operations are reversible, this would take $|\Psi_i\rangle$ back to an equal superposition. $U_w|\Psi_i\rangle \rightarrow |\Psi_0\rangle$ To get this information to the $|1\rangle^{\otimes N}$ computational basis state, we apply parallel Hadamard and X gates. $H^{\otimes N}|\Psi_0\rangle \rightarrow |0\rangle^{\otimes N}$ followed by $X^{\otimes N}|0\rangle^{\otimes N} \rightarrow |1\rangle^{\otimes N}$

$H^{\otimes N}|0\rangle \rightarrow |\Psi_0\rangle \xrightarrow{U_i} |\Psi_i\rangle \xrightarrow{U_w} |\Psi_0\rangle \xrightarrow{H^{\otimes N}} |0\rangle^{\otimes N} \xrightarrow{X^{\otimes N}} |1\rangle^{\otimes N}$

Step 4 - Extract the information and measure

As noted above, the inner product information should now be contained in the amplitude of the $|1\rangle^{\otimes N}$ computational basis state. We can extract this information by using a multi-controlled-X gate (Toffoli gate for 3 qubits) using our first two qubits as the controls and ancillary qubit as the target. Then we measure the ancillary qubit.

A crucial part of perceptron-based algorithms is a nonlinear activation function. Measurement being a quadratic probabilistic function actually serves as an `activation function`!

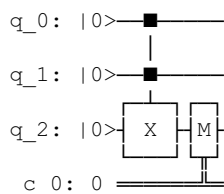
In [8]:

```
# to measure we'll need a classical register
c = ClassicalRegister(1, 'c')

measure_circ = QuantumCircuit(q, c)

# use Toffoli gate to move inner product result to the ancillary qubit
measure_circ.ccx(q[0], q[1], q[2])
measure_circ.measure(q[2], c[0])

print(measure_circ)
```



Putting it all together!

We can simulate our circuit by combining the pieces from above and using the statevector and QASM simulators.

First we look at the statevector simulation, which produces the ideal (noiseless) result of running the circuit. You can then see the Bloch sphere representation of the resulting state of the qubits. We can see mostly interested in the state of the ancillary qubit, qubit-2, as that is what is measured.

For the example that we've been working with of $\vec{i} = \vec{k}_3$ and $\vec{w} = \vec{k}_8$ we would not want the perceptron to be activated because the input and weight vectors are not very similar. For two qubits, the perceptron will only be activated when $\vec{i} = \vec{w}$ and it's negative $-\vec{i} = \vec{w}$.

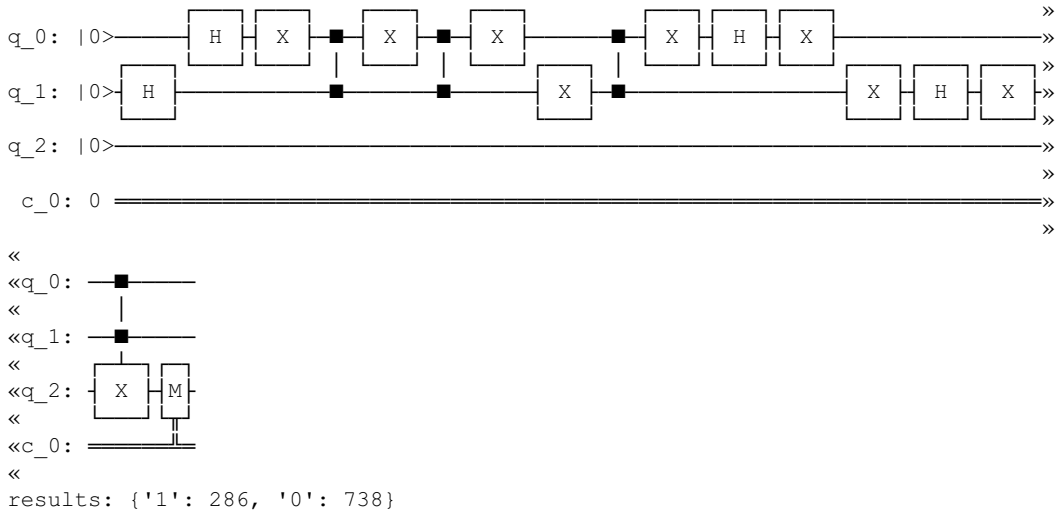
In [9]:

```
full_circ = superposition_circ + input_circ + weight_circ + rotation_circ + measure_circ
print(full_circ)

# Use Aer's qasm_simulator
backend_sim = BasicAer.get_backend('qasm_simulator')
# Execute the circuit on the qasm simulator.
job_sim = execute(full_circ, backend_sim, shots=1024)
# Grab the results from the job.
result_sim = job_sim.result()
```



```
# Results with noise
print("results:", result_sim.get_counts(full_circ))
```



In [10]:

```
# Use Aer's qasm_simulator
backend_sim = BasicAer.get_backend('qasm_simulator')
# Execute the circuit on the qasm simulator.
job_sim = execute(full_circ, backend_sim, shots=1024)
# Grab the results from the job.
result_sim = job_sim.result()

# Results with noise
print("results:", result_sim.get_counts(full_circ))
```

results: {'1': 286, '0': 738}

Quantum computation is inherently probabilistic. However, if we set our perceptron threshold properly, we can get a perceptron that classifies as we would want. For example, by considering the collection of measurements and define a threshold that activates the perceptron if over 50% of those measurements were in the '1' state.

Analysis of classification Results

Let's try for two more examples

1. $\vec{w} = \vec{w}$ corresponding to a boolean function
2. XOR data

In [11]:

```
# input vector equal to the weight vector
k = 10
m = 4

input10_vector = k2vec(k, m)
print(input_vector)
input10_circ = SF_encoder(input10_vector)
weight10_circ = input10_circ

# create quantum register and circuit
q = QuantumRegister(3, 'q')
circ10 = QuantumCircuit(q)
circ10 += superposition_circ + input10_circ + weight10_circ + rotation_circ + measure_circ

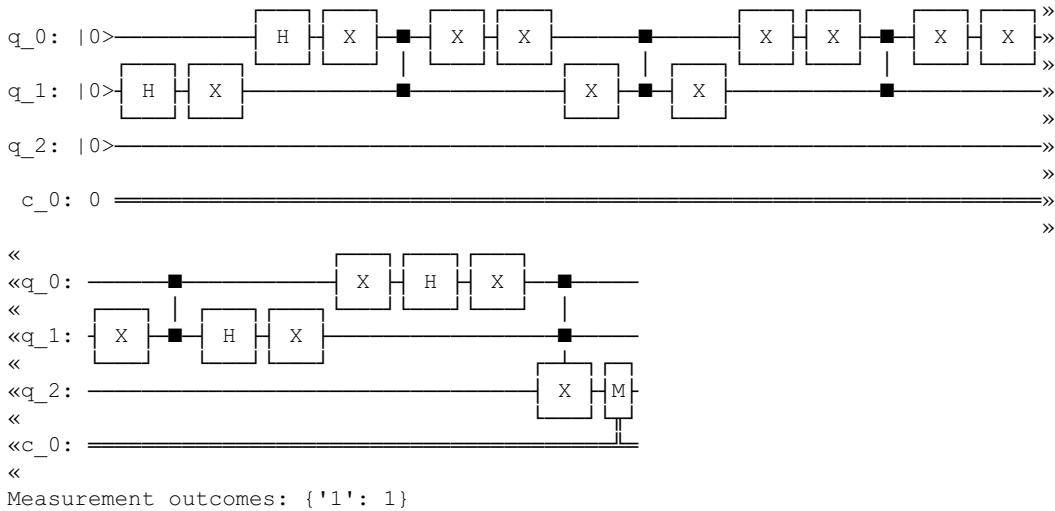
draw_state_vector(circ10)

# Select the StatevectorSimulator from the Aer provider
simulator = Aer.get_backend('statevector_simulator')
result = execute(circ10, simulator).result()
statevector = result.get_statevector(circ10)
```

```
print("Measurement outcomes:", result.get_counts(circ10))
```

```
[1.0, 1.0, -1.0, -1.0]
```

```
state vector after circuit: [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.-0.j]
```



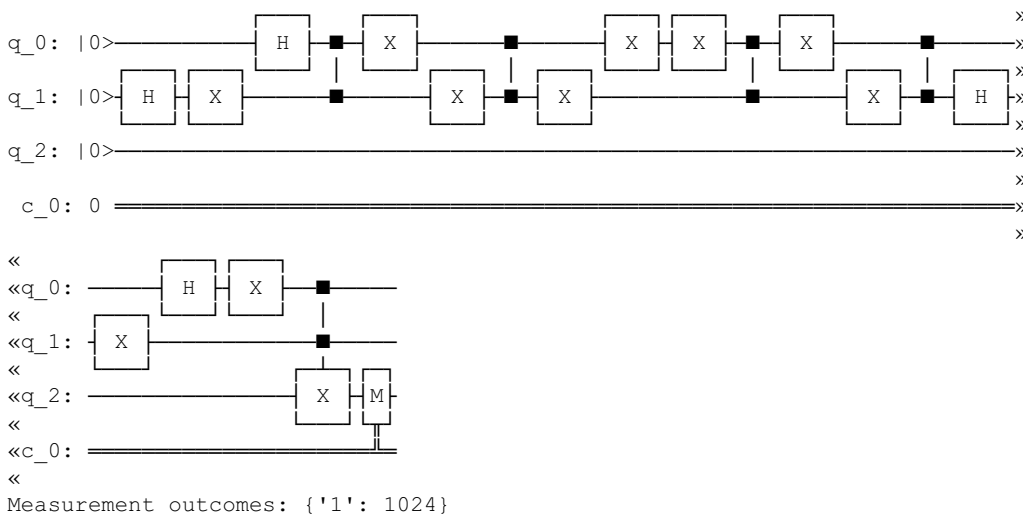
As we can see this circuit this circuit has an amplitude of 1 for the $|1\rangle^{\otimes N}$ state. Meaning that it will deterministically (without noise) give an outcome of '1', i.e. an activated perceptron!

In [12]:

```
# an XOR data set with -i = w
print("input vector", k2vec(6,4))
print("weight vector", k2vec(9,4))

qx = QuantumRegister(3, 'q')
cx = ClassicalRegister(1, 'c')
circx = QuantumCircuit(qx, cx)
circx += superposition_circ + SF_encoder(k2vec(6,4)) + SF_encoder(k2vec(9, 4)) + rotation_circ + measure_circ
simulator = Aer.get_backend('qasm_simulator')
result = execute(circx, simulator, shots=1024).result()
counts = result.get_counts(circx)
print(circx)
print("Measurement outcomes:", counts)
```

```
input vector [1.0, -1.0, -1.0, 1.0]
weight vector [-1.0, 1.0, 1.0, -1.0]
```



We see that deterministically the perceptron is activated! The above result shows two interesting results of the quantum perceptron algorithm.

1. It can correctly classify non-linear data such as the boolean XOR data set. This is not possible in the classical perceptron algorithm. It is made possible due to the non-linearity induced by measuring a quantum system.

2. The algorithm evaluates a pattern $|\psi\rangle$ and its negative $|\psi\rangle$ on equal footing. This is because global phase is physically insignificant in quantum systems, e.g. $|\psi\rangle = -|\psi\rangle$

All pairs of inputs and weights

In [13]:

```
def i_w_circuit(i, w):
    q = QuantumRegister(3, 'q')
    c = ClassicalRegister(1, 'c')
    circ = QuantumCircuit(q, c)

    # superposition
    circ.h(q[0])
    circ.h(q[1])
    # encode i
    circ += SF_encoder(i)
    # encode w
    circ += SF_encoder(w)
    # rotation
    circ.h(q[0])
    circ.h(q[1])
    circ.x(q[0])
    circ.x(q[1])
    # measure
    circ.ccx(q[0], q[1], q[2])
    circ.measure(q[2], c[0])

    #print(circ)
    simulator = Aer.get_backend('qasm_simulator')
    result = execute(circ, simulator, shots=1024).result()
    counts = result.get_counts(circ)

    return counts
```

In [14]:

```
outcomes = np.zeros((16,16))
for m in range(16):
    i = k2vec(m)
    for n in range(16):
        w = k2vec(n)
        counts = i_w_circuit(i, w)
        #print(counts)
        if len(counts) == 1:
            outcomes[m][n] = int([*counts][0]) # unpacking
        else:
            outcomes[m][n] = counts['1']/float(1024)
```

In [15]:

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
#ax.matshow(outcomes, cmap=plt.cm.Blues)
ax.set_xlabel("weight")
ax.set_ylabel("input")

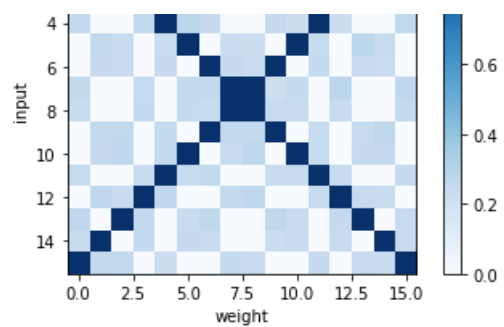
out_plot = ax.imshow(outcomes, cmap='Blues', interpolation='none')
fig.colorbar(out_plot, ax=ax)

#fig.savefig('inputs_v_weights.png')
```

Out[15]:

<matplotlib.colorbar.Colorbar at 0x7f66aa05cf60>





In []:

Part 3

Learning weights (2-qubit example)

From part 2 we have an algorithm that serves as a classifier for given inputs and weights now it's time to actually learn the proper weights for a data set.

Inspired by work by Francesco Tacchino, Chiara Macchiavello, Dario Gerace & Daniele Bajoni

<https://www.nature.com/articles/s41534-019-0140-4?amp%3Bcode=4cf1b507-7e23-4df0-a2a1-e82a3fe2bc4b>

In [8]:

```
import math
import numpy as np
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute, BasicAer, Aer
from qiskit.tools.visualization import plot_histogram, plot_state_city, plot_bloch_multivector

import numpy as np
import matplotlib.pyplot as plt
import random

import import_ipynb
from Part2 import SF_encoder, k2vec, draw_state_vector
```

The training procedure

- Start with randomly initialized \vec{w}_0
- Loop through the training data, whenever a perceptron outcome is incorrect, adjust the weights
 - If an a training example is classified as positive (activated) when it should have been negative move \vec{w} further from \vec{i} by randomly flipping a fraction (I_n) of the signs where \vec{w} and \vec{i} coincide
 - If an a training example is classified as negative when it should have been should move \vec{w} closer to \vec{i} by randomly flipping a fraction (I_p) of the signs where \vec{w} and \vec{i} differ
- Repeat until all training data is correctly classified

Generate training set

2 qubits can encode a 4-dimensional classical vector. Since we considering binary vectors, there are $2^{2^2}=16$ possible vectors that will use as our training set. We now just need to generate labels.

Some visualization code

In [2]:

```
binary_labels = [{"{:0%db}".format(i) for i in range(16)}]
#print(binary_labels)

binary_matrices = []
for b in binary_labels:
    mat = np.array([[int(b[0]), int(b[1])], [int(b[2]), int(b[3])]])
    binary_matrices.append(mat)

binary_matrices.reverse() # reverse to match with presentation in paper
```

Generate labels

Let's say we want to learn the pattern corresponding to $k_i = 1$ or $\vec{i} = [1,1,1,-1]$ as depicted below.

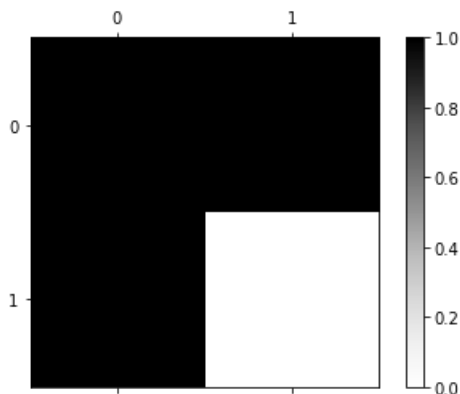
In [3]:

```
labels = -np.ones(16)
labels[1] = 1
```

```
# visualize k_i=1
fig, ax = plt.subplots()
ax.matshow(binary_matrices[1])
out_plot = ax.imshow(binary_matrices[1], cmap='Greys', interpolation='none')
fig.colorbar(out_plot, ax=ax)
```

Out[3]:

<matplotlib.colorbar.Colorbar at 0x7f285bcd2ba8>



Note that due to global phase, the negative of the pattern is treated as the same. So $k_i=14$ or $\vec{i} = [-1,-1,-1,1]$ will also be given a positive label.

In [4]:

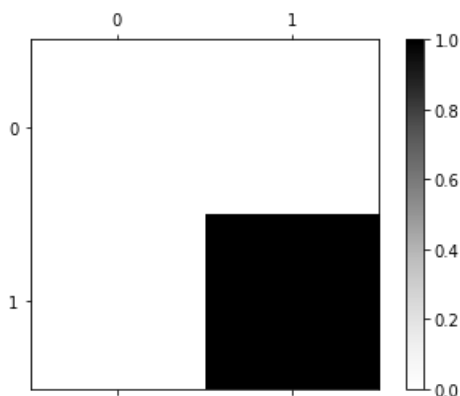
```
labels[14] = 1
print("labels:", labels)

# visualize k_i=14
fig, ax = plt.subplots()
ax.matshow(binary_matrices[14])
out_plot = ax.imshow(binary_matrices[14], cmap='Greys', interpolation='none')
fig.colorbar(out_plot, ax=ax)
```

labels: [-1. 1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. 1. -1.]

Out[4]:

<matplotlib.colorbar.Colorbar at 0x7f285bc63668>



In [5]:

```
# function to construct a quantum circuit for a given input and weight
def i_w_circuit(i, w):
    q = QuantumRegister(3, 'q')
    c = ClassicalRegister(1, 'c')
    circ = QuantumCircuit(q, c)

    # superposition
```

```

circ.h(q[0])
circ.h(q[1])
# encode i
circ += SF_encoder(i)
# encode w
circ += SF_encoder(w)
# rotation
circ.h(q[0])
circ.h(q[1])
circ.x(q[0])
circ.x(q[1])
# measure
circ.ccx(q[0], q[1], q[2])
circ.measure(q[2], c[0])

#print(circ)
simulator = Aer.get_backend('qasm_simulator')
result = execute(circ, simulator, shots=1024).result()
counts = result.get_counts(circ)

```

```

return counts

```

Now to do training algorithm

```

def train(labels, threshold=0.5, lp=0.5, ln=0.5, max_iter=50):
    # Initialize a random w
    w = np.array(random.choices([-1,1], k=int(math.log(len(labels), 2))))
    #print("starting w:", w)

    epochs = 1
    while True:
        m = 0
        for k_i in range(16):
            i = k2vec(k_i)

            # Get the average of the results of running the circuit
            counts = i_w_circuit(i,w)
            if len(counts) == 1:
                avg_counts = int([*counts][0]) # unpacking
            else:
                avg_counts = counts['1']/float(1024)

            # Apply threshold
            if avg_counts >= threshold:
                output = 1
            else:
                output = -1

            # Check for misclassification
            if np.sign(output) != np.sign(labels[k_i]):
                m += 1
                # case when classified positive (activated) but should be negative
                if np.sign(output) > 0:
                    same_idx = [i for i, z in enumerate(zip(i,w)) if z[0] == z[1]]
                    # randomly select elements to switch signs
                    if len(same_idx) > len(w)*ln:
                        random.shuffle(same_idx)
                        same_idx = same_idx[:int(len(w)*ln)]
                    # change signs of w
                    w[same_idx] *= -1

                # case when classified negative but should be positive
            else:
                diff_idx = [i for i, z in enumerate(zip(i,w)) if z[0] != z[1]]
                # randomly select elements to switch signs
                if len(diff_idx) > len(w)*lp:
                    random.shuffle(diff_idx)
                    diff_idx = diff_idx[:int(len(w)*lp)]
                # change signs of w
                w[diff_idx] *= -1
            #print("w:", w)

        if m == 0:
            #print("converged in {} epochs".format(epochs))
            #print("w:", w)
            break
        elif epochs == max_iter:
            print("Not converging")

```

```
        print("w:", w)
        break
    else:
        epochs += 1

    return w
```

In [6]:

```
weights = train(labels)
```

Et Voila!

Our training function learns the weights the exactly correspond to our target pattern (or it's negative).

In [7]:

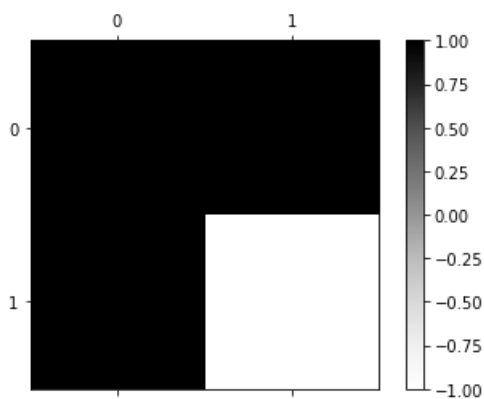
```
print(weights)
weights_mat = weights.reshape(2,2)

# visualize weights
fig, ax = plt.subplots()
ax.matshow(weights_mat)
out_plot = ax.imshow(weights_mat, cmap='Greys', interpolation='none')
fig.colorbar(out_plot, ax=ax)
```

```
[ 1  1  1 -1]
```

Out[7]:

<matplotlib.colorbar.Colorbar at 0x7f285bed6898>



In []:

A 1-qubit perceptron

My custom perceptron algorithm on a single qubit

In [1]:

```
import math
import numpy as np
import random
import matplotlib.pyplot as plt
from collections import defaultdict
import itertools

from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute, BasicAer, Aer
from qiskit.tools.visualization import plot_histogram, plot_state_city, plot_bloch_multivector
```

In [2]:

```
def k2vec(k, m=2):
    """
    Parameters:
        - k (integer) the integer number for the corresponding binary vector
        - m (integer) the length of binary string (needed to distinguish all possibilities)
    Returns:
        - v (list) a binary vector corresponding to integer k

    Description: By taking a fixed total ordering of binary strings of fixed length we
    can associate each binary vector with an integer. The integer is converted to a
    binary string which can be used to generate unique vectors.
    """

    v = -1*np.ones(m)
    binary_string = ("{:0%db}"%m).format(k) # convert k to an m-digit binary number
    #print(binary_string)
    v = list(map(lambda v, b : v**int(b), v, binary_string))

    return np.array(v)

def draw_state_vector(circ):
    """
    Input:
        - circ (qiskit.circuit.quantumcircuit.QuantumCircuit) quantum circuit
        to be executed
    Description:
        - Runs the quantum circuit and prints the resulting quantum state vector
        and draws the circuit.
    """

    backend = BasicAer.get_backend('statevector_simulator')
    job = execute(circ, backend)
    result = job.result()
    output_state = result.get_statevector(circ, decimals=3)
    print("state vector after circuit:", output_state)
    print(circ.draw())
```

All possible 2 bit boolean vectors with elements {-1, 1}

In [3]:

```
# generate all 2-bit boolean data
data = np.array([k2vec(i,2) for i in range(4)])
for d in data:
    print(d)
```

```
[1.  1.]
[ 1. -1.]
[-1.  1.]
```

```
[-1. -1.]
```

Create a dictionary with the circuits to encode the inputs and weights

In [4]:

```
k_circ = {}  
vec_circ_dict = {}
```



In [5]:

```
# encodes k=0  
q = QuantumRegister(1, 'q')  
c = ClassicalRegister(1, 'c')  
circ = QuantumCircuit(q, c)  
  
i = k2vec(0, 2)  
  
# add to dictionary  
k_circ[0] = circ  
vec_circ_dict[repr(i)] = circ
```

In [6]:

```
# encodes k=1  
q = QuantumRegister(1, 'q')  
c = ClassicalRegister(1, 'c')  
circ = QuantumCircuit(q, c)  
  
i = k2vec(1, 2)  
print("i:", i)  
  
#circ.h(q[0]) # uncomment to see that this is the correct circuit  
circ.z(q[0])  
draw_state_vector(circ)  
  
# add to dictionary  
k_circ[1] = circ  
vec_circ_dict[repr(i)] = circ
```

```
i: [ 1. -1.]  
state vector after circuit: [1.+0.j 0.+0.j]
```

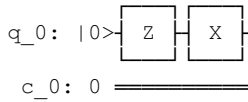
q_0: $|0\rangle$ 
c_0: 0 

In [7]:

```
# encodes k=2  
q = QuantumRegister(1, 'q')  
c = ClassicalRegister(1, 'c')  
circ = QuantumCircuit(q, c)  
  
i = k2vec(2, 2)  
print("i:", i)  
  
#circ.h(q[0]) # uncomment to see that this is the correct circuit  
circ.z(q[0])  
circ.x(q[0])  
  
draw_state_vector(circ)  
  
# add to dictionary  
k_circ[2] = circ  
vec_circ_dict[repr(i)] = circ
```

```
i: [-1.  1.]
```

state vector after circuit: [0.+0.j 1.+0.j]



In [8]:

```
# encodes k=3
q = QuantumRegister(1, 'q')
c = ClassicalRegister(1, 'c')
circ = QuantumCircuit(q, c)

i = k2vec(3, 2)
print("i:", i)

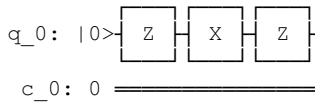
#circ.h(q[0]) # uncomment to see that this is the correct circuit
circ.z(q[0])
circ.x(q[0])
circ.z(q[0])

draw_state_vector(circ)

# add to dictionary
k_circ[3] = circ
vec_circ_dict[repr(i)] = circ
```

i: [-1. -1.]

state vector after circuit: [0.+0.j -1.+0.j]



Taking the inner product

We do this in a similar fashion to part 2. First create a superposition, then encode the inputs and weights, finally rotate the quantum state vector in such a way that the inner product information is contained in the amplitude for the $|1\rangle$ outcome.

One change from the previous classifier is that the ancillary qubit is no longer needed, as the inner product information is contained in β as defined below, which directly determines the probability of a $|1\rangle$ measurement outcome.

$$|\Psi\rangle = \alpha |0\rangle + \beta |1\rangle$$

In [9]:

```
def counts_to_output(counts, threshold=0.5):
    # average the results of the circuit
    if len(counts) == 1:
        avg_counts = int(*counts[0]) # unpacking
    else:
        avg_counts = counts['1']/float(1024)

    # Apply threshold
    if avg_counts >= threshold:
        output = 1
    else:
        output = -1

    return output

def one_qubit_classifier(i, w, threshold=0.5, vec_circ_dict=vec_circ_dict, print_circ=False):
    """
    Inputs:
    - i (list) 2-dim classical input vector with elements {-1, 1}
    - w (list) 2-dim classical weight vector with elements {-1, 1}
    - vec_circ_dict (dictionary) keys are the vectors and the values are the circuits that
      encode those vectors
    Returns:
    - counts (dictionary) the results of running the circuit
    """
```

```

q = QuantumRegister(1, 'q')
c = ClassicalRegister(1, 'c')
circ = QuantumCircuit(q, c)

circ.h(q[0])
# encode i
circ += vec_circ_dict[repr(i)]
# encode w
circ += vec_circ_dict[repr(w)]
# rotation
circ.h(q[0])
circ.x(q[0])
# measure
circ.measure(q[0], c[0])

if print_circ==True:
    print(circ)

simulator = Aer.get_backend('qasm_simulator')
result = execute(circ, simulator, shots=1024).result()
counts = result.get_counts(circ)

output = counts_to_output(counts, threshold)
return output

```

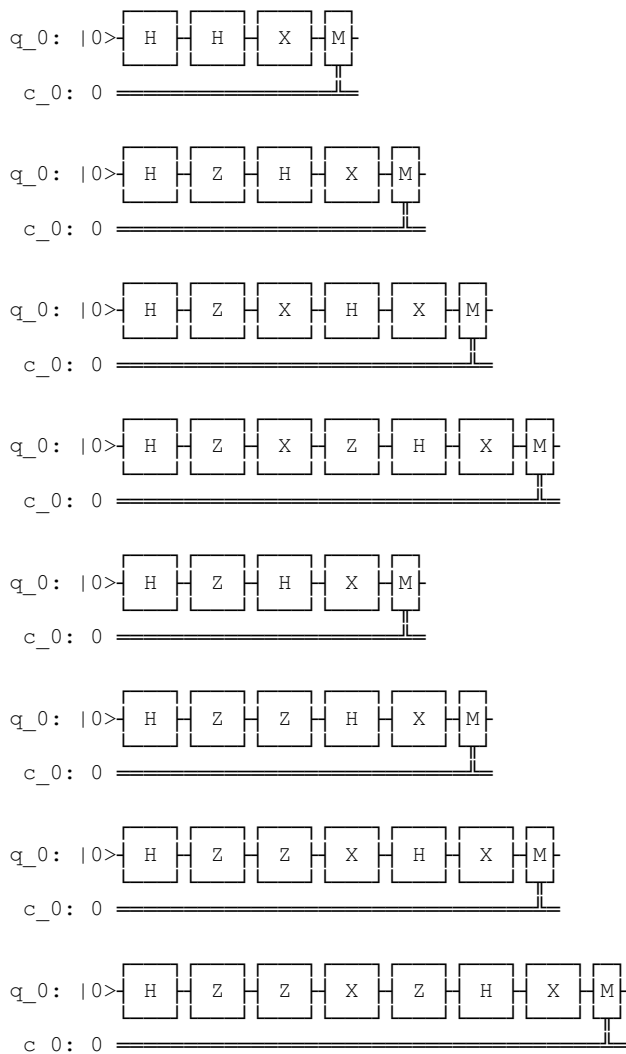
In [10]:

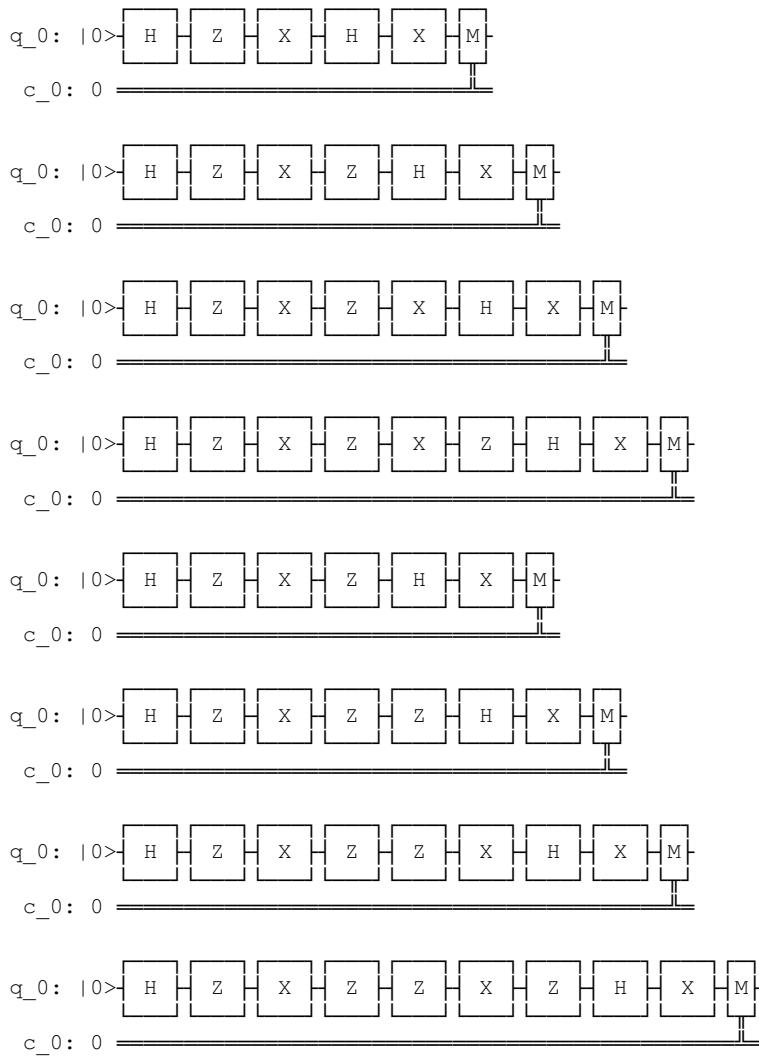
```

outcomes = np.zeros((4,4))

for m in range(4):
    i = k2vec(m, 2)
    for n in range(4):
        w = k2vec(n, 2)
        outcomes[m][n] = one_qubit_classifier(i, w, print_circ=True)

```





In [11]:

```
import numpy as np
import matplotlib.pyplot as plt

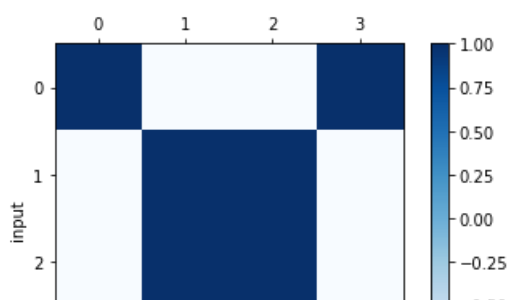
fig, ax = plt.subplots()
ax.matshow(outcomes, cmap=plt.cm.Blues)
ax.set_xlabel("weight")
ax.set_ylabel("input")

out_plot = ax.imshow(outcomes, cmap='Blues', interpolation='none')
fig.colorbar(out_plot, ax=ax)

#fig.savefig('one-qubit-perceptron.png')

print(outcomes)
```

```
[[ 1. -1. -1.  1.]
 [-1.  1.  1. -1.]
 [-1.  1.  1. -1.]
 [ 1. -1. -1.  1.]]
```




```

        # change signs of w
        w[same_idx] *= -1

        # case when classified negative but should be positive
    else:
        diff_idx = [i for i, z in enumerate(zip(i, w)) if z[0] != z[1]]
        # randomly select elements to switch signs
        if len(diff_idx) > len(w)*lp:
            random.shuffle(diff_idx)
            diff_idx = diff_idx[:int(len(w)*lp)]
        # change signs of w
        w[diff_idx] *= -1
        #print("w:", w)

    if m == 0:
        #print("converged in {} epochs".format(epochs))
        #print("w:", w)
        break
    elif epochs == max_iter:
        print("Not converging")
        print("w:", w)
        break
    else:
        epochs += 1

    return w

```

changed the call to `one_qubit_classifier()` to be integers not vectors. Need to change to a a and make a k2vec_dic

In [14]:

```
XOR_weight = one_qubit_train(data, XOR_labels)
```

Inference

In [15]:

```

def predict(data, weights, classifier=one_qubit_classifier):
    preds = []
    for d in data:
        # Get the average of the results of running the circuit
        preds.append(one_qubit_classifier(d, weights))

    return preds

```

In [16]:

```

predictions = predict(data, XOR_weight)
print(predictions)

```

```
[-1, 1, 1, -1]
```

Results of running the quantum classifier trained on XOR data set

Plotting our data and coloring the points red if the results of the classifier puts them in the negative class and blue for the positive class.

In [17]:

```

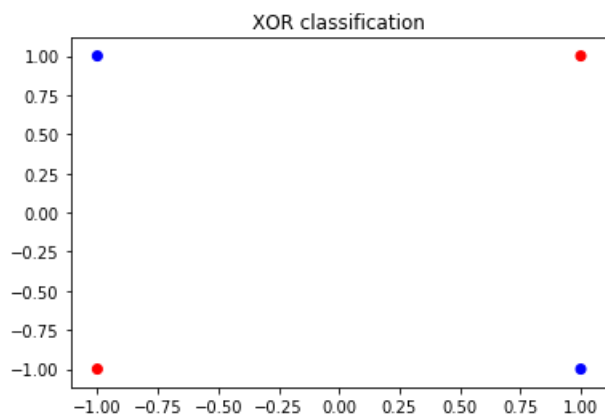
x = data[:, 0]
y = data[:, 1]
colors = ['red' if p == -1 else 'blue' for p in predictions]
fig, _ = plt.subplots()
plt.scatter(x, y, c=colors)
plt.title("XOR classification")
fig.savefig('one-qubit-XOR.png')

```

Out [17]:

```
out[1]:
```

```
Text(0.5, 1.0, 'XOR classification')
```



Global phase and the encoding scheme

Our particular quantum encoding scheme does map one-to-one from classical vectors to quantum state vectors; however, two quantum vectors are considered physically indistinguishable from each other if they only differ by a constant ($|c|^2=1$), called global phase. This actually comes in handy in our classification of the XOR data and leads to two possible weight vectors for the data.

```
In [18]:
```

```
def sample(data, labels, train=one_qubit_train, n=100):
    d = {}
    d = defaultdict(lambda:0,d)
    for i in range(n):
        d[repr(train(data, labels))] += 1
    return d
```

```
d_XOR =sample(data, XOR_labels)
print(d_XOR)
```

```
defaultdict(<function sample.<locals>.<lambda> at 0x7facc44b9158>, {'array([ 1., -1.])': 62, 'array([-1., 1.])': 38})
```

```
In [19]:
```

```
def print_weights(sample_dict, title="", save=False):
    all_strings = ['array([1., 1.])', 'array([ 1., -1.])', 'array([-1., 1.])', 'array([-1., -1.])']
    key_strings = [k[6:-1] for k in all_strings]
    weight_counts = [0,0,0,0]

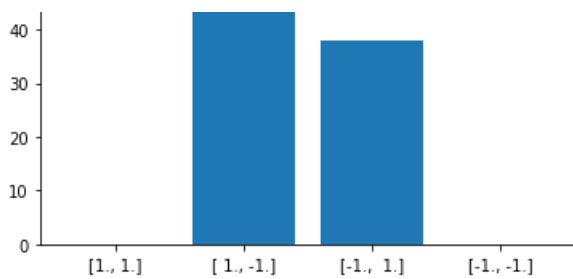
    for key in sample_dict.keys():
        idx = all_strings.index(key)
        weight_counts[idx] = sample_dict[key]

    fig, _ = plt.subplots()
    plt.bar([0, 1, 2, 3], weight_counts)
    plt.xticks([0, 1, 2, 3], key_strings)
    plt.title(title)
    if save==True:
        fig.savefig(title + '.png')
```

```
In [20]:
```

```
print_weights(d_XOR, title="Learned XOR weight vectors", save=True)
#print_weights(d_XOR, title="Learned XOR weight vectors")
```





The limits of a pattern classifier

For low dimensional data (e.g. data that can be encoded on 1 or 2 qubits) the classifier acts as if it is picking out a single input and thus the negative of that input to result in positive classification. In other words the perceptron will only be activated when the input is equal to weights (or it's negative).

input	output
00	0
01	0
10	1
11	1

Could we learn this data?

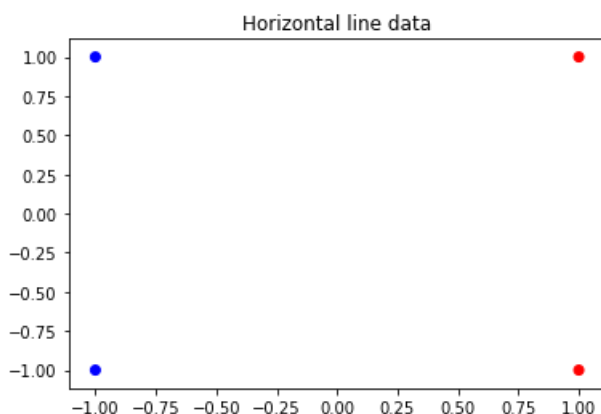
In [21]:

```
limit_labels = np.array([-1, -1, 1, 1])

x = data[:, 0]
y = data[:, 1]
colors = ['red' if l==1 else 'blue' for l in limit_labels]
fig, _ = plt.subplots()
plt.scatter(x, y, c=colors)
plt.title("Horizontal line data")
#fig.savefig('Horizontal line data.png')
```

Out[21]:

Text(0.5, 1.0, 'Horizontal line data')



Can we learn this?

In [22]:

```
limit_weight = one_qubit_train(data, limit_labels)
```

Not converging
w: [-1. -1.]

The 1-qubit classifier cannot activate for both of the positive labeled inputs (see the figure showing all possible outcomes), thus we

The 1-qubit classifier cannot activate for both of the positive labeled inputs (see the figure showing all possible outcomes), thus we cannot learn this function.

What about boolean AND

Let's generate data for the Boolean AND function.

input	AND
00	0
01	0
10	0
11	1

I'll consider AND output of '1' to be a positive label (1) and AND output of '0' to be a negative label (-1)

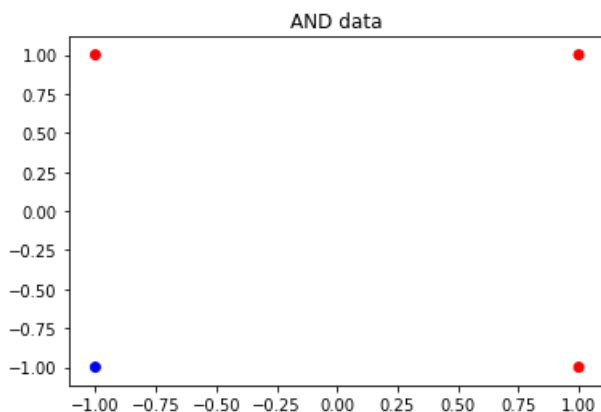
In [23]:

```
AND_labels = np.array([-1, -1, -1, 1])

x = data[:, 0]
y = data[:, 1]
colors = ['red' if l==1 else 'blue' for l in AND_labels]
fig, _ = plt.subplots()
plt.scatter(x, y, c=colors)
plt.title("AND data")
#fig.savefig('Horizontal line data.png')
```

Out[23]:

Text(0.5, 1.0, 'AND data')



What about learning this?

In [24]:

```
AND_weight = one_qubit_train(data, AND_labels)
```

Not converging
w: [-1. -1.]

Again. This function does correspond to a possible output of the perceptron. However, below I propose a modified training algorithm that does converge for the boolean AND function, and gives the weight output exactly equal to the postively labeled input (and \neg the negative).

A modified training algorithm

Due to global phase, the algorithm cannot learn the AND data set. It cannot classify as an input of $[-1, -1]$ $([1, 1])$ as positive without also classifying $[1, 1]$ $([0, 0])$ as positive.

Here, I propose the a modification to the training algorithm to learn the "correct" weights, the weights you would expect a classical perceptron to learn.

This algorithm gives the weight vector one would expect in the classical version of the algorithm, and can be used when an input and it's negative have different labels. It can also be used in cases when one does not wants only a specific weight vector, and does not want the negative of a desired weight vector to be returned.

The algorithm:

- First, for each positively labeled data, find the negative data.
- If the negative data is not also positively labeled, add it to a list of negative labels
- Initialize a boolean variable full_loop=False, this serves as a second misclassification check
- Training:
 - Loop through all data
 - In the training procedure, if an example is misclassified and does not belong to the list of negative labels, adjust the weights and increment the misclassification counter and set the boolean variable full_loop=False
 - if an example is misclassified and DOES belong to the list of negative labels, adjust the weights but do NOT increment the misclassification counter
 - At the end of an epoch (training loop)
 - if m>0 repeat Training
 - if m=0 and full_loop=False set full_loop=True and repeat Training
 - if m=0 and full_loop=True return the learned weights

In [25]:

```
def find_neg_labels(data, labels):
    pos_data = [data[i] for i,l in enumerate(labels) if l == 1]
    neg_pattern_neg_label = []
    for p in pos_data:
        neg_data = -1*p
        if neg_data in data:
            neg_idx = [i for i, x in enumerate(data) if np.array_equal(x, neg_data)]
            for idx in neg_idx:
                if labels[idx] != 1:
                    neg_pattern_neg_label.append(idx)

    #print(pos_data)
    return neg_pattern_neg_label

def modified_train(data, labels, neg_labels=[], vec_circ_dict=vec_circ_dict,
                  threshold=0.5, lp=0.5, ln=0.5, max_iter=25):
    # Initialize a random w
    w = np.array(random.choices([-1.0,1.0], k=int(math.log(len(labels), 2)))) # k is classical dim

    full_loop = False
    epochs = 1
    while True:
        m = 0
        for d_idx, d in enumerate(data):
            i = d

            # Get output from classification circuit
            output = one_qubit_classifier(i, w)

            # Check for misclassification
            if np.sign(output) != np.sign(labels[d_idx]):
                if d_idx not in neg_labels:
                    m += 1
                    full_loop = False

            # case when classified positive (activated) but should be negative
            if np.sign(output) > 0:
                same_idx = [i for i, z in enumerate(zip(i,w)) if z[0] == z[1]]
                # randomly select elements to switch signs
                if len(same_idx) > len(w)*ln:
                    random.shuffle(same_idx)
                    same_idx = same_idx[:int(len(w)*ln)]
                # change signs of w
                w[same_idx] *= -1

            # case when classified negative but should be positive
        else:
            diff_idx = [i for i, z in enumerate(zip(i,w)) if z[0] != z[1]]
```

```

        # randomly select elements to switch signs
        if len(diff_idx) > len(w)*lp:
            random.shuffle(diff_idx)
            diff_idx = diff_idx[:int(len(w)*lp)]
        # change signs of w
        w[diff_idx] *= -1
        #print("w:", w)

    if m == 0:
        if full_loop == False:
            full_loop = True
        else:
            #print("converged in {} epochs".format(epochs))
            #print("w:", w)
            break
    elif epochs == max_iter:
        print("Not converging")
        print("w:", w)
        break
    else:
        epochs += 1

    return w

def modified_sample(data, labels, neg_labels, train=modified_train, n=100):
    d = {}
    d = defaultdict(lambda: 0, d) # set initial values to 0
    for i in range(n):
        d[repr(train(data, labels, neg_labels))] += 1
    return d

```

In [26]:

```

AND_labels = np.array([-1, -1, -1, 1])
print("data\n", data)
print("labels", AND_labels)

```

```

data
[[ 1.  1.]
 [ 1. -1.]
 [-1.  1.]
 [-1. -1.]]
labels [-1 -1 -1  1]

```

In [27]:

```

# shuffle data
AND_labels = np.array([-1, -1, -1, 1])
indices = np.arange(len(AND_labels))
np.random.shuffle(indices)

print(indices)
shuffle_data = data[indices]
AND_labels = AND_labels[indices]
print("data\n", shuffle_data)
print("labels", AND_labels)

```

```

[1 3 2 0]
data
[[ 1. -1.]
 [-1. -1.]
 [-1.  1.]
 [ 1.  1.]]
labels [-1  1 -1 -1]

```

In [28]:

```

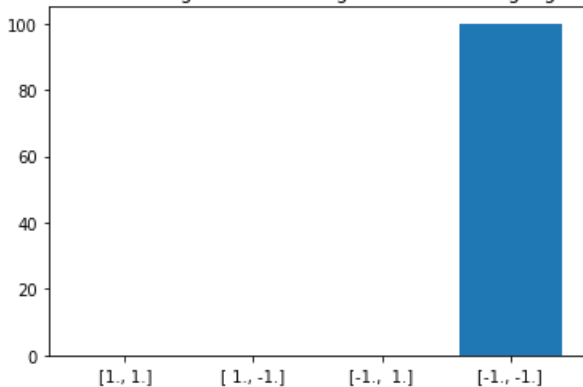
neg_labels = find_neg_labels(shuffle_data, AND_labels)
AND_weight = modified_train(shuffle_data, AND_labels, neg_labels)

```

In [29]:

```
d_AND = modified_sample(shuffle_data, AND_labels, neg_labels, train=modified_train, n=100)
# print_weights(d_AND, title="Learned AND weight vectors using modified training algorithm", save=True)
print_weights(d_AND, title="Learned AND weight vectors using modified training algorithm")
```

Learned AND weight vectors using modified training algorithm



We can see that the modified training algorithm does converge for the boolean AND data set and converges to the weight you would expect the a classical version of the algorithm to do so. However, it should be noted that using this weights for classification in the quantum circuit would cause an activation for inputs that are negative to the weight vector, in this case the [1, 1] vector, which is not true to the boolean AND.

This algorithm may be seen a way to control the outcomes of training when one cares about global phase. Furthermore, if one wanted to use the quantum perceptron algorithm to learn weights, and then used the learned weights for future classification in using a classical algorithm, it would give the desired results. Hypothetically, this might occur if there was a speed-up available in using the quantum version of the algorithm for training.

Further verification of the algorithm

First, I'll show that the algorithm is invariant to the order of the data by testing the algorithm on all permutations of training data order.

In [30]:

```
indices = np.arange(len(AND_labels))
orderings = list(itertools.permutations(indices))

AND_labels = np.array([-1, -1, -1, 1])
data = np.array([k2vec(i,2) for i in range(4)])

for o in orderings:
    print(o)
    o = np.array(o)
    order_labels = AND_labels[o]
    order_data = data[o]

    neg_labels = find_neg_labels(order_data, order_labels)
    d_AND = modified_sample(order_data, order_labels, neg_labels, train=modified_train, n=3)
    errors = 0
    if d_AND['array([-1., -1.])'] != 3:
        errors += 1
        print("error on permutation", o)

if errors == 0:
    print("Algorithm is invariant to the order of data!")
```

```
(0, 1, 2, 3)
(0, 1, 3, 2)
(0, 2, 1, 3)
(0, 2, 3, 1)
(0, 3, 1, 2)
(0, 3, 2, 1)
(1, 0, 2, 3)
(1, 0, 3, 2)
(1, 2, 0, 3)
(1, 2, 3, 0)
(1, 3, 0, 2)
```

```
(1, 3, 2, 0)
(2, 0, 1, 3)
(2, 0, 3, 1)
(2, 1, 0, 3)
(2, 1, 3, 0)
(2, 3, 0, 1)
(2, 3, 1, 0)
(3, 0, 1, 2)
(3, 0, 2, 1)
(3, 1, 0, 2)
(3, 1, 2, 0)
(3, 2, 0, 1)
(3, 2, 1, 0)
```

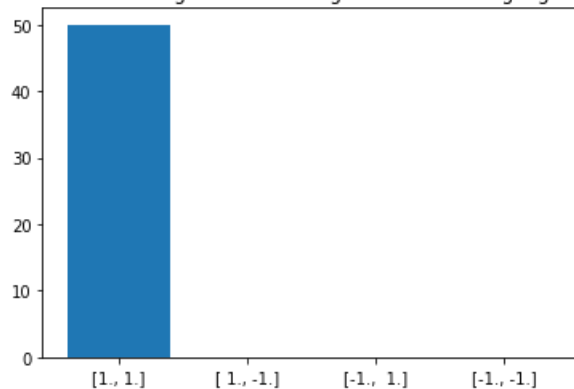
Algorithm is invariant to the order of data!

Now I will show that the selective training algorithm works for all data with one positive label

In [31]:

```
one_labels = np.array([1, -1, -1, -1])
neg_labels = find_neg_labels(data, one_labels)
d_one = modified_sample(data, one_labels, neg_labels, train=modified_train, n=50)
#print_weights(d_one, title="Learned ONE weight vectors using modified training algorithm", save=True)
print_weights(d_one, title="Learned ONE weight vectors using modified training algorithm")
```

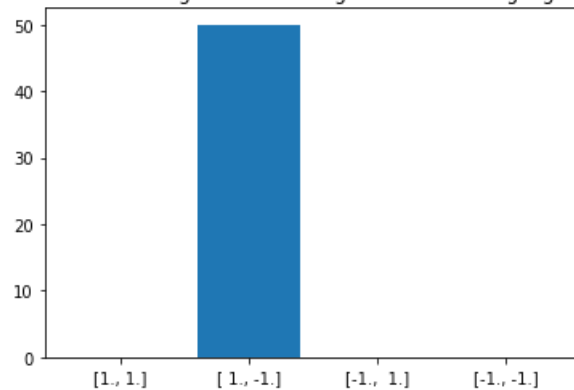
Learned ONE weight vectors using modified training algorithm



In [32]:

```
two_labels = np.array([-1, 1, -1, -1])
neg_labels = find_neg_labels(data, two_labels)
d_two = modified_sample(data, two_labels, neg_labels, train=modified_train, n=50)
#print_weights(d_two, title="Learned TWO weight vectors using modified training algorithm", save=True)
print_weights(d_two, title="Learned TWO weight vectors using modified training algorithm")
```

Learned TWO weight vectors using modified training algorithm

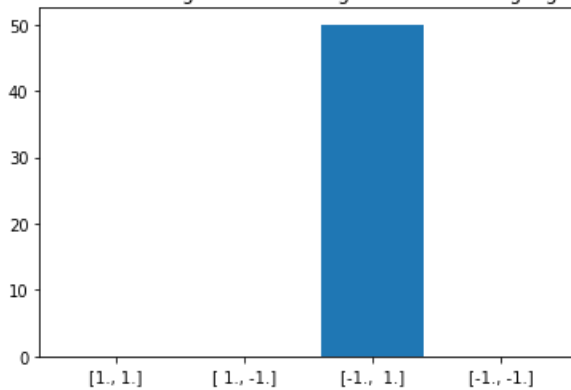


In [33]:

```
three_labels = np.array([-1, -1, 1, -1])
neg_labels = find_neg_labels(data, three_labels)
```

```
d_three = modified_sample(data, three_labels, neg_labels, train=modified_train, n=50)
#print_weights(d_three, title="Learned THREE weight vectors using modified training algorithm", save=True)
print_weights(d_three, title="Learned THREE weight vectors using modified training algorithm")
```

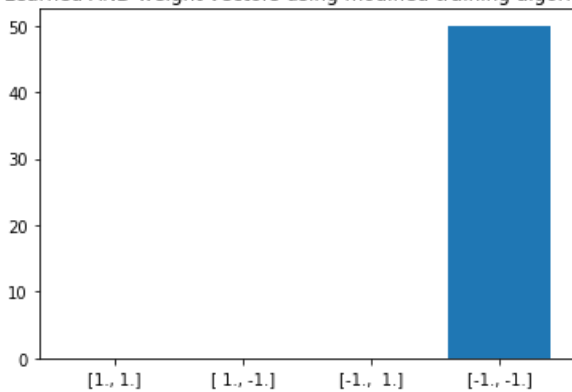
Learned THREE weight vectors using modified training algorithm



In [34]:

```
AND_labels = np.array([-1, -1, -1, 1])
neg_labels = find_neg_labels(data, AND_labels)
d_AND = modified_sample(data, AND_labels, neg_labels, train=modified_train, n=50)
#print_weights(d_AND, title="Learned AND weight vectors using modified training algorithm", save=True)
print_weights(d_AND, title="Learned AND weight vectors using modified training algorithm")
```

Learned AND weight vectors using modified training algorithm



In []: