PHYS 576 Final Project

Cory Lorenz

June 2020

1 Project Overview

The goal of this project is to understand the technique and replicate the results of the paper "An Artificial Neuron Implemented on an Actual Quantum Processor" [1].

The next section will provide an overview of the paper by both re-deriving key results with additional detail, and providing examples and pseudo code to aid understanding. The subsequent section will show the results of experiments to replicate the results in the paper, both simulated and on the IBM Quantum Experience hardware. The final section provides the source code used to run the experiments.

2 Overview of Paper

2.1 The Classical Perceptron

On a classical computer, a perceptron is computed via the dot (or inner) product of two vectors known as the input vector and the weight vector. The weight vector represents your model of the system, and the input vector the new information you are evaluating. The value of the dot product is then passed through a non-linear activation function which defines the perceptron output y [2].

$$y = f(w^T i) = f(\overrightarrow{w} \cdot \overrightarrow{i}) = f(\sum_j w_j i_j)$$

Canonical activation functions (f) are the sigmoid function, or the hyperbolic tangent function since their output is over the domain $[0, 1]$ and can be thought of as a probability. One can add a simple threshold (e.g. 0.5) if a binary decision is required.

Some of the earliest work on perceptrons was done by McCullogh and Pitts who worked with binary values of w and i, that is each $w_j, i_j \in \{-1, 1\}$. These neurons are less powerful, but can still be used for pattern recognition. The authors introduce a new design for quantum computation that mimics a perceptron of this form.

2.2 The Quantum Perceptron

Given a system with N qubits, and therefore $m = 2^N$ basis states, the referenced design starts by encoding the m-dimensional weight and input vectors as follows:

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

So for example, with an $N = 2$ qubit system, we have $2^2 = 4$ basis states, and would encode a 4 element vector as follows:

$$\vec{i} = \begin{bmatrix} 1 \\ \text{-1} \\ 1 \\ \text{-1} \end{bmatrix} \rightarrow |\psi_i\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$$

One can then compute the dot product of the two vectors:

$$\begin{aligned} \langle\psi_i|\psi_w\rangle &= \left(\frac{1}{\sqrt{m}}\sum_{j=0}^{m-1} i_j\langle j|\right)\left(\frac{1}{\sqrt{m}}\sum_{j'=0}^{m-1} w_{j'}|j'\rangle\right) \\ &= \frac{1}{m}\sum_{j=0}^{m-1}\sum_{j'=0}^{m-1} i_j w_{j'}\langle j|j'\rangle \\ &= \frac{1}{m}\sum_{j=0}^{m-1}\sum_{j'=0}^{m-1} i_j w_{j'}\delta_{j'j} \\ &= \frac{1}{m}\sum_{j=0}^{m-1} i_j w_j \\ &= \frac{1}{m}\vec{i}\cdot\vec{w} \end{aligned}$$

And the measurement operation can be used as the non-linear activation function (since it squares the result) and to put our output on the probabalistic interval $[0, 1]$. Note that because the measurement of the state results in a probability that is squared (e.g. $|\langle\psi_i|\psi_w\rangle|^2$) we are unable to distinguish the overall sign of the dot product. For instance, anti-parallel vectors will have the same value as parallel ones. This we be important to keep in mind when reviewing the results.

2.3 Circuit Overview

In a future quantum computing system, we would in theory obtain $|\psi_i\rangle$ from some sort of quantum memory. However, with the current state of the art, we need to construct the state starting from the $|00...0\rangle$ state that our quantum circuits start in. In mathematical terms, we need to construct a unitary transform U_i such that:

$$|\psi_i\rangle = U_i|0\rangle^{\otimes N}$$

Once we have set up our input vector, we need to find a unitary transform U_w such that:

$$U_w|\psi_w\rangle = |m-1\rangle = |1\rangle^{\otimes N}$$

The completely general result of applying this unitary on the input ket is:

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

Since $U_w^\dagger U_w = \mathbb{1}$ and $\langle\psi_w|U_w^\dagger = (U_w|\psi_w\rangle)^\dagger = \langle m-1|$ we can insert the identity to see how we can measure the dot product:

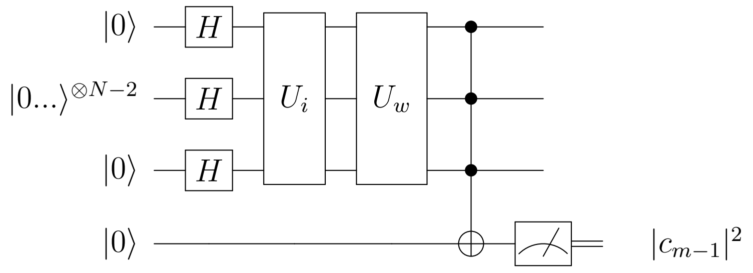
$$\begin{aligned} \langle\psi_w|\psi_i\rangle = \langle\psi_w|\mathbb{1}|\psi_i\rangle &= \langle\psi_w|U_w^\dagger U_w|\psi_i\rangle \\ &= (\langle\psi_w|U_w^\dagger)(U_w|\psi_i\rangle) \\ &= \langle m-1|\phi_{i,w}\rangle \\ &= c_{m-1} \end{aligned}$$

Where orthonormality of the computational basis states was used to cancel all terms except the one associated with $m-1$. Thus, by using a $C^N X$ gate targeting an ancilla qubit and measuring it, we can obtain c_{m-1} and thus the dot product $\overrightarrow{i} \cdot \overrightarrow{w}$, since:

$$C^N X|\phi_{i,w}\rangle|0\rangle = C^N X \sum_{j=0}^{m-1} c_j|j\rangle|0\rangle = \sum_{j=0}^{m-2} c_j|j\rangle|0\rangle + c_{m-1}|m-1\rangle|1\rangle$$

The key fact is that $C^N X|m-1\rangle|0\rangle = C^N X|1\rangle^{\otimes N}|0\rangle = |1\rangle^{\otimes N}|1\rangle$.

Thus, our general circuit diagram is as follows:



Our final step is to determine how to construct the necessary unitary transforms.

2.4 Implementing the Unitaries: Sign Flip Blocks Hypergraph States

The paper proposes two methods to do this: a brute force technique based on Sign-Flip Blocks (SFB) and a more efficient Hypergraph States Generation Subroutine (HSGS). As explained below, the brute-force technique requires many gates and ancilla qubits to implement. The authors showed in their paper that this caused the experimental results to be poor, even for a 2 qubit state. Thus, after explaining the SFB technique which is useful conceptually, this project focuses on implementing the HSGS technique.

In essence, the SFB technique starts by creating equi-superposition state via Hadamard gates. Next, each negative sign on a computational basis state is applied via an SFB, defined as:

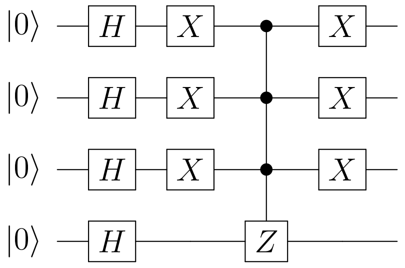
$$SFB_{N,j}|j'\rangle = \begin{cases} |j'\rangle \text{ if } j \neq j' \\ -|j'\rangle \text{ if } j = j' \end{cases}$$

A single SFB can be implemented with a multiply-controlled-Z operation that matches the state of interest in order to flip its sign. In the worst case, 2^{m-1} SFBs would be required to implement U_i (the -1 in the exponent is due to the 2-norm applied when measuring, as described earlier).

As an illustration, imagine the following state for an $N = 4$ system:

$$|\psi_i\rangle = |0000\rangle - |0001\rangle + \sum_{j=2}^{m-1} |j\rangle$$

The unitary transform U_i for this requires a single SFB, as illustrated here:



However, if half of the signs were negative (the worst-case for $N = 4$), 8 SFBs would be required in total. As a further complication, most toolkits would require that the multiply controlled gate must be built up with ancilla qubits and Toffoli gates, making the resulting circuit more complex and providing even more opportunities for errors to occur. This was demonstrated in Figure 3e of [1].

Thus, the authors developed the HSGS technique the reduce the number of gates needed. Apparently hypergraph states are used in many quantum algorithms, and they sound really cool, but that knowledge is not required to understand the cleverness of the technique. The efficiency of the technique is obtained by applying the required controlled Z gates with the fewest number of controls (e.g. the least number of "1" qubits) first, tracking their impact on other states, and fixing any sign errors induced on the states which require higher numbers of controlled qubits. What is important is that a controlled Z with a higher number of controls can never change the basis states which required a controlled Z with a lesser number of controls. (Note that, as we learned in class, the application of the controlled Z gate can be done on any of the qubits under target or control since $Z|0\rangle = |0\rangle$ and $Z|1\rangle = -|1\rangle$). An additional advantage is that the pairs of X gates are not required, and you have a chance of flipping the sign for a state that needed it anyways, saving additional gates.

Some examples can aid understanding. Let $C^N Z_{i_1 \ldots i_N}$ be a Z gate with controls on the $i_1, i_2, ..i_N^{th}$ qubits. A state like $|0100\rangle$ would require only a singly controlled Z gate (i.e. $C^1 Z_2$) and applying that gate on the 2nd qubit would also change the sign of a state like $|0101\rangle$. If that higher-controlled state needed to have its sign flipped back, we would require a $C^2 Z_{2,4}$ gate, but application of that gate would not change any lower-controlled state since $C^2 Z_{2,4}|0100\rangle = |0100\rangle$ due to the fourth qubit being zero.

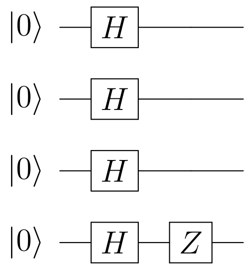
Pseudocode for the HSGS technqiue is given below:

```
def hsgs(circuit, desired_state_signs, N):

current_state_signs = all_ones(2^N)

for i=1 to N:
    i_states = get_all_states_by_count_of_one_qubits(count=i)
    for state in i_states:
        if desired_state_signs[state] != current_state_signs[state]:
            one_qubits = get_one_qubits(state)
            circuit.apply_controlled_z(one_qubits)

            update_states = get_all_states_with_specific_one_qubits(one_qubits)
            for up_state in update_states:
                current_state_signs[up_state] *= -1

return circuit
```

As a cherry-picked example of the efficiency of HSGS, consider the following state:

$$|\psi_i\rangle = |0000\rangle - |0001\rangle + |0010\rangle - |0011\rangle + |0100\rangle - |0101\rangle + ... = \sum_{j=0}^{m-1} (-1)^{j(\text{mod}2)}|j\rangle$$

The HSGS circuit to implement this unitary would be as follows (in contrast to the worst-case 8 SFBs which would be required to brute-force flip the sign of each state):



Luckily, with some additional cleverness, we can use HSGS and a few extra gates to construct U_w as well. We note that given $|\psi_w\rangle$ we can return it to the equi-superposition state by flipping the sign of each basis state in $|\psi_w\rangle$ which has a negative coefficient. From the equiposition state, we can obtain the desired $|m-1\rangle$ state via Hadamard and NOT gates. That is to say:

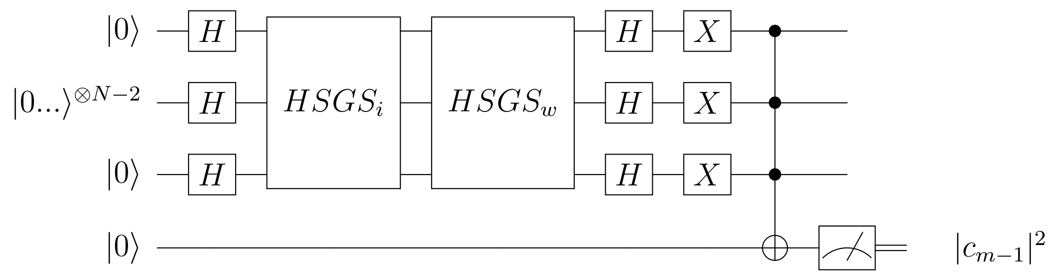
$$U_w = X^{\otimes N} H^{\otimes N} HSGS(\psi_w)$$

Since:

$$\begin{aligned} X^{\otimes N} H^{\otimes N} HSGS(\psi_w)|\psi_w\rangle &= X^{\otimes N} H^{\otimes N} \sum_{j=0}^{m-1} |j\rangle \\ &= X^{\otimes N} |0\rangle^{\otimes N} \\ &= |1\rangle^{\otimes N} = |m-1\rangle \end{aligned}$$

Which is the definition of the unitary transform U_w.

The final circuit diagram is given by:



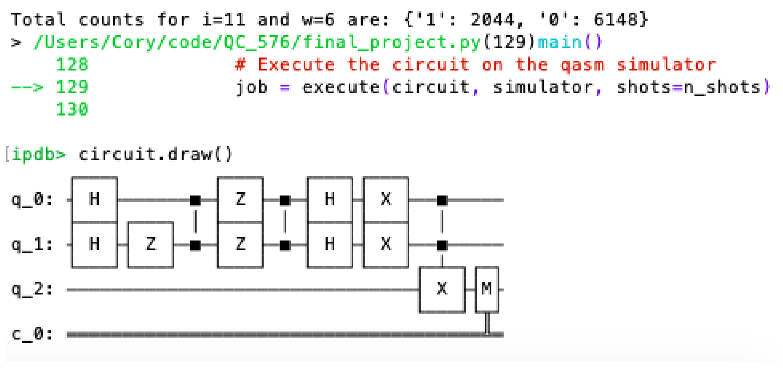
where the HSGS blocks are compiled via the pseudocode given above, for the i and w vectors.

3 Experimental Results

With the techniques outlined above, Python code was generated to implement the HSGS unitaries for both the data initialization and calculating the inner product with the weight vector for arbitrary choices of the i and w vectors.

Figure 1 shows the circuit diagram as output by Qiskit of the $i = 11$ and $w = 7$ case which was shown in Figure 3c of the original paper. Comparing the two figures, we see that the implemented HSGS alorgithm for 2 qubits matches the one used by the authors. (Note that the controls with the lines between them represent the controlled-Z gate, a notational shorthand used because the Z gate can be applied on any of the control qubits, as discussed above).

Figure 2 shows the result of simulating the developed quantum circuits for all possible combinations of i and w vectors. The circuits were simulated using IBM's Aer toolkit. Upon comparison to Figure 3d of the original paper, we see that the results are replicated.



```
Total counts for i=11 and w=6 are: {'1': 2044, '0': 6148}
> /Users/Cory/code/QC_576/final_project.py(129)main()
    128             # Execute the circuit on the qasm simulator
--> 129             job = execute(circuit, simulator, shots=n_shots)
    130

ipdb> circuit.draw()
```

Figure 1: Qiskit circuit representation for the i=11 and w=7 experiment

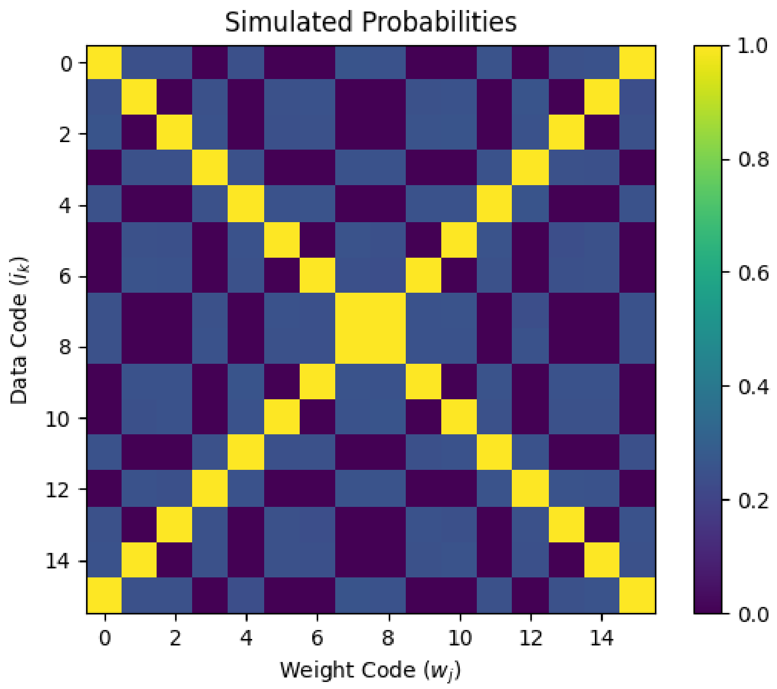


Figure 2: Results of simulating the output of all combinations of i and w vectors

It is instructive to explain the pattern of values observed in Figure 2. There are only 3 values which are observed: 0, 1/4, and 1. How these values are obtained is shown below for $i = 0, 1, 3$ and $w = 0$ (e.g. the upper left of the first column).

$$\begin{aligned}
|\psi_0\rangle = |\psi_{0000}\rangle &= \frac{1}{2}((-1)^0|00\rangle + (-1)^0|01\rangle + (-1)^0|10\rangle + (-1)^0|11\rangle) \\
&= \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \\
|\psi_1\rangle = |\psi_{0001}\rangle &= \frac{1}{2}((-1)^0|00\rangle + (-1)^0|01\rangle + (-1)^0|10\rangle + (-1)^1|11\rangle) \\
&= \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle) \\
|\psi_3\rangle = |\psi_{0011}\rangle &= \frac{1}{2}((-1)^0|00\rangle + (-1)^0|01\rangle + (-1)^1|10\rangle + (-1)^1|11\rangle) \\
&= \frac{1}{2}(|00\rangle + |01\rangle - |10\rangle - |11\rangle)
\end{aligned}$$

We can then compute, using the orthonormality of the computational basis states:

$$
\begin{aligned}
\langle \psi_{i=0} | \psi_{w=0} \rangle &= \frac{1}{4}(\langle 00|00\rangle + \langle 01|01\rangle + \langle 10|10\rangle + \langle 11|11\rangle) \\
&= \frac{1}{4}(1+1+1+1) = 1 \\
|\langle \psi_{i=0} | \psi_{w=0} \rangle|^2 &= 1 \\
\langle \psi_{i=1} | \psi_{w=0} \rangle &= \frac{1}{4}(\langle 00|00\rangle + \langle 01|01\rangle + \langle 10|10\rangle - \langle 11|11\rangle) \\
&= \frac{1}{4}(1+1+1+1) = \frac{1}{2} \\
|\langle \psi_{i=1} | \psi_{w=0} \rangle|^2 &= \frac{1}{4} \\
\langle \psi_{i=3} | \psi_{w=0} \rangle &= \frac{1}{4}(\langle 00|00\rangle - \langle 01|01\rangle + \langle 10|10\rangle - \langle 11|11\rangle) \\
&= \frac{1}{4}(1+1-1-1) = 0 \\
|\langle \psi_{i=3} | \psi_{w=0} \rangle|^2 &= 0
\end{aligned}
$$

Thus, we see that the values of 0, 1/4, and 1 correspond to having sign differences in 2, (1 or 3), (0 or 4) computational basis states, respectively. We are unable to distinguish between certain differences due to the 2-norm eliminating sign information when we measure.

Figure 3 shows the result of running the developed quantum circuits for all possible combinations of i and w vectors on IBM's quantum computers in Yorktown (backend "ibmq_5_yorktown"). We see that there is some noise that is introduced upon running the system on real hardware, but the expected pattern seen in Figure 2 is maintained. Upon comparison to Figure 3f of the original paper we see that, up to random noise variation, the results are replicated. A similar maximum probability near 0.8 is achieved, with the underlying structure remaining intact.

Figure 4 shows the results of simulating the HSGS quantum circuits for the $N = 4$ qubit case with a handful of selected data vectors for a cross-patterned weight vector. Upon comparison to

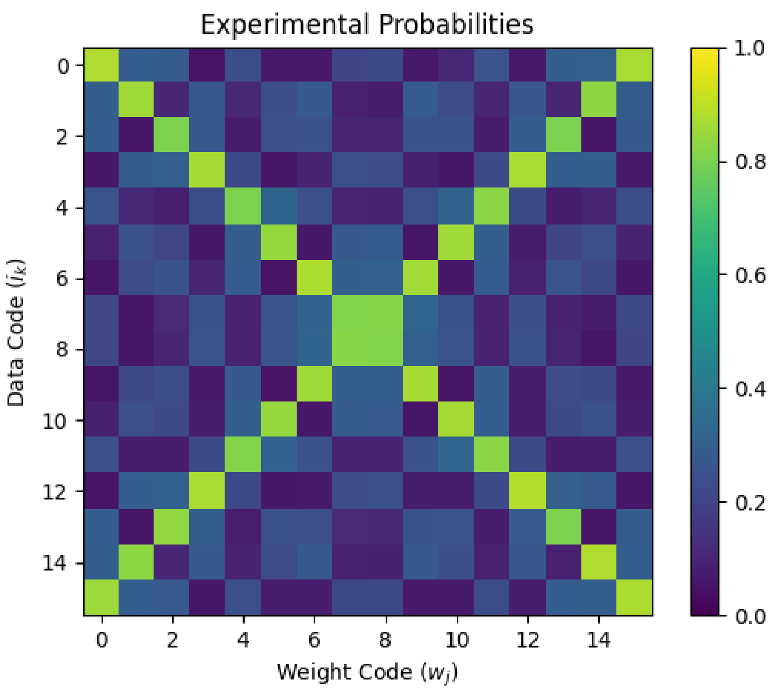


Figure 3: Results of hardware experiments for all combinations of i and w vectors

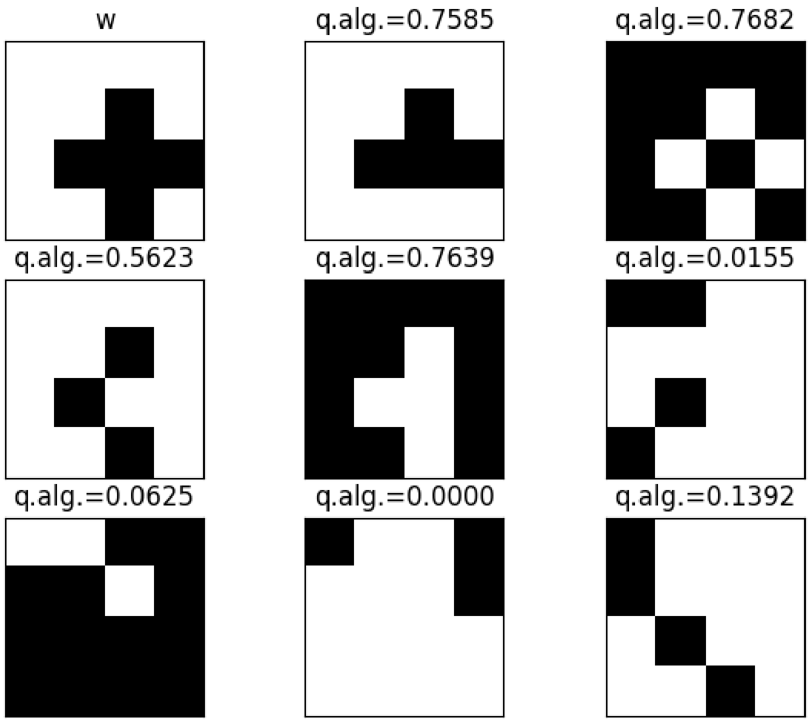


Figure 4: Results of simulated experiments for selected 4 Qubit cases

Figure 4 of the original paper (which matched well with the theoretical values) we see that, up to some small variations due to random value generation in the simulator, the results are replicated.

As an explicit example of the theoretical calculation, take the case of the image in the second row, first column. All but 14 of the 16 pixels have the same value, with 2 pixels different. We can thus compute the probability similar to the $N = 2$ qubit case as:

$$Pr = \left(\frac{14-2}{16}\right)^2 = \left(\frac{3}{4}\right)^2 = 0.5625$$

Which agrees well with the simulated result of 0.5623.

The Python code used generate these plots is provided in the Appendix.

4 Conclusions

In conclusion, the major results of [1] have been replicated using the methods originally described in the paper, and re-derived with additional detail and examples in this report. Python code to compile HSGS unitary transforms for the IBM Qiskit has been developed and hopefully aids in understanding.

Appendix - Python Source Code

Shown below is the code used to generate both the simulated and experimental results. The code can also be found in an easier to read format with syntax highlighting on Github:

```
import argparse
from collections import defaultdict

import numpy as np
import matplotlib.pyplot as plt

from qiskit import QuantumCircuit, execute, Aer, IBMQ


def code_to_sign_pattern(code, n_qubits=2):
    """
    Converts an integer code to its binary vector representation.
    For instance, 7 = 0111 -> (-1)^[0,1,1,1] = [1, -1, -1, -1]

    Currently only works for the 2 qubit case

    inputs: integer,
    outputs: list of +-1 values
    """

    m = 2**n_qubits

    signs = [1]*m
    modulus = code
    for idx in range(m):
        if modulus >= 2**(m-1-idx):
            signs[idx] = -1
            modulus -= 2**(m-1-idx)

    return signs

def implement_HSGS_nqubits(circuit, signs, n_qubits=2):
    """
    Implementation of the hypergraph states generation subroutine for N qubits

    HSGS is an efficient method for implementing the unitary transforms required
    to both initialize a data vector, as well as the rotation which implements
    the dot product of it with a weight vector

    inputs:
        circuit: work in progress IBM QuantumCircuit object
        signs: list of signs on the computational basis states
        count_map: (dict) maps the count of "1" qubits to
            a list of the comp basis indexes with that number of "1" qubits

    outputs:
        circuit: IBM QuantumCircuit object with the HSGS unitary transform added
             to it
    """

    m = 2**n_qubits
    # Define the binary representation of each state
    # e.g. for 2 qubits binary_reps = ["00", "01", "10", "11"]
    str_format = "{:0" + str(n_qubits) + "b}"
    binary_reps = [str_format.format(j) for j in range(m)]

    # Need to generate the count map
    count_map = defaultdict(list)
    for idx, bin_rep in enumerate(binary_reps):
        n_ones = sum(s == "1" for s in bin_rep)
        count_map[n_ones].append(idx)
    count_map = dict(count_map)

    # Initialize array to track which sign we have applied to each state
    implemented_signs = [1]*m

    # Flip all of the signs if coefficient for |00> is -1
    if signs[0] == -1:
        signs = [s*-1 for s in signs]

    # Increment over the count of "1" qubits in the basis state
    for count in range(1,n_qubits+1):
        # Loop over all of the states which have that count of "1" qubits
        for idx in count_map.get(count, []):
            # See if the sign that is implemented is not correct
            if signs[idx] != implemented_signs[idx]:
                # Get the binary representation of this index
                bin_rep = binary_reps[idx]
                # The flip is controlled by the qubits that are a "1"
                apply_qbits = [i for i, x in enumerate(bin_rep) if x == "1"]

                # Flip the sign of those qubits, using the appropriate gate
                if count == 1:
                    circuit.z(apply_qbits[0])
                elif count == 2:
                    # With a controlled Z gate, it doesn't matter which is
                       control or target
                    circuit.cz(apply_qbits[0], apply_qbits[1])
                else:
                    ctrl_qubits = [circuit.qubits[i] for i in apply_qbits[1:]]
                    targ_qubit = circuit.qubits[apply_qbits[0]]
                    # CRZ gate with theta=-pi is equiv to a CZ gate up to global
                        phase
                    circuit.mcrz(-np.pi, ctrl_qubits, targ_qubit)

                # Update the signs to be what is currently implemented.
                for i, bin_rep in enumerate(binary_reps):
                    # Convert to numpy array so we can do vector ops
                    bool_rep = np.array([s == '1' for s in bin_rep])
                    if np.all(bool_rep[apply_qbits]):
                        implemented_signs[i] *= -1

    return circuit

def create_perceptron_circuit(data_code, weight_code, n_qubits=2):
    """
    Implements a quantum circuit to compute the perceptron output a data and
       weight vector

    The data and weight vectors are binary, and specified by their integer code.
    The circuit returned  is suitable for simulation or execution with IBM
       qiskit

    inputs:
        data_code (int)
        weight_code (int)

    outputs:
        circuit qiskit QuantumCircuit Type
    """

    # Initialize the quantum circuit: N+1 qubits (N state, 1 work), 1 cbit
    circuit = QuantumCircuit(n_qubits+1, 1)
    # Set the 4 pixel qubits to equi-superposition state
    for qubit_idx in range(n_qubits):
        circuit.h(qubit_idx)

    # Convert the codes to the patterns of +1 and -1
    data_signs = code_to_sign_pattern(data_code, n_qubits)
    weight_signs = code_to_sign_pattern(weight_code, n_qubits)

    # Apply the unitary transform to initialize the data
    circuit = implement_HSGS_nqubits(circuit, data_signs, n_qubits)
    # Apply the unitary transform to implement dot product with weight vector
    circuit = implement_HSGS_nqubits(circuit, weight_signs, n_qubits)

    # Do the final H and X gates:
    for qubit_idx in range(n_qubits):
        circuit.h(qubit_idx)
    for qubit_idx in range(n_qubits):
        circuit.x(qubit_idx)

    # Final Toffoli gate
    circuit.mcx(control_qubits=list(range(n_qubits)), target_qubit=n_qubits)
    # Measure the ancilla qubit
    circuit.measure(
        qubit=[n_qubits],
        cbit=[0]
    )

    return circuit


def replicate_2qbit_results(backend, n_shots=8192):
    """
    Code to replicate HSGS results in Figure 3 of original paper
    """

    results_matrix = np.zeros((16,16))

    for data_code in range(16):
        for weight_code in range(16):
            circuit = create_perceptron_circuit(data_code, weight_code)

            # Pause simulation to check circuit is implemented as in the paper
            if data_code == 11 and weight_code == 7:
                # circuit.draw()
                import pdb; pdb.set_trace()

            # Execute the circuit on the qasm simulator
            job = execute(circuit, backend, shots=n_shots)

            # Grab results from the job and get counts
            result = job.result()
            counts = result.get_counts(circuit)
            print("Total counts for i={} and w={} are:".format(data_code,
                weight_code),counts)

            # Store normalized result in the matrix
            results_matrix[data_code, weight_code] = counts.get('1', 0)/n_shots
    # Visualize the results as an image
    plt.imshow(results_matrix)
    plt.xlabel('Weight Code ($w_j$)')
    plt.ylabel('Data Code ($i_k$)')
    plt.colorbar()
    plt.clim([0, 1])
    plt.title('Experimental Probabilities')
    plt.show()


def bit_string_to_img(bit_string):
    """
    Coverts from the bit string representation to its image
    """
    m = len(bit_string)
    img = np.zeros((m))
    for ii, bit in enumerate(bit_string):
        if bit == '0':
            img[ii] = 255

    hm = int(np.sqrt(m))
    return img.reshape((hm, hm))

def plot_bit_string(bit_string):
    """
    Plots the image associated with a data bit string
    """
    ax = plt.imshow(bit_string_to_img(bit_string), cmap='gray')
    ax.axes.xaxis.set_visible(False)
    ax.axes.yaxis.set_visible(False)

    return ax

def replicate_4qbit_results(backend, n_shots=8192):
    """
    Code to replicate the 4 qubit results from the original paper
    """
    # Read off the pixel values from Fig 4 of the paper
    w = '0000001001110010'
    i_bit_strings = [
        '0000001001110000',
        '1111110110101101',
        '0000001001000010',
        '1111110110011101',
        '1100000001001000',
        '0011110111111111',
        '1001000100000000',
        '1000100001000010'
    ]
    # Convert these to the integer codes
    weight_code = int(w,2)

    # Plot the weight image first
    sub = plt.subplot(3, 3, 1)
    plot_bit_string(w)
    sub.set_title('w')

    results = []
    for ii, i_bit_str in enumerate(i_bit_strings):
        data_code = int(i_bit_str, 2)
        circuit = create_perceptron_circuit(data_code, weight_code, n_qubits=4)
        # Run the quantum circuit and grab results
        job = execute(circuit, backend, shots=n_shots)
        result = job.result()

        # Get counts and normalize to get a probability
        counts = result.get_counts(circuit)
        prob = counts.get('1', 0)/n_shots
        results.append(prob)

        # Plot the data image and the associated probability
        sub = plt.subplot(3, 3, ii+2)
        plot_bit_string(i_bit_str)
        sub.set_title("q.alg.={:0.4f}".format(prob))

    plt.show()


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("--token", "-t",  help="Auth Token from IBM Website")
    parser.add_argument(
        "--backend", "-b",
        help="Hardware backend to use, eg 'ibmq_5_yorktown'. Uses Aer simulator
           if unprovided"
    )
    parser.add_argument(
        "-e", "--experiment", type=str, default='2qubits',
        help="Which experiment to run, either '2qubits' or '4qubits'"
    )
    cli_args = parser.parse_args()

    if cli_args.token is not None and cli_args.backend is not None:
        provider = IBMQ.enable_account(cli_args.token)
        backend = provider.get_backend(cli_args.backend)
        print("Using IBM Backend ", cli_args.backend)
    else:
        print("Backend not properly specified. Using Aer simulator.")
        backend = Aer.get_backend('qasm_simulator')

    if cli_args.experiment.lower() == "2qubits":
        print("Running the 2 qubit experiment")
        replicate_2qbit_results(backend)
    elif cli_args.experiment.lower() == "4qubits":
        print("Running the 4 qubit experiment")
        replicate_4qbit_results(backend)
    else:
        raise RuntimeError("Unknown experiment type ", cli_args.experiment)
```

References

[1] Francesco Tacchino et al. "An artificial neuron implemented on an actual quantum processor". In: *npj Quantum Information* 5.1 (Mar. 2019). ISSN: 2056-6387. DOI: 10.1038/s41534-019-0140-4. URL: http://dx.doi.org/10.1038/s41534-019-0140-4.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.