Untitled

April 11, 2021

1 IBM Quantum Creative Challenge (Our Team Project)

Quantum computers operate on different sets of physical laws than classical computers. One consequence of this is "interference". Like with waves, there can be constructive (additive) or destructive (subtraction) interference. Harnessing this property in a quantum circuit can lead to 'correct' answers interfering constructively and 'wrong' answers destructively, essentially nullifying them.

In this challenge, we'd like you to use the concept of quantum interference to make a creative project showcasing the idea.

2 Quantum Interference and Decoherence

A General Idea:

A fundamental idea in quantum computing is to control the probability a system of qubits collapses into particular measurement states. Quantum interference, a byproduct of superposition, is what allows us to bias the measurement of a qubit toward a desired state or set of states.

To fix ideas, consider the qubit in superposition $|\rangle$=(1,1)/2 and H$|\rangle$=$|1\rangle$. In other words, if the Hadamard gate is applied to $|\rangle$ then it will be observed in the pure state $|1\rangle$ with theoretical certainty upon measurement. This is quantum interference at its purest.

Observe, however, that even though $|\rangle$ may be measured in state $|0\rangle$ and $|1\rangle$ with equal probability, this is not the same as saying that H$|\rangle$=H$|0\rangle$ with probability 1/2, and H$|\rangle$=H$|1\rangle$ with probability 1/2. In fact, neither H$|0\rangle$ nor H$|1\rangle$ equals a pure state. Further, for H$|\rangle$=H$|0\rangle$ and H$|\rangle$=H$|1\rangle$ with equal probability, $|\rangle$ should have been incidentally measured before the application of the Hadamard gate. (After an unintended measurement, a qubit is said to be in a mixed state.) Quantum interference may therefore be disrupted by an incidental measurement of a system qubit. This phenomenon is called quantum decoherence and can be a major source of error when working with physical quantum computers.

3 Probability distribution

Suppose that Asja starts in (1,0) and secretly applies the probabilistic operator ((0.3,0.6),(0.7,0.4)) .

Because she applies her operator secretly, our information about her state is probabilistic, which is calculated as

(0.3,0.7)=((0.3,0.6),(0.7,0.4))(1,0).

Asja is either in state 0 or in state 1.

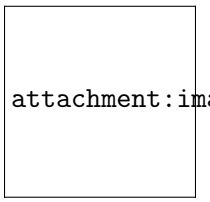
However, from our point of view, Asja is in state 0 with probability 0.3 and in state 1 with probability 0.7 .

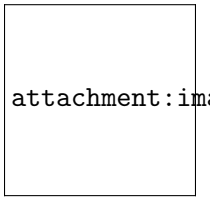
We can say that Asja is in a probability distribution of states 0 and 1, being in both states at the same time but with different weights.

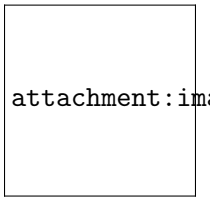
On the other hand, if we observe Asja's state, then our information about Asja becomes deterministic: either (1,0) or (0,1) .

We can say that, after observing Asja's state, the probabilistic state (0.3,0.7) collapses to either (1,0) or (0,1) .

4 We trace it step by step by matrix-vector multiplication.







The photon was in both states at the same time with certain amplitudes.

After the second Hadamard, the "outcomes" are interfered with each other.

The interference can be constructive or destructive.

In our examples, the outcome $|0\rangle$ s are interfered constructively, but the outcome $|1\rangle$ s are interfered destructively.

5 Observations

Observations

Probabilistic systems: If there is a nonzero transition to a state, then it contributes to the probability of this state positively.

Quantum systems: If there is a nonzero transition to a state, then we cannot know its contribution without knowing the other transitions to this state.

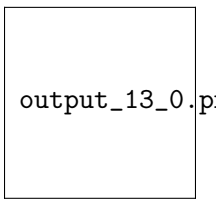
If it is the only transition, then it contributes to the amplitude (and probability) of the state, and it does not matter whether the sign of the transition is positive or negative.

If there is more than one transition, then depending on the summation of all transitions, we can determine whether a specific transition contributes or not.

As a simple rule, if the final amplitude of the state and nonzero transition have the same sign, then it is a positive contribution; and, if they have the opposite signs, then it is a negative contribution.

[8]:
```
# import all necessary objects and methods for quantum circuits
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit, execute,␣
 ↪Aer

# define a quantum register with a single qubit
q = QuantumRegister(1,"q")
# define a classical register with a single bit
c = ClassicalRegister(1,"c")
# define a quantum circuit
qc = QuantumCircuit(q,c)

# apply the first Hadamard
qc.h(q[0])

# the first measurement
qc.measure(q,c)

# apply the second Hadamard if the measurement outcome is 0
qc.h(q[0]).c_if(c,0)

# the second measurement
qc.measure(q[0],c)

# draw the circuit
qc.draw(output="mpl")
```

[8]:



[ ]:

[ ]:

6 Observe the constructive and destructive interferences, when calculating

Being in a superposition A quantum system can be in more than one state with nonzero amplitudes.

Then, we say that our system is in a superposition of these states.

When evolving from a superposition, the resulting transitions may affect each other constructively and destructively.

This happens because of having opposite sign transition amplitudes.

Otherwise, all nonzero transitions are added up to each other as in probabilistic systems.

Interference is the situation where intervention from noise in the environment damages the quantum object, and also the possibility that the wave functions of particles can either reinforce or diminish each other.

7 Taking Quantum Interference in major account

to generate the results for the publication 'Implementing a distance-based classifier with a quantum interference circuit'

[7]:
```
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute, BasicAer

class DistanceBasedClassifier:

    def initialize_registers(self, num_registers):
        """
        Creates quantum and classical registers
        with `num_registers` qubits each.
        """
        self.q = QuantumRegister(4)
        self.c = ClassicalRegister(4)

        # name the individual qubits for more clarity
        self.ancilla_qubit = self.q[0]
        self.index_qubit = self.q[1]
        self.data_qubit = self.q[2]
        self.class_qubit = self.q[3]
```

```
    def create_circuit(self, angles):
        """
        Creating the quantum circuit
        by filling in the gaps with the
        defined `angles` that are required
        to load the test and training vectors.
        """

        # create empty quantum circuit
        qc = QuantumCircuit(self.q, self.c)

        ######################################
        #START of the state preparation routine

        # put the ancilla and the index qubits into uniform superposition
        qc.h(self.ancilla_qubit)
        qc.h(self.index_qubit)

        # loading the test vector (which we wish to classify)
        qc.cx(self.ancilla_qubit, self.data_qubit)
        qc.u3(-angles[0], 0, 0, self.data_qubit)
        qc.cx(self.ancilla_qubit, self.data_qubit)
        qc.u3(angles[0], 0, 0, self.data_qubit)

        # barriers make sure that our circuit is being executed the way we want
        # otherwise some gates might be executed before we want to
        qc.barrier()

        # flipping the ancilla qubit > this moves the input vector to the |0>␣
↪state of the ancilla
        qc.x(self.ancilla_qubit)
        qc.barrier()

        # loading the first training vector
        # [0,1] -> class 0
        # we can load this with a straightforward Toffoli

        qc.ccx(self.ancilla_qubit, self.index_qubit, self.data_qubit)
        qc.barrier()

        # flip the index qubit > moves the first training vector to the |0>␣
↪state of the index qubit
        qc.x(self.index_qubit)
        qc.barrier()

        # loading the second training vector
        # [0.78861, 0.61489] -> class 1
```

```
        qc.ccx(self.ancilla_qubit, self.index_qubit, self.data_qubit)

        qc.cx(self.index_qubit, self.data_qubit)
        qc.u3(angles[1], 0, 0, self.data_qubit)
        qc.cx(self.index_qubit, self.data_qubit)
        qc.u3(-angles[1], 0, 0, self.data_qubit)

        qc.ccx(self.ancilla_qubit, self.index_qubit, self.data_qubit)

        qc.cx(self.index_qubit, self.data_qubit)
        qc.u3(-angles[1], 0, 0, self.data_qubit)
        qc.cx(self.index_qubit, self.data_qubit)
        qc.u3(angles[1], 0, 0, self.data_qubit)

        qc.barrier()

        # END of state preparation routine
        ##################################################

        # at this point we would usually swap the data and class qubit
        # however, we can be lazy and let the Qiskit compiler take care of it

        # flip the class label for training vector #2
        qc.cx(self.index_qubit, self.class_qubit)

        qc.barrier()

        ##############################################
        # START of the mini distance-based classifier

        # interfere the input vector with the training vectors
        qc.h(self.ancilla_qubit)

        qc.barrier()

        # Measure all qubits and record the results in the classical registers
        qc.measure(self.q, self.c)

        # END of the mini distance-based classifier
        ##############################################

        return qc

    def simulate(self, quantum_circuit):
        """
        Compile and run the quantum circuit
```

```
        on a simulator backend.
        """

        # noisy simulation
        backend_sim = BasicAer.get_backend('qasm_simulator')

        job_sim = execute(quantum_circuit, backend_sim)

        # retrieve the results from the simulation
        return job_sim.result()

    def get_angles(self, test_vector, training_vectors):
        """
        Return the angles associated with
        the `test_vector` and the `training_vectors`.
        Note: if you want to extend this classifier
        for other test and training vectors you need to
        specify the angles here!
        """
        angles = []

        if test_vector == [-0.549, 0.836]:
            angles.append(4.30417579487669/2)
        elif test_vector == [0.053 , 0.999]:
            angles.append(3.0357101997648965/2)
        else:
            print('No angle defined for this test vector.')

        if training_vectors[0] == [0, 1] and training_vectors[1] == [0.78861006,␣
↪0.61489363]:
            angles.append(1.3245021469658966/4)
        else:
            print('No angles defined for these training vectors.')

        return angles

    def interpret_results(self, result_counts):
        """
        Post-selecting only the results where
        the ancilla was measured in the |0> state.
        Then computing the statistics of the class
        qubit.
        """

        total_samples = sum(result_counts.values())
```

```
        # define lambda function that retrieves only results where the ancilla␣
↪is in the |0> state
        post_select = lambda counts: [(state, occurences) for state, occurences␣
↪in counts.items() if state[-1] == '0']

        # perform the postselection
        postselection = dict(post_select(result_counts))
        postselected_samples = sum(postselection.values())

        print(f'Ancilla post-selection probability was found to be␣
↪{postselected_samples/total_samples}')

        retrieve_class = lambda binary_class: [occurences for state, occurences␣
↪in postselection.items() if state[0] == str(binary_class)]

        prob_class0 = sum(retrieve_class(0))/postselected_samples
        prob_class1 = sum(retrieve_class(1))/postselected_samples

        print(f'Probability for class 0 is {prob_class0}')
        print(f'Probability for class 1 is {prob_class1}')

        return prob_class0, prob_class1

    def classify(self, test_vector, training_set):
        """
        Classifies the `test_vector` with the
        distance-based classifier using the `training_vectors`
        as the training set.
        This functions combines all other functions of this class
        in order to execute the quantum classification.
        """

        # extract training vectors
        training_vectors = [tuple_[0] for tuple_ in training_set]

        # initialize the Q and C registers
        self.initialize_registers(num_registers=4)

        # get the angles needed to load the data into the quantum state
        angles = self.get_angles(
                test_vector=test_vector,
                training_vectors=training_vectors
        )

        # create the quantum circuit
        qc = self.create_circuit(angles=angles)
```

```
        # simulate and get the results
        result = self.simulate(qc)

        prob_class0, prob_class1 = self.interpret_results(result.get_counts(qc))

        if prob_class0 > prob_class1:
            return 0
        elif prob_class0 < prob_class1:
            return 1
        else:
            return 'inconclusive. 50/50 results'


if __name__ == "__main__":

    # initiate an instance of the distance-based classifier
    classifier = DistanceBasedClassifier()

    x_prime = [-0.549, 0.836] # x' in publication
    x_double_prime = [0.053 , 0.999] # x'' in publication

    # training set must contain tuples: (vector, class)
    training_set = [
        ([0, 1], 0), # class 0 training vector
        ([0.78861006, 0.61489363], 1) # class 1 training vector
    ]

    print(f"Classifying x' = {x_prime} with noisy simulator backend")
    class_result = classifier.classify(test_vector=x_prime,␣
 ↪training_set=training_set)
    print(f"Test vector x'' was classified as class {class_result}\n")

    print('==============================================\n')

    print(f"Classifying x'' = {x_double_prime} with noisy simulator backend")
    class_result = classifier.classify(test_vector=x_double_prime,␣
 ↪training_set=training_set)
    print(f"Test vector x' was classified as class {class_result}")
```

```
Classifying x' = [-0.549, 0.836] with noisy simulator backend
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:40:
DeprecationWarning: The QuantumCircuit.u3 method is deprecated as of 0.16.0. It
will be removed no earlier than 3 months after the release date. You should use
QuantumCircuit.u instead, which acts identically. Alternatively, you can
decompose u3 in terms of QuantumCircuit.p and QuantumCircuit.sx: u3(,,) =
p(+) sx p(+) sx p() (2 pulses on hardware).
```

```
Ancilla post-selection probability was found to be 0.732421875
Probability for class 0 is 0.6533333333333333
Probability for class 1 is 0.3466666666666667
Test vector x'' was classified as class 0

================================================

Classifying x'' = [0.053, 0.999] with noisy simulator backend
Ancilla post-selection probability was found to be 0.912109375
Probability for class 0 is 0.5535331905781584
Probability for class 1 is 0.4464668094218415
Test vector x' was classified as class 0
```

We start with the most simple quantum circuit and show that it can be used as a - likewise simple - model of a classifier.

[ ]: