Generative Learning Using a Quantum Computer

In this tutorial, we implement Data-driven quantum circuit learning (DDQCL), based on the 2019 paper Training of quantum circuits on a hybrid quantum computer by Zhu et. al, using the Qiskit SDK.

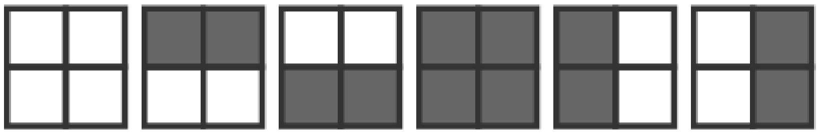
DQCCL

DDQCL is a hybrid quantum-classical technique for generative modeling of classical data, using a parametrized quantum circuit as the model being trained. Training occurs by sampling the output of a parametrized circuit run on a quantum computer and updating the circuit parameters using classical optimization techniques. Convergence produces a circuit that captures the correlations in the training data, thus serving as a generative model for the data itself.

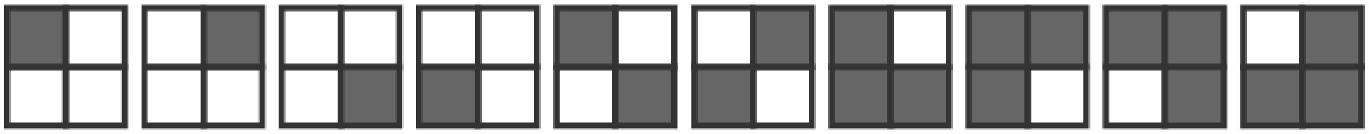
The Bars-And-Stripes (BAS) Dataset

The data we will be training against is a canonical ensemble that works like so: for an $n \times m$ matrix, each cell can either be on (filled-in) or off (blank); it is a valid BAS example if it contains only completely filled rows (bars) or columns (stripes).

For a 2x2 matrix, like we will be using in this example, this allows for six valid patterns of 16 total permutations:



Valid BAS patterns



Not BAS patterns

We encode this as a binary string by reading left-to-right, top-to-bottom, with a filled-in cell representing a 1 and a blank cell a 0; the above valid examples translate (in order) to 0000, 1100, 0011, 0000, 1010, and 0101.

Implementation

We will be implementing this technique using noisyopt, a python library for optimizing noisy functions. To do that, we'll need to define

1. A generator for our BAS training data
2. A generator for our parametrized circuit
3. A cost function for our optimizer
4. The hybrid learning algorithm itself, using all of the above

Imports and Setup

First, we'll do some setup that will allow us to build, optimize, visualize, store, and retrieve the circuits we'll be running.

First, get an API key from IonQ. This will be used by the IonQ provider inside Qiskit to submit circuits to the IonQ platform.

After securing an API key, install the qiskit-ionq provider at https://github.com/Qiskit-Partners/qiskit-ionq/

(Optional) Extra Dependencies

Some examples use additional Python dependencies; please make sure to `pip install` them as needed.

Dependencies:

- `matplotlib` : To run `qiskit.visualization.plot_histogram` .

NOTE: The provider expects an API key to be supplied via the `token` keyword argument to its constructor. If no token is directly provided, the provider will check for one in the `QISKIT_IONQ_API_TOKEN` environment variable.

Now that the Python package has been installed, you can import and instantiate the provider:

In [1]:

```
# general imports
import math
import time
import pickle
import random
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from datetime import datetime

# Noisy optimization package — you could also use scipy's optimization functions,
# but this is a little better suited to the noisy output of NISQ devices.
%pip install noisyopt
import noisyopt

# magic invocation for producing visualizations in notebook
%matplotlib inline
```

```
Defaulting to user installation because normal site-packages is not writeable
Looking in indexes: https://pypi.org/simple, https://pkgs.dev.azure.com/ms-quantum-public/9af4e09e
-a436-4aca-9559-2094cfe8d80c/_packaging/alpha/pypi/simple/
Requirement already satisfied: noisyopt in /home/sjohri/.local/lib/python3.6/site-packages (0.2.2)
Requirement already satisfied: scipy in /home/sjohri/.local/lib/python3.6/site-packages (from nois
yopt) (1.5.2)
Requirement already satisfied: numpy in /home/sjohri/.local/lib/python3.6/site-packages (from nois
yopt) (1.19.1)
WARNING: You are using pip version 21.0; however, version 21.0.1 is available.
You should consider upgrading via the '/usr/bin/python3 -m pip install --upgrade pip' command.
Note: you may need to restart the kernel to use updated packages.
```

In [2]:

```
# Qiskit imports
from qiskit import Aer

from qiskit_ionq import IonQProvider

#Call provider and set token value
provider = IonQProvider(token='My token')
```

```
/home/sjohri/.local/lib/python3.6/site-packages/qiskit/__init__.py:67: DeprecationWarning: Using Q
iskit with Python 3.6 is deprecated as of the 0.17.0 release. Support for running Qiskit with Pyth
```

```
on 3.6 will be removed in a future release.
  "future release.", DeprecationWarning)
```

The `provider` instance can now be used to create and submit circuits to IonQ.

Backend Types

The IonQ provider supports two backend types:

- `ionq_simulator` : IonQ's simulator backend.
- `ionq_qpu` : IonQ's QPU backend.

To view all current backend types, use the `.backends` property on the provider instance:

In [3]:
```
provider.backends()
```

Out[3]:
```
[<IonQSimulatorBackend('ionq_simulator')>, <IonQQPUBackend('ionq_qpu')>]
```

In [4]:
```
# fix random seed for reproducibility – this allows us to re-run the process and get the same resu
seed = 42
np.random.seed(seed)
random.seed(a=seed)
```

BAS Generator

Here we generate our bars and stripes — as this is just a proof-of-principle, we are limiting the possible outputs to the four single-bar or single-stripe outputs to keep convergence time (and therefore training cost) low.

In [5]:
```
#Generate BAS, only one bar or stripe allowed
def generate_target_distribution(rows, columns):
    #Stripes
    states=[]
    for i in range(rows):
        s=['0']*rows*columns
        for j in range(columns):
            s[j+columns*i]='1'
        states.append(''.join(s))


    #Bars
    for j in range(columns):
        s=['0']*rows*columns
        for i in range(rows):
            s[j+columns*i]='1'
        states.append(''.join(s))

    return states
```

Circuit Generator

Here we define several driver functions which consist of parametrized one qubit gates that help to explore the Hilbert space, and several entangler functions that generate entanglement in the system.

The complete circuit ansatz consists of repeating layers of the drivers and entanglers. The parameters (γ and β) of the ansatz are iteratively optimized (learned) by our hybrid quantum-clasical routine.

In [6]:

```python
#Parameters for driver
def generate_beta(ansatz_type,random_init):
    if ansatz_type[0]==0: #No driver
        beta=[]
    elif ansatz_type[0]==1: #Rz(t1)Rx(t2)Rz(t3), angles different for each qubit
        beta=(
            [random.uniform(0.0,2.*math.pi) for i in range(3*n*(layers-1))]
            if random_init
            else [0]*3*n*(layers-1)
        )
    elif ansatz_type[0]==2: #Rz(t1)Rx(t2)Rz(t3), angles same for all qubits
        beta=(
            [random.uniform(0.0,2.*math.pi) for i in range(3*(layers-1))]
            if random_init
            else [0]*3*(layers-1)
        )
    elif ansatz_type[0]==3: #Rz(t1), angles different for each qubit
        beta=(
            [random.uniform(0.0,2.*math.pi) for i in range(n*(layers-1))]
            if random_init
            else [0]*n*(layers-1)
        )
    else:
        raise Exception("Undefined driver type")

    return beta

#Parameters for entangler
def generate_gamma(ansatz_type,random_init, n, conn):
    length_gamma=int(n*conn-conn*(conn+1)/2.)
    if ansatz_type[1]==0: #No entangler
        gamma=[]
    elif ansatz_type[1]==1: #XX(t1), angles different for each qubit
        gamma=(
            [random.uniform(0.0,2.*math.pi) for i in range(length_gamma*layers)]
            if random_init
            else [0]*length_gamma*layers
        )
    elif ansatz_type[1]==2: #XX(t1), angles same for all qubits
        gamma=(
            [random.uniform(0.0,2.*math.pi) for i in range(layers)]
            if random_init
            else gamma[0]*layers
        )
    else:
        raise Exception("Undefined entangler type")

    return gamma
```

In [7]:

```python
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.circuit import Gate

def driver(circ,qr,beta,n,ansatz_type):
    #qr=QuantumRegister(n)
    #circ = QuantumCircuit(qr)

    if ansatz_type==0:
        pass
    elif ansatz_type==1:
        for i_q in range(n):
            circ.rz(beta[3*i_q], qr[i_q])
            circ.rx(beta[3*i_q+1], qr[i_q])
            circ.rz(beta[3*i_q+2], qr[i_q])
    elif ansatz_type==2:
        for i_q in range(n):
            circ.rz(beta[0], qr[i_q])
            circ.rx(beta[1], qr[i_q])
            circ.rz(beta[2], qr[i_q])
    elif ansatz_type==3:
        for i_q in range(n):
            circ.rz(beta[i_q], qr[i_q])

    return

def entangler(circ,qr,gamma,n,conn,ansatz_type):
    #qr=QuantumRegister(n)
    #cr=ClassicalRegister(n)
    #circ = QuantumCircuit(qr)

    if ansatz_type==0:
        pass
    elif ansatz_type==1:
        i_gamma=0
        for i_conn in range(1,conn+1):
            for i_q in range(0,n-i_conn):
                circ.cx(qr[i_q],qr[i_q+i_conn])
                circ.rx(gamma[i_gamma],qr[i_q])
                circ.cx(qr[i_q],qr[i_q+i_conn])
                #circ.rxx(gamma[i_gamma], qr[i_q], qr[i_q+i_conn])
                i_gamma+=1

    elif ansatz_type==2:
        for i_conn in range(1,conn+1):
            for i_q in range(0,n-i_conn):
                circ.cx(qr[i_q],qr[i_q+i_conn])
                circ.rx(gamma[0],qr[i_q])
                circ.cx(qr[i_q],qr[i_q+i_conn])

    #circ.measure(qr,cr)
    return

#Define circuit ansatz
def circuit_ansatz(n,params,conn=1, layers=1, ansatz_type=[1,1]):
    qr=QuantumRegister(n)
    cr=ClassicalRegister(n)
    circ = QuantumCircuit(qr,cr)
    if ansatz_type[0]==0:
        length_beta=0
    elif ansatz_type[0]==1:
        length_beta=3*n
    elif ansatz_type[0]==2:
        length_beta=3
    elif ansatz_type[0]==3:
        length_beta=n

    if ansatz_type[1]==0:
        length_gamma=0
    elif ansatz_type[1]==1:
        length_gamma=int(n*conn-conn*(conn+1)/2.)
    elif ansatz_type[1]==2:
        length_gamma=1


    for i_layer in range(layers-1):

        beta=params[(length_beta+length_gamma)*i_layer:(length_beta+length_gamma)*i_layer+length_b
        gamma=params[(length_beta+length_gamma)*i_layer+length_beta:(length_beta+length_gamma)*(i_

        entangler(circ,qr,gamma,n,conn,ansatz_type[1])
        driver(circ,qr,beta,n,ansatz_type[0])

    gamma=params[(length_beta+length_gamma)*(layers-1):]
    entangler(circ,qr,gamma,n,conn,ansatz_type[1])
    circ.measure(qr, cr)

    return circ
```

Cost function

Here, we define our cost function to feed to the optimization routine, which takes the form of a regularized Kullback-Leibler Divergence (Quick Overview, Wikipedia).

In [8]:

```
def cost(counts, shots, target_states,tol=0.0001):
    cost=0
    for state in target_states:
        if state in counts:
            cost-=1./len(target_states)*np.log2(max(tol,counts[state]/shots))
        else:
            cost-=1./len(target_states)*np.log2(tol)
    return cost
```

Helper Methods

Finally, we define two helper methods: `run_iteration` , to run and return each iteration of our circuit (with some helpful tools for managing timeouts and waiting for results) and `run_circuit_and_calc_cost` , which uses `run_iteration` to both run an interation of the circuit and calculate a new cost function, effectively taking a training step.

In [9]:

```
from qiskit.providers.jobstatus import JobStatus

def run_iteration(circ, num_qubits, shots=100):

    # submit task: define task (asynchronous)
    task_status=''
    while task_status != JobStatus.DONE:

            task = backend.run(circ, shots=shots)
            #print("Job submitted")

            # Get ID of submitted task
            task_id = task.job_id()
            #print('Task ID :', task_id)
            while (task_status == JobStatus.INITIALIZING) or (task_status == JobStatus.QUEUED)   c
                time.sleep(1)
                try:
                    task_status=task.status()
                except:
                    print("Error querying status. Trying again.")
                    pass


        #print('Task status is', task_status)

    # get result
    counts = task.result().get_counts()
    return counts


#Run an iteration of the circuit and calculate its cost
def run_circuit_and_calc_cost(params, *args):
    n, conn, layers, ansatz_type, target_states, shots = args
    circ=circuit_ansatz(n, params, conn, layers,ansatz_type)
    counts=run_iteration(circ, n, shots=shots)
    iter_cost=cost(counts, shots, target_states)
    cost_history.append(iter_cost)
    print("Current cost:", iter_cost)
    return iter_cost
```

Running The Learning Algorithm

Now we're ready to run the learning algorithm and train our circuit.

Choose Target Device

We need to do this first because we check against its capabilities when building the training data and circuit. We do not recommend running this this algorithm against a real hardware device until you have run it on a simulator to understand estimated price and if it will converge with the parameters you've selected.

So first, let's test it on the simulator

In [10]:
```
backend = provider.get_backend("ionq_simulator")
```

Set Up Training Data

First, we'll set up our training data for the problem using the method we defined earlier.

In [11]:
```
#Set problem parameters
r=2 #Number of rows — make sure r*c is less than qubit count
c=2 #Number of columns — make sure r*c is less than qubit count
n=r*c #Qubits needed
conn=2 #Connectivity of the entangler.

#Check on qubit count
if n>backend.configuration().n_qubits:
    raise Exception("Too many qubits")

#Check on conn
if conn>(n-1):
    raise Exception("Connectivity is too large")

target_states=generate_target_distribution(r,c)

# Check expected output
print('Bitstrings that should be generated by trained circuit are', target_states, 'corresponding
for state in target_states:
    for i in range(r):
        print(state[c*i:][:c].replace('0','□ ').replace('1','■ '))
    print('')
```

```
Bitstrings that should be generated by trained circuit are ['1100', '0011', '1010', '0101'] corres
ponding to the following BAS patterns:

■ ■
□ □
```

□ □
■ ■

■ □
■ □

□ ■
□ ■

Set Up Training Parameters

Now, we'll set up our parameters for the training routine itself — these are all tunable; feel free to play with them to see how they impact the routine (especially when running on a local simulator!). Again, because this is just a proof-of-principle demo, we've selected some defaults that keep convergence time and training cost low.

In [12]:

```python
# Choose entangler and driver type
layers=1 # Number of layers in each circuit
shots=100 # Number of shots per iteration
ansatz_type=[1,1]
random_init=False # Set to true for random initialization; will be slower to converge

# Set up args
beta=generate_beta(ansatz_type, random_init)
gamma=generate_gamma(ansatz_type, random_init, n, conn)
params=beta+gamma

base_bounds=(0,2.*math.pi)
bnds=((base_bounds , ) * len(params))

#Set up list to track cost history
cost_history=[]

max_iter=100 # max number of optimizer iterations

args=(n, conn, layers, ansatz_type, target_states, shots)
opts = {'disp': True, 'maxiter': max_iter, 'maxfev': max_iter, 'return_all': True}

#Visualize the circuit
circ=circuit_ansatz(n, params, conn,layers,ansatz_type)
circ.draw()
```

Out[12]:

```
                ┌───────┐                                                                  ┌───────┐ »
q0_0: ────■─────┤ RX(0) ├───■──────────────────────────────────────────────────────■───────┤ RX(0) ├─»
        ┌─┴─┐   └───────┘ ┌─┴─┐            ┌───────┐                                │       └───────┘ »
q0_1: ──┤ X ├─────────────┤ X ├────■───────┤ RX(0) ├───■────────────────────────────┼──────────■──────»
        └───┘             └───┘  ┌─┴─┐     └───────┘ ┌─┴─┐          ┌───────┐     ┌─┴─┐        │      »
q0_2: ───────────────────────────┤ X ├───────────────┤ X ├───■──────┤ RX(0) ├──■──┤ X ├────────┼──────»
                                 └───┘               └───┘ ┌─┴─┐    └───────┘┌─┴─┐└───┘      ┌─┴─┐    »
q0_3: ─────────────────────────────────────────────────────┤ X ├─────────────┤ X ├───────────┤ X ├────»
                                                           └───┘             └───┘           └───┘    »
c0: 4/════════════════════════════════════════════════════════════════════════════════════════════════»
                                                                                                      »
«                                   ┌─┐
«q0_0: ──────────────■──────────────┤M├─────────
«      ┌───────┐     │              └╥┘┌─┐
«q0_1: ┤ RX(0) ├─────┼───────■───────╫─┤M├──────
«      └───────┘   ┌─┴─┐     │       ║ └╥┘┌─┐
«q0_2: ────────────┤ X ├─────┼───────╫──╫─┤M├───
«                  └───┘   ┌─┴─┐     ║  ║ └╥┘┌─┐
«q0_3: ────────────────────┤ X ├─────╫──╫──╫─┤M├
«                          └───┘     ║  ║  ║ └╥┘
```

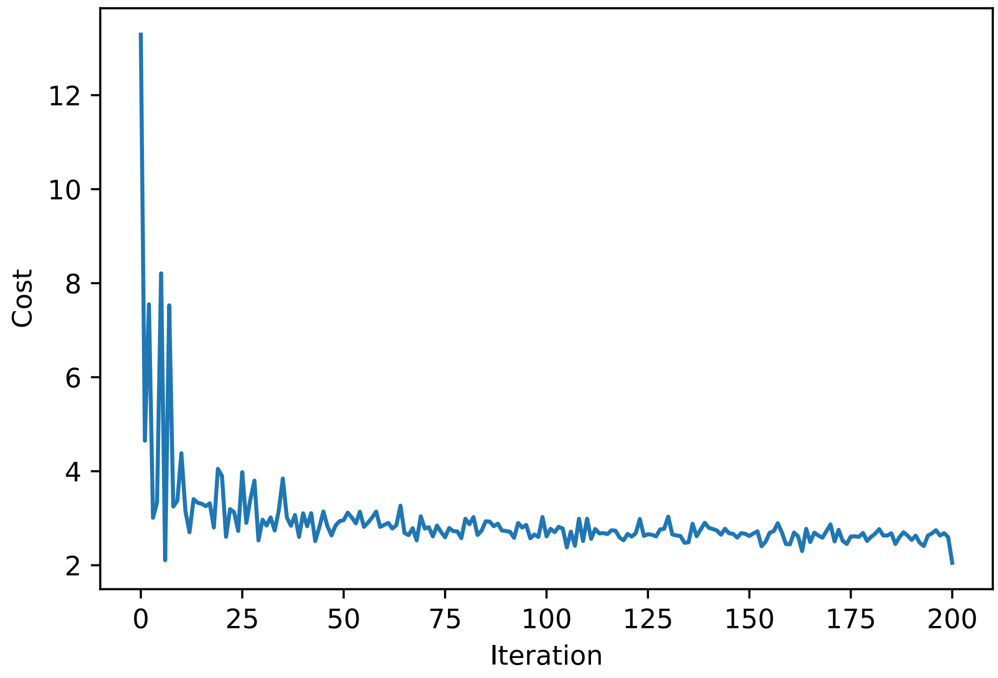
```
«c0: 4/══════════════════════════╩══╩══╩══╩═
«                                0  1  2  3
```

In [13]:

```python
from qiskit.visualization import plot_histogram

#Train Circuit
result=noisyopt.minimizeSPSA(run_circuit_and_calc_cost, params, args=args, bounds=bnds, niter=max_

print("Success: ", result.success)
print(result.message)
print("Final cost function is ", result.fun)
print("Min possible cost function is ", np.log2(len(target_states)))
print("Number of iterations is ", result.nit)
print("Number of function evaluations is ", result.nfev)
print("Number of parameters was ", len(params))
print("Approximate cost on hardware: $", 0.01*len(cost_history)*shots)

#Plot the evolution of the cost function
plt.figure()
plt.plot(cost_history)
plt.ylabel('Cost')
plt.xlabel('Iteration')
```

```
Current cost: 13.287712379549449
Current cost: 4.649536208899746
Current cost: 7.552125277530095
Current cost: 3.0091877015040573
Current cost: 3.336651517150754
Current cost: 8.210192331404974
Current cost: 2.107681672441811
Current cost: 7.5295758252872
Current cost: 3.249027205379721
Current cost: 3.3706669045447613
Current cost: 4.382757659935112
Current cost: 3.14704018675659
Current cost: 2.7011321562532893
Current cost: 3.4060350110329813
Current cost: 3.3300990473200422
Current cost: 3.307782212795703
Current cost: 3.2561694934289145
Current cost: 3.3176257723793063
Current cost: 2.803301735796455
Current cost: 4.050639924746577
Current cost: 3.899536208899745
Current cost: 2.6051516625116986
Current cost: 3.193429100045609
Current cost: 3.1271657066779976
Current cost: 2.730197182642733
Current cost: 3.981093266979992
Current cost: 2.901554534033666
Current cost: 3.399536208899746
Current cost: 3.801456853205056
Current cost: 2.5290863509833077
Current cost: 2.968800452074911
Current cost: 2.8448593652675975
Current cost: 3.0164124386929902
Current cost: 2.738572161456064
Current cost: 3.1636284232938614
Current cost: 3.844859365267597
Current cost: 3.0153009624351923
Current cost: 2.839669725225898
Current cost: 3.068410699261948
Current cost: 2.6038883555801275
Current cost: 3.1049144551633114
Current cost: 2.8309208698868473
Current cost: 3.107054958539168
Current cost: 2.5153009624351923
Current cost: 2.811407105219549
Current cost: 3.143944255642632
Current cost: 2.8295092291015616
Current cost: 2.6348986715659706
Current cost: 2.8461685273591346
Current cost: 2.93851234168159
Current cost: 2.95938592056028
Current cost: 3.1190603372549033
Current cost: 3.0153009624351923
Current cost: 2.8913115887695895
Current cost: 3.1380475505632277
Current cost: 2.816061024761386
Current cost: 2.906035011032982
Current cost: 3.014500354425606
Current cost: 3.1413115887695895
Current cost: 2.8160610247613858
Current cost: 2.8608919496225265
Current cost: 2.9009478496850307
Current cost: 2.780409925900686
Current cost: 2.855151662511699
Current cost: 3.2669075297250507
Current cost: 2.686023192787796
Current cost: 2.6396758773028965
Current cost: 2.7870279787409835
Current cost: 2.5295911428924707
Current cost: 3.0411771063736577
Current cost: 2.7798157218288813
Current cost: 2.8088936890535683
Current cost: 2.6129470763237688
Current cost: 2.8423315362757755
Current cost: 2.7089110373314096
Current cost: 2.5948593652675975
Current cost: 2.7922717165013013
Current cost: 2.7244262793644722
Current cost: 2.7209748294832146
Current cost: 2.5762823581567145
Current cost: 2.9885721614560645
Current cost: 2.871659130095498
Current cost: 3.02391112376745
Current cost: 2.643944255642632
Current cost: 2.7355790010395453
Current cost: 2.9359304149711503
Current cost: 2.927676650664159
Current cost: 2.8295092291015616
Current cost: 2.884169300720397
Current cost: 2.738658046385681
Current cost: 2.7295265773794033
Current cost: 2.711251923592057
Current cost: 2.583146399572021
Current cost: 2.899536208899746
Current cost: 2.801456853205056
Current cost: 2.857054958539168
Current cost: 2.5733057915043274
Current cost: 2.653041850616463
Current cost: 2.602786580199432
Current cost: 3.0265729348173274
Current cost: 2.6135537606724037
Current cost: 2.7747962131937616
Current cost: 2.704196280518581
Current cost: 2.817447617888047
Current cost: 2.7800243846985846
Current cost: 2.3794428547837407
Current cost: 2.715241142660718
Current cost: 2.4148863450657814
Current cost: 2.988572161456065
Current cost: 2.5162037758232874
Current cost: 2.990293757852374
Current cost: 2.5609738634134356
Current cost: 2.770892915692306
Current cost: 2.6781968503850266
Current cost: 2.6832686039212725
Current cost: 2.6637679759209547
Current cost: 2.7437618014977634
Current cost: 2.736005077592608
Current cost: 2.5902764095745336
Current cost: 2.5306510986012283
Current cost: 2.6670983205454126
Current cost: 2.609568374705148
Current cost: 2.683059941051569
Current cost: 2.9852668644902307
Current cost: 2.624652290512017
Current cost: 2.659938308534438
Current cost: 2.647985305435945
Current cost: 2.6135537606724037
Current cost: 2.761315576467937
Current cost: 2.7765729348173265
Current cost: 3.0338011077961364
Current cost: 2.654399299566289
Current cost: 2.634169300720397
Current cost: 2.6229136487871543
Current cost: 2.4773001890739295
Current cost: 2.488572161456064
Current cost: 2.882672283260497
Current cost: 2.618136716365468
Current cost: 2.7597943858526923
Current cost: 2.9030757246489105
Current cost: 2.7979469898877283
Current cost: 2.7701891617905425
Current cost: 2.740293757852374
Current cost: 2.649536208899746
Current cost: 2.7761914712595868
Current cost: 2.679312362130852
Current cost: 2.6685451816578807
Current cost: 2.587554330538849
Current cost: 2.684695652518262
Current cost: 2.6671335408725954
Current cost: 2.621659130095498
Current cost: 2.6770244375384697
Current cost: 2.722696425879525
Current cost: 2.403957568022742
Current cost: 2.499027205379721
Current cost: 2.6904108919704655
Current cost: 2.7254185049152087
Current cost: 2.893944255642632
Current cost: 2.699927902178845
Current cost: 2.4541962805185804
Current cost: 2.444314189471921
Current cost: 2.6960909110954865
Current cost: 2.6165719687475475
Current cost: 2.3009651979628725
Current cost: 2.7746696387898577
Current cost: 2.492023611337322
Current cost: 2.694035784394244
Current cost: 2.6337837595182956
Current cost: 2.5846844563174196
Current cost: 2.7290801989063094
Current cost: 2.868034237671719
Current cost: 2.506530347646331
Current cost: 2.753176333012395
Current cost: 2.519954881977029
Current cost: 2.4541962805185804
Current cost: 2.6111331223230687
Current cost: 2.6153392779834084
Current cost: 2.6038883555801275
Current cost: 2.687045327018825
Current cost: 2.5175273507406652
Current cost: 2.6027865801994325
Current cost: 2.6715633897330244
Current cost: 2.7683191705103365
Current cost: 2.633575096648592
Current cost: 2.632053906033348
Current cost: 2.679312362130852
Current cost: 2.4523954631848603
Current cost: 2.598801194232177
Current cost: 2.7009838223995755
Current cost: 2.6301839147531414
Current cost: 2.5384550273379736
Current cost: 2.6300355808994276
Current cost: 2.4751150351789866
Current cost: 2.4090532191081255
Current cost: 2.630332309637038
Current cost: 2.678584463275876
Current cost: 2.745858747870943
Current cost: 2.6318196262198343
Current cost: 2.683059941051569
Current cost: 2.597477619314798
Current cost: 2.0529593781435804
Success:  True
terminated after reaching max number of iterations
Final cost function is  2.0529593781435804
Min possible cost function is  2.0
Number of iterations is  100
Number of function evaluations is  200
Number of parameters was  5
Approximate cost on hardware: $ 201.00000000000003
```

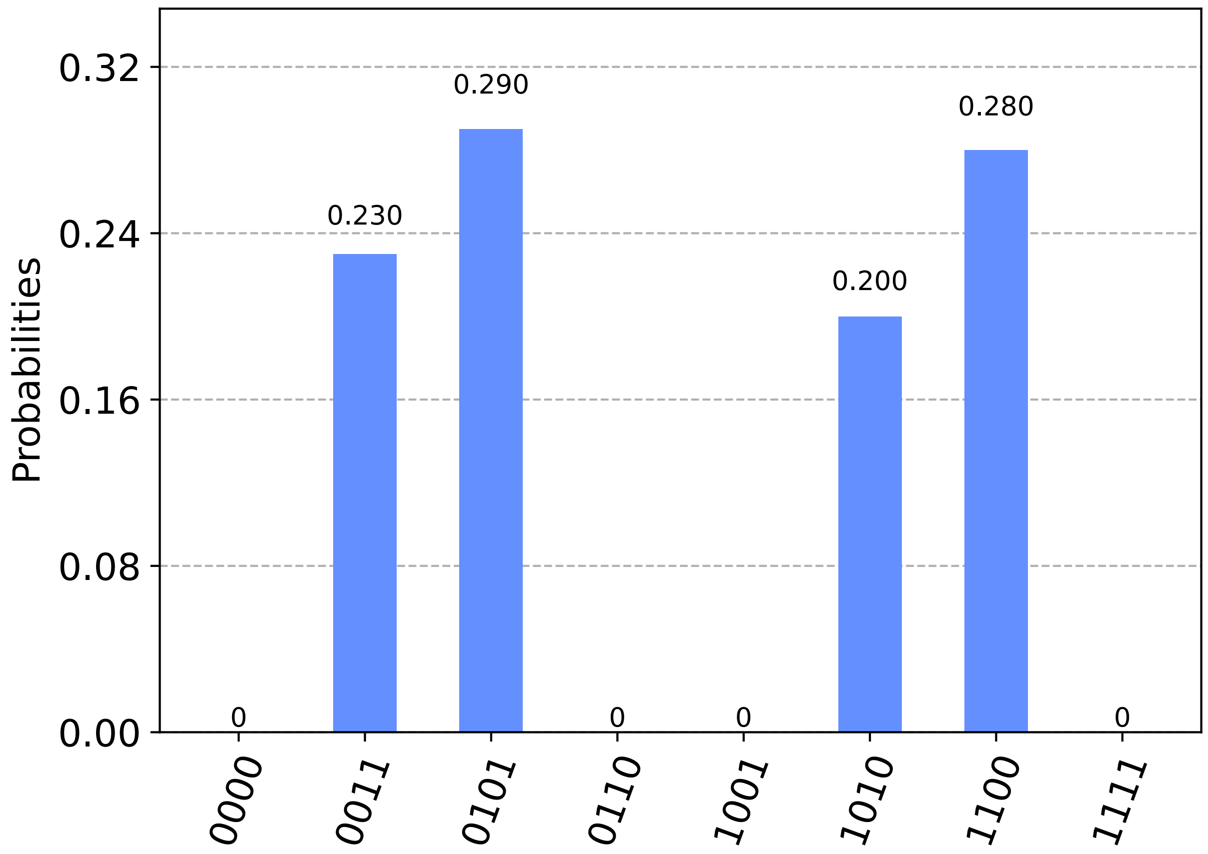
Out[13]:
```
Text(0.5, 0, 'Iteration')
```



In [14]:

```
#Visualize the result
result_final=run_iteration(circuit_ansatz(n, result.x, conn,layers,ansatz_type), shots)
plot_histogram(result_final)
```

Out[14]:



Next let's test it on the qpu backend.

In [15]:

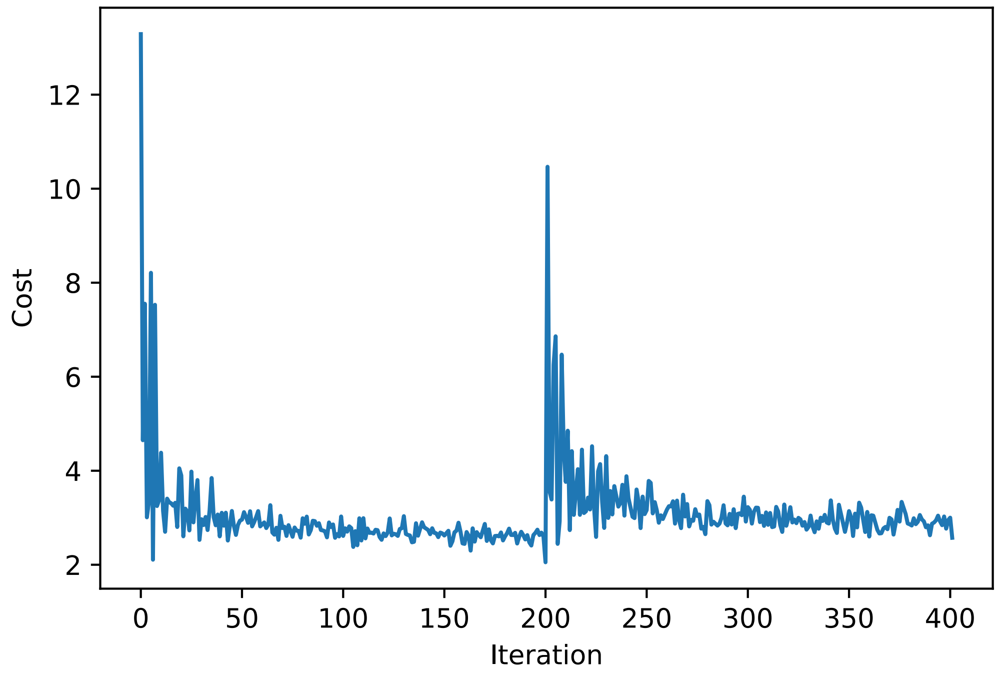
```
#Train Circuit
backend = provider.get_backend("ionq_qpu")
result=noisyopt.minimizeSPSA(run_circuit_and_calc_cost, params, args=args, bounds=bnds, niter=max_

print("Success: ", result.success)
print(result.message)
print("Final cost function is ", result.fun)
print("Min possible cost function is ", np.log2(len(target_states)))
print("Number of iterations is ", result.nit)
print("Number of function evaluations is ", result.nfev)
print("Number of parameters was ", len(params))
print("Approximate cost on hardware: $", 0.01*len(cost_history)*shots)

#Plot the evolution of the cost function
plt.figure()
plt.plot(cost_history)
plt.ylabel('Cost')
plt.xlabel('Iteration')
```

```
Current cost: 10.465784284662087
Current cost: 3.562670412151121
Current cost: 3.388264236517611
Current cost: 6.254902151009315
Current cost: 6.859480308837241
Current cost: 2.4441876150680177
Current cost: 2.9262650098788736
Current cost: 6.46985773649725
Current cost: 4.4476974783853445
Current cost: 3.7634560798437926
Current cost: 4.851374939414146
Current cost: 2.7373005974358553
Current cost: 4.417133540872595
Current cost: 3.061004479580868
Current cost: 3.554330762914513
Current cost: 4.033120315812543
Current cost: 3.061792646421651
Current cost: 4.44851626139356
Current cost: 3.1004369056988694
Current cost: 3.1384376790601323
Current cost: 3.428196850385026
Current cost: 3.1758317680727597
Current cost: 4.520892915692306
Current cost: 3.2152948103581944
Current cost: 2.5924174212053925
Current cost: 3.9901787287459225
Current cost: 4.141049375918911
Current cost: 3.334404321343941
Current cost: 2.783801107796137
Current cost: 4.31056735219707
Current cost: 2.996034044963202
Current cost: 3.569054185177905
Current cost: 3.070344611456849
Current cost: 3.672813559997616
Current cost: 3.4623016499416748
Current cost: 3.2365170347548227
Current cost: 3.313374166052884
Current cost: 3.7005713880948017
Current cost: 3.044838800364758
Current cost: 3.8827576599351117
Current cost: 3.4171335408725954
Current cost: 3.2185295742850686
Current cost: 3.0208929156923063
Current cost: 2.992197053879843
Current cost: 3.6004369056988694
Current cost: 3.343938103565634
Current cost: 2.7798157218288813
Current cost: 3.4495423609767437
Current cost: 3.077013206958406
Current cost: 3.163279777775923
Current cost: 3.7831203158125426
Current cost: 3.7420236113373218
Current cost: 3.0923315362757755
Current cost: 3.332171040151439
Current cost: 3.158548427198962
Current cost: 2.893944255642632
Current cost: 3.051456853205055
Current cost: 2.9726964258795245
Current cost: 3.0771595546009993
Current cost: 3.172813559997616
Current cost: 3.246034044963202
Current cost: 3.240178728745923
Current cost: 3.35043690569887
Current cost: 2.874652290512017
Current cost: 3.3642831315179773
Current cost: 2.9495423609767437
Current cost: 2.7790863509833077
Current cost: 3.4901787287459225
Current cost: 3.0265729348173265
Current cost: 3.2911771063736572
Current cost: 2.81506879921065
Current cost: 2.9551343142338578
Current cost: 2.93593041497115
Current cost: 3.181945029003894
Current cost: 3.0367334309416636
Current cost: 3.068410699261948
Current cost: 2.767899755275787
Current cost: 2.805535016988957
Current cost: 2.650947849685031
Current cost: 3.3538883555801275
Current cost: 3.2683968935491508
Current cost: 2.8536716696257445
Current cost: 2.9151664800392605
Current cost: 2.8771657066779976
Current cost: 2.8324494452291162
Current cost: 2.874652290512017
Current cost: 3.0032955837194564
Current cost: 3.2653009624351923
Current cost: 2.878429013609569
Current cost: 2.8386104770042806
Current cost: 3.0821710401514393
Current cost: 2.855299996365413
Current cost: 3.1809189294207103
Current cost: 2.7790863509833077
Current cost: 3.0820226452675423
Current cost: 3.093938103565634
Current cost: 3.0543307629145127
Current cost: 3.44851626139356
Current cost: 2.9185451816578802
Current cost: 3.225749854281851
Current cost: 3.1683330829983003
Current cost: 2.8755299952113047
Current cost: 3.124652290512017
Current cost: 3.2146513244422374
Current cost: 3.2117697610969804
Current cost: 2.907061110616166
Current cost: 3.03939306105325
Current cost: 2.8271595546009998
Current cost: 3.1135537606724037
Current cost: 2.849559709254585
Current cost: 3.1013749394141463
Current cost: 2.800639924746578
Current cost: 2.8134622319207914
Current cost: 3.230925081497709
Current cost: 3.133206219346495
Current cost: 2.843938103565634
Current cost: 2.69851626139356
Current cost: 3.281053411816642
Current cost: 2.829509229101562
Current cost: 2.9901787287459225
Current cost: 3.2244262793644722
Current cost: 2.899536208899746
Current cost: 2.9608143333588783
Current cost: 2.884812786636353
Current cost: 2.997967957158104
Current cost: 2.965294810358194
Current cost: 2.831059563893641
Current cost: 2.9085484271989626
Current cost: 2.746677530879159
Current cost: 2.8056211891934577
Current cost: 3.045776834080035
Current cost: 2.81392052138553
Current cost: 2.691822532755751
Current cost: 2.922813559997616
Current cost: 2.7664124386929902
Current cost: 3.000799561576301
Current cost: 2.9319450290038946
Current cost: 3.061004479580868
Current cost: 2.8985981751844685
Current cost: 2.8771657066779976
Current cost: 3.368937051059814
Current cost: 2.9586738299830224
Current cost: 2.7683191705103365
Current cost: 2.6762650098788736
Current cost: 3.2765729348173274
Current cost: 3.092331536275775
Current cost: 2.9228135599976155
Current cost: 2.6995423609767437
Current cost: 2.8922746701434914
Current cost: 3.14131158876959
Current cost: 3.0175273507406657
Current cost: 2.611918049205711
Current cost: 3.0868770087243522
Current cost: 2.7953745208646206
Current cost: 3.32034461145685
Current cost: 3.215294810358194
Current cost: 2.9364255060032103
Current cost: 2.696090911095486
Current cost: 3.123402120247425
Current cost: 2.6002905580562765
Current cost: 3.052275636213271
Current cost: 3.0441702667901764
Current cost: 2.9026356101828013
Current cost: 2.7421970538798432
Current cost: 2.6651664800392605
Current cost: 2.6706730566217596
Current cost: 2.7720217859729295
Current cost: 2.8046916171319296
Current cost: 2.756492741815747
Current cost: 2.999027205379721
Current cost: 2.9652948103581944
Current cost: 2.6420993730512325
Current cost: 2.873651807872643
Current cost: 3.1671335408725954
Current cost: 2.9182624111532185
Current cost: 3.336651517150754
Current cost: 3.2041962805185804
Current cost: 3.077159554600999
Current cost: 2.87767049858716
Current cost: 2.8599288682486246
Current cost: 2.8346844563174196
Current cost: 2.988572161456064
Current cost: 2.8632300686962484
Current cost: 2.9137679759209547
Current cost: 3.0579556553382914
Current cost: 2.9646513244422374
Current cost: 2.922813559997616
Current cost: 2.7979469898877283
Current cost: 2.83834615030852
Current cost: 2.628191806261182
Current cost: 2.8729863579417643
Current cost: 2.8979296416098874
Current cost: 2.945556975009489
Current cost: 3.044170266790176
Current cost: 2.921147627427364
Current cost: 2.8417976001897776
Current cost: 3.026572934817327
Current cost: 2.7683191705103365
Current cost: 2.950571388094802
Current cost: 3.001155080343601
Current cost: 2.5739172758444484
```

```
Success:  True
terminated after reaching max number of iterations
Final cost function is  2.5739172758444484
Min possible cost function is  2.0
Number of iterations is  100
Number of function evaluations is  200
Number of parameters was  5
Approximate cost on hardware: $ 402.00000000000006
```

Out[15]:

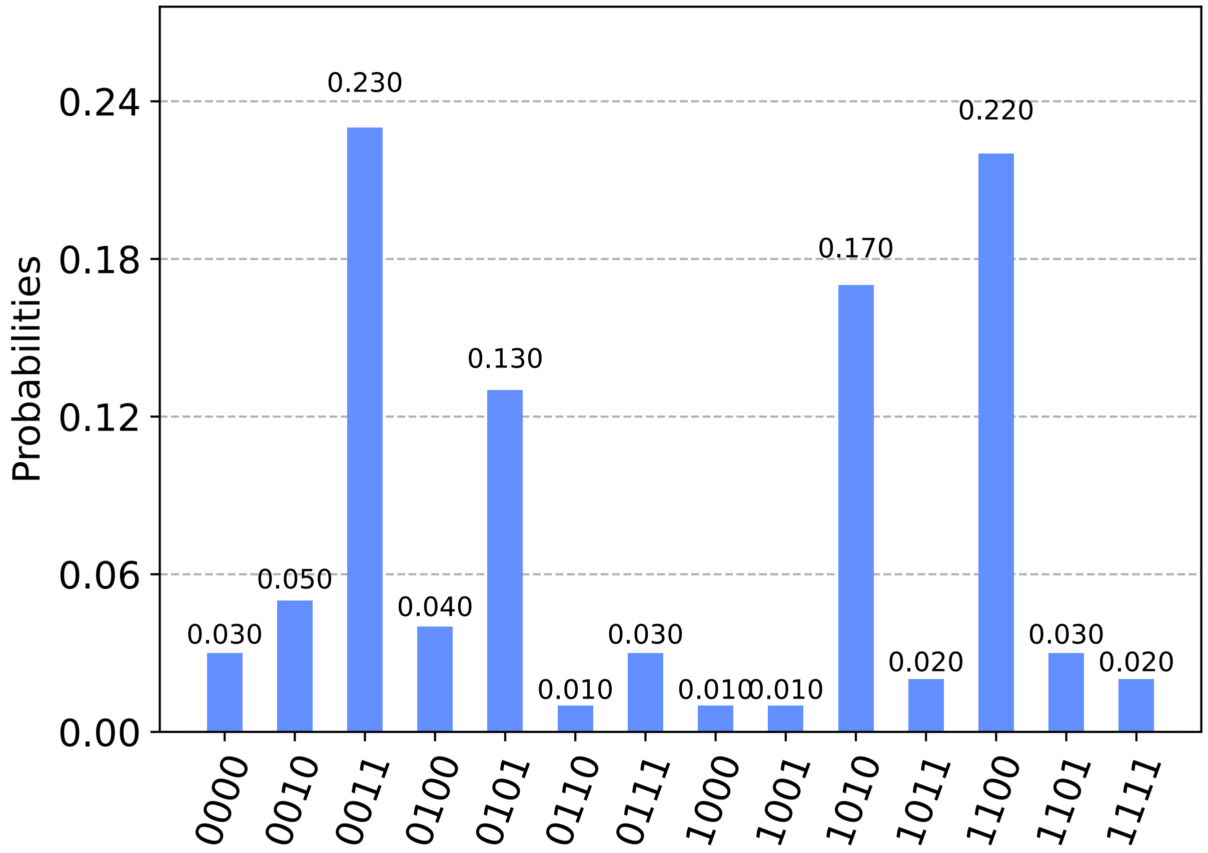
```
Text(0.5, 0, 'Iteration')
```



In [16]:

```python
#Visualize the result
result_final=run_iteration(circuit_ansatz(n, result.x, conn,layers,ansatz_type), shots)
plot_histogram(result_final)
```

Out[16]:



```
In [ ]:
```