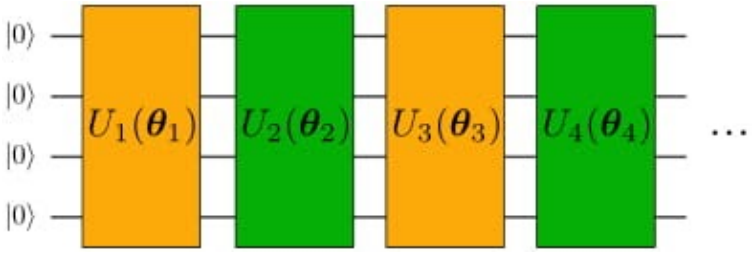
QOSF Screening Task 1 Solution

by Cenk Tüysüz

September 25, 2020

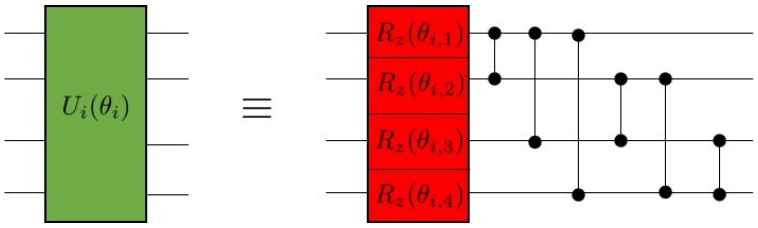
1 Task1

Implement, on a quantum simulator of your choice, the following 4 qubits state $|\psi(\theta)\rangle$:

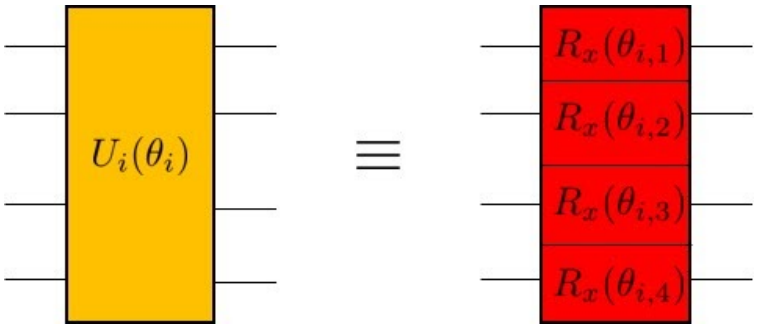


Where the number of layers, denoted with L, has to be considered as a parameter. We call "Layer" the combination of 1 yellow + 1 green block, so, for example, U1 + U2 is a layer. The odd/even variational blocks are given by:

Even blocks:



Odd blocks:



The angles $\theta_{i,n}$ are variational parameters, lying in the interval $(0, 2\pi)$, initialized at random. Double qubit gates are CZ gates.

Report with a plot, as a function of the number of layers, L, the minimum distance

$$\varepsilon = min_\theta \parallel |\psi(\theta)\rangle - |\phi\rangle \parallel$$

Where $|\phi\rangle$ is a randomly generated vector on 4 qubits and the norm $\parallel |v\rangle \parallel$, of a state $|v\rangle$, simply denotes the sum of the squares of the components of $|v\rangle$. The right set of parameters $\theta_{i,n}$ can be found via any method of choice (e.g. grid-search or gradient descent)

1.1 Bonus question:

Try using other gates for the parametrized gates and see what happens.

2 Solution

Let's start by importing NumPy, Qiskit and other relevant libraries.

[1]:
```
import numpy as np
from qiskit import Aer, QuantumRegister, QuantumCircuit, execute
from qiskit.quantum_info import random_statevector
from scipy.optimize import minimize

%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.dpi']= 200

import time
```

Now, we can define the Parametrized Quantum Circuits. We construct a QNN class to build our model under a single object. The QNN class contains several functions that we need. Let's go over them quickly. (Read the comments in the code for more details).

- `__init__()` is called when the QNN() is first called. It creates the random set of parameters, as well as the random Haar state. A fixed seed is chosen for the random state, such that results are consistent for each execution of the program.
- `qc()` takes a set of parameters as input and applies the Quantum Circuit defined in the question.
- `gate()` takes the gate name as a string and applies that gate to the specified qubit.
- `cost_fn()` calculates the ε defined in the question w.r.t. a random Haar state.
- `train()` uses the COBYLA optimizer to minimize the cost and find optimal parameters and returns the final ε.

[2]:
```
class QNN():
    ''' Constructs the Quantum Neural Network (QNN) object.
    Args:
    =====
    n_layers: int
        Number of layers, which the even and odd blocks are repeated.
    steps: int
        Number of optimization steps
    lr: float
        Learning rate
    '''
    def __init__(self,n_layers=1, gate_set=['rz','rx']):
        '''Initializer function '''
        self.n_layers = n_layers
        self.gate_set = gate_set
        self.n_qubits = 4

        # Initialize random parameters
        self.params = np.random.RandomState().uniform(low=0.0,high=2*np.
↪pi,size=(n_layers*8,))
        # Obtain a Haar Random State from Qiskit, uses fixed seed so that␣
↪results are consistent
        self.random_state = random_statevector(16,seed=32).data

    def gate(self,circ,gate,param,qr):
        '''Parametrized gate function.
        Args:
        =====
        circ: QuantumCircuit
            Quantum Circuit object from Qiskit
        gate: str
            Type of the Quantum gate
        param: float
            Parameter of the Quantum gate
        qr: QuantumRegister
            Quantum Register that we will apply the gate to
        '''
        if gate=='rz'  : circ.rz(param,qr)
        elif gate=='ry': circ.ry(param,qr)
        elif gate=='rx': circ.rx(param,qr)
        else: raise ValueError('Instruction Not Defined.')

    def qc(self, params):
        '''Defines the the Quantum Circuit.'''
        # Setup the circuit
        qr = QuantumRegister(self.n_qubits, 'qr')
        circ = QuantumCircuit(qr)
```

```
        # Repeats the block n_layers times
        for layer in range(self.n_layers):
            # Even Block
            for idx in range(self.n_qubits):
                self.gate(circ,self.gate_set[0],params[layer*8+idx],qr[idx])
            circ.cz(qr[0],qr[1])
            circ.cz(qr[0],qr[2])
            circ.cz(qr[0],qr[3])
            circ.cz(qr[1],qr[2])
            circ.cz(qr[1],qr[3])
            circ.cz(qr[2],qr[3])
            # Odd Block
            for idx in range(4):
                self.gate(circ,self.gate_set[1],params[layer*8+idx+self.
↪n_qubits],qr[idx])

        # Select the StatevectorSimulator from the Aer provider
        simulator = Aer.get_backend('statevector_simulator')

        # Execute
        result = execute(circ, simulator).result()
        return result.get_statevector(circ)


    def cost_fn(self, params):
        '''Defines the cost function: distance between two states.
        Args:
        =====
        params: float list
            Parameters of the model.
        Returns:
        ========
        float
            Cost of the Paremetrized Quantum Circuit
        '''
        return np.linalg.norm(self.qc(params) - self.random_state, ord = 2)

    def train(self):
        '''Trainer of the circuit.

        Returns:
        =======
        float
            Final cost of the model
        '''
        return minimize(self.cost_fn, self.params, method='COBYLA').fun
```

We are ready to define how many layers we want to test. Here, it is also good idea to run the same model several times and take an average, as Quantum Circuits sometimes do not train very well due to a bad initialization. Since we randomly initialize the circuits every time, we will run each model 3 times and use averaged results.

[3]:
```
# Sets number of layers to be tested.
n_layers = 20
# A list of number of layers.
layer_list = np.linspace(1,n_layers,n_layers,dtype=int)
n_runs = 3
# Arrays to store the costs for each model.
costs = []
errors = []
```

Let's train the model up to 20 layers and log the ε.

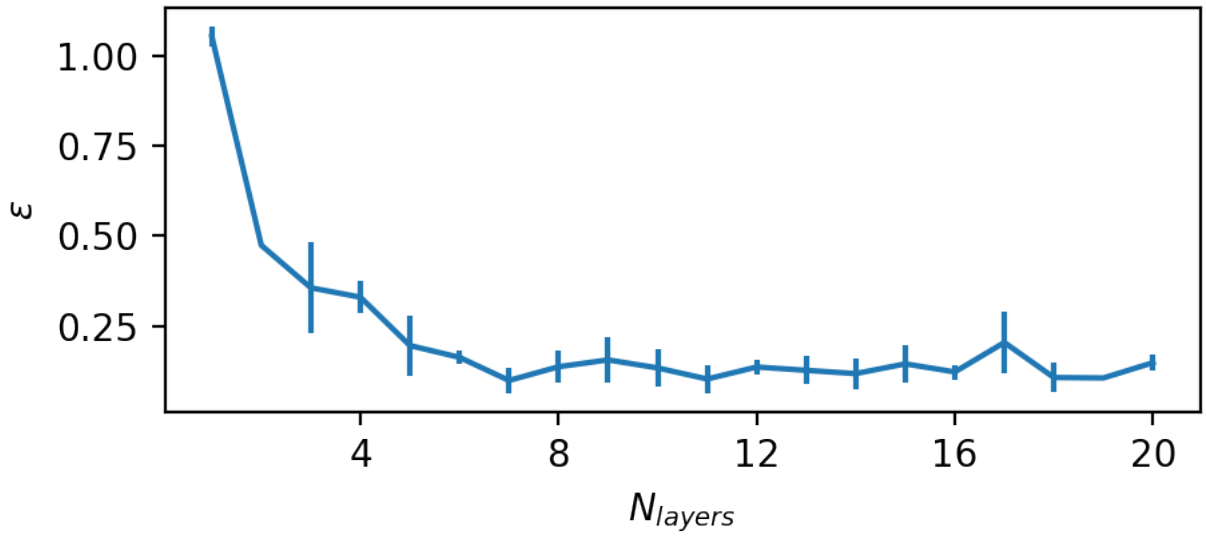
[4]:
```
# Starts a loop where the model is independently trained up to n_layer layers.
# all models are independently trained n_runs and the average of the loss curves␣
 ↪are recorded.
for idx, n_layer in enumerate(layer_list):
    # start timer
    t0 = time.time()
    run_costs = np.zeros(n_runs)
    # Train the model and append the costs to run_cost list.
    for i in range(n_runs):
        # Instantiate the QNN model with the given parameters.
        model = QNN(n_layers=n_layer)
        run_costs[i] = model.train()
    costs.append(np.mean(run_costs))
    errors.append(np.std(run_costs))
    # end timer and record duration
    duration = time.time() - t0
    # calculate the mean and std. for multiple runs
    print('Training completed for {:d} layers. Final Cost: {:1.3f} +/- {:1.3f}␣
 ↪in {:2.0f}m{:2.0f}s'.format(n_layer, costs[idx], errors[idx], duration//60,␣
 ↪duration%60))
```

```
Training completed for 1 layers. Final Cost: 1.052 +/- 0.029 in  0m 2s
Training completed for 2 layers. Final Cost: 0.474 +/- 0.000 in  0m14s
Training completed for 3 layers. Final Cost: 0.356 +/- 0.127 in  0m26s
Training completed for 4 layers. Final Cost: 0.330 +/- 0.045 in  0m31s
Training completed for 5 layers. Final Cost: 0.196 +/- 0.082 in  0m36s
Training completed for 6 layers. Final Cost: 0.163 +/- 0.018 in  0m38s
Training completed for 7 layers. Final Cost: 0.099 +/- 0.036 in  0m44s
Training completed for 8 layers. Final Cost: 0.137 +/- 0.044 in  0m49s
Training completed for 9 layers. Final Cost: 0.156 +/- 0.064 in  0m54s
Training completed for 10 layers. Final Cost: 0.134 +/- 0.051 in  1m 0s
Training completed for 11 layers. Final Cost: 0.103 +/- 0.038 in  1m 4s
```

```
Training completed for 12 layers. Final Cost: 0.136 +/- 0.021 in  1m15s
Training completed for 13 layers. Final Cost: 0.127 +/- 0.040 in  1m 8s
Training completed for 14 layers. Final Cost: 0.118 +/- 0.043 in  1m14s
Training completed for 15 layers. Final Cost: 0.145 +/- 0.051 in  1m19s
Training completed for 16 layers. Final Cost: 0.123 +/- 0.020 in  1m26s
Training completed for 17 layers. Final Cost: 0.204 +/- 0.086 in  1m48s
Training completed for 18 layers. Final Cost: 0.107 +/- 0.042 in  2m 1s
Training completed for 19 layers. Final Cost: 0.106 +/- 0.002 in  2m 5s
Training completed for 20 layers. Final Cost: 0.148 +/- 0.024 in  2m11s
```

Plot ε vs. N_{layers}

[ ]:
```
# define plot area
fig, ax = plt.subplots(1,figsize=(5, 2))
# set xticks
plt.xticks([i*(n_layers//5) for i in range(2+(n_layers//5))])
# plot with error bars
plt.errorbar(x=layer_list, y=costs, yerr=errors)
# set labels
ax.set_xlabel(r'$N_{layers}$')
ax.set_ylabel(r'$\varepsilon$')
# show plot
plt.show()
```



Above plot shows us that, after some N_{layers} the model stops improving. The main reason for this is that the generalizability of the Quantum Circuits saturates after some layers [1]. Also, it gets harder to train Quantum Circuits as the depth increases [2,3]. As a result, we obtain a convergence plateau.

In this question, we try to show the effect of increasing the depth of the Quantum Circuits. Although, there are methods [3,4] to obtain better ε, the aim is not to get the best possible result.

3 Solution to Bonus Question

The bonus question asks to vary the gates of the model. We define a set of gate combinations and test them using the setting $N_{layer} = 9$. Here, we only use the R_X, R_Y and R_Z gates but this example can be extended with the use of other parametrized gates such as U_1, U_2, U_3, etc.

[6]:
```
# Choose a layer
n_layer = 9
n_runs = 3
# Creates a set of gates to run
gate_list = [
    ['rz','rz'],
    ['rz','ry'],
    ['rz','rx'],
    ['ry','rz'],
    ['ry','ry'],
    ['ry','rx'],
    ['rx','rz'],
    ['rx','ry'],
    ['rx','rx'],
    ]
n_gate_sets = len(gate_list)
# Arrays to store the costs for each model.
costs = []
errors = []
```

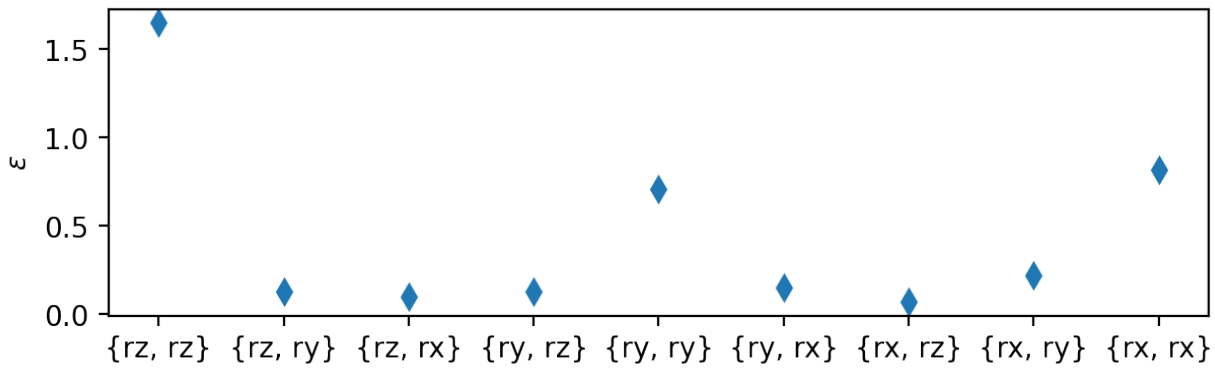
[7]:
```
# Training loop
for idx, gates in enumerate(gate_list):
    # start timer
    t0 = time.time()
    run_costs = np.zeros(n_runs)
    for run in range(n_runs):
        model = QNN(n_layers=n_layer,gate_set=gates)
        run_costs[run] = model.train()

    costs.append(np.mean(run_costs))
    errors.append(np.std(run_costs))
    # end timer and record duration
    duration = time.time() - t0
    # calculate the mean and std. for multiple runs
    print('Training completed for even gates: {}, odd gates: {} layers. Final␣
 ↪Cost: {:1.3f} +/- {:1.3f} in {:2.0f}m{:2.0f}s'.format(gates[0],gates[1],␣
 ↪costs[idx], errors[idx], duration//60, duration%60))
```

```
Training completed for even gates: rz, odd gates: rz layers. Final Cost: 1.649
+/- 0.000 in  0m23s
Training completed for even gates: rz, odd gates: ry layers. Final Cost: 0.128
+/- 0.052 in  0m58s
Training completed for even gates: rz, odd gates: rx layers. Final Cost: 0.101
+/- 0.032 in  0m59s
Training completed for even gates: ry, odd gates: rz layers. Final Cost: 0.127
+/- 0.041 in  0m57s
Training completed for even gates: ry, odd gates: ry layers. Final Cost: 0.710
+/- 0.000 in  1m22s
Training completed for even gates: ry, odd gates: rx layers. Final Cost: 0.153
+/- 0.062 in  1m22s
Training completed for even gates: rx, odd gates: rz layers. Final Cost: 0.072
+/- 0.032 in  0m57s
Training completed for even gates: rx, odd gates: ry layers. Final Cost: 0.218
+/- 0.062 in  1m24s
Training completed for even gates: rx, odd gates: rx layers. Final Cost: 0.820
+/- 0.000 in  1m22s
```

Plot the final ε of each model against the gate sets.

[ ]:
```
fig, ax = plt.subplots(1, figsize=(7, 2))
x_ticks = ['{'+gate_list[i][0]+', '+gate_list[i][1]+'}' for i in␣
 ↪range(n_gate_sets)]
plt.xticks(range(n_gate_sets), x_ticks)
plt.plot(range(n_gate_sets), costs, marker="d", linestyle="None")
ax.set_ylabel(r'$\varepsilon$')
plt.show()
```



This plot shows us that we get the best results when we have combinations of different gate types. The perfomance decreases significantly, when we have the same gate types for even and odd blocks. However, there is a much wore case, which is the case where there are only R_Z gates.

The reason for getting the worse ε in the only R_Z case is that the qubits being initialized in the

$|z;0\rangle$ state. When we use only the R_Z gate, we can't get out of the $|0\rangle$ state.

This situation also extends to the cases when we only use R_X or R_Y gates. In these cases, the sytem can get out of the $|0\rangle$ state. However, since there is only 1 degree of freedom, the model can't use the full potential of the bloch sphere.

The models with different gates performs the best as we can make us of complete bloch sphere. This result can also be inferred from linear algebra, where a general U_3 unitary transformation can be decomposed to a set of rotations in different axes.

4 References

[1] S. Sim, P. D. Johnson, and A. Aspuru-Guzik, "Expressibility and Entangling Capability of Parameterized Quantum Circuits for Hybrid Quantum-Classical Algorithms," Adv. Quantum Technol., vol. 2, no. 12, p. 1900070, 2019, doi: 10.1002/qute.201900070. Available: https://arxiv.org/abs/1905.10876

[2] J. R. McClean, S. Boixo, V. N. Smelyanskiy, R. Babbush, and H. Neven, "Barren plateaus in quantum neural network training landscapes," Nat. Commun., vol. 9, no. 1, pp. 1–6, 2018, doi: 10.1038/s41467-018-07090-4. Available: https://arxiv.org/abs/1803.11173

[3] E. Grant, L. Wossnig, M. Ostaszewski, and M. Benedetti, "An initialization strategy for addressing barren plateaus in parametrized quantum circuits," Quantum, vol. 3, p. 214, 2019, doi: 10.22331/q-2019-12-09-214. Available: https://arxiv.org/abs/1903.05076

[4] A. Skolik, J. R. McClean, M. Mohseni, P. van der Smagt, and M. Leib, "Layerwise learning for quantum neural networks," 2020, [Online]. Available: http://arxiv.org/abs/2006.14904.

End Note

This document is partially auto-generated and originally written as a jupyter notebook. There might be some discrepancies in means of formatting. Please alse refer to https://github.com/cnktysz/qosf-screening-tasks/blob/master/task1/Task1.ipynb