Quantum Fourier transform via variational quantum circuits

Steiropoulou Evangelia

1 Code implementation

[ ]:
```
import torch
import numpy as np
import scipy.linalg
import sys
import matplotlib.pyplot as plt
import numpy as np
import time
```

1.1 Pauli Matrices:

1.1.1 Pauli-X matrix:

[ ]:
```
ss1 = [[0,1],[1,0]]
ss1 = torch.tensor(ss1)
```

1.1.2 Pauli-Y matrix:

[ ]:
```
ss2 = [[0,-1j],[1j,0]]
ss2 = torch.tensor(ss2)
```

1.1.3 Pauli-Z matrix:

[ ]:
```
ss3 = [[1,0],[0,-1]]
ss3 = torch.tensor(ss3)
```

1.1.4 Identity matrix:

[ ]:
```
ss4 = torch.eye(2)
```

1.2 QFT matrix, row by row:

```
[ ]: #fill a tensor 1*8 with 1/(2*sqrt(2)
QF3 = torch.full((1,8),1/(2*torch.sqrt(torch.tensor(2.0))))
QF3 = torch.tensor(QF3)
```

```
[ ]: #fill a tensor 1*8 with 1/(2*sqrt(2), exponential(1j*pi/4)/2*sqrt(2), (1j)/2*sqrt(2), exponential(3j*pi/4)/2*sqrt(2), - 1/(2*sqrt(2), exponential(-3j*pi/4)/2*sqrt(2), -(1j)/2*sqrt(2), exponential(-1j*pi/4)/2*sqrt(2)
QF4 = [[1/(2*torch.sqrt(torch.tensor(2.0))), (torch.exp(1j*torch.tensor(np.pi/4)))/(2*torch.sqrt(torch.tensor(2.0))), 1j/(2*torch.sqrt(torch.tensor(2.0))) , torch.exp(3j*torch.tensor(np.pi/4))/(2*torch.sqrt(torch.tensor(2.0))),-1/(2*torch.sqrt(torch.tensor(2.0))), torch.exp(-3j*torch.tensor(np.pi/4))/(2*torch.sqrt(torch.tensor(2.0))) , -1j/(2*torch.sqrt(torch.tensor(2.0))), torch.exp(-1j*torch.tensor(np.pi/4))/(2*torch.sqrt(torch.tensor(2.0)))]]
QF4 = torch.tensor(QF4)
```

```
[ ]: #fill a tensor 1*8, with 1/(2*sqrt(2), (1j)/2*sqrt(2), - 1/(2*sqrt(2),  -(1j)/2*sqrt(2), 1/(2*sqrt(2), (1j)/2*sqrt(2), - 1/(2*sqrt(2),  -(1j)/2*sqrt(2)
QF5 = [[1/(2*torch.sqrt(torch.tensor(2.0))), 1j/(2*torch.sqrt(torch.tensor(2.0))), -1/(2*torch.sqrt(torch.tensor(2.0))), -1j/(2*torch.sqrt(torch.tensor(2.0))), 1/(2*torch.sqrt(torch.tensor(2.0))), 1j/(2*torch.sqrt(torch.tensor(2.0))), -1/(2*torch.sqrt(torch.tensor(2.0))), -1j/(2*torch.sqrt(torch.tensor(2.0)))]]
QF5 = torch.tensor(QF5)
```

```
[ ]: #fill a tensor 1*8, with 1/(2*sqrt(2), exponential(3j*pi/4)/2*sqrt(2), -(1j)/2*sqrt(2), exponential(1j*pi/4)/2*sqrt(2), - 1/(2*sqrt(2),  exponential(-1j*pi/4)/2*sqrt(2), (1j)/2*sqrt(2), exponential(-3j*pi/4)/2*sqrt(2)
QF6 = [[1/(2*torch.sqrt(torch.tensor(2.0))), torch.exp(3j*torch.tensor(np.pi/4))/(2*torch.sqrt(torch.tensor(2.0))), -1j/(2*torch.sqrt(torch.tensor(2.0))), torch.exp(1j*torch.tensor(np.pi/4))/(2*torch.sqrt(torch.tensor(2.0))),-1/(2*torch.sqrt(torch.tensor(2.0))), torch.exp(-1j*torch.tensor(np.pi/4))/(2*torch.sqrt(torch.tensor(2.0))) , 1j/(2*torch.sqrt(torch.tensor(2.0))), torch.exp(-3j*torch.tensor(np.pi/4))/(2*torch.sqrt(torch.tensor(2.0)))]]
QF6 = torch.tensor(QF6)
```

```
[ ]: #fill a tensor 1*8 with 1/(2*sqr(2) and -1/(2*sqr(2)
QF7 = [[1/(2*torch.sqrt(torch.tensor(2.0))), -1/(2*torch.sqrt(torch.tensor(2.0))), 1/(2*torch.sqrt(torch.tensor(2.0))), -1/(2*torch.sqrt(torch.tensor(2.0))), 1/(2*torch.sqrt(torch.tensor(2.0))), -1/(2*torch.sqrt(torch.tensor(2.0))), 1/(2*torch.sqrt(torch.tensor(2.0))), -1/(2*torch.sqrt(torch.tensor(2.0)))]]
QF7 = torch.tensor(QF7)
```

```
[ ]:
```

```
QF8 = [[1/(2*torch.sqrt(torch.tensor(2.0))), torch.exp(-3j*torch.tensor(np.pi/
 ↪4))/(2*torch.sqrt(torch.tensor(2.0))), 1j/(2*torch.sqrt(torch.tensor(2.0))),␣
 ↪torch.exp(-1j*torch.tensor(np.pi/4))/(2*torch.sqrt(torch.tensor(2.0))), -1/
 ↪(2*torch.sqrt(torch.tensor(2.0))), torch.exp(1j*torch.tensor(np.pi/4))/
 ↪(2*torch.sqrt(torch.tensor(2.0))) , -1j/(2*torch.sqrt(torch.tensor(2.0))),␣
 ↪torch.exp(3j*torch.tensor(np.pi/4))/(2*torch.sqrt(torch.tensor(2.0)))]]
QF8 = torch.tensor(QF8)
```

[ ]:
```
#fill a tensor 1*8, with 1/(2*sqrt(2), (-1j)/2*sqrt(2), - 1/(2*sqrt(2),  (1j)/
 ↪2*sqrt(2), 1/(2*sqrt(2), (-1j)/2*sqrt(2), - 1/(2*sqrt(2),  (1j)/2*sqrt(2)
QF9 = [[1/(2*torch.sqrt(torch.tensor(2.0))), -1j/(2*torch.sqrt(torch.tensor(2.
 ↪0))), -1/(2*torch.sqrt(torch.tensor(2.0))), 1j/(2*torch.sqrt(torch.tensor(2.
 ↪0))), 1/(2*torch.sqrt(torch.tensor(2.0))), -1j/(2*torch.sqrt(torch.tensor(2.
 ↪0))), -1/(2*torch.sqrt(torch.tensor(2.0))), 1j/(2*torch.sqrt(torch.tensor(2.
 ↪0)))]]
QF9 = torch.tensor(QF9)
```

1.2.1 QFT matrix:

[ ]:
```
#fill a tensor 1*8 with 1/(2*sqrt(2), exponential(-1j*pi/4)/2*sqrt(2), (-1j)/
 ↪2*sqrt(2), exponential(-3j*pi/4)/2*sqrt(2), - 1/(2*sqrt(2), exponential(3j*pi/
 ↪4)/2*sqrt(2), (1j)/2*sqrt(2), exponential(1j*pi/4)/2*sqrt(2)
QF10 = [[1/(2*torch.sqrt(torch.tensor(2.0))), torch.exp(-1j*torch.tensor(np.pi/
 ↪4))/(2*torch.sqrt(torch.tensor(2.0))), -1j/(2*torch.sqrt(torch.tensor(2.0))),␣
 ↪torch.exp(-3j*torch.tensor(np.pi/4))/(2*torch.sqrt(torch.tensor(2.0))),-1/
 ↪(2*torch.sqrt(torch.tensor(2.0))), torch.exp(3j*torch.tensor(np.pi/4))/
 ↪(2*torch.sqrt(torch.tensor(2.0))) , 1j/(2*torch.sqrt(torch.tensor(2.0))),␣
 ↪torch.exp(1j*torch.tensor(np.pi/4))/(2*torch.sqrt(torch.tensor(2.0)))]]
QF10 = torch.tensor(QF10)
```

$$\frac{1}{\sqrt{8}}\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & e^{i\frac{\pi}{4}} & e^{i\frac{\pi}{2}} & e^{i\frac{3\pi}{4}} & e^{i\pi} & e^{i\frac{5\pi}{4}} & e^{i\frac{3\pi}{2}} & e^{i\frac{7\pi}{4}} \\
1 & e^{i\frac{\pi}{2}} & e^{i\pi} & e^{i\frac{3\pi}{2}} & 1 & e^{i\frac{\pi}{2}} & e^{i\pi} & e^{i\frac{3\pi}{2}} \\
1 & e^{i\frac{3\pi}{4}} & e^{i\frac{3\pi}{2}} & e^{i\frac{9\pi}{4}} & e^{i\pi} & e^{i\frac{5\pi}{4}} & e^{i\frac{7\pi}{2}} & e^{i\frac{15\pi}{4}} \\
1 & e^{i\pi} & 1 & e^{i\pi} & 1 & e^{i\pi} & 1 & e^{i\pi} \\
1 & e^{i\frac{5\pi}{4}} & e^{i\frac{\pi}{2}} & e^{i\frac{7\pi}{4}} & e^{i\pi} & e^{i\frac{\pi}{4}} & e^{i\frac{\pi}{2}} & e^{i\frac{3\pi}{4}} \\
1 & e^{i\frac{3\pi}{2}} & e^{i\pi} & e^{i\frac{7\pi}{2}} & 1 & e^{i\frac{\pi}{2}} & e^{i\pi} & e^{i\frac{3\pi}{2}} \\
1 & e^{i\frac{7\pi}{4}} & e^{i\frac{3\pi}{2}} & e^{i\frac{15\pi}{4}} & e^{i\pi} & e^{i\frac{3\pi}{4}} & e^{i\frac{7\pi}{2}} & e^{i\frac{15\pi}{4}}
\end{bmatrix}$$

[ ]:
```
#make tensor with all the above tensors in it
QF = torch.cat((QF3, QF4, QF5, QF6, QF7, QF8, QF9, QF10), 0)
```

1.3 Generators:

Here we create the generators, the quantum gates that are goint to be used in the circuit. The generators, are combinations of Kronecker products of the gates we mentioned above.

1.3.1 Single qubit gates:

(1,4,4)

```
[ ]: c1 = torch.kron(ss1, ss4)
     c1 = torch.kron(c1, ss4)
```

(4,2,4)

```
[ ]: c2 = torch.kron(ss4, ss2)
     c2 = torch.kron(c2, ss4)
```

(4,3,4)

```
[ ]: c3 = torch.kron(ss4, ss3)
     c3 = torch.kron(c3, ss4)
```

(4,1,4)

```
[ ]: c4 = torch.kron(ss4, ss1)
     c4 = torch.kron(c4, ss4)
```

(4,4,3)

```
[ ]: c5 = torch.kron(ss4, ss4)
     c5 = torch.kron(c5, ss3)
```

1.3.2 Two - qubit gates:

(4,3,3)

```
[ ]: c6 = torch.kron(ss4, ss3)
     c6 = torch.kron(c6, ss3)
```

(4,1,3)

```
[ ]: c7 = torch.kron(ss4, ss1)
     c7 = torch.kron(c7, ss3)
```

(1,1,4)

```
[ ]: c8 = torch.kron(ss1, ss1)
     c8 = torch.kron(c8, ss4)
```

(1,2,4)

```
[ ]: c9 = torch.kron(ss1, ss2)
     c9 = torch.kron(c9, ss4)
```

(3,4,2)

```
[ ]: c10 = torch.kron(ss3, ss4)
     c10 = torch.kron(c10, ss2)
```

(1,4,3)

```
[ ]: c11 = torch.kron(ss1, ss4)
     c11 = torch.kron(c11, ss3)
```

In vv3 we will add all the generators, for later use.

```
[ ]: #c1 - c5 are single qubit gates, c6 - c11 are two qubit gates
     vv3 = torch.stack((c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11))
```

1.4 Variational Gates:

```
[ ]: #Gi_, j_, k_, x_ := MatrixExp[I x KroneckerProduct[Assi, ssj, ssk]]
     G = torch.zeros(11, 8, 8, dtype=torch.complex64)
     for i in range(G.size(dim = 0)):
         G[i]= torch.linalg.matrix_exp(1j*vv3[i])
```

```
[ ]: def Gx(x):
         Gx = torch.zeros(11, 8, 8, dtype=torch.complex64)
         for i in range(Gx.size(dim = 0)):
             Gx[i] = torch.tensor(scipy.linalg.fractional_matrix_power(G[i], x)) #G␣
      ↪to the power of x
         return Gx
```

```
[ ]: #find conjugate transpose of QF
     B = torch.conj(torch.transpose(QF, 0,1))
     B.requires_grad = True
```

1.5 Circuit generator:

The circuit created below, was firstly designed by hand, in order to combine both single and 2-qubit variational gates.

```
[ ]: def create_circuit(x_var, parameters_num):
         Gm = []
         # loop over the x values to generate the corresponding G matrices
         for i in range(x_var.size(dim=0)):
             Gx_i = torch.zeros(11, 8, 8, dtype=torch.complex64)
             Gx_i = Gx(x_var[i].item())
             Gm.append(Gx_i)

         # multiply the 18 G matrices to get the final G matrix/circuit
         i = 0

         G1 = Gm[i][5]   #get the first 2-qubit gate(of the first x-modified␣
      ↪Gx_i(i==0)), 4-3-3
         G2 = Gm[i+1][1] #get the second single qubit gate, 4-2-4
         G3 = Gm[i+2][4] #get the last single qubit gate 4-4-3
         G4 = Gm[i+3][7] #1-1-4
         G5 = Gm[i+4][0] #1-4-4
```

```
    G6 = Gm[i+5][2] #4-3-4
    G7 = Gm[i+6][9] #3-4-2
    G8 = Gm[i+7][4] #4-4-3
    G9 = Gm[i+8][0] #1-4-4
    G10 = Gm[i+9][6] #4-1-3
    G11 = Gm[i+10][2] #4-3-4
    G12 = Gm[i+11][4] #4-4-3
    G13 = Gm[i+12][8] #1-2-4
    G14 = Gm[i+13][0] #1-4-4
    G15 = Gm[i+14][3] #4-1-4
    G16 = Gm[i+15][10]#1-4-3
    G17 = Gm[i+16][0] #1-4-4
    G18 = Gm[i+17][4] #4-4-3

    G_final = G1@G2@G3@G4@G5@G6@G7@G8@G9@G10@G11@G12@G13@G14@G15@G16@G17@G18

    #In order to expand the initial 18-parameter circuit to a n-parameter circuit, we need to add more gates at the end of the circuit.
    #As additional gates we will use those in the initial 18-parameter circuit. For example if the desired circuit has 20 parameters,
    #we need to add 2 additional gates to the initial 18-parameter circuit. In order to do that, we will add the first 2 gates of the initial circuit
    #at the end of the circuit. If we want a 28-parameter circuit, we will add the first 10 gates of the initial circuit, at the end etc.

    #Initial gate indices
    gate_indices = [5, 1, 4, 7, 0, 2, 9, 4, 0, 6, 2, 4, 8, 0, 3, 10, 0, 4]   # Example gate indices
    # Additional gates
    G_additional = torch.eye(8, dtype=torch.complex64)  # identity matrix

    # Multiply additional gates based on the number of parameters
    for i in range(parameters_num - 18):
        gate_idx = gate_indices[i % 18]  # Cycle through the gate_indices list
        G_additional = G_additional @ Gm[i+18][gate_idx]
        G_final = G_final @ G_additional # Multiply G_final with G_additional

    return G_final
```

1.6 Cost function:

[ ]:
```
def cost_function(x_var):
    G_final = create_circuit(x_var, len(x_var))
    cost = 1 - 1/64 * ((torch.abs(torch.trace(G_final @ B)))**2)
    return cost
```

1.7 Optimization methods:

[ ]:
```
def learning_rate_step_scheduler(learning_rate, step_size):
    return learning_rate * step_size
```

1.7.1 Gradient Descent optimizer:

[ ]:
```
#function performs gradient descent of cost to find the optimal x values
#x_var is the initial x values, gamma is the learning rate, delta is the␣
↪perturbation value
def optimize_parameters(x_var, gamma, delta):
    #print("x initial is: \n\n", x_var)
    #print("cost initial = ", cost_function(x_var))
    x_new = x_var.clone()

    for i in range(len(x_var)):
        x_var_sum = x_var.clone() #create a copy of the x_var tensor
        x_var_sum[i] = x_var[i] + delta
        cost_sum = cost_function(x_var_sum)

        x_var_diff = x_var.clone()
        x_var_diff[i] = x_var[i] - delta
        cost_diff = cost_function(x_var_diff)
        x_new[i] = x_var[i] - gamma * ((cost_sum - cost_diff) / (2* delta))

    return x_new, cost_function(x_new)
```

[ ]:
```
#function that calls the optimize_parameters function until the cost stops␣
↪changing more than a certain value(epsilon)
def gradient_descent_cost_optimizer(x_var, learning_rate, delta, epsilon,␣
↪threshold, step_size):
    iterations = 0
    x_init, cost_init = optimize_parameters(x_var, learning_rate, delta) #get␣
↪the initial cost after the first optimization
    x_old = x_init.clone()
    cost_old = cost_init.clone()
    cost_history = [cost_init]  # List to store the cost at each iteration

    while True:
        #print("ITERATION = \n", iterations)
        x_new, cost_new = optimize_parameters(x_old, learning_rate, delta)
        #print("new cost = ", cost_new)
        if torch.abs(cost_new - cost_old) < epsilon:
            break
        else:
            if(torch.abs(cost_new - cost_old) < threshold and iterations != 0):
```

```
                learning_rate = learning_rate_step_scheduler(learning_rate,␣
↪step_size)
                #print("ITERATION = ", iterations, "    LEARNING RATE = ",␣
↪learning_rate, "\n")
            x_old = x_new.clone()
            cost_old = cost_new.clone()
            iterations += 1
            cost_history.append(cost_new)  # Add the current cost to the history

    return x_new, cost_new, iterations, cost_history
```

1.7.2 Stochastic gradient descent:

```
[ ]: # Perform optimization on a single data point -> this will be used in the␣
     ↪stochastic gradient descent
     def optimize_stochastic_parameters(x_var, learning_rate, delta, data_point):  #␣
     ↪Perform optimization on a single data point
         x_var_sum = x_var.clone()
         x_var_sum[data_point] = x_var[data_point] + delta
         cost_sum = cost_function(x_var_sum)

         x_var_diff = x_var.clone()
         x_var_diff[data_point] = x_var[data_point] - delta
         cost_diff = cost_function(x_var_diff)

         x_new = x_var.clone()
         x_new[data_point] = x_var[data_point] - learning_rate* ((cost_sum -␣
     ↪cost_diff) / (2 * delta))

         return x_new, cost_function(x_new)
```

```
[ ]: def stochastic_gradient_descent(x_var, learning_rate, delta, epsilon, threshold,␣
     ↪step_size, scheduler, num_epochs):
         iterations = 0
         num_data_points = len(x_var)
         x_init, cost_init = optimize_stochastic_parameters(x_var, learning_rate,␣
     ↪delta, np.random.randint(num_data_points))
         x_old = x_init.clone()
         cost_old = cost_init.clone()
         cost_difference = torch.abs(cost_old - cost_init)
         cost_history = [cost_init]  # List to store the cost at each iteration

         while True:
             #print("ITERATION =", iterations)
             data_point = np.random.randint(num_data_points)
```

```
        x_new, cost_new = optimize_stochastic_parameters(x_old, learning_rate,␣
↪delta, data_point)
        #print("x new =", x_new)
        #print("new cost =", cost_new)

        if torch.abs(cost_new - cost_old) != cost_difference:
            cost_difference = torch.abs(cost_new - cost_old)
            #print("cost difference =", cost_difference)
        if iterations > num_epochs and cost_difference < epsilon:
            break
        else:
            if cost_difference < threshold and iterations != 0:
                #print("Scheduler called", scheduler)
                learning_rate = scheduler(learning_rate, step_size)
                #print("ITERATION =", iterations, " LEARNING RATE =",␣
↪learning_rate, "\n")

            x_old = x_new.clone()
            cost_old = cost_new.clone()
            iterations += 1
            cost_history.append(cost_new)  # Add the current cost to the history

    return x_new, cost_new, iterations, cost_history
```

1.8 Execution of the program, and cost optimization

[297]:
```
num_parameters = [18, 22, 26, 28]
iterations = [100, 150, 200, 300] # Number of iterations for each num of␣
↪parameters. Used only in stochastic gradient descent.
#In this implementation, we perform only the gradient descent algorithm

counter = 1  # Initialize the counter

# Create an empty list to store the results
results = []
for algorithm in [gradient_descent_cost_optimizer]:
    for i, j in zip(num_parameters, iterations):

        # Create an empty list to store the results
        results_algorithm = []

        # Count time for each iteration
        start = time.time()
        x_var = torch.rand(i, dtype=torch.float32) * 2 * np.pi
        print("Algorithm is: ", algorithm.__name__, "\n")
        print("Number of parameters is: ", i, "\n")
        print("x initial is: \n", x_var, "\n")
```

```
        if algorithm == stochastic_gradient_descent:
            learning_rate = 0.05
            delta = 0.0005
            epsilon = 1e-08
            threshold = 0.00001
            step_size = 0.1
            x, cost, iters, cost_history = stochastic_gradient_descent(x_var,␣
↪learning_rate, delta, epsilon, threshold, step_size,␣
↪learning_rate_step_scheduler,j)
        else:
            learning_rate =  0.05
            delta = 0.005
            epsilon =  1e-08
            threshold =  0.0001
            step_size =   0.1
            x, cost, iters, cost_history =␣
↪gradient_descent_cost_optimizer(x_var, learning_rate, delta, epsilon,␣
↪threshold, step_size)

        results_algorithm.append((x_var, cost_function(x_var), x, iters, cost))
        end = time.time()

        print("Parameters are: \n" , i, " X INITIAL is:\n", x_var)
        print("initial cost: ", cost_function(x_var))
        print("X FINAL is:\n\n", x)
        print("iterations =", iters, "final cost: ", cost)
        print("time taken =", end - start, "\n\n")
        print("learning_rate = ", learning_rate, "\n")
        print("delta = ", delta, "\n")
        print("epsilon = ", epsilon, "\n")
        print("threshold = ", threshold, "\n")
        print("step_size = ", step_size, "\n")

        cost_history_np = np.array([cost.detach().numpy() for cost in␣
↪cost_history])

        # Plot the cost progression
        plt.figure()
        plt.plot(cost_history_np)
        plt.xlabel('Iteration')
        plt.ylabel('Cost')
        plt.title(f'Cost Progression for: {i} Parameters, {algorithm.__name__}')

        plt.ylim(bottom=0.0, top=1)

        # Show the plot without blocking program execution
```

```
        plt.show(block=False)

        # Save the figure as a PNG file
        filename = f'cost_progression_{i}_{algorithm.__name__}_{counter}.png'
        plt.savefig(filename)

        # Increment the counter
        counter += 1

        # Append the results to the list
        for result in results_algorithm:
            results.append({
                'Algorithm': algorithm.__name__,
                'Number of Parameters': i,
                'Initial Values': result[0].detach().numpy(),
                'Final Values': result[2].detach().numpy(),
                'Iterations': result[3],
                'Initial Cost': result[1].item(),
                'Final Cost': result[4].item(),
                'Execution Time': end - start
            })

        # Open the file in write mode
            output_file = f'{i}_{algorithm.__name__}_{counter}.txt'
            with open(output_file, 'w') as file:
                # Iterate over the 'results' list and write each element to the␣
↪file
                for result in results:
                    file.write("Algorithm: {}\n".format(result['Algorithm']))
                    file.write("Number of Parameters: {}\n".
↪format(result['Number of Parameters']))
                    file.write("Initial Values: {}\n".format(result['Initial␣
↪Values']))
                    file.write("Final Values: {}\n".format(result['Final␣
↪Values']))
                    file.write("Initial Cost: {}\n".format(result['Initial␣
↪Cost']))
                    file.write("Final Cost: {}\n".format(result['Final Cost']))
                    file.write("Iterations: {}\n".format(result['Iterations']))
                    file.write("Execution Time: {}\n".format(result['Execution␣
↪Time']))
```