# -\*- coding: utf-8 -\*-  
  
import qiskit  
import typing  
import numpy as np  
from qiskit import QuantumCircuit  
from qiskit.circuit import ParameterVector  
from pqc import Interface  
from pqc import Multiprocess  
  
class Simulation(Multiprocess):  
   
 def \_\_init\_\_(self,circ=None,samples:int=10,num\_proc:int=1):  
 Multiprocess.\_\_init\_\_(self,circ,samples,num\_proc)  
 self.thetas,self.phis = self.get\_params()  
   
 def simulate(self,theta):  
 circuit = self.\_circ.assign\_parameters(theta)  
 circuit.snapshot("final", snapshot\_type="statevector")  
 result = qiskit.execute(  
 circuit, qiskit.Aer.get\_backend("aer\_simulator\_statevector")  
 ).result()  
 result\_data = result.data(0)["snapshots"]["statevector"]["final"][0]  
  
 return result\_data  
   
 def get\_params(self) -> typing.Tuple[typing.List, typing.List]:  
 """Generate parameters for the calculation of expressibility  
 :returns theta (np.array): first list of parameters for the parameterized quantum circuit  
 :returns phi (np.array): second list of parameters for the parameterized quantum circuit  
 """  
 #np.random.seed(1234)  
 theta = [  
 {p: 2 \* np.random.random() \* np.pi for p in self.\_circ.parameters}  
 for \_ in range(self.\_samples)  
 ]  
 #np.random.seed(1234)  
 phi = [  
 {p: 2 \* np.random.random() \* np.pi for p in self.\_circ.parameters}  
 for \_ in range(self.\_samples)  
 ]  
   
 params = theta,phi  
   
 return params  
   
   
 def get\_circuits(self):  
 """  
 Function that needs multiprocessing.  
   
 """  
   
 thetas = self.thetas  
 phis = self.phis  
 arr1,arr2 = self.job(self.simulate,[(i,thetas[i],phis[i]) for i in range(self.\_samples)])  
 arr1 = [arr1[i] for i in range(self.\_samples)]  
 arr2 = [arr2[i] for i in range(self.\_samples)]  
   
 return arr1, arr2  
   
   
if \_\_name\_\_=="\_\_main\_\_":  
 Num = 4  
 qc = QuantumCircuit(Num)  
 x = ParameterVector(r'$\theta$', length=8)  
 [qc.h(i) for i in range(Num)]   
 [qc.ry(x[int(2\*i)], i) for i in range(Num)]  
 [qc.rz(x[int(2\*i+1)], i) for i in range(Num)]  
 qc.cx(0, range(1, Num))  
 circ = qc  
 run = Simulation(circ,samples=10,num\_proc=2)  
 print(type(run.thetas),type(run.thetas[0]))  
 print(run.thetas[0].values(),run.phis[0].values())  
   
   
# -\*- coding: utf-8 -\*-  
  
  
from pqc import Analyser  
from qiskit import QuantumCircuit  
from qiskit.circuit import ParameterVector  
  
  
def analyse(circ=None,samples:int=100,descriptor='both',method\_ex:str='kl',  
 method\_ec:str='mw',num\_proc:int=1):  
 """  
 function that implements the package.  
 -------------------------------------  
 descriptor: 'ex','ec' or both, specifying the descriptor(s) one wants to use.  
 circ: a qiskit QuantumCircuit object.  
 samples: int, number of samples for the circuit parameters.  
 method\_ex: string, 'kl', or 'js', method for evaluating the circuit expressibility.  
 method\_ec: string, 'mw' or 'scott', method for evaluating the circuit entanglement capability.  
 num\_proc: int, number of processes for multiprocessing, the default is 1.   
 """  
   
 analyser = Analyser(circ=circ,samples=samples,method\_ex=method\_ex,method\_ec=method\_ec,num\_proc=num\_proc)  
   
 if descriptor == 'both':  
 return analyser.get\_expressibility(),analyser.get\_entanglement()  
 elif descriptor == 'ex':  
 return analyser.get\_expressibility()  
 elif descriptor == 'ec':  
 return analyser.get\_entanglement()  
   
 else:  
 raise ValueError(  
 "Invalid descriptor, choose from 'ex', 'ec' or 'both'."  
 )  
   
  
  
if \_\_name\_\_=="\_\_main\_\_":  
 Num = 4  
 qc = QuantumCircuit(Num)  
 x = ParameterVector(r'$\theta$', length=8)  
 [qc.h(i) for i in range(Num)]   
 [qc.ry(x[int(2\*i)], i) for i in range(Num)]  
 [qc.rz(x[int(2\*i+1)], i) for i in range(Num)]  
 qc.cx(0, range(1, Num))  
 circ = qc  
  
 analyse(circ=qc)# -\*- coding: utf-8 -\*-  
  
class Interface:  
   
 def \_\_init\_\_(self,circ=None,samples:int=1000)->None:  
 """  
 Interface which initialise the user input.  
 ------------------------------------------  
 circ: qiskit.QuantumCircuit, a user defined quantum circuit, must contain parametrised quantum gates  
 samples: int, number of samples we want to draw for the parameters.  
 """  
 self.\_circ = circ   
 self.\_num\_qubits = circ.num\_qubits  
 self.\_samples = samples  
 self.\_num\_params = circ.num\_parameters  
 # -\*- coding: utf-8 -\*-  
  
  
import time  
import ctypes  
import pickle  
import numpy as np  
from functools import partial  
from pqc import Interface  
from multiprocessing import Array, Pool  
  
  
class Multiprocess(Interface):  
   
 def \_\_init\_\_(self,circ=None,samples:int=10,num\_proc:int=1):  
 Interface.\_\_init\_\_(self,circ,samples)  
 self.\_num\_proc = num\_proc  
   
  
 def init(self,arr1\_base,arr2\_base):  
 global arr1, arr2  
 arr1 = np.ctypeslib.as\_array(arr1\_base.get\_obj())  
 arr1 = arr1.view(np.complex128).reshape(self.\_samples, 2\*\*self.\_num\_qubits)  
 arr2 = np.ctypeslib.as\_array(arr2\_base.get\_obj())  
 arr2 = arr2.view(np.complex128).reshape(self.\_samples, 2\*\*self.\_num\_qubits)  
  
 # Parallel processing  
 def target(self,func,i,p1,p2):  
 """  
 p1: theta  
 p2: phi  
 """  
 out1 = func(p1)  
 out2 = func(p2)  
 arr1[i] = out1  
 arr2[i] = out2  
   
   
   
 def job(self,func,iterable):  
 arr1\_base = Array(ctypes.c\_double, self.\_samples\*2\*\*self.\_num\_qubits\*2)  
 arr2\_base = Array(ctypes.c\_double, self.\_samples\*2\*\*self.\_num\_qubits\*2)  
 pool = Pool(processes=self.\_num\_proc, initializer=self.init, initargs=(arr1\_base,arr2\_base,))  
 pool.starmap(partial(self.target,func), iterable)  
 pool.close()   
 pool.join()  
 arr1 = np.ctypeslib.as\_array(arr1\_base.get\_obj())  
 arr1 = arr1.view(np.complex128).reshape(self.\_samples, 2\*\*self.\_num\_qubits)  
 arr2 = np.ctypeslib.as\_array(arr2\_base.get\_obj())  
 arr2 = arr2.view(np.complex128).reshape(self.\_samples, 2\*\*self.\_num\_qubits)  
   
 return arr1, arr2  
   
   
  
import numpy as np  
from qiskit import QuantumCircuit  
from qiskit.circuit import ParameterVector  
  
  
class FeatureMap:  
   
 def \_\_init\_\_(self,qubits:int=4):  
 self.qubits = qubits  
   
   
 def circuit1(self):  
   
 qc = QuantumCircuit(self.qubits)  
 # create all parameters inside the qc   
 x = ParameterVector(r'$\theta$', length=8)  
 # add parametrised gates  
 [qc.rx(x[int(2\*i)], i) for i in range(self.qubits)]  
 # add parametrised gates  
 [qc.rz(x[int(2\*i+1)], i) for i in range(self.qubits)]  
   
   
 return qc  
   
 def circuit2(self):  
 qc = QuantumCircuit(self.qubits)  
 # create all parameters inside the qc   
 x = ParameterVector(r'$\theta$', length=8)  
 # add parametrised gates  
 [qc.rx(x[int(2\*i)], i) for i in range(self.qubits)]  
 # add parametrised gates  
 [qc.rz(x[int(2\*i+1)], i) for i in range(self.qubits)]  
 qc.cx(range(1, self.qubits), range(self.qubits-1))  
   
 return qc  
   
 def circuit3(self):  
 qc = QuantumCircuit(self.qubits)  
 # create all parameters inside the qc   
 x = ParameterVector(r'$\theta$', length=11)  
 # add parametrised gates  
 [qc.rx(x[int(2\*i)], i) for i in range(self.qubits)]  
 # add parametrised gates  
 [qc.rz(x[int(2\*i+1)], i) for i in range(self.qubits)]  
 [qc.crz(x[8+i],i+1,i)for i in range(self.qubits-1)]  
   
 return qc  
   
 def circuit4(self):  
   
 qc = QuantumCircuit(self.qubits)  
 # create all parameters inside the qc   
 x = ParameterVector(r'$\theta$', length=11)  
 # add parametrised gates  
 [qc.rx(x[int(2\*i)], i) for i in range(self.qubits)]  
 # add parametrised gates  
 [qc.rz(x[int(2\*i+1)], i) for i in range(self.qubits)]  
 [qc.crx(x[8+i],i+1,i)for i in range(self.qubits-1)]  
   
   
 return qc  
   
 def circuit5(self):  
   
 qc = QuantumCircuit(self.qubits)  
 # create all parameters inside the qc   
 x = ParameterVector(r'$\theta$', length=11)  
 # add parametrised gates  
 [qc.rx(x[int(2\*i)], i) for i in range(self.qubits)]  
 # add parametrised gates  
 [qc.rz(x[int(2\*i+1)], i) for i in range(self.qubits)]  
 [qc.cry(x[8+i],i+1,i)for i in range(self.qubits-1)]  
   
 return qc  
   
   
 def circuit6(self):  
   
 qc = QuantumCircuit(self.qubits)  
 # create all parameters inside the qc   
 x = ParameterVector(r'$\theta$', length=8)  
 # add parametrised gates  
 [qc.ry(x[int(2\*i)], i) for i in range(self.qubits)]  
 # add parametrised gates  
 [qc.rz(x[int(2\*i+1)], i) for i in range(self.qubits)]  
 qc.cx(range(1, self.qubits), range(self.qubits-1))  
   
 return qc  
   
   
 def circuit7(self):  
   
 qc = QuantumCircuit(self.qubits)  
 # create all parameters inside the qc   
 x = ParameterVector(r'$\theta$', length=11)  
 # add parametrised gates  
 [qc.ry(x[int(2\*i)], i) for i in range(self.qubits)]  
 # add parametrised gates  
 [qc.rz(x[int(2\*i+1)], i) for i in range(self.qubits)]  
 [qc.crx(x[8+i],0,i+1)for i in range(self.qubits-1)]  
   
 return qc  
   
   
 def circuit8(self):  
   
 qc = QuantumCircuit(self.qubits)  
 # create all parameters inside the qc   
 x = ParameterVector(r'$\theta$', length=11)  
 # add parametrised gates  
 [qc.ry(x[int(2\*i)], i) for i in range(self.qubits)]  
 # add parametrised gates  
 [qc.rz(x[int(2\*i+1)], i) for i in range(self.qubits)]  
 [qc.crz(x[8+i],i+1,i)for i in range(self.qubits-1)]  
   
 return qc  
   
   
 def circuit9(self):  
   
 qc = QuantumCircuit(self.qubits)  
 # create all parameters inside the qc   
 x = ParameterVector(r'$\theta$', length=11)  
 # add hadamard gate to each qubit  
 [qc.h(i) for i in range(self.qubits)]   
 # add parametrised gates  
 [qc.ry(x[int(2\*i)], i) for i in range(self.qubits)]  
 # add parametrised gates  
 [qc.rz(x[int(2\*i+1)], i) for i in range(self.qubits)]  
 [qc.crz(x[8+i],i+1,i)for i in range(self.qubits-1)]  
   
 return qc  
   
   
 def circuit10(self):  
   
 qc = QuantumCircuit(self.qubits)  
 # create all parameters inside the qc   
 x = ParameterVector(r'$\theta$', length=11)  
 # add parametrised gates  
 [qc.ry(x[int(2\*i)], i) for i in range(self.qubits)]  
 # add parametrised gates  
 [qc.rz(x[int(2\*i+1)], i) for i in range(self.qubits)]  
   
 [qc.crz(x[8+i],i+1,i)for i in range(self.qubits-1)]  
   
 return qc  
   
 def circuit11(self):  
   
 qc = QuantumCircuit(self.qubits)  
 # create all parameters inside the qc   
 x = ParameterVector(r'$\theta$', length=8)  
 # add hadamard gate to each qubit  
 [qc.h(i) for i in range(self.qubits)]   
 [qc.cz(i+1,i)for i in range(self.qubits-1)]  
 # add parametrised gates  
 [qc.rx(x[int(2\*i)], i) for i in range(self.qubits)]  
 # add parametrised gates  
 [qc.rz(x[int(2\*i+1)], i) for i in range(self.qubits)]  
   
 return qc  
   
   
   
   
   
 # -\*- coding: utf-8 -\*-  
  
  
import qiskit  
import itertools  
import typing  
import numpy as np  
from qiskit import QuantumCircuit  
from qiskit.circuit import ParameterVector  
from qiskit.quantum\_info import state\_fidelity  
from pqc import Simulation  
from .descriptors.expressibility import Expressibility  
from .descriptors.entanglement import Entanglement\_Capability  
  
  
class Analyser(Simulation,Expressibility,Entanglement\_Capability):  
   
 def \_\_init\_\_(self,descriptor:str='both',  
 circ=None,samples:int=10,method\_ex:str='kl',  
 method\_ec:str='mw',num\_proc:int=1):  
   
 Simulation.\_\_init\_\_(self,circ,samples,num\_proc)  
 self.theta\_circuits, self.phi\_circuits = self.get\_circuits()   
 if descriptor == "ex":  
 Expressibility.\_\_init\_\_(self,circ,samples,method\_ex)  
 elif descriptor == "ec":  
 Entanglement\_Capability.\_\_init\_\_(self,circ,samples,method\_ec)  
 elif descriptor == "both":  
 Expressibility.\_\_init\_\_(self,circ,samples,method\_ex)  
 Entanglement\_Capability.\_\_init\_\_(self,circ,samples,method\_ec)  
   
 else:  
 raise ValueError(  
 "Invalid descriptor, choose from 'ex', 'ec' or 'both'."  
 )  
   
 def fidelity(self, shots: int = 1024) -> np.ndarray:  
 """Return fidelities for PQC  
 :param shots: number of shots for circuit execution  
 :returns fidelities (np.array): np.array of fidelities  
 """  
 theta\_circuits, phi\_circuits = self.get\_circuits()  
 fidelity = np.array(  
 [state\_fidelity(a,b) for a,b in zip(theta\_circuits,phi\_circuits)]  
 )  
   
 return np.asarray(fidelity)  
   
   
 def get\_entanglement(self):  
 num\_qubits = self.\_num\_qubits  
 theta\_circuits = self.theta\_circuits  
 phi\_circuits = self.phi\_circuits  
   
 method = self.\_method\_ec  
 if method=='mw':  
   
 circ\_entanglement\_capability = self.meyer\_wallach\_measure(  
 theta\_circuits + phi\_circuits, num\_qubits  
 ) / (2 \* self.\_samples)  
   
 elif method=='scott':  
   
 circ\_entanglement\_capability = self.scott\_measure(  
 theta\_circuits + phi\_circuits, num\_qubits  
 ) / (2 \* self.\_samples)  
   
 else:  
 raise ValueError("Invalid measure provided, choose from 'mw' or 'scott'")  
   
 return circ\_entanglement\_capability  
   
 def get\_expressibility(self, shots: int = 1024):  
 samples = self.\_samples  
 measure = self.\_method\_ex  
 haar = self.prob\_haar()  
 haar\_prob: np.ndarray = haar / float(haar.sum())  
  
 if len(self.\_circ.parameters) > 0:  
 fidelity = self.fidelity(shots)  
 else:  
 fidelity = np.ones(self.\_samples \*\* 2)  
  
 bin\_edges: np.ndarray  
 pqc\_hist, bin\_edges = np.histogram(  
 fidelity, samples, range=(0, 1), density=True  
 )  
 pqc\_prob: np.ndarray = pqc\_hist / float(pqc\_hist.sum())  
  
 if measure == "kl":  
 pqc\_expressibility = self.kl\_divergence(pqc\_prob, haar\_prob)  
 elif measure == "js":  
 pqc\_expressibility = self.jensenshannon(pqc\_prob, haar\_prob)  
 else:  
 raise ValueError("Invalid measure provided, choose from 'kl' or 'js'")  
   
  
 return pqc\_expressibility  
   
   
   
if \_\_name\_\_=="\_\_main\_\_":  
   
 Num = 4  
 qc = QuantumCircuit(Num)  
 x = ParameterVector(r'$\theta$', length=8)  
 [qc.h(i) for i in range(Num)]   
 [qc.ry(x[int(2\*i)], i) for i in range(Num)]  
 [qc.rz(x[int(2\*i+1)], i) for i in range(Num)]  
 qc.cx(0, range(1, Num))  
 circ = qc   
 out = Analyser(circ=qc,samples=1000,num\_proc=4)  
 print(out.\_\_dict\_\_.keys())  
 print("exp: ",out.get\_expressibility())  
 print("ent: ",out.get\_entanglement())  
   
   
# -\*- coding: utf-8 -\*-  
  
import qiskit  
import itertools  
import typing  
import numpy as np  
from pqc import Interface  
from qiskit.quantum\_info import state\_fidelity  
from scipy.spatial import distance  
  
  
class Expressibility(Interface):  
   
 def \_\_init\_\_(self,circ=None,samples:int=1000,method\_ex:str='kl'):  
 Interface.\_\_init\_\_(self,circ,samples)  
 self.\_method\_ex = method\_ex  
 #print("Entering expressibility init ...")  
 #print("method ex: ",method\_ex)  
  
  
 @staticmethod  
 def kl\_divergence(prob\_a: np.ndarray, prob\_b: np.ndarray) -> float:  
 """Returns KL divergence between two probabilities"""  
 prob\_a[prob\_a == 0] = 1e-10  
 kl\_div = np.sum(np.where(prob\_a != 0, prob\_a \* np.log(prob\_a / prob\_b), 0))  
 return typing.cast(float, kl\_div)  
   
 @staticmethod  
 def js\_distance(prob\_a: np.ndarray, prob\_b: np.ndarray,base: float=2.)-> float:  
 return distance.jessenshannon(prob\_a,prob\_b,base)  
   
 def prob\_haar(self) -> np.ndarray:  
 """Returns probability density function of fidelities for Haar Random States"""  
 fidelity = np.linspace(0, 1, self.\_samples)  
 num\_qubits = self.\_num\_qubits  
   
 return (2 \*\* num\_qubits - 1) \* (1 - fidelity + 1e-8) \*\* (2 \*\* num\_qubits - 2)  
  
   
# -\*- coding: utf-8 -\*-  
  
import qiskit  
import itertools  
import numpy as np  
from pqc import Interface  
from qiskit.quantum\_info import partial\_trace  
  
class Entanglement\_Capability(Interface):  
   
 def \_\_init\_\_(self,circ=None,samples:int=1000,method\_ec:str='mw'):  
 Interface.\_\_init\_\_(self,circ,samples)  
 self.\_method\_ec = method\_ec  
 #print("Entering entanglement init ...")  
 #print("method ec: ",method\_ec)  
  
 @staticmethod  
 def scott\_helper(state, perms):  
   
 dems = np.linalg.matrix\_power(  
 [partial\_trace(state, list(qb)).data for qb in perms], 2  
 )  
 trace = np.trace(dems, axis1=1, axis2=2)  
 return np.sum(trace).real  
  
 def meyer\_wallach\_measure(self,states, num\_qubits):  
 r"""Returns the meyer-wallach entanglement measure for the given circuit.  
 .. math::  
 Q = \frac{2}{|\vec{\theta}|}\sum\_{\theta\_{i}\in \vec{\theta}}  
 \Bigg(1-\frac{1}{n}\sum\_{k=1}^{n}Tr(\rho\_{k}^{2}(\theta\_{i}))\Bigg)  
 """  
 num\_qubits = self.\_num\_qubits  
 permutations = list(itertools.combinations(range(num\_qubits), num\_qubits - 1))  
 ns = 2 \* sum(  
 [  
 1 - 1 / num\_qubits \* self.scott\_helper(state, permutations)  
 for state in states  
 ]  
 )  
   
 return ns.real  
   
 def scott\_measure(self,states, num\_qubits):  
 r"""Returns the scott entanglement measure for the given circuit.  
 .. math::  
 Q\_{m} = \frac{2^{m}}{(2^{m}-1) |\vec{\theta}|}\sum\_{\theta\_i \in \vec{\theta}}\  
 \bigg(1 - \frac{m! (n-m)!)}{n!}\sum\_{|S|=m} \text{Tr} (\rho\_{S}^2 (\theta\_i)) \bigg)\  
 \quad m= 1, \ldots, \lfloor n/2 \rfloor  
 """  
 num\_qubits = self.\_num\_qubits  
 m = range(1, num\_qubits // 2 + 1)  
 permutations = [  
 list(itertools.combinations(range(num\_qubits), num\_qubits - idx))  
 for idx in m  
 ]  
 combinations = [1 / comb(num\_qubits, idx) for idx in m]  
 contributions = [2 \*\* idx / (2 \*\* idx - 1) for idx in m]  
 ns = []  
  
 for ind, perm in enumerate(permutations):  
 ns.append(  
 contributions[ind]  
 \* sum(  
 [  
 1 - combinations[ind] \* self.scott\_helper(state, perm)  
 for state in states  
 ]  
 )  
 )  
  
 return np.array(ns)  
   
   
   
  
  
"""The qLEET Package for visualizing quantum circuit behavior"""  
  
import qleet.examples  
import qleet.analyzers  
import qleet.simulators  
import qleet.interface  
  
from qleet.interface.metas import AnalyzerList  
from qleet.interface.circuit import CircuitDescriptor  
  
from qleet.analyzers.training\_path import OptimizationPathPlotter  
from qleet.analyzers.loss\_landscape import LossLandscapePlotter  
from qleet.analyzers.expressibility import Expressibility  
from qleet.analyzers.entanglement import EntanglementCapability  
  
from qleet.simulators.circuit\_simulators import CircuitSimulator  
from qleet.simulators.pqc\_trainer import PQCSimulatedTrainer  
  
from qleet.examples.qaoa\_maxcut import QAOACircuitMaxCut  
import typing  
  
import numpy as np  
from sklearn.manifold import TSNE  
from sklearn.decomposition import PCA  
import plotly.express as px  
import plotly.graph\_objects as pg  
  
from .loss\_landscape import LossLandscapePlotter  
from ..interface.metas import MetaLogger  
  
  
class OptimizationPathPlotter(MetaLogger):  
 def \_\_init\_\_(self, mode="tSNE"):  
 super().\_\_init\_\_()  
 assert mode in [  
 "tSNE",  
 "PCA",  
 ], "Mode of Dimensionality Reduction is not implemented, use PCA or tSNE."  
 self.dimensionality\_reduction = TSNE if mode == "tSNE" else PCA  
  
 def log(self, solver, \_loss):  
 self.data.append(solver.model.trainable\_variables[0].numpy())  
 self.runs.append(self.trial)  
 self.item.append(self.counter)  
 self.counter += 1  
  
 def plot(self):  
 raw\_params = np.stack(self.data)  
 final\_params = self.dimensionality\_reduction(n\_components=2).fit\_transform(  
 raw\_params  
 )  
 max\_number\_of\_runs = max(self.item)  
 size\_values = [5 if size > max\_number\_of\_runs - 5 else 1 for size in self.item]  
 fig = px.scatter(  
 x=final\_params[:, 0],  
 y=final\_params[:, 1],  
 color=self.runs,  
 size=size\_values,  
 )  
 return fig  
  
  
class LossLandscapePathPlotter(MetaLogger):  
 def \_\_init\_\_(self, base\_plotter: LossLandscapePlotter):  
 super().\_\_init\_\_()  
 self.loss: typing.List[float] = []  
 self.plotter = base\_plotter  
  
 def log(self, solver, loss: float):  
 self.data.append(  
 self.plotter.axes @ solver.model.trainable\_variables[0].numpy()  
 )  
 self.loss.append(loss)  
 self.runs.append(self.trial)  
 self.item.append(self.counter)  
 self.counter += 1  
  
 def plot(self):  
 data = np.array(self.data)  
 loss = np.array(self.loss)  
 max\_number\_of\_runs = max(self.item)  
 size\_values = np.array(  
 [12 if size > max\_number\_of\_runs - 5 else 5 for size in self.item]  
 )  
 fig = pg.Figure(  
 data=[  
 pg.Scatter3d(  
 x=data[:, 0],  
 y=data[:, 1],  
 z=-loss,  
 mode="markers",  
 marker=dict(color=self.runs, size=size\_values),  
 )  
 ]  
 )  
 return fig  
"""Module to test the expressibility of circuits."""  
  
import itertools  
import typing  
  
from qiskit.providers.aer.noise import NoiseModel as qiskitNoiseModel  
from cirq.devices.noise\_model import NoiseModel as cirqNoiseModel  
from pyquil.noise import NoiseModel as pyquilNoiseModel  
  
from qiskit.quantum\_info import state\_fidelity  
from scipy.spatial.distance import jensenshannon  
  
import matplotlib.pyplot as plt  
import numpy as np  
  
from ..interface.metas import MetaExplorer  
from ..interface.circuit import CircuitDescriptor  
from ..simulators.circuit\_simulators import CircuitSimulator  
  
NOISE\_MODELS = {  
 "cirq": cirqNoiseModel,  
 "pyquil": pyquilNoiseModel,  
 "qiskit": qiskitNoiseModel,  
}  
  
  
class Expressibility(MetaExplorer):  
 """Calculates expressibility of a parameterized quantum circuit"""  
  
 def \_\_init\_\_(  
 self,  
 circuit: CircuitDescriptor,  
 noise\_model: typing.Union[  
 cirqNoiseModel, qiskitNoiseModel, pyquilNoiseModel, None  
 ] = None,  
 samples: int = 1000,  
 ):  
 """Constructor the the Expressibility analyzer  
  
 :param circuit: input circuit as a CircuitDescriptor object  
 :param noise\_model: (dict, NoiseModel) initialization noise-model dictionary  
 :param samples: number of samples for the experiment  
 :raises ValueError: If circuit and noise model does not correspond to same framework  
 """  
 super().\_\_init\_\_()  
 self.circuit = circuit  
  
 if noise\_model is not None:  
 if isinstance(noise\_model, NOISE\_MODELS[circuit.default\_backend]):  
 self.noise\_model = noise\_model  
 else:  
 raise ValueError(  
 f"Circuit and noise model must correspond to the same \  
 framework but circuit:{circuit.default\_backend} and \  
 noise\_model:{type(noise\_model)} were provided."  
 )  
 else:  
 self.noise\_model = None  
  
 self.num\_samples = samples  
 self.expr = 0.0  
 self.plot\_data: typing.List[np.ndarray] = []  
  
 @staticmethod  
 def kl\_divergence(prob\_a: np.ndarray, prob\_b: np.ndarray) -> float:  
 """Returns KL divergence between two probabilities"""  
 prob\_a[prob\_a == 0] = 1e-10  
 kl\_div = np.sum(np.where(prob\_a != 0, prob\_a \* np.log(prob\_a / prob\_b), 0))  
 return typing.cast(float, kl\_div)  
  
 def gen\_params(self) -> typing.Tuple[typing.List, typing.List]:  
 """Generate parameters for the calculation of expressibility  
  
 :returns theta (np.array): first list of parameters for the parameterized quantum circuit  
 :returns phi (np.array): second list of parameters for the parameterized quantum circuit  
 """  
 theta = [  
 {p: 2 \* np.random.random() \* np.pi for p in self.circuit.parameters}  
 for \_ in range(self.num\_samples)  
 ]  
 phi = [  
 {p: 2 \* np.random.random() \* np.pi for p in self.circuit.parameters}  
 for \_ in range(self.num\_samples)  
 ]  
 return theta, phi  
  
 def prob\_haar(self) -> np.ndarray:  
 """Returns probability density function of fidelities for Haar Random States"""  
 fidelity = np.linspace(0, 1, self.num\_samples)  
 num\_qubits = self.circuit.num\_qubits  
 return (2 \*\* num\_qubits - 1) \* (1 - fidelity + 1e-8) \*\* (2 \*\* num\_qubits - 2)  
  
 def prob\_pqc(self, shots: int = 1024) -> np.ndarray:  
 """Return probability density function of fidelities for PQC  
  
 :param shots: number of shots for circuit execution  
 :returns fidelities (np.array): np.array of fidelities  
 """  
 thetas, phis = self.gen\_params()  
  
 theta\_circuits = [  
 CircuitSimulator(self.circuit, self.noise\_model).simulate(theta, shots)  
 for theta in thetas  
 ]  
 phi\_circuits = [  
 CircuitSimulator(self.circuit, self.noise\_model).simulate(phi, shots)  
 for phi in phis  
 ]  
 fidelity = np.array(  
 [  
 state\_fidelity(rho\_a, rho\_b)  
 for rho\_a, rho\_b in itertools.product(theta\_circuits, phi\_circuits)  
 ]  
 )  
 return np.array(fidelity)  
  
 def expressibility(self, measure: str = "kld", shots: int = 1024) -> float:  
 r"""Returns expressibility for the circuit  
  
 .. math::  
 Expr = D\_{KL}(\hat{P}\_{PQC}(F; \theta) | P\_{Haar}(F))\\  
 Expr = D\_{\sqrt{JSD}}(\hat{P}\_{PQC}(F; \theta) | P\_{Haar}(F))  
  
 :param measure: specification for the measure used in the expressibility calculation  
 :param shots: number of shots for circuit execution  
 :returns pqc\_expressibility: float, expressibility value  
 :raises ValueError: if invalid measure is specified  
 """  
 haar = self.prob\_haar()  
 haar\_prob: np.ndarray = haar / float(haar.sum())  
  
 if len(self.circuit.parameters) > 0:  
 fidelity = self.prob\_pqc(shots)  
 else:  
 fidelity = np.ones(self.num\_samples \*\* 2)  
  
 bin\_edges: np.ndarray  
 pqc\_hist, bin\_edges = np.histogram(  
 fidelity, self.num\_samples, range=(0, 1), density=True  
 )  
 pqc\_prob: np.ndarray = pqc\_hist / float(pqc\_hist.sum())  
  
 if measure == "kld":  
 pqc\_expressibility = self.kl\_divergence(pqc\_prob, haar\_prob)  
 elif measure == "jsd":  
 pqc\_expressibility = jensenshannon(pqc\_prob, haar\_prob, 2.0)  
 else:  
 raise ValueError("Invalid measure provided, choose from 'kld' or 'jsd'")  
 self.plot\_data = [haar\_prob, pqc\_prob, bin\_edges]  
 self.expr = pqc\_expressibility  
  
 return pqc\_expressibility  
  
 def plot(self, figsize=(6, 4), dpi=300, \*\*kwargs):  
 """Returns plot for expressibility visualization"""  
 if self.plot\_data is None:  
 raise "Perform expressibility calculation first"  
  
 haar\_prob, pqc\_prob, bin\_edges = self.plot\_data  
 expr = self.expr  
  
 bin\_middles = (bin\_edges[1:] + bin\_edges[:-1]) / 2.0  
 bin\_width = bin\_edges[1] - bin\_edges[0]  
  
 fig = plt.figure(figsize=figsize, dpi=dpi, \*\*kwargs)  
 plt.bar(bin\_middles, haar\_prob, width=bin\_width, label="Haar")  
 plt.bar(bin\_middles, pqc\_prob, width=bin\_width, label="PQC", alpha=0.6)  
 plt.xlim((-0.05, 1.05))  
 plt.ylim(bottom=0.0, top=max(max(pqc\_prob), max(haar\_prob)) + 0.01)  
 plt.grid(True)  
 plt.title(f"Expressibility: {np.round(expr,5)}")  
 plt.xlabel("Fidelity")  
 plt.ylabel("Probability")  
 plt.legend()  
  
 return fig  
"""Module to test the achievable entanglement in circuits."""  
  
import itertools  
import typing  
  
from qiskit.providers.aer.noise import NoiseModel as qiskitNoiseModel  
from cirq.devices.noise\_model import NoiseModel as cirqNoiseModel  
from pyquil.noise import NoiseModel as pyquilNoiseModel  
  
from qiskit.quantum\_info import partial\_trace  
from scipy.special import comb  
  
import numpy as np  
  
from ..interface.metas import MetaExplorer  
from ..interface.circuit import CircuitDescriptor  
from ..simulators.circuit\_simulators import CircuitSimulator  
  
NOISE\_MODELS = {  
 "cirq": cirqNoiseModel,  
 "pyquil": pyquilNoiseModel,  
 "qiskit": qiskitNoiseModel,  
}  
  
  
class EntanglementCapability(MetaExplorer):  
 """Calculates entangling capability of a parameterized quantum circuit"""  
  
 def \_\_init\_\_(  
 self,  
 circuit: CircuitDescriptor,  
 noise\_model: typing.Union[  
 cirqNoiseModel, qiskitNoiseModel, pyquilNoiseModel, None  
 ] = None,  
 samples: int = 1000,  
 ):  
 """Constructor for entanglement capability plotter  
  
 :param circuit: input circuit as a CircuitDescriptor object  
 :param noise\_model: (dict, NoiseModel) initialization noise-model dictionary for  
 generating noise model  
 :param samples: number of samples for the experiment  
 :returns Entanglement object instance  
 :raises ValueError: If circuit and noise model does not correspond to same framework  
 """  
 super().\_\_init\_\_()  
 self.circuit = circuit  
  
 if noise\_model is not None:  
 if isinstance(noise\_model, NOISE\_MODELS[circuit.default\_backend]):  
 self.noise\_model = noise\_model  
 else:  
 raise ValueError(  
 f"Circuit and noise model must correspond to the same \  
 framework but circuit:{circuit.default\_backend} and \  
 noise\_model:{type(noise\_model)} were provided."  
 )  
 else:  
 self.noise\_model = None  
  
 self.num\_samples = samples  
  
 def gen\_params(self) -> typing.Tuple[typing.List, typing.List]:  
 """Generate parameters for the calculation of expressibility  
  
 :return theta (np.array): first list of parameters for the parameterized quantum circuit  
 :return phi (np.array): second list of parameters for the parameterized quantum circuit  
 """  
 theta = [  
 {p: 2 \* np.random.random() \* np.pi for p in self.circuit.parameters}  
 for \_ in range(self.num\_samples)  
 ]  
 phi = [  
 {p: 2 \* np.random.random() \* np.pi for p in self.circuit.parameters}  
 for \_ in range(self.num\_samples)  
 ]  
 return theta, phi  
  
 @staticmethod  
 def scott\_helper(state, perms):  
 """Helper function for entanglement measure. It gives trace of the output state"""  
 dems = np.linalg.matrix\_power(  
 [partial\_trace(state, list(qb)).data for qb in perms], 2  
 )  
 trace = np.trace(dems, axis1=1, axis2=2)  
 return np.sum(trace).real  
  
 def meyer\_wallach\_measure(self, states, num\_qubits):  
 r"""Returns the meyer-wallach entanglement measure for the given circuit.  
  
 .. math::  
 Q = \frac{2}{|\vec{\theta}|}\sum\_{\theta\_{i}\in \vec{\theta}}  
 \Bigg(1-\frac{1}{n}\sum\_{k=1}^{n}Tr(\rho\_{k}^{2}(\theta\_{i}))\Bigg)  
  
 """  
 permutations = list(itertools.combinations(range(num\_qubits), num\_qubits - 1))  
 ns = 2 \* sum(  
 [  
 1 - 1 / num\_qubits \* self.scott\_helper(state, permutations)  
 for state in states  
 ]  
 )  
 return ns.real  
  
 def scott\_measure(self, states, num\_qubits):  
 r"""Returns the scott entanglement measure for the given circuit.  
  
 .. math::  
 Q\_{m} = \frac{2^{m}}{(2^{m}-1) |\vec{\theta}|}\sum\_{\theta\_i \in \vec{\theta}}\  
 \bigg(1 - \frac{m! (n-m)!)}{n!}\sum\_{|S|=m} \text{Tr} (\rho\_{S}^2 (\theta\_i)) \bigg)\  
 \quad m= 1, \ldots, \lfloor n/2 \rfloor  
  
 """  
 m = range(1, num\_qubits // 2 + 1)  
 permutations = [  
 list(itertools.combinations(range(num\_qubits), num\_qubits - idx))  
 for idx in m  
 ]  
 combinations = [1 / comb(num\_qubits, idx) for idx in m]  
 contributions = [2 \*\* idx / (2 \*\* idx - 1) for idx in m]  
 ns = []  
  
 for ind, perm in enumerate(permutations):  
 ns.append(  
 contributions[ind]  
 \* sum(  
 [  
 1 - combinations[ind] \* self.scott\_helper(state, perm)  
 for state in states  
 ]  
 )  
 )  
  
 return np.array(ns)  
  
 def entanglement\_capability(  
 self, measure: str = "meyer-wallach", shots: int = 1024  
 ) -> float:  
 """Returns entanglement measure for the given circuit  
  
 :param measure: specification for the measure used in the entangling capability  
 :param shots: number of shots for circuit execution  
 :returns pqc\_entangling\_capability (float): entanglement measure value  
 :raises ValueError: if invalid measure is specified  
 """  
 thetas, phis = self.gen\_params()  
  
 theta\_circuits = [  
 CircuitSimulator(self.circuit, self.noise\_model).simulate(theta, shots)  
 for theta in thetas  
 ]  
 phi\_circuits = [  
 CircuitSimulator(self.circuit, self.noise\_model).simulate(phi, shots)  
 for phi in phis  
 ]  
  
 num\_qubits = self.circuit.num\_qubits  
  
 if measure == "meyer-wallach":  
 pqc\_entanglement\_capability = self.meyer\_wallach\_measure(  
 theta\_circuits + phi\_circuits, num\_qubits  
 ) / (2 \* self.num\_samples)  
 elif measure == "scott":  
 pqc\_entanglement\_capability = self.scott\_measure(  
 theta\_circuits + phi\_circuits, num\_qubits  
 ) / (2 \* self.num\_samples)  
 else:  
 raise ValueError(  
 "Invalid measure provided, choose from 'meyer-wallach' or 'scott'"  
 )  
  
 return pqc\_entanglement\_capability  
import numpy as np  
import tqdm.auto as tqdm  
import plotly.graph\_objects as pg  
  
from ..simulators.pqc\_trainer import PQCSimulatedTrainer  
from ..interface.metric\_spec import MetricSpecifier  
  
  
class LossLandscapePlotter:  
 def \_\_init\_\_(self, solver: PQCSimulatedTrainer, metric: MetricSpecifier, dim=2):  
 self.n = len(solver.circuit.parameters)  
 self.metric = metric  
 self.solver = solver  
 self.dim = dim  
 self.axes = self.\_\_random\_subspace(dim=self.dim)  
  
 def \_\_random\_subspace(self, dim):  
 axes = []  
 for \_i in range(dim):  
 axis = np.random.random(self.n)  
 for other\_axis in axes:  
 projection = np.dot(axis, other\_axis)  
 axis = axis - projection \* other\_axis  
 axis = axis / np.sum(axis)  
 axes.append(axis)  
 return np.stack(axes, axis=0)  
  
 def scan(self, points, distance, origin):  
 chained\_range = [  
 np.linspace(-distance, distance, points) for \_i in range(self.dim)  
 ]  
 coords = np.meshgrid(\*chained\_range)  
 coords = np.reshape(np.stack(coords, axis=-1), (-1, self.dim))  
 values = np.zeros(len(coords), dtype=np.float)  
 with tqdm.trange(len(coords)) as iterator:  
 iterator.set\_description("Contour Plot Scan")  
 for i in iterator:  
 # TODO: Incorporate state vector and density matrix modes for higher speed  
 values[i] = self.metric.from\_circuit(  
 circuit\_descriptor=self.solver.circuit,  
 parameters=coords[i] @ self.axes + origin,  
 mode="samples",  
 )  
 return values, coords  
  
 def plot(self, mode="surface", points=25, distance=np.pi):  
 assert mode in ["line", "contour", "surface"]  
 if mode == "contour":  
 assert (  
 self.dim == 2  
 ), "Contour plots can only be drawn with 2-dimensional axes"  
 origin = self.solver.model.trainable\_variables[0]  
 data, \_coords = self.scan(points, distance, origin)  
 data = np.reshape(data, (points, points))  
 scan\_range = np.linspace(-distance, +distance, points)  
 fig = pg.Figure(data=pg.Contour(z=data, x=scan\_range, y=scan\_range))  
 return fig  
 elif mode == "surface":  
 assert (  
 self.dim == 2  
 ), "Contour plots can only be drawn with 2-dimensional axes"  
 origin = self.solver.model.trainable\_variables[0]  
 data, \_coords = self.scan(points, distance, origin)  
 data = np.reshape(data, (points, points))  
 scan\_range = np.linspace(-distance, +distance, points)  
 fig = pg.Figure(data=pg.Surface(z=data, x=scan\_range, y=scan\_range))  
 return fig  
 else:  
 raise NotImplementedError("This plotting mode has not been implemented yet")  
import typing  
  
import numpy as np  
import sympy  
  
import cirq  
import qiskit  
import pyquil  
  
from cirq.contrib.qasm\_import import circuit\_from\_qasm  
from cirq.contrib.quil\_import import circuit\_from\_quil  
import qiskit.quantum\_info  
import pyquil.paulis  
  
  
def convert\_to\_cirq(  
 circuit: typing.Union[qiskit.QuantumCircuit, cirq.Circuit, pyquil.Program]  
) -> cirq.Circuit:  
 """Converts any circuit to cirq  
 :param circuit: input circuit in any framework  
 :return: circuit in cirq  
 :raises ValueError: if the circuit is not from one of the supported frameworks  
 """  
 if isinstance(circuit, cirq.Circuit):  
 return circuit  
 elif isinstance(circuit, qiskit.QuantumCircuit):  
 return circuit\_from\_qasm(circuit.qasm())  
 elif isinstance(circuit, pyquil.Program):  
 return circuit\_from\_quil(str(circuit))  
 else:  
 raise ValueError(  
 f"Expected a circuit object in cirq, qiskit or pyquil, got {type(circuit)}"  
 )  
  
  
def convert\_to\_qiskit(  
 circuit: typing.Union[qiskit.QuantumCircuit, cirq.Circuit, pyquil.Program]  
) -> qiskit.QuantumCircuit:  
 """Converts any circuit to qiskit  
 :param circuit: input circuit in any framework  
 :raises ValueError: if the circuit is not from one of the supported frameworks  
 :return: circuit in qiskit  
 """  
 if isinstance(circuit, cirq.Circuit):  
 return qiskit.QuantumCircuit.from\_qasm\_str(circuit.to\_qasm())  
 elif isinstance(circuit, qiskit.QuantumCircuit):  
 return circuit  
 elif isinstance(circuit, pyquil.Program):  
 raise convert\_to\_qiskit(convert\_to\_cirq(circuit))  
 else:  
 raise ValueError(  
 f"Expected a circuit object in cirq, qiskit or pyquil, got {type(circuit)}"  
 )  
  
  
class CircuitDescriptor:  
 """The interface for users to provide a circuit in any framework and visualize it in qLEET."""  
  
 def \_\_init\_\_(  
 self,  
 circuit: typing.Union[qiskit.QuantumCircuit, cirq.Circuit, pyquil.Program],  
 params: typing.List[typing.Union[sympy.Symbol, qiskit.circuit.Parameter]],  
 cost\_function: typing.Union[  
 cirq.PauliSum, qiskit.quantum\_info.PauliList, pyquil.paulis.PauliSum, None  
 ] = None,  
 ):  
 self.\_circuit = circuit  
 self.\_params = params  
 self.\_cost = cost\_function  
  
 @property  
 def default\_backend(self):  
 if isinstance(self.\_circuit, cirq.Circuit):  
 return "cirq"  
 elif isinstance(self.\_circuit, qiskit.QuantumCircuit):  
 return "qiskit"  
 elif isinstance(self.\_circuit, pyquil.Program):  
 return "pyquil"  
 else:  
 raise ValueError("Unsupported framework of circuit")  
  
 @classmethod  
 def from\_qasm(cls, qasm\_str: str, params, cost\_function):  
 """Generate the descriptor from OpenQASM string  
 :param qasm\_str:OpenQASM string for each part of the circuit  
 :param params: list of sympy symbols which act as parameters  
 :param cost\_function: pauli-string operator to implement cost function  
 :return: The CircuitDescriptor object  
 """  
 cirq\_circuit = circuit\_from\_qasm(qasm\_str)  
 return CircuitDescriptor(  
 circuit=cirq\_circuit, params=params, cost\_function=cost\_function  
 )  
  
 @property  
 def parameters(  
 self,  
 ) -> typing.List[typing.Union[sympy.Symbol, qiskit.circuit.Parameter]]:  
 """The list of sympy symbols to resolve as parameters, will be swept from 0 to 2\*pi  
 :return: list of parameters  
 """  
 return self.\_params  
  
 def \_\_len\_\_(self) -> int:  
 """Number of parameters in the variational circuit  
 :return: number of parameters in the circuit  
 """  
 return len(self.parameters)  
  
 @property  
 def cirq\_circuit(self) -> cirq.Circuit:  
 """Get the circuit in cirq  
 :return: the cirq representation of the circuit  
 """  
 return convert\_to\_cirq(self.\_circuit)  
  
 @property  
 def qiskit\_circuit(self) -> qiskit.QuantumCircuit:  
 """Get the circuit in qiskit  
 :return: the cirq representation of the circuit  
 """  
 return convert\_to\_qiskit(self.\_circuit)  
  
 @property  
 def num\_qubits(self) -> int:  
 """Get the number of qubits for a circuit  
 :return: the number of qubits in the circuit  
 :raises ValueError: if unsupported circuit framework is given  
 """  
 if isinstance(self.\_circuit, cirq.Circuit):  
 return len(self.\_circuit.all\_qubits())  
 elif isinstance(self.\_circuit, qiskit.QuantumCircuit):  
 return self.\_circuit.num\_qubits  
 elif isinstance(self.\_circuit, pyquil.Program):  
 return len(self.\_circuit.get\_qubits())  
 else:  
 raise ValueError("Unsupported framework of circuit")  
  
 @property  
 def cirq\_cost(self) -> cirq.PauliSum:  
 """Returns the cost function, which is a function that takes in the state vector or the  
 density matrix and returns the loss value of the solution envisioned by the Quantum Circuit.  
 :raises ValueError: if the circuit is not from one of the supported frameworks  
 :raises NotImplementedError: Long as qiskit and pyquil ports of pauli-string aren't written  
 :return: cost function  
 TODO: Implement conversions into Cirq PauliSum  
 """  
 if isinstance(self.\_cost, cirq.PauliSum):  
 return self.\_cost  
 elif isinstance(self.\_cost, qiskit.quantum\_info.PauliList):  
 raise NotImplementedError("Qiskit PauliString support is not implemented")  
 elif isinstance(self.\_cost, pyquil.paulis.PauliSum):  
 raise NotImplementedError("PyQuil PauliString support is not implemented")  
 else:  
 raise ValueError("Cost object should be a Pauli-Sum object")  
  
 def \_\_eq\_\_(self, other: typing.Any) -> bool:  
 if isinstance(other, CircuitDescriptor):  
 return (  
 np.array\_equal(self.parameters, other.parameters)  
 and self.cirq\_circuit == other.cirq\_circuit  
 )  
 else:  
 return False  
  
 def \_\_repr\_\_(self) -> str:  
 return f"qleet.CircuitDescriptor({repr(self.\_circuit)}, {repr(self.\_params)})"  
  
 def \_\_str\_\_(self) -> str:  
 return f"qleet.CircuitDescriptor({repr(self.\_circuit)})"  
import abc  
  
import numpy as np  
import tensorflow\_quantum as tfq  
  
from ..interface.circuit import CircuitDescriptor  
  
  
class MetricSpecifier(abc.ABC):  
 def \_\_init\_\_(self, default\_call\_mode="samples"):  
 self.\_\_mode\_to\_function\_map = {  
 "samples": self.from\_samples\_vector,  
 "state\_vector": self.from\_state\_vector,  
 "density\_matrix": self.from\_density\_matrix,  
 }  
 assert default\_call\_mode in ["samples", "state\_vector", "density\_matrix"]  
 self.default\_call\_mode = default\_call\_mode  
 self.default\_call\_function = self.\_\_mode\_to\_function\_map[default\_call\_mode]  
  
 def from\_circuit(  
 self, circuit\_descriptor: CircuitDescriptor, parameters, mode="samples"  
 ):  
 if mode == "samples":  
 samples = sample\_solutions(  
 circuit=circuit\_descriptor.cirq\_circuit,  
 param\_symbols=circuit\_descriptor.parameters,  
 param\_values=parameters,  
 )  
 return self.from\_samples\_vector(samples)  
 elif mode == "state\_vector":  
 raise NotImplementedError  
 elif mode == "density\_matrix":  
 raise NotImplementedError  
 else:  
 raise ValueError(  
 "Provided mode should be one of [samples, state\_vector, density\_matrix]"  
 )  
  
 @abc.abstractmethod  
 def from\_state\_vector(self, state\_vector: np.ndarray) -> float:  
 raise NotImplementedError  
  
 @abc.abstractmethod  
 def from\_density\_matrix(self, density\_matrix: np.ndarray) -> float:  
 raise NotImplementedError  
  
 @abc.abstractmethod  
 def from\_samples\_vector(self, samples\_vector: np.ndarray) -> float:  
 raise NotImplementedError  
  
  
def sample\_solutions(circuit, param\_symbols, param\_values, samples=1000):  
 """Get the computed cuts for a given ansatz  
 :param circuit: Circuit to be sampled  
 :param param\_symbols: The symbols of model parameters (betas and gammas)  
 :param param\_values: The value of model parameters (betas and gammas) to sample at, 1-D vector  
 :param samples: Number of times to sample the resulting quantum state  
 :return: 2-D matrix, n\_samples rows of boolean vectors showing the cut  
 """  
 output = tfq.layers.Sample()(  
 circuit,  
 symbol\_names=param\_symbols,  
 symbol\_values=[param\_values],  
 repetitions=samples,  
 )  
 return output.numpy()[0]  
import typing  
from abc import abstractmethod, ABC  
  
if typing.TYPE\_CHECKING:  
 from ..simulators.pqc\_trainer import PQCSimulatedTrainer  
  
  
class MetaLogger(ABC):  
 def \_\_init\_\_(self):  
 self.trial, self.counter = 0, 0  
 self.data = []  
 self.runs = []  
 self.item = []  
  
 @abstractmethod  
 def log(self, solver, loss):  
 raise NotImplementedError  
  
 @abstractmethod  
 def plot(self):  
 raise NotImplementedError  
  
 def next(self):  
 self.trial += 1  
 self.counter = 0  
  
  
class MetaExplorer(ABC):  
 def \_\_init\_\_(self):  
 pass  
  
  
class AnalyzerList:  
 def \_\_init\_\_(self, \*args: typing.Union[MetaLogger, MetaExplorer]):  
 self.\_analyzers: typing.Tuple[  
 typing.Union[MetaLogger, MetaExplorer]  
 ] = typing.cast(typing.Tuple[typing.Union[MetaLogger, MetaExplorer]], args)  
  
 def \_\_str\_\_(self) -> str:  
 return "\n".join(str(self.\_analyzers))  
  
 def log(self, solver: "PQCSimulatedTrainer", loss: float) -> None:  
 for analyzer in self.\_analyzers:  
 if isinstance(analyzer, MetaLogger):  
 analyzer.log(solver, loss)  
  
 def next(self) -> None:  
 for analyzer in self.\_analyzers:  
 if isinstance(analyzer, MetaLogger):  
 analyzer.next()  
  
 def \_\_getitem\_\_(self, item: int) -> typing.Union[MetaLogger, MetaExplorer]:  
 return self.\_analyzers[item]  
  
 def \_\_iter\_\_(self) -> typing.Iterable[typing.Union[MetaLogger, MetaExplorer]]:  
 return self.\_analyzers  
import networkx as nx  
  
import dash  
import dash\_core\_components as dash\_core  
import dash\_html\_components as dash\_html  
  
from qleet.interface.metas import AnalyzerList  
from qleet.analyzers.loss\_landscape import LossLandscapePlotter  
from qleet.analyzers.training\_path import OptimizationPathPlotter  
from qleet.analyzers.training\_path import LossLandscapePathPlotter  
  
  
def launch\_dashboard(trainer, plottable\_metric):  
 plot = LossLandscapePlotter(trainer, plottable\_metric, dim=2)  
 trainer.train(n\_samples=50)  
 fig\_loss\_surface = plot.plot("surface", points=5)  
  
 trackers = AnalyzerList(  
 LossLandscapePathPlotter(plot),  
 OptimizationPathPlotter(mode="tSNE"),  
 )  
 for \_i in range(5):  
 trainer.train(loggers=trackers)  
 trackers.next()  
  
 # Make the app to plot all of this  
  
 external\_stylesheets = [  
 "https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css",  
 "./app.css",  
 ]  
 app = dash.Dash(\_\_name\_\_, external\_stylesheets=external\_stylesheets)  
  
 fig\_loss\_traversal = trackers[0].plot()  
 fig\_training\_trace = trackers[1].plot()  
  
 fig\_loss\_surface.layout.update(  
 height=800,  
 width=1200,  
 )  
 fig\_loss\_traversal.layout.update(  
 height=800,  
 width=1200,  
 )  
  
 app.layout = dash\_html.Div(  
 className="container",  
 id="mainApp",  
 children=[  
 dash\_html.H1(children="Variational Quantum Circuit Analyzer"),  
 dash\_html.H3(children="Visualizing QAOA Landscapes and Plotting Paths."),  
 dash\_html.H2(children="tSNE of Optimization Path"),  
 dash\_core.Graph(id="training-path", figure=fig\_training\_trace),  
 dash\_html.H2(children="Loss Landscape along random 2-D subspace"),  
 dash\_core.Graph(  
 id="loss-landscape",  
 figure=fig\_loss\_surface,  
 ),  
 dash\_html.H2(children="Traversal on the Loss Landscape above"),  
 dash\_core.Graph(  
 id="loss-traversal",  
 figure=fig\_loss\_traversal,  
 ),  
 ],  
 )  
 app.run\_server(debug=False)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 from qleet.interface.circuit import CircuitDescriptor  
 from qleet.simulators.pqc\_trainer import PQCSimulatedTrainer  
 from qleet.examples.qaoa\_maxcut import QAOACircuitMaxCut, MaxCutMetric  
  
 graph = nx.gnm\_random\_graph(n=10, m=40)  
 qaoa = QAOACircuitMaxCut(graph, p=1)  
 circuit = CircuitDescriptor(qaoa.qaoa\_circuit, qaoa.params, qaoa.qaoa\_cost)  
 solver = PQCSimulatedTrainer(circuit)  
 metric = MaxCutMetric(graph)  
 launch\_dashboard(solver, metric)  
import typing  
  
import cirq  
import numpy as np  
import qiskit  
  
from qiskit.providers.aer.noise import NoiseModel as qiskitNoiseModel  
from cirq.devices.noise\_model import NoiseModel as cirqNoiseModel  
from pyquil.noise import NoiseModel as pyquilNoiseModel  
  
from ..interface.circuit import CircuitDescriptor  
  
  
class CircuitSimulator:  
 """The interface for users to execute their CircuitDescriptor objects"""  
  
 def \_\_init\_\_(  
 self,  
 circuit: CircuitDescriptor,  
 noise\_model: typing.Union[  
 cirqNoiseModel, qiskitNoiseModel, pyquilNoiseModel, None  
 ] = None,  
 ):  
 """Initialize the state simulator  
 :param circuit: the target circuit to simulate  
 :param noise\_model: the noise model as dict or empty dict for density matrix simulations,  
 None if performing state vector simulations  
 """  
 self.circuit = circuit  
 self.noise\_model = noise\_model  
 self.\_result = None  
  
 @property  
 def result(  
 self,  
 ) -> typing.Optional[np.ndarray]:  
 """Get the results stored from the circuit simulator  
 :return: stored result of the circuit simulation if it has been performed, else None.  
 """  
 return self.\_result  
  
 def simulate(  
 self,  
 param\_resolver: typing.Dict[qiskit.circuit.Parameter, float],  
 shots: int = 1024,  
 ) -> np.ndarray:  
 """Simulate to get the state vector or the density matrix  
 :param param\_resolver: a dictionary of all the symbols/parameters mapping to their values  
 :param shots: number of times to run the qiskit density matrix simulator  
 :returns: state vector or density matrix resulting from the simulation  
 :raises NotImplementedError: if circuit simulation is not supported for a backend  
 """  
 if self.circuit.default\_backend == "qiskit":  
 circuit = self.circuit.qiskit\_circuit.bind\_parameters(param\_resolver)  
 if self.noise\_model is not None:  
 circuit.snapshot("final", snapshot\_type="density\_matrix")  
 result = qiskit.execute(  
 circuit,  
 qiskit.Aer.get\_backend("qasm\_simulator"),  
 shots=shots,  
 noise\_model=self.noise\_model,  
 backend\_options={"method": "density\_matrix"},  
 ).result()  
 result\_data = result.data(0)["snapshots"]["density\_matrix"]["final"][0][  
 "value"  
 ]  
 else:  
 circuit.snapshot("final", snapshot\_type="statevector")  
 result = qiskit.execute(  
 circuit, qiskit.Aer.get\_backend("aer\_simulator\_statevector")  
 ).result()  
 result\_data = result.data(0)["snapshots"]["statevector"]["final"][0]  
  
 elif self.circuit.default\_backend == "cirq":  
  
 circuit = self.circuit.cirq\_circuit  
 non\_unitary\_flag = False  
 for op in circuit.all\_operations():  
 op\_name = str(op).split("(")[0]  
 if op\_name in [  
 "phase\_flip",  
 "phase\_damp",  
 "amplitude\_damp",  
 "depolarize",  
 "asymmetric\_depolarize",  
 ]:  
 non\_unitary\_flag = True  
 break  
  
 if self.noise\_model is None and not non\_unitary\_flag:  
 simulator = cirq.Simulator() # type: ignore  
 result = simulator.simulate(self.circuit.cirq\_circuit, param\_resolver)  
 result\_data = result.final\_state\_vector  
 else:  
 simulator = cirq.DensityMatrixSimulator(noise=self.noise\_model) # type: ignore  
 result = simulator.simulate(self.circuit.cirq\_circuit, param\_resolver)  
 result\_data = result.final\_density\_matrix  
  
 else:  
 raise NotImplementedError(  
 "Parametrized circuit simulation is not implemented for this backend."  
 )  
  
 self.\_result = result\_data  
 return result\_data  
import typing  
  
import cirq  
import tqdm.auto as tqdm  
  
import tensorflow as tf  
import tensorflow\_quantum as tfq  
  
from ..interface.metas import AnalyzerList  
from ..interface.circuit import CircuitDescriptor  
  
  
class PQCSimulatedTrainer:  
 def \_\_init\_\_(self, circuit: CircuitDescriptor):  
 self.optimizer = tf.keras.optimizers.Adam(lr=0.01)  
 self.model = tf.keras.models.Sequential(  
 [  
 tf.keras.layers.Input(shape=(), dtype=tf.dtypes.string),  
 tfq.layers.PQC(  
 circuit.cirq\_circuit,  
 circuit.cirq\_cost,  
 differentiator=tfq.differentiators.Adjoint(),  
 ),  
 ]  
 )  
 self.circuit = circuit  
  
 def train(self, n\_samples=100, loggers: typing.Optional[AnalyzerList] = None):  
 dummy\_input = tfq.convert\_to\_tensor([cirq.Circuit()])  
 total\_error = 0.0  
 with tqdm.trange(n\_samples) as iterator:  
 iterator.set\_description("QAOA Optimization Loop")  
 for step in iterator:  
 with tf.GradientTape() as tape:  
 error = self.model(dummy\_input)  
 grads = tape.gradient(error, self.model.trainable\_variables)  
 self.optimizer.apply\_gradients(  
 zip(grads, self.model.trainable\_variables)  
 )  
 error = error.numpy()[0][0]  
 if loggers is not None:  
 loggers.log(self, error)  
 total\_error += error  
 iterator.set\_postfix(error=total\_error / (step + 1))  
 return self.model  
  
 def evaluate(self, n\_samples: int = 1000):  
 dummy\_input = tfq.convert\_to\_tensor([cirq.Circuit()])  
 total\_error = 0.0  
 with tqdm.trange(n\_samples) as iterator:  
 iterator.set\_description("QAOA Evaluation Loop")  
 for step in iterator:  
 error = self.model(dummy\_input)  
 error = error.numpy()[0][0]  
 total\_error += error  
 iterator.set\_postfix(error=total\_error / (step + 1))  
 return total\_error / n\_samples  
import itertools  
import typing  
  
import numpy as np  
import networkx as nx  
import sympy  
import cirq  
  
from ..interface.metric\_spec import MetricSpecifier  
  
  
class QAOACircuitMaxCut:  
 def \_\_init\_\_(self, graph: nx.Graph = None, p: int = 2):  
 self.\_graph = nx.gnm\_random\_graph(n=6, m=15) if graph is None else graph  
 self.\_qubits = cirq.GridQubit.rect(1, self.\_graph.number\_of\_nodes())  
 self.params = sympy.symbols("q0:%d" % (2 \* p))  
  
 self.qaoa\_circuit = cirq.Circuit()  
 for qubit in self.\_qubits:  
 self.qaoa\_circuit.append(cirq.H(qubit))  
 for i in range(p):  
 # Cost Hamiltonian  
 for edge in self.\_graph.edges():  
 self.qaoa\_circuit += cirq.CNOT(  
 self.\_qubits[edge[0]], self.\_qubits[edge[1]]  
 )  
 self.qaoa\_circuit += cirq.rz(self.params[2 \* i]).on(  
 self.\_qubits[edge[1]]  
 )  
 self.qaoa\_circuit += cirq.CNOT(  
 self.\_qubits[edge[0]], self.\_qubits[edge[1]]  
 )  
 # Mixing Hamiltonian  
 for j in range(len(self.\_qubits)):  
 self.qaoa\_circuit += cirq.rx(2 \* self.params[2 \* i + 1]).on(  
 self.\_qubits[j]  
 )  
  
 self.qaoa\_cost: cirq.PauliSum = cirq.PauliSum()  
 for edge in self.\_graph.edges():  
 self.qaoa\_cost += cirq.PauliString(  
 1 / 2 \* cirq.Z(self.\_qubits[edge[0]]) \* cirq.Z(self.\_qubits[edge[1]])  
 )  
  
 def solve\_classically(self):  
 """Solve the combinatorial problem using a full, exponentially sized search  
 :return: Value of the max the cut  
 """  
 subsets\_list = itertools.chain.from\_iterable(  
 itertools.combinations(self.\_graph.nodes(), r)  
 for r in range(self.\_graph.number\_of\_nodes() + 1)  
 )  
 max\_cut = [  
 nx.algorithms.cuts.cut\_size(self.\_graph, assignment)  
 for assignment in subsets\_list  
 ]  
 return np.max(max\_cut)  
  
  
class MaxCutMetric(MetricSpecifier):  
 def \_\_init\_\_(self, graph):  
 super().\_\_init\_\_("samples")  
 self.graph = graph  
  
 def from\_samples\_vector(self, samples\_vector: np.ndarray) -> float:  
 return typing.cast(  
 float,  
 np.mean(  
 [  
 nx.algorithms.cuts.cut\_size(self.graph, np.where(cut)[0])  
 for cut in samples\_vector  
 ]  
 ),  
 )  
  
 def from\_density\_matrix(self, density\_matrix: np.ndarray) -> float:  
 raise NotImplementedError  
  
 def from\_state\_vector(self, state\_vector: np.ndarray) -> float:  
 raise NotImplementedError