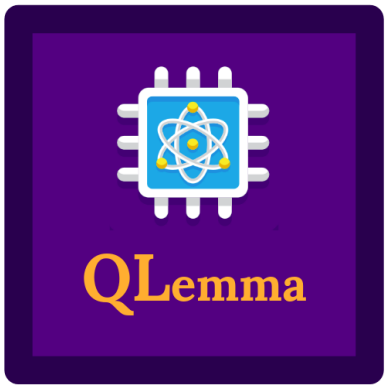
qLEET

Release v0.1.0

Animesh Sinha, Utkarsh Azad

Dec 08, 2022

CONTENTS:



qLEET is an open-source library for exploring Loss landscape, Expressibility, Entangling capability and Training trajectories of noisy parameterized quantum circuits.

Will support Qiskit's, Cirq's and pyQuil's quantum circuits and noise models.

Our package provides opportunities to improve existing algorithms like VQE, QAOA by utilizing intuitive insights from the ansatz capability and structure of loss landscape.

The aim of the library is to facilitate research in designing new hybrid quantum-classical algorithms.

CHAPTER

ONE

TYPES OF PLOTS

There are several types of analysis supported by our module. They are:

- Expressibility Plot.
- Loss Landscape Plot.
- Training Path Plot.
- Entanglement Alibility Value.
- Histogram Plot.

More types of analysis will be added, be sure to raise an issue for feature requests on the repository.

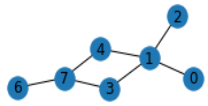
CHAPTER

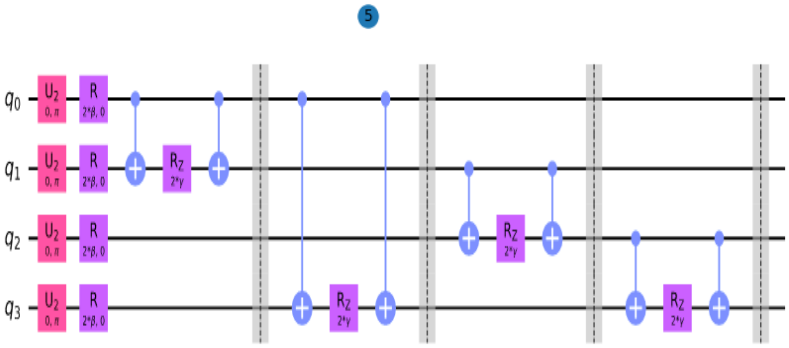
TWO

EXAMPLES OF CIRCUIT

The library comes with a few Pre-built circuits to which you can analyze out of the box.

One of those is QAOA for computing Max Cut of a Graph.





2.1 qleet package

2.1.1 Subpackages

2.1.1.1 qleet.analyzers package

Submodules

qleet.analyzers.entanglement module

Module to evaluate the achievable entanglement in circuits.

class qleet.analyzers.entanglement.**EntanglementCapability**(*circuit:* CircuitDescriptor, *noise_model: Optional[Union[cirq.devices.noise_model.NoiseModel, qiskit.providers.aer.noise.NoiseModel, pyquil.noise.NoiseModel]] = None*, *samples: int = 1000*)

Bases: *MetaExplorer*

Calculates entangling capability of a parameterized quantum circuit

entanglement_capability(*measure: str = 'meyer-wallach'*, *shots: int = 1024*) → float

Returns entanglement measure for the given circuit

Parameters

- **measure** – specification for the measure used in the entangling capability
- **shots** – number of shots for circuit execution

Returns pqc_entangling_capability (float)
entanglement measure value

Raises
ValueError – if invalid measure is specified

gen_params() → Tuple[List, 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

meyer_wallach_measure(*states*, *num_qubits*)

Returns the meyer-wallach entanglement measure for the given circuit.

$$Q = \frac{2}{|\vec{\theta}|} \sum_{\theta_i \in \vec{\theta}} \left(1 - \frac{1}{n} \sum_{k=1}^{n} Tr(\rho_k^2(\theta_i)) \right)$$

static scott_helper(*state*, *perms*)

Helper function for entanglement measure. It gives trace of the output state

scott_measure(*states*, *num_qubits*)

Returns the scott entanglement measure for the given circuit.

$$Q_m = \frac{2^m}{(2^m - 1)|\vec{\theta}|} \sum_{\theta_i \in \vec{\theta}} \left(1 - \frac{m!(n-m)!)}{n!} \sum_{|S|=m} \mathrm{Tr}(\rho_S^2(\theta_i)) \right) \quad m = 1, \ldots, \lfloor n/2 \rfloor$$

qleet.analyzers.expressibility module

Module to evaluate the expressibility of circuits.

class qleet.analyzers.expressibility.**Expressibility**(*circuit:* CircuitDescriptor, *noise_model: Optional[Union[cirq.devices.noise_model.NoiseModel, qiskit.providers.aer.noise.NoiseModel, pyquil.noise.NoiseModel]] = None*, *samples: int = 1000*)

Bases: *MetaExplorer*

Calculates expressibility of a parameterized quantum circuit

expressibility(*measure: str = 'kld'*, *shots: int = 1024*) → float

Returns expressibility for the circuit

$$Expr = D_{KL}(\hat{P}_{PQC}(F;\theta)|P_{Haar}(F))$$
$$Expr = D_{\sqrt{JSD}}(\hat{P}_{PQC}(F;\theta)|P_{Haar}(F))$$

Parameters

- **measure** – specification for the measure used in the expressibility calculation
- **shots** – number of shots for circuit execution

Returns pqc_expressibility
float, expressibility value

Raises
ValueError – if invalid measure is specified

gen_params() → Tuple[List, 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

static kl_divergence(*prob_a: numpy.ndarray*, *prob_b: numpy.ndarray*) → float

Returns KL divergence between two probabilities

plot(*figsize=(6, 4)*, *dpi=300*, ***kwargs*)

Returns plot for expressibility visualization

prob_haar() → numpy.ndarray

Returns probability density function of fidelities for Haar Random States

prob_pqc(*shots: int = 1024*) → numpy.ndarray

Return probability density function of fidelities for PQC

Parameters
shots – number of shots for circuit execution

Returns fidelities (np.array)
np.array of fidelities

qleet.analyzers.loss_landscape module

Module to plot the loss landscapes of circuits.

For any variational quantum algorithm being trained to optimize on a given metric, the plot of a projected subspace of the metric is of value because it helps us confirm along random axes that our point is indeed the local minima / maxima and also helps visualize how rough the landscape is giving clues on how likely the variational models might converge.

We hope that these visualizations can help improve the choice of optimizers and ansatz we have for these quantum circuits.

class qleet.analyzers.loss_landscape.**LossLandscapePlotter**(*solver:* PQCSimulatedTrainer, *metric:* MetricSpecifier, *dim: int = 2*)

Bases: *MetaExplorer*

This class plots the loss landscape for a given PQC trainer object.

It can plot the true loss that we are training on or on some other metric, this can help use proxy metrics as loss functions and seeing if they help optimize on the true target metric.

These plots can support 1-D and 2-D subspace projections for now, since we have to plot the loss value on the second or third axis. A 3-D projection of the plot will also be supported by v1.0.0 and onwards, which will use colors and point density to show the metric values.

plot(*mode: str = 'surface'*, *points: int = 25*, *distance: float = numpy.pi*) → plotly.graph_objects.Figure

Plots the loss landscape The surface plot is the best 3D visualization, but it uses the plotly dynamic interface, it also has an overhead contour. For simple 2D plots which can be used as matplotlib graphics or easily used in publications, use line and contour modes.

Parameters

- **mode** (*str*) – line, contour or surface, what type of plot do we want?
- **points** (*int*) – number of points to sample for the metric
- **distance** (*float*) – the range around the current parameters that we need to sample to

Returns
The figure object that has been generated

Return type
Plotly or matplotlib figure object

Raises
NotImplementedError – For the 1D plotting. TODO Implement 1D plots.

Increasing the number of points improves the quality of the plot but takes a lot more time, it scales quadratically in the number of points. Lowering the distance is a good idea if using fewer points, since you get the same number of points for a small region. Note that these plots can be deceptive, there might be large ridges that get missed due to lack of resolution of the points, always be careful and try to use as many points as possible before making a final inference.

scan(*points: int*, *distance: float*, *origin: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]

Scans the target vector-subspace for values of the metric Returns the sampled coordinates in the grid and the values of the metric at those coordinates. The sampling of the subspace is done uniformly, and evenly in all directions.

Parameters

- **points** (*int*) – Number of points to sample
- **distance** (*float*) – The range of parameters around the current value to scan over

- **origin** (*np.ndarray*) – The value of the current parameter to be used as origin of our plot

Returns
tuple of the coordinates and the metric values at those coordinates

Return type
a tuple of np.array, shapes being (n, dims) and (n,)

qleet.analyzers.training_path module

Module responsible to generating plots of the training trajectory.

The training trajectory is the set of parameter values (projected down to some low dimensional space) that the model had through the different epochs of it's training process. This when plotted for one model tells us if the loss was decreasing always, if the learning rate should be lowered, increased, or what the schedule should look like, etc. When plotted for more than one model, it let's us know if the paths are converging or not, giving us a view of how likely is our generated solution optimal. If many of the models converge to the same path and start mixing, then they likely are optimal, if not they they are more likely to be just random chance solutions.

class qleet.analyzers.training_path.**LossLandscapePathPlotter**(*base_plotter:* LossLandscapePlotter)

Bases: *MetaLogger*

An module to plot the training path of the PQC on the loss landscape

This class is an extension of the Loss Landscape plotter and the Training Path plotter, puts both the ideas together and shows how the different models ended us at different parts of the loss landscape.

log(*solver:* PQCSimulatedTrainer, *loss: float*)

Logs the value of the parameters that the circuit currently has. The parameter values should be a numpy vector.

Parameters

- **solver** (PQCSimulatedTrainer) – The trainer module which has the parameters to be plotted
- **loss** (*float*) – The value of the loss at the current epoch

plot()

Plots the 2D parameter projections with the loss value on the 3rd dimension. For the entire set of runs, the class has logged the parameter values. Now it reduces the dimensionality of those parameter vectors using PCA or tSNE and then plots them on a 2D plane, associates them with a loss value to put on the third dimension. This output is coupled with the actual loss landscape drawing and returned.

Returns
The figure on which the parameter projections are plotted

Return type
Plotly figure

class qleet.analyzers.training_path.**OptimizationPathPlotter**(*mode: str = 'tSNE'*)

Bases: *MetaLogger*

Class which logs the parameter information and plots it over the iterations of training.

This will be used to plot the parameter values in 2-D or 3-D, rather a t-SNE or PCA projection or the parameter values. For getting the loss values for the associated training points as a part of the plot too, see *LossLandscapePathPlotter*.

This class conforms to the *MetaLogger* interface and can be used as part of an *AnalyzerList* when plotting the training properties of a circuit.

log(*solver:* PQCSimulatedTrainer, *_loss: float*) → None

Logs the value of the parameters that the circuit currently has. The parameter values should be a numpy vector.

Parameters

- **solver** (PQCSimulatedTrainer) – The trainer module which has the parameters to be plotted
- **_loss** (*float*) – The loss value at that epoch, not used by this class

plot() → plotly.graph_objects.Figure

Plots the 2D parameter projections. For the entire set of runs, the class has logged the parameter values. Now it reduces the dimensionality of those parameter vectors using PCA or tSNE and then plots them on a 2D plane.

Returns
The figure on which the parameter projections are plotted

Return type
Plotly figure

qleet.analyzers.histogram module

Module contents

2.1.1.2 qleet.examples package

Submodules

qleet.examples.qaoa_maxcut module

Implements an example of QAOA for Max Cut. Allows the user to analyze QAOA with easy setup.

class qleet.examples.qaoa_maxcut.**MaxCutMetric**(*graph*)

Bases: *MetricSpecifier*

The metric for the Max Cut problem, generates using a classical process.

from_density_matrix(*density_matrix: numpy.ndarray*) → float

Computes the vector from the samples vector output from the quantum circuit. :type density_matrix: np.array, 2-D matrix of size (2^n, 2^n) :param density_matrix: The 2-D density matrix to generate the output metric :returns: The value of the max-cut :rtype: float :raises NotImplemetedError: Computing from density-matrix is not implemented yet

from_samples_vector(*samples_vector: numpy.ndarray*) → float

Computes the vector from the samples vector output from the quantum circuit. :type samples_vector: np.array, 2-D matrix of size (num_samples, n) :param samples_vector: *num_samples* measurements each of size *n*, as a 2-D matrix :returns: The value of the max-cut :rtype: float

from_state_vector(*state_vector: numpy.ndarray*) → float

Computes the vector from the samples vector output from the quantum circuit. :type state_vector: np.array, 2-D matrix of size (2^n,) :param state_vector: The 2-D state vector to generate the output metric :returns:

The value of the max-cut :rtype: float :raises NotImplemetedError: Computing from density-matrix is not implemented yet

class qleet.examples.qaoa_maxcut.**QAOACircuitMaxCut**(*graph: Optional[Graph] = None*, *p: int = 2*)

Bases: object

The class to specify a QAOA circuit and metric for computing Max Cut of a graph.

solve_classically()

Solve the combinatorial problem using a full, exponentially sized search :return: Value of the max the cut :rtype: float

Module contents

2.1.1.3 qleet.interface package

Submodules

qleet.interface.circuit module

This module provides the interface to the circuits that the user specifies and library uses.

This makes the abstractions which ensures all operations in the library are backend agnostic, by allowing us to get the circuit back in the desired library's form, cirq, qiskit or pytket. It allows the user to specify the circuit in any form. It also takes the loss function specification as a Pauli String.

It also exposes functions that the user can use to convert their circuits to a qiskit or cirq backend. WARNING: the conversion is done through a OpenQASM intermediate, operations not supported on QASM cannot be converted directly, please provide your circuit in a Cirq or Qiskit backend in that case.

class qleet.interface.circuit.**CircuitDescriptor**(*circuit: Union[qiskit.QuantumCircuit, cirq.Circuit, pyquil.Program]*, *params: List[Union[sympy.Symbol, qiskit.circuit.Parameter]]*, *cost_function: Optional[Union[cirq.PauliSum, qiskit.quantum_info.PauliList, pyquil.paulis.PauliSum]] = None*)

Bases: object

The interface for users to provide a circuit in any framework and visualize it in qLEET.

It consists of 3 parts: * Circuit: which has the full ansatz preparation from the start where * Params: list of parameters which are used to parameterize the circuit * Cost Function: presently a pauli string, which we measure to get the

output we are optimizing over

Combined they form the full the parameterized quantum circuit from the initial qubits to the end measurement.

property cirq_circuit: cirq.Circuit

Get the circuit in cirq :return: the cirq representation of the circuit :rtype: cirq.Circuit

property cirq_cost: 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

property default_backend: str

Returns the backend in which the user had provided the circuit. :returns: The name of the default backend :rtype: str :raises ValueError: if the given circuit is not from a supported library

classmethod from_qasm(*qasm_str: str*, *params: List[Union[sympy.Symbol, qiskit.circuit.Parameter]]*, *cost_function: Optional[Union[cirq.PauliSum, qiskit.quantum_info.PauliList, pyquil.paulis.PauliSum]]*, *backend: str = 'cirq'*)

Generate the descriptor from OpenQASM string

Parameters

- **qasm_str** (*str*) – OpenQASM string for each part of the circuit
- **params** (*list[sympy.Symbol]*) – list of sympy symbols which act as parameters for the PQC
- **cost_function** (*PauliSum*) – pauli-string operator to implement cost function
- **backend** (*str*) – backend for the circuit descriptor objects

Returns
The CircuitDescriptor object

Return type
CircuitDescriptor

property num_qubits: int

Get the number of qubits for a circuit :return: the number of qubits in the circuit :rtype: int :raises ValueError: if unsupported circuit framework is given

property parameters: List[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

property pyquil_circuit: pyquil.Program

Get the circuit in pyquil :return: the pyquil representation of the circuit :rtype: pyquil.Program

property qiskit_circuit: qiskit.QuantumCircuit

Get the circuit in qiskit :return: the qiskit representation of the circuit :rtype: qiskit.QuantumCircuit

qleet.interface.circuit.**convert_to_cirq**(*circuit: Union[qiskit.QuantumCircuit, cirq.Circuit, pyquil.Program]*) → cirq.Circuit

Converts any circuit to cirq :type circuit: Circuit in any supported library :param circuit: input circuit in any framework :return: circuit in cirq :rtype: cirq.Circuit :raises ValueError: if the circuit is not from one of the supported frameworks

qleet.interface.circuit.**convert_to_pyquil**(*circuit: Union[qiskit.QuantumCircuit, cirq.Circuit, pyquil.Program]*) → qiskit.QuantumCircuit

Converts any circuit to pyquil :type circuit: Circuit in any supported library :param circuit: input circuit in any framework :raises ValueError: if the circuit is not from one of the supported frameworks :return: circuit in pyquil :rtype: pyquil.Program

qleet.interface.circuit.**convert_to_qiskit**(*circuit: Union[qiskit.QuantumCircuit, cirq.Circuit, pyquil.Program]*) → qiskit.QuantumCircuit

Converts any circuit to qiskit :type circuit: Circuit in any supported library :param circuit: input circuit in any framework :raises ValueError: if the circuit is not from one of the supported frameworks :return: circuit in qiskit :rtype: qiskit.QuantumCircuit

qleet.interface.dashboard module

qleet.interface.metas module

This module houses the interfaces for analyzers, and provide a utility container AnalyzerList

- **MetaLogger is interface for those analyzers which need the state of the** circuit at each timestep in training.
- **MetaExplorer is the interaface for those analyzers which can generate** properties from a single snapshot of the circuit.
- **AnalyzerList is the convinience container which acts as a list of analyzers** which easy to use API.

class qleet.interface.metas.**AnalyzerList**(**args: Union[MetaLogger, MetaExplorer]*)

Bases: object

Container class, Stores a list of loggers.

All the loggers can be asked to log the information they need together. The information to be logged can be provided to the Analyzer List in one convinient function call, and all the associated functions for all the loggers get called which can accept that form of data. All the loggers can also together be moved to the next model.

log(*solver:* PQCSimulatedTrainer, *loss: float*) → None

Logs the current state of model in all the loggers. Does not ask the *MetaAnalyzers* to log the information since they don't implement the logging interface. :type solver: PQCSimulatedTrainer :param solver: The PQC trainer whose parameters are to be logged :type loss: float :param loss: Loss value on the current epoch

next() → None

Moves the loggers to logging of the next model. Completes the logging for the current training path.

class qleet.interface.metas.**MetaExplorer**

Bases: ABC

Abstract class to represent interface of analyzing a the current state of the circuit. Treats the parameters of the circuit as a snapshot.

class qleet.interface.metas.**MetaLogger**

Bases: ABC

Abstract class to represent interface of logging. Logs the present state of the model during training.

abstract log(*solver:* PQCSimulatedTrainer, *loss: float*)

Logs information at one timestep about either the solver or the present loss.

Parameters

- **solver** (PQCSimulatedTrainer) – The state of the PQC trainer at the current timestep
- **loss** (*float*) – The loss at the current timestep

next()

Moves the logger to analyzing the next run or model path.

abstract plot()

Plots the values logged by the logger.

qleet.interface.metric_spec module

This module houses the ways to specify a metric, and sample solutions to compute it's value.

Metrics are a useful abstraction which given the state of the circuit compute some classical value which we need to interpret or plot.

class qleet.interface.metric_spec.**MetricSpecifier**(*default_call_mode: str = 'samples'*)

Bases: ABC

Class to specify classical metrics which are a function of the sampled quantum state. Examples would be an arbitrary cost function, Mean Squared Error for some regression task, size of the max cut, etc.

This is an abstract class, all metrics should be it's subclasses

How the metric is computed is left for the user to decide, it can be from the actual samples drawn from the circuit, or from the state vector, or from the density matrix.

from_circuit(*circuit_descriptor:* CircuitDescriptor, *parameters: Union[numpy.ndarray, List]*, *mode: str = 'samples'*) → float

Computes the value of the metric from the circuit, by using the default mode or metric computation. :type circuit_descriptor: CircuitDescriptor :param circuit_descriptor: The provided circuit :type parameters: List or Numpy array :param parameters: List of values of the parameters to sample the circuit at :type mode: str :param mode: From what to compute the metric, samples, state_vector, or density_matrix :return: The value of the metric at those parameters :rtype: float :raises NotImplementedError: if required mode of evaluating metric wasn't implemented :raises ValueError: if the mode specified wasn't valid

abstract from_density_matrix(*density_matrix: numpy.ndarray*) → float

Returns the value of the loss function given the density matrix of the state prepared from the circuit using the noise model provided. :type density_matrix: np.ndarray, 2-D of shape (2^n, 2^n) :param density_matrix: Vector of samples drawn from the circuit :return: value of the loss function :rtype: float

abstract from_samples_vector(*samples_vector: numpy.ndarray*) → float

Returns the value of the loss function from one set of measurements sampled from the circuit. :type samples_vector: np.ndarray, 1-D of shape (n,) :param samples_vector: Vector of samples drawn from the circuit :return: value of the loss function :rtype: float

abstract from_state_vector(*state_vector: numpy.ndarray*) → float

Returns the value of the loss function given the state vector of the state prepared from the circuit. :type state_vector: np.ndarray, 1-D of shape (2^n,) :param state_vector: State vector of state prepared by circuit :return: value of the loss function :rtype: float

qleet.interface.metric_spec.**sample_solutions**(*circuit: cirq.Circuit*, *param_symbols: List[sympy.Symbol]*, *param_values: Iterable*, *samples: int = 1000*) → numpy.ndarray

Get the computed cuts for a given ansatz :type circuit: cirq.Circuit :param circuit: Circuit to be sampled :type param_symbols: List of sympy.Symbols :param param_symbols: The symbols of model parameters :type param_values: List of floats :param param_values: The value of model parameters to sample at, 1-D vector :type samples: int :param samples: Number of times to sample the resulting quantum state :return: 2-D matrix, n_samples rows of boolean vectors showing the cut :rtype: np.array

Module contents

2.1.1.4 qleet.simulators package

Submodules

qleet.simulators.circuit_simulators module

Module to draw samples from the circuit. Used for computing properties of the circuit like Entanglability and Expressibility.

class qleet.simulators.circuit_simulators.**CircuitSimulator**(*circuit:* CircuitDescriptor, *noise_model: Optional[Union[cirq.devices.noise_model.NoiseModel, qiskit.providers.aer.noise.NoiseModel, pyquil.noise.NoiseModel]] = None*)

Bases: object

The interface for users to execute their CircuitDescriptor objects

property result: Optional[numpy.ndarray]

Get the results stored from the circuit simulator :return: stored result of the circuit simulation if it has been performed, else None. :rtype: np.array or None

simulate(*param_resolver: Dict[qiskit.circuit.Parameter, float]*, *shots: int = 1024*) → numpy.ndarray

Simulate to get the state vector or the density matrix :type param_resolver: Dict to resolve all parameters to a static float value :param param_resolver: a dictionary of all the symbols/parameters mapping to their values :type shots: int :param shots: number of times to run the qiskit density matrix simulator :returns: state vector or density matrix resulting from the simulation :rtype: np.array :raises NotImplementedError: if circuit simulation is not supported for a backend

qleet.simulators.pqc_trainer module

The module which houses the Parametrized Quantum Circuit trainer class.

It generates the TensorFlow Quantum model, and allows Keras like API to train and evaluate a model.

class qleet.simulators.pqc_trainer.**PQCSimulatedTrainer**(*circuit:* CircuitDescriptor)

Bases: object

A class to train parametrized Quantum Circuits in Tensorflow Quantum Uses gradient descent over the provided parameters, using the TFQ Adjoin differentiator.

evaluate(*n_samples: int = 1000*) → float

Evaluates the Parametrized Quantum Circuit. :type n_samples: int :param n_samples: The number of samples to evaluate the circuit over :returns: The average loss of the circuit over all the samples :rtype: float

train(*n_samples=100*, *loggers: Optional[AnalyzerList] = None*) → tensorflow.keras.Model

Trains the parameter of the circuit to minimize the loss. :type n_samples: int :param n_samples: Number of samples to train the circuit over :type loggers: *AnalyzerList* :param loggers: The AnalyzerList that tracks the training of the model :returns: The trained model :rtype: tf.keras.Model

Module contents

2.1.2 Module contents

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

q

INDEX

N

O

P

Q

R

S

T