
tensorcircuit Documentation

refraction-ray

Feb 03, 2023

CONTENTS

1	Links	3
2	Reference Documentation	5
2.1	Quick Start	5
2.2	Advanced Usage	20
2.3	Frequently Asked Questions	24
2.4	TensorCircuit: The Sharp Bits	28
2.5	TensorCircuit: What is inside?	31
2.6	Guide for Contributors	34
3	Tutorials	39
3.1	Jupyter Tutorials	39
3.2	Whitepaper Tutorials	145
4	API References	195
4.1	tensorcircuit	195
5	Indices and Tables	731
Python Module Index		733
Index		735



TensorCircuit is an open source quantum circuit and algorithm simulation framework.

- It is built for human beings.
- It is designed for speed, flexibility and elegance.
- It is empowered by advanced tensor network simulator engine.
- It is implemented with industry-standard machine learning frameworks: TensorFlow, JAX, and PyTorch.
- It is compatible with machine learning engineering paradigms: automatic differentiation, just-in-time compilation, vectorized parallelism and GPU acceleration.

**CHAPTER
ONE**

LINKS

TensorCircuit is created and maintained by [Shi-Xin Zhang](#) and this version of the software is released by [Tencent Quantum Lab](#). The current core authors of TensorCircuit are [Shi-Xin Zhang](#) and [Yu-Qin Chen](#). We also thank [contributions](#) from the lab and the open source community.

- Source code: <https://github.com/tencent-quantum-lab/tensorcircuit>
- Software Whitepaper in Quantum: <https://quantum-journal.org/papers/q-2023-02-02-912/>
- Documentation: <https://tensorcircuit.readthedocs.io>
- Issue Tracker: <https://github.com/tencent-quantum-lab/tensorcircuit/issues>
- Forum: <https://github.com/tencent-quantum-lab/tensorcircuit/discussions>
- PyPI page: <https://pypi.org/project/tensorcircuit>
- DockerHub page: <https://hub.docker.com/repository/docker/tensorcircuit/tensorcircuit>

REFERENCE DOCUMENTATION

The following documentation sections briefly introduce TensorCircuit to the users and developpers.

2.1 Quick Start

2.1.1 Installation

- For x86 Linux or Mac,

```
pip install tensorcircuit
```

is in general enough. Either pip from conda or other python env managers is fine.

Since there are many optional packages for various features, the users may need to install more pip packages when required.

- For Linux with Nvidia GPU,

please refer to the GPU aware installation guide of corresponding machine learning frameworks: [TensorFlow](#), [Jax](#), or [PyTorch](#).

Docker is also recommended (especially Linux + Nvidia GPU setup):

```
sudo docker run -it --network host --gpus all tensorcircuit/tensorcircuit.
```

- For Windows, due to the lack of support for Jax, we recommend to use docker or WSL, please refer to [TC via windows docker](#) or [TC via WSL](#).
- For Mac with M series chips (arm architecture), please refer to [TC on Mac M series](#).

Overall, the installation of TensorCircuit is simple, since it is purely in Python and hence very portable. As long as the users can take care of the installation of ML frameworks on the corresponding system, TensorCircuit will work as expected.

Note: We also provide a nightly build of tensorcircuit via PyPI which can be accessed by `pip uninstall tensorcircuit`, then `pip install tensorcircuit-nightly`

2.1.2 Circuit Object

The basic object for TensorCircuit is `tc.Circuit`.

Initialize the circuit with the number of qubits `c=tc.Circuit(n)`.

Input States:

The default input function for the circuit is $|0^n\rangle$. One can change this to other wavefunctions by directly feeding the inputs state vectors `w: c=tc.Circuit(n, inputs=w)`.

One can also feed matrix product states as input states for the circuit, but we leave MPS/MPO usage for future sections.

Quantum Gates:

We can apply gates on circuit objects. For example, using `c.H(1)` or `c.rx(2, theta=0.2)`, we can apply Hadamard gate on qubit 1 (0-based) or apply Rx gate on qubit 2 as $e^{-i\theta/2X}$.

The same rule also applies to multi-qubit gates, such as `c.cnot(0, 1)`.

There are also highly customizable gates, two instances are:

- `c.exp1(0, 1, unitary=m, theta=0.2)` which is for the exponential gate $e^{i\theta m}$ of any matrix `m` as long as $m^2 = 1$.
- `c.any(0, 1, unitary=m)` which is for applying the unitary gate `m` on the circuit.

These two examples are flexible and support gates on any number of qubits.

Measurements and Expectations:

The most straightforward way to get the output from the circuit object is by getting the output wavefunction in vector form as `c.state()`.

For bitstring sampling, we have `c.perfect_sampling()` which returns the bitstring and the corresponding probability amplitude.

To measure part of the qubits, we can use `c.measure(0, 1)`, if we want to know the corresponding probability of the measurement output, try `c.measure(0, 1, with_prob=True)`. The measure API is by default non-jittable, but we also have a jittable version as `c.measure_jit(0, 1)`.

The measurement and sampling utilize advanced algorithms based on tensornetwork and thus require no knowledge or space for the full wavefunction.

See the example below:

```
K = tc.set_backend("jax")
@K.jit
def sam(key):
    K.set_random_state(key)
    n = 50
    c = tc.Circuit(n)
    for i in range(n):
        c.H(i)
    return c.perfect_sampling()

sam(jax.random.PRNGKey(42))
sam(jax.random.PRNGKey(43))
```

To compute expectation values for local observables, we have `c.expectation([tc.gates.z(), [0]], [tc.gates.z(), [1]])` for $\langle Z_0 Z_1 \rangle$ or `c.expectation([tc.gates.x(), [0]])` for $\langle X_0 \rangle$.

This expectation API is rather flexible, as one can measure an m on several qubits as `c.expectation([m, [0, 1, 2]])`.

We can also extract the unitary matrix underlying the whole circuit as follows:

```
>>> n = 2
>>> c = tc.Circuit(n, inputs=tc.backend.eye(2**n))
>>> c.X(1)
>>> tc.backend.reshape(c.state())
array([[0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j],
       [1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j],
       [0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j]], dtype=complex64)
```

Circuit Transformations:

We currently support transform `tc.Circuit` from and to Qiskit `QuantumCircuit` object.

Export to Qiskit (possible for further hardware experiment, compiling, and visualization): `c.to_qiskit()`.

Import from Qiskit: `c = tc.Circuit.from_qiskit(QuantumCircuit, n)`. Parameterized Qiskit circuit is supported by passing the parameters to the `binding_parameters` argument of the `from_qiskit` function, similar to the `assign_parameters` function in Qiskit.

Circuit Visualization:

`c.vis_tex()` can generate tex code for circuit visualization based on LaTeX `quantikz` package.

There are also some automatic pipeline helper functions to directly generate figures from tex code, but they require extra installations in the environment.

`render_pdf(tex)` function requires full installation of LaTeX locally. And in the Jupyter environment, we may prefer `render_pdf(tex, notebook=True)` to return jpg figures, which further require wand magicwand library installed, see [here](#).

Or since we can transform `tc.Circuit` into `QuantumCircuit` easily, we have a simple pipeline to first transform `tc.Circuit` into Qiskit and then call the visualization built in Qiskit. Namely, we have `c.draw()` API.

Circuit Intermediate Representation:

TensorCircuit provides its own circuit IR as a python list of dicts. This IR can be further utilized to run compiling, generate serialization `qasm`, or render circuit figures.

The IR is given as a list, each element is a dict containing information on one gate that is applied to the circuit. Note `gate attr` in the dict is a python function that returns the gate's node.

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.crx(1, 0, theta=0.2)
>>> c.to_qir()
[{'gate': cnot, 'index': (0, 1), 'name': 'cnot', 'split': None}, {'gate': crx, 'index': (1, 0), 'name': 'crx', 'split': None, 'parameters': {'theta': 0.2}}]
```

2.1.3 Programming Paradigm

The most common case and the most typical programming paradigm for TensorCircuit are to evaluate the circuit output and the corresponding quantum gradients, which is common in variational quantum algorithms.

```
import tensorcircuit as tc

K = tc.set_backend("tensorflow")

n = 1

def loss(params, n):
    c = tc.Circuit(n)
    for i in range(n):
        c.rx(i, theta=params[0, i])
    for i in range(n):
        c.rz(i, theta=params[1, i])
    loss = 0.0
    for i in range(n):
        loss += c.expectation([tc.gates.z(), [i]])
    return K.real(loss)

vgf = K.jit(K.value_and_grad(loss), static_argnums=1)
params = K.implicit_rndn([2, n])
print(vgf(params, n)) # get the quantum loss and the gradient
```

Also for a non-quantum example (linear regression) demonstrating the backend agnostic feature, variables with pytree support, AD/jit/vmap usage, and variational optimization loops. Please refer to the example script: [linear regression example](#). This example might be more friendly to the machine learning community since it is purely classical while also showcasing the main features and paradigms of tensorcircuit.

If the user has no intention to maintain the application code in a backend agnostic fashion, the API for ML frameworks can be more handily used and interleaved with the TensorCircuit API.

```
import tensorcircuit as tc
import tensorflow as tf

K = tc.set_backend("tensorflow")

n = 1

def loss(params, n):
    c = tc.Circuit(n)
    for i in range(n):
        c.rx(i, theta=params[0, i])
    for i in range(n):
        c.rz(i, theta=params[1, i])
    loss = 0.0
    for i in range(n):
        loss += c.expectation([tc.gates.z(), [i]])
    return tf.math.real(loss)
```

(continues on next page)

(continued from previous page)

```

def vgf(params, n):
    with tf.GradientTape() as tape:
        tape.watch(params)
        l = loss(params, n)
    return l, tape.gradient(l, params)

vgf = tf.function(vgf)
params = tf.random.normal([2, n])
print(vgf(params, n)) # get the quantum loss and the gradient

```

2.1.4 Automatic Differentiation, JIT, and Vectorized Parallelism

For concepts of AD, JIT and VMAP, please refer to [Jax documentation](#).

The related API design in TensorCircuit closely follows the functional programming design pattern in Jax with some slight differences. So we strongly recommend users learn some basics about Jax no matter which ML backend they intend to use.

AD Support:

Gradients, vjps, jvps, natural gradients, Jacobians, and Hessians. AD is the base for all modern machine learning libraries.

JIT Support:

Parameterized quantum circuits can run in a blink. Always use jit if the circuit will get evaluations multiple times, it can greatly boost the simulation with two or three order time reduction. But also be cautious, users need to be familiar with jit, otherwise, the jitted function may return unexpected results or recompile on every hit (wasting lots of time). To learn more about the jit mechanism, one can refer to documentation or blogs on `tf.function` or `jax.jit`, though these two still have subtle differences.

VMAP Support:

Inputs, parameters, measurements, circuit structures, and Monte Carlo noise can all be evaluated in parallel. To learn more about vmap mechanism, one can refer to documentation or blogs on `tf.vectorized_map` or `jax.vmap`.

2.1.5 Backend Agnosticism

TensorCircuit supports TensorFlow, Jax, and PyTorch backends. We recommend using TensorFlow or Jax backend since PyTorch lacks advanced jit and vmap features.

The backend can be set as `K=tc.set_backend("jax")` and K is the backend with a full set of APIs as a conventional ML framework, which can also be accessed by `tc.backend`.

```

>>> import tensorcircuit as tc
>>> K = tc.set_backend("tensorflow")
>>> K.ones([2,2])
<tf.Tensor: shape=(2, 2), dtype=complex64, numpy=
array([[1.+0.j, 1.+0.j],
       [1.+0.j, 1.+0.j]], dtype=complex64)>
>>> tc.backend.eye(3)
<tf.Tensor: shape=(3, 3), dtype=complex64, numpy=
array([[1.+0.j, 0.+0.j, 0.+0.j],
       [0.+0.j, 1.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j, 1.+0.j]], dtype=complex64)>

```

(continues on next page)

(continued from previous page)

```
[0.+0.j, 1.+0.j, 0.+0.j],
[0.+0.j, 0.+0.j, 1.+0.j]], dtype=complex64)>
>>> tc.set_backend("jax")
<tensorcircuit.backends.jax_backend.JaxBackend object at 0x7fb00e0fd6d0>
>>> tc.backend.name
'jax'
>>> tc.backend.implicit_randu()
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
˓→rerun for more info.)
DeviceArray([0.7400521], dtype=float32)
```

The supported APIs in the backend come from two sources, one part is implemented in [TensorNetwork package](#) and the other part is implemented in [TensorCircuit package](#). To see all the backend agnostic APIs, try:

```
>>> [s for s in dir(tc.backend) if not s.startswith("_")]
['abs',
'acos',
'acosh',
'addition',
'adjoint',
'arange',
'argmax',
'argmin',
'asin',
'asinh',
'atan',
'atan2',
'atanh',
'broadcast_left_multiplication',
'broadcast_right_multiplication',
'cast',
'cholesky',
'concat',
'cond',
'conj',
'convert_to_tensor',
'coo_sparse_matrix',
'coo_sparse_matrix_from_numpy',
'copy',
'cos',
'cosh',
'cumsum',
'deserialize_tensor',
'device',
'device_move',
'diagflat',
'diagonal',
'divide',
'dtype',
'eigh',
'eigs',
'eigsh',
```

(continues on next page)

(continued from previous page)

```
'eigsh_lanczos',
'eigvalsh',
'einsum',
'eps',
'exp',
'expm',
'eye',
'from_dlpack',
'gather1d',
'get_random_state',
'gmres',
'grad',
'hessian',
'i',
'imag',
'implicit_randc',
'implicit_randn',
'implicit_randu',
'index_update',
'inv',
'is_sparse',
'is_tensor',
'item',
'jacwd',
'jacfd',
'jacrev',
'jit',
'jvp',
'kron',
'left_shift',
'log',
'matmul',
'max',
'mean',
'min',
'mod',
'multiply',
'name',
'norm',
'numpy',
'one_hot',
'onehot',
'ones',
'outer_product',
'pivot',
'power',
'probability_sample',
'qr',
'randn',
'random_split',
'random_uniform',
'real',
```

(continues on next page)

(continued from previous page)

```
'relu',
'reshape',
'reshape2',
'reshapem',
'reverse',
'right_shift',
'rq',
'scatter',
'searchsorted',
'serialize_tensor',
'set_random_state',
'shape_concat',
'shape_prod',
'shape_tensor',
'shape_tuple',
'sigmoid',
'sign',
'sin',
'sinh',
'size',
'sizen',
'slice',
'softmax',
'solve',
'sparse_dense_matmul',
'sparse_shape',
'sqrt',
'sqrtnh',
'stack',
'stateful_randc',
'stateful_randn',
'stateful_randu',
'std',
'stop_gradient',
'subtraction',
'sum',
'svd',
'switch',
'tan',
'tanh',
'tensordot',
'tile',
'to_dense',
'to_dlpack',
'trace',
'transpose',
'tree_flatten',
'tree_map',
'tree_unflatten',
'unique_with_counts',
'value_and_grad',
'vectorized_value_and_grad',
```

(continues on next page)

(continued from previous page)

```
'vjp',
'vmap',
'vvag',
'zeros']
```

2.1.6 Switch the Dtype

TensorCircuit supports simulation using 32/64 bit precession. The default dtype is 32-bit as “complex64”. Change this by `tc.set_dtype("complex128")`.

`tc.dtypestr` always returns the current dtype string: either “complex64” or “complex128”.

2.1.7 Setup the Contractor

TensorCircuit is a tensornetwork contraction-based quantum circuit simulator. A contractor is for searching for the optimal contraction path of the circuit tensornetwork.

There are various advanced contractors provided by third-party packages, such as `opt-einsum` and `cotengra`.

`opt-einsum` is shipped with TensorNetwork package. To use `cotengra`, one needs to pip install it; `kahypar` is also recommended to install with `cotengra`.

Some setup cases:

```
import tensorcircuit as tc

# 1. cotengra contractors, have better and consistent performance for large circuit_
# simulation
import cotengra as ctg

optr = ctg.ReusableHyperOptimizer(
    methods=["greedy", "kahypar"],
    parallel=True,
    minimize="flops",
    max_time=120,
    max_repeats=4096,
    progbar=True,
)
tc.set_contractor("custom", optimizer=optr, preprocessing=True)
# by preprocessing set as True, tensorcircuit will automatically merge all single-qubit_
# gates into entangling gates

# 2. RandomGreedy contractor
tc.set_contractor("custom_stateful", optimizer=oem.RandomGreedy, max_time=60, max_
repeats=128, minimize="size")

# 3. state simulator like contractor provided by tensorcircuit, maybe better when there_
# is ring topology for two-qubit gate layout
tc.set_contractor("plain-experimental")
```

For advanced configurations on `cotengra` contractors, please refer to `cotengra` doc and more fancy examples can be found at `contractor tutorial`.

Setup in Function or Context Level

Beside global level setup, we can also setup the backend, the dtype, and the contractor at the function level or context manager level:

```
with tc.runtime_backend("tensorflow"):
    with tc.runtime_dtype("complex128"):
        m = tc.backend.eye(2)
n = tc.backend.eye(2)
print(m, n) # m is tf tensor while n is numpy array

@tc.set_function_backend("tensorflow")
@tc.set_function_dtype("complex128")
def f():
    return tc.backend.eye(2)
print(f()) # complex128 tf tensor
```

2.1.8 Noisy Circuit Simulation

Monte Carlo State Simulator:

For the Monte Carlo trajectory noise simulator, the unitary Kraus channel can be handled easily. TensorCircuit also supports fully jittable and differentiable general Kraus channel Monte Carlo simulation, though.

```
def noisecircuit(random):
    c = tc.Circuit(1)
    c.x(0)
    c.thermalrelaxation(
        0,
        t1=300,
        t2=400,
        time=1000,
        method="ByChoi",
        excitedstatepopulation=0,
        status=random,
    )
    return c.expectation_ps(z=[0])

K = tc.set_backend("tensorflow")
noisec_vmap = K.jit(K.vmap(noisecircuit, vectorized_argnums=0))
nmc = 10000
random = K.implicit_randu(nmc)
valuemc = K.mean(K.numpy(noisec_vmap(random)))
# (0.931+0j)
```

Density Matrix Simulator:

Density matrix simulator `tc.DMCircuit` simulates the noise in a full form, but takes twice qubits to do noiseless simulation. The API is the same as `tc.Circuit`.

```
def noisecircuitdm():
    dmc = tc.DMCircuit(1)
    dmc.x(0)
```

(continues on next page)

(continued from previous page)

```

dmc.thermalrelaxation(
    0, t1=300, t2=400, time=1000, method="ByChoi", excitedstatepopulation=0
)
return dmc.expectation_ps(z=[0])

K = tc.set_backend("tensorflow")
noisec_jit = K.jit(noisecircuitdm)
valuedm = noisec_jit()
# (0.931+0j)

```

Experiment with quantum errors:

Multiple quantum errors can be added on circuit.

```

c = tc.Circuit(1)
c.x(0)
c.thermalrelaxation(
    0, t1=300, t2=400, time=1000, method="ByChoi", excitedstatepopulation=0
)
c.generaldepolarizing(0, p=0.01, num_qubits=1)
c.phasedamping(0, gamma=0.2)
c.amplitudedamping(0, gamma=0.25, p=0.2)
c.reset(0)
c.expectation_ps(z=[0])

```

Experiment with readout error:

Readout error can be added in experiments for sampling and expectation value calculation.

```

c = tc.Circuit(3)
c.X(0)
readout_error = []
readout_error.append([0.9, 0.75]) # readout error of qubit 0 p0/0=0.9, p1/1=0.75
readout_error.append([0.4, 0.7]) # readout error of qubit 1
readout_error.append([0.7, 0.9]) # readout error of qubit 2
value = c.sample_expectation_ps(z=[0, 1, 2], readout_error=readout_error)
# tf.Tensor(0.039999977, shape=(), dtype=float32)
instances = c.sample(
    batch=3,
    allow_state=True,
    readout_error=readout_error,
    random_generator=tc.backend.get_random_state(42),
    format_="sample_bin"
)
# tf.Tensor(
# [[1 0 0]
# [1 0 0]
# [1 0 1]], shape=(3, 3), dtype=int32)

```

2.1.9 MPS and MPO

TensorCircuit has its class for MPS and MPO originally defined in TensorNetwork as `tc.QuVector`, `tc.QuOperator`. `tc.QuVector` can be extracted from `tc.Circuit` as the tensor network form for the output state (uncontracted) by `c.quvector()`.

The QuVector forms a wavefunction `w`, which can also be fed into Circuit as the inputs state as `c=tc.Circuit(n, mps_inputs=w)`.

- MPS as input state for circuit

The MPS/QuVector representation of the input state has a more efficient and compact form.

```
n = 3
nodes = [tc.gates.Gate(np.array([0.0, 1.0])) for _ in range(n)]
mps = tc.quantum.QuVector([nd[0] for nd in nodes])
c = tc.Circuit(n, mps_inputs=mps)
c.x(0)
c.expectation_ps(z=[0])
# 1.0
```

- MPS as (uncomputed) output state for circuit

For example, a quick way to calculate the wavefunction overlap without explicitly computing the state amplitude is given as below:

```
>>> c = tc.Circuit(3)
>>> [c.H(i) for i in range(3)]
[None, None, None]
>>> c.cnot(0, 1)
>>> c2 = tc.Circuit(3)
>>> [c2.H(i) for i in range(3)]
[None, None, None]
>>> c2.cnot(1, 0)
>>> q = c.quvector()
>>> q2 = c2.quvector().adjoint()
>>> (q2@q).eval_matrix()
array([[0.9999998+0.j]], dtype=complex64)
```

- MPO as the gate on the circuit

Instead of a common quantum gate in matrix/node format, we can directly apply a gate in MPO/QuOperator format.

```
>>> x0, x1 = tc.gates.x(), tc.gates.x()
>>> mpo = tc.quantum.QuOperator([x0[0], x1[0]], [x0[1], x1[1]])
>>> c = tc.Circuit(2)
>>> c.mpo(0, 1, mpo=mpo)
>>> c.state()
array([0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j], dtype=complex64)
```

The representative gate defined in MPO format is the `multicontrol` gate.

- MPO as the operator for expectation evaluation on a circuit

We can also measure operator expectation on the circuit output state where the operator is in MPO/QuOperator format.

```
>>> z0, z1 = tc.gates.z(), tc.gates.z()
>>> mpo = tc.quantum.QuOperator([z0[0], z1[0]], [z0[1], z1[1]])
>>> c = tc.Circuit(2)
>>> c.X(0)
>>> tc.templates.measurements.mpo_expectation(c, mpo)
-1.0
```

2.1.10 Interfaces

PyTorch Interface to Hybrid with PyTorch Modules:

As we have mentioned in the backend section, the PyTorch backend may lack advanced features. This doesn't mean we cannot hybrid the advanced circuit module with PyTorch neural module. We can run the quantum function on TensorFlow or Jax backend while wrapping it with a Torch interface.

```
import tensorcircuit as tc
from tensorcircuit.interfaces import torch_interface
import torch

tc.set_backend("tensorflow")

def f(params):
    c = tc.Circuit(1)
    c.rx(0, theta=params[0])
    c.ry(0, theta=params[1])
    return c.expectation([tc.gates.z(), [0]])

f_torch = torch_interface(f, jit=True)

a = torch.ones([2], requires_grad=True)
b = f_torch(a)
c = b ** 2
c.backward()

print(a.grad)
```

For a GPU/CPU, torch/tensorflow, quantum/classical hybrid machine learning pipeline enabled by tensorcircuit, see example script.

We also provider wrapper of quantum function for torch module as `tensorcircuit.TorchLayer()` alias to `tensorcircuit.torchnn.QuantumNet()`.

For `TorchLayer`, `use_interface=True` is by default, which natively allow the quantum function defined on other tensorcircuit backends, such as jax or tf for speed consideration.

`TorchLayer` can process multiple input arguments as multiple function inputs, following torch practice.

```
n = 3
p = 0.1
K = tc.backend
torchb = tc.get_backend("pytorch")
```

(continues on next page)

(continued from previous page)

```

def f(state, noise, weights):
    c = tc.Circuit(n, inputs=state)
    for i in range(n):
        c.rz(i, theta=weights[i])
    for i in range(n):
        c.depolarizing(i, px=p, py=p, pz=p, status=noise[i])
    return K.real(c.expectation_ps(x=[0]))

layer = tc.TorchLayer(f, [n], use_vmap=True, vectorized_argnums=[0, 1])
state = torchb.ones([2, 2**n]) / 2 ** (n / 2)
noise = 0.2 * torchb.ones([2, n], dtype="float32")
l = layer(state, noise)
lsum = torchb.sum(l)
print(l)
lsum.backward()
for p in layer.parameters():
    print(p.grad)

```

TensorFlow interfaces:

Similar rules apply similar as torch interface. The interface can even be used within jit environment outside. See [tensorcircuit.interfaces.tensorflow.tensorflow_interface\(\)](#).

We also provider enable_dlpark=True option in torch and tf interfaces, which allow the tensor transformation happen without memory transfer via dlpark, higher version of tf or torch package required.

We also provider wrapper of quantum function for keras layer as `tensorcircuit.KerasLayer()` alias to `tensorcircuit.keras.KerasLayer()`.

KerasLayer can process multiple input arguments with the input as a dict, following the common keras practice, see example below.

```

def f(inputs, weights):
    state = inputs["state"]
    noise = inputs["noise"]
    c = tc.Circuit(n, inputs=state)
    for i in range(n):
        c.rz(i, theta=weights[i])
    for i in range(n):
        c.depolarizing(i, px=p, py=p, pz=p, status=noise[i])
    return K.real(c.expectation_ps(x=[0]))

layer = tc.KerasLayer(f, [n])
v = {"state": K.ones([1, 2**n]) / 2 ** (n / 2), "noise": 0.2 * K.ones([1, n])}
with tf.GradientTape() as tape:
    l = layer(v)
grad = tape.gradient(l, layer.trainable_variables)

```

Scipy Interface to Utilize Scipy Optimizers:

Automatically transform quantum functions as scipy-compatible values and grad functions as provided for scipy interface with `jac=True`.

```
n = 3
```

(continues on next page)

(continued from previous page)

```

def f(param):
    c = tc.Circuit(n)
    for i in range(n):
        c.rx(i, theta=param[0, i])
        c.rz(i, theta=param[1, i])
    loss = c.expectation(
        [
            tc.gates.y(),
            [
                0,
            ],
        ]
    )
    return tc.backend.real(loss)

f_scipy = tc.interfaces.scipy_optimize_interface(f, shape=[2, n])
r = optimize.minimize(f_scipy, np.zeros([2 * n]), method="L-BFGS-B", jac=True)

```

2.1.11 Templates as Shortcuts

Measurements:

- Ising type Hamiltonian defined on a general graph

See `tensorcircuit.templates.measurements.spin_glass_measurements()`

- Heisenberg Hamiltonian on a general graph with possible external fields

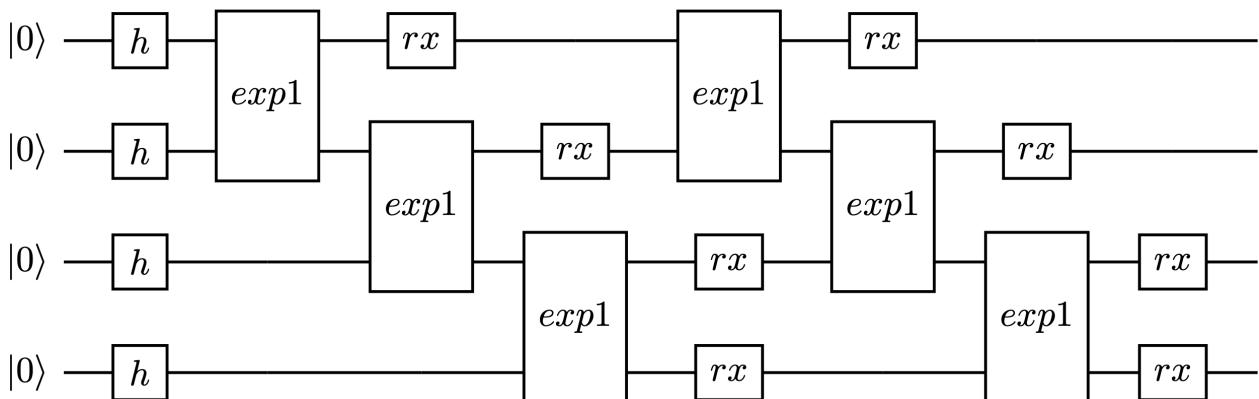
See `tensorcircuit.templates.measurements.heisenberg_measurements()`

Circuit Blocks:

```

c = tc.Circuit(4)
c = tc.templates.blocks.example_block(c, tc.backend.ones([16]))

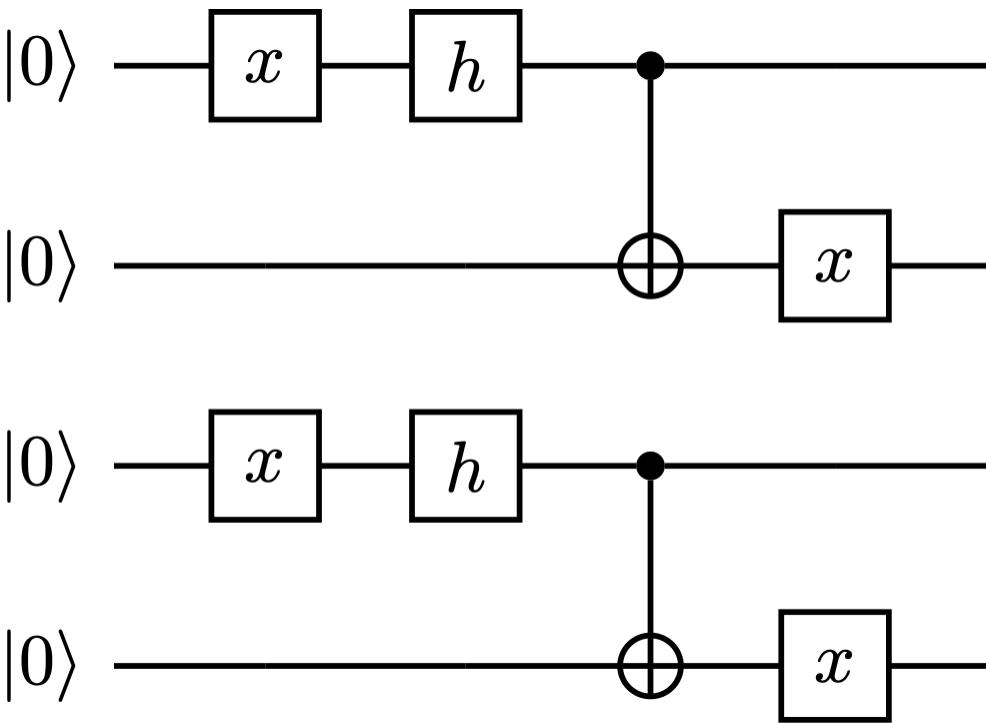
```



```

c = tc.Circuit(4)
c = tc.templates.blocks.Bell_pair_block(c)

```



2.2 Advanced Usage

2.2.1 MPS Simulator

(Still experimental support)

Very simple, we provide the same set of API for `MPSCircuit` as `Circuit`, the only new line is to set the bond dimension for the new simulator.

```
c = tc.MPSCircuit(n)
c.set_split_rules({"max_singular_values": 50})
```

The larger bond dimension we set, the better approximation ratio (of course the more computational cost we pay)

2.2.2 Split Two-qubit Gates

The two-qubit gates applied on the circuit can be decomposed via SVD, which may further improve the optimality of the contraction pathfinding.

`split` configuration can be set at circuit-level or gate-level.

```
split_conf = {
    "max_singular_values": 2, # how many singular values are kept
    "fixed_choice": 1, # 1 for normal one, 2 for swapped one
}

c = tc.Circuit(nwires, split=split_conf)

# or
```

(continues on next page)

(continued from previous page)

```
c.exp1(
    i,
    (i + 1) % nwires,
    theta=paramc[2 * j, i],
    unitary=tc.gates._zz_matrix,
    split=split_conf
)
```

Note `max_singular_values` must be specified to make the whole procedure static and thus jittable.

2.2.3 Jitted Function Save/Load

To reuse the jitted function, we can save it on the disk via support from the TensorFlow `SavedModel`. That is to say, only jitted quantum function on the TensorFlow backend can be saved on the disk.

For the JAX-backend quantum function, one can first transform them into the tf-backend function via JAX experimental support: `jax2tf`.

We wrap the tf-backend `SavedModel` as very easy-to-use function `tensorcircuit.keras.save_func()` and `tensorcircuit.keras.load_func()`.

2.2.4 Parameterized Measurements

For plain measurements API on a `tc.Circuit`, eg. `c = tc.Circuit(n=3)`, if we want to evaluate the expectation $\langle Z_1 Z_2 \rangle$, we need to call the API as `c.expectation((tc.gates.z(), [1]), (tc.gates.z(), [2]))`.

In some cases, we may want to tell the software what to measure but in a tensor fashion. For example, if we want to get the above expectation, we can use the following API: `tensorcircuit.templates.measurements.parameterized_measurements()`.

```
c = tc.Circuit(3)
z1z2 = tc.templates.measurements.parameterized_measurements(c, tc.array_to_tensor([0, 3, ↵
    3, 0]), onehot=True) # 1
```

This API corresponds to measure $I_0 Z_1 Z_2 I_3$ where 0, 1, 2, 3 are for local I, X, Y, and Z operators respectively.

2.2.5 Sparse Matrix

We support COO format sparse matrix as most backends only support this format, and some common backend methods for sparse matrices are listed below:

```
def sparse_test():
    m = tc.backend.coo_sparse_matrix(indices=np.array([[0, 1], [1, 0]]), values=np.
        array([1.0, 1.0]), shape=[2, 2])
    n = tc.backend.convert_to_tensor(np.array([[1.0], [0.0]]))
    print("is sparse: ", tc.backend.is_sparse(m), tc.backend.is_sparse(n))
    print("sparse matmul: ", tc.backend.sparse_dense_matmul(m, n))

for K in ["tensorflow", "jax", "numpy"]:
    with tc.runtime_backend(K):
```

(continues on next page)

(continued from previous page)

```
print("using backend: ", K)
sparse_test()
```

The sparse matrix is specifically useful to evaluate Hamiltonian expectation on the circuit, where sparse matrix representation has a good tradeoff between space and time. Please refer to [`tensorcircuit.templates.measurements.sparse_expectation\(\)`](#) for more detail.

For different representations to evaluate Hamiltonian expectation in tensorcircuit, please refer to [`VQE on 1D TFIM with Different Hamiltonian Representation`](#).

2.2.6 Randoms, Jit, Backend Agnostic, and Their Interplay

```
import tensorcircuit as tc
K = tc.set_backend("tensorflow")
K.set_random_state(42)

@K.jit
def r():
    return K.implicit_rndn()

print(r(), r()) # different, correct
```

```
import tensorcircuit as tc
K = tc.set_backend("jax")
K.set_random_state(42)

@K.jit
def r():
    return K.implicit_rndn()

print(r(), r()) # the same, wrong
```

```
import tensorcircuit as tc
import jax
K = tc.set_backend("jax")
key = K.set_random_state(42)

@K.jit
def r(key):
    K.set_random_state(key)
    return K.implicit_rndn()

key1, key2 = K.random_split(key)

print(r(key1), r(key2)) # different, correct
```

Therefore, a unified jittable random infrastructure with backend agnostic can be formulated as

```
import tensorcircuit as tc
import jax
K = tc.set_backend("tensorflow")
```

(continues on next page)

(continued from previous page)

```

def ba_key(key):
    if tc.backend.name == "tensorflow":
        return None
    if tc.backend.name == "jax":
        return jax.random.PRNGKey(key)
    raise ValueError("unsupported backend %s"%tc.backend.name)

@K.jit
def r(key=None):
    if key is not None:
        K.set_random_state(key)
    return K.implicit_randn()

key = ba_key(42)

key1, key2 = K.random_split(key)

print(r(key1), r(key2))

```

And a more neat approach to achieve this is as follows:

```

key = K.get_random_state(42)

@K.jit
def r(key):
    K.set_random_state(key)
    return K.implicit_randn()

key1, key2 = K.random_split(key)

print(r(key1), r(key2))

```

It is worth noting that since `Circuit.unitary_kraus` and `Circuit.general_kraus` call `implicit_rand*` API, the correct usage of these APIs is the same as above.

One may wonder why random numbers are dealt in such a complicated way, please refer to the [Jax design note](#) for some hints.

If `vmap` is also involved apart from `jit`, I currently find no way to maintain the backend agnosticity as TensorFlow seems to have no support of `vmap` over random keys (ping me on GitHub if you think you have a way to do this). I strongly recommend the users using Jax backend in the `vmap+random` setup.

2.3 Frequently Asked Questions

2.3.1 How can I run TensorCircuit on GPU?

This is done directly through the ML backend. GPU support is determined by whether ML libraries are can run on GPU, we don't handle this within tensorcircuit. It is the users' responsibility to configure a GPU-compatible environment for these ML packages. Please refer to the installation documentation for these ML packages and directly use the official dockerfiles provided by TensorCircuit. With GPU compatible environment, we can switch the use of GPU or CPU by a backend agnostic environment variable `CUDA_VISIBLE_DEVICES`.

2.3.2 When should I use GPU for the quantum simulation?

In general, for a circuit with qubit count larger than 16 or for circuit simulation with large batch dimension more than 16, GPU simulation will be faster than CPU simulation. That is to say, for very small circuits and the very small batch dimensions of vectorization, GPU may show worse performance than CPU. But one have to carry out detailed benchmarks on the hardware choice, since the performance is determined by the hardware and task details.

2.3.3 When should I jit the function?

For a function with “tensor in and tensor out”, wrapping it with jit will greatly accelerate the evaluation. Since the first time of evaluation takes longer time (staging time), jit is only good for functions which have to be evaluated frequently.

Warning: Be caution that jit can be easily misused if the users are not familiar with jit mechanism, which may lead to:

1. very slow performance due to recompiling/staging for each run,
2. error when run function with jit,
3. or wrong results without any warning.

The most possible reasons for each problem are:

1. function input are not all in the tensor form,
2. the output shape of all ops in the function may require the knowledge of the input value more than the input shape, or use mixed ops from numpy and ML framework
3. subtle interplay between random number generation and jit (see [Randoms, Jit, Backend Agnostic, and Their Interplay](#) for the correct solution), respectively.

2.3.4 Which ML framework backend should I use?

Since the Numpy backend has no support for AD, if you want to evaluate the circuit gradient, you must set the backend as one of the ML frameworks beyond Numpy.

Since PyTorch has very limited support for vectorization and jit while our package strongly depends on these features, it is not recommended to use. Though one can always wrap a quantum function on another backend using a PyTorch interface, say `tensorcircuit.interfaces.torch_interface()`.

In terms of the choice between TensorFlow and Jax backend, the better one may depend on the use cases and one may want to benchmark both to pick the better one. There is no one-for-all recommendation and this is why we maintain the backend agnostic form of our software.

Some general rules of thumb:

- On both CPU and GPU, the running time of a jitted function is faster for jax backend.
- But on GPU, jit staging time is usually much longer for jax backend.
- For hybrid machine learning tasks, TensorFlow has a better ML ecosystem and reusable classical ML models.
- Jax has some built-in advanced features that are lacking in TensorFlow, such as checkpoint in AD and pmap for distributed computing.
- Jax is much insensitive to dtype where type promotion is handled automatically which means easier debugging.
- TensorFlow can cache the jitted function on the disk via SavedModel, which further amortizes the staging time.

2.3.5 What is the counterpart of QuantumLayer for PyTorch and Jax backend?

Since PyTorch doesn't have mature vmap and jit support and Jax doesn't have native classical ML layers, we highly recommend TensorFlow as the backend for quantum-classical hybrid machine learning tasks, where `QuantumLayer` plays an important role. For PyTorch, we can in principle wrap the corresponding quantum function into a PyTorch module, we currently have the built-in support for this wrapper as `tc.TorchLayer`. In terms of the Jax backend, we highly suggested keeping the functional programming paradigm for such machine learning tasks. Besides, it is worth noting that, jit and vmap are automatically taken care of in `QuantumLayer`.

2.3.6 When do I need to customize the contractor and how?

As a rule of thumb, for the circuit with qubit counts larger than 16 and circuit depth larger than 8, customized contraction may outperform the default built-in greedy contraction strategy.

To set up or not set up the customized contractor is about a trade-off between the time on contraction pathfinding and the time on the real contraction via matmul.

The customized contractor costs much more time than the default contractor in terms of contraction path searching, and via the path it finds, the real contraction can take less time and space.

If the circuit simulation time is the bottleneck of the whole workflow, one can always try customized contractors to see whether there is some performance improvement.

We recommend to using [cotengra library](#) to set up the contractor, since there are lots of interesting hyperparameters to tune, we can achieve a better trade-off between the time on contraction path search and the time on the real tensor network contraction.

It is also worth noting that for jitted function which we usually use, the contraction path search is only called at the first run of the function, which further amortizes the time and favors the use of a highly customized contractor.

In terms of how-to on contractor setup, please refer to [Setup the Contractor](#).

2.3.7 Is there some API less cumbersome than expectation for Pauli string?

Say we want to measure something like $\langle X_0 Z_1 Y_2 Z_4 \rangle$ for a six-qubit system, the general expectation API may seem to be cumbersome. So one can try one of the following options:

- `c.expectation_ps(x=[0], y=[2], z=[1, 4])`
- `tc.templates.measurements.parameterized_measurements(c, np.array([1, 3, 2, 0, 3, 0]), onehot=True)`

2.3.8 Can I apply quantum operation based on previous classical measurement results in TensorCircuit?

Try the following: (the pipeline is even fully jittable!)

```
c = tc.Circuit(2)
c.H(0)
r = c.cond_measurement(0)
c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
```

`cond_measurement` will return 0 or 1 based on the measurement result on z-basis, and `conditional_gate` applies `gate_list[r]` on the circuit.

2.3.9 How to understand the difference between different measurement methods for Circuit?

- `tensorcircuit.circuit.Circuit.measure()` : used at the end of the circuit execution, return bitstring based on quantum amplitude probability (can also with the probability), the circuit and the output state are unaffected (no collapse). The jittable version is `measure_jit`.
- `tensorcircuit.circuit.Circuit.cond_measure()`: also with alias `cond_measurement`, usually used in the middle of the circuit execution. Apply a POVM on z basis on the given qubit, the state is collapsed and normalized based on the measurement projection. The method returns an integer Tensor indicating the measurement result 0 or 1 based on the quantum amplitude probability.
- `tensorcircuit.circuit.Circuit.post_select()`: also with alias `mid_measurement`, usually used in the middle of the circuit execution. The measurement result is fixed as given from `keep` arg of this method. The state is collapsed but unnormalized based on the given measurement projection.

Please refer to the following demos:

```
c = tc.Circuit(2)
c.H(0)
c.H(1)
print(c.measure(0, 1))
# ('01', -1.0)
print(c.measure(0, with_prob=True))
# ('0', (0.4999999657714588+0j))
print(c.state()) # unaffected
# [0.49999998+0.j 0.49999998+0.j 0.49999998+0.j 0.49999998+0.j]

c = tc.Circuit(2)
c.H(0)
c.H(1)
print(c.cond_measure(0)) # measure the first qubit return +z
# 0
print(c.state()) # collapsed and normalized
# [0.70710678+0.j 0.70710678+0.j 0.          +0.j 0.          +0.j]

c = tc.Circuit(2)
c.H(0)
c.H(1)
print(c.post_select(0, keep=1)) # measure the first qubit and it is guaranteed to return
# -z
```

(continues on next page)

(continued from previous page)

```
# 1
print(c.state()) # collapsed but unnormalized
# [0.          +0.j 0.         +0.j 0.49999998+0.j 0.49999998+0.j]
```

2.3.10 How to understand difference between `tc.array_to_tensor` and `tc.backend.convert_to_tensor`?

`tc.array_to_tensor` convert array to tensor as well as automatically cast the type to the default dtype of TensorCircuit, i.e. `tc.dtypestr` and it also support to specify dtype as `tc.array_to_tensor(, dtype="complex128")`. Instead, `tc.backend.convert_to_tensor` keeps the dtype of the input array, and to cast it as complex dtype, we have to explicitly call `tc.backend.cast` after conversion. Besides, `tc.array_to_tensor` also accepts multiple inputs as `a_tensor, b_tensor = tc.array_to_tensor(a_array, b_array)`.

2.3.11 How to arrange the circuit gate placement in the visualization from `c.tex()`?

Try `lcompress=True` or `rcompress=True` option in `tensorcircuit.circuit.Circuit.tex()` API to make the circuit align from the left or from the right.

Or try `c.unitary(0, unitary=tc.backend.eye(2), name="invisible")` to add placeholder on the circuit which is invisible for circuit visualization.

2.3.12 How to get the entanglement entropy from the circuit output?

Try the following:

```
c = tc.Circuit(4)
# omit circuit construction

rho = tc.quantum.reduced_density_matrix(s, cut=[0, 1, 2])
# get the reduced density matrix, where cut list is the index to be traced out

rho.shape
# (2, 2)

ee = tc.quantum.entropy(rho)
# get the entanglement entropy

renyi_ee = tc.quantum.renyi_entropy(rho, k=2)
# get the k-th order renyi entropy
```

2.4 TensorCircuit: The Sharp Bits

Be fast is never for free, though much cheaper in TensorCircuit, but you have to be cautious especially in terms of AD, JIT compatibility. We will go through the main sharp edges in this note.

2.4.1 Jit Compatibility

Non tensor input or varying shape tensor input

The input must be in tensor form and the input tensor shape must be fixed otherwise the recompilation is incurred which is time-consuming. Therefore, if there are input args that are non-tensor or varying shape tensors and frequently change, jit is not recommended.

```
K = tc.set_backend("tensorflow")

@K.jit
def f(a):
    print("compiling")
    return 2*a

f(K.ones([2]))
# compiling
# <tf.Tensor: shape=(2,), dtype=complex64, numpy=array([2.+0.j, 2.+0.j],  
#   dtype=complex64)>

f(K.zeros([2]))
# <tf.Tensor: shape=(2,), dtype=complex64, numpy=array([0.+0.j, 0.+0.j],  
#   dtype=complex64)>

f(K.ones([3]))
# compiling
# <tf.Tensor: shape=(3,), dtype=complex64, numpy=array([2.+0.j, 2.+0.j, 2.+0.j],  
#   dtype=complex64)>
```

Mix use of numpy and ML backend APIs

To make the function jittable and ad-aware, every ops in the function should be called via ML backend (`tc.backend` API or direct API for the chosen backend `tf` or `jax`). This is because the ML backend has to create the computational graph to carry out AD and JIT transformation. For numpy ops, they will be only called in jit staging time (the first run).

```
K = tc.set_backend("tensorflow")

@K.jit
def f(a):
    return np.dot(a, a)

f(K.ones([2]))
# NotImplementedError: Cannot convert a symbolic Tensor (a:0) to a numpy array. This  
# error may indicate that you're trying to pass a Tensor to a NumPy call, which is not  
# supported
```

Numpy call inside jitted function can be helpful if you are sure of the behavior is what you expect.

```
K = tc.set_backend("tensorflow")

@K.jit
def f(a):
    print("compiling")
    n = a.shape[0]
    m = int(np.log(n)/np.log(2))
    return K.reshape(a, [2 for _ in range(m)])

f(K.ones([4]))
# compiling
# <tf.Tensor: shape=(2, 2), dtype=complex64, numpy=
# array([[1.+0.j, 1.+0.j],
#        [1.+0.j, 1.+0.j]], dtype=complex64>

f(K.zeros([4]))
# <tf.Tensor: shape=(2, 2), dtype=complex64, numpy=
# array([[0.+0.j, 0.+0.j],
#        [0.+0.j, 0.+0.j]], dtype=complex64>

f(K.zeros([2]))
# compiling
# <tf.Tensor: shape=(2,), dtype=complex64, numpy=array([0.+0.j, 0.+0.j],  

# dtype=complex64>
```

list append under if

Append something to a Python list within if whose condition is based on tensor values will lead to wrong results. Actually values of both branch will be attached to the list. See example below.

```
K = tc.set_backend("tensorflow")

@K.jit
def f(a):
    l = []
    one = K.ones([])
    zero = K.zeros([])
    if a > 0:
        l.append(one)
    else:
        l.append(zero)
    return l

f(-K.ones([], dtype="float32"))

# [<tf.Tensor: shape=(), dtype=complex64, numpy=(1+0j)>,
# <tf.Tensor: shape=(), dtype=complex64, numpy=0j>]
```

The above code raise `ConcretizationTypeError` exception directly for Jax backend since Jax jit doesn't support tensor value if condition.

Similarly, conditional gate application must be takend carefully.

```
K = tc.set_backend("tensorflow")

@K.jit
def f():
    c = tc.Circuit(1)
    c.h(0)
    a = c.cond_measure(0)
    if a > 0.5:
        c.x(0)
    else:
        c.z(0)
    return c.state()

f()
# InaccessibleTensorError: tf.Graph captured an external symbolic tensor.

# The correct implementation is

@K.jit
def f():
    c = tc.Circuit(1)
    c.h(0)
    a = c.cond_measure(0)
    c.conditional_gate(a, [tc.gates.z(), tc.gates.x()], 0)
    return c.state()

f()
# <tf.Tensor: shape=(2,), dtype=complex64, numpy=array([0.9999994+0.j, 0.           +0.j], ...)
# dtype=complex64>
```

2.4.2 AD Consistency

TF and JAX backend manage the differentiation rules differently for complex-valued function (actually up to a complex conjugate). See issue discussion [tensorflow issue](#).

In TensorCircuit, currently we make the difference in AD transparent, namely, when switching the backend, the AD behavior and result for complex valued function can be different and determined by the nature behavior of the corresponding backend framework. All AD relevant ops such as `grad` or `jacrev` may be affected. Therefore, the user must be careful when dealing with AD on complex valued function in a backend agnostic way in TensorCircuit.

See example script on computing Jacobian with different modes on different backends: `jacobian_cal.py`. Also see the code below for a reference:

```
bks = ["tensorflow", "jax"]
n = 2
for bk in bks:
    print(bk, "backend")
    with tc.runtime_backend(bk) as K:
        def wfn(params):
            c = tc.Circuit(n)
            for i in range(n):
                c.H(i)
```

(continues on next page)

(continued from previous page)

```

for i in range(n):
    c.rz(i, theta=params[i])
    c.rx(i, theta=params[i])
return K.real(c.expectation_ps(z=[0])+c.expectation_ps(z=[1]))
print(K.grad(wfn)(K.ones([n], dtype="complex64"))) # default
print(K.grad(wfn)(K.ones([n], dtype="float32")))

# tensorflow backend
# tf.Tensor([0.90929717+0.9228758j 0.90929717+0.9228758j], shape=(2,), dtype=complex64)
# tf.Tensor([0.90929717 0.90929717], shape=(2,), dtype=float32)
# jax backend
# [0.90929747-0.9228759j 0.90929747-0.9228759j]
# [0.90929747 0.90929747]

```

2.5 TensorCircuit: What is inside?

This part of the documentation is mainly for advanced users and developers who want to learn more about what happened behind the scene and delve into the codebase.

2.5.1 Overview of Modules

Core Modules:

- `tensorcircuit.abstractcircuit` and `tensorcircuit.basecircuit`: Hierarchical abstraction of circuit class.
- `tensorcircuit.circuit`: The core object `tensorcircuit.circuit.Circuit`. It supports circuit construction, simulation, representation, and visualization without noise or with noise using the Monte Carlo trajectory approach.
- `tensorcircuit.cons`: Runtime ML backend, dtype and contractor setups. We provide three sets of set methods for global setup, function level setup using function decorators, and context setup using `with` context managers. We also include customized contractor infrastructures in this module.
- `tensorcircuit.gates`: Definition of quantum gates, either fixed ones or parameterized ones, as well as `tensorcircuit.gates.GateF` class for gates.

Backend Agnostic Abstraction:

- `tensorcircuit.backends` provides a set of backend API and the corresponding implementation on Numpy, Jax, TensorFlow, and PyTorch backends. These backends are inherited from the TensorNetwork package and are highly customized.

Noisy Simulation Related Modules:

- `tensorcircuit.channels`: Definition of quantum noise channels.
- `tensorcircuit.densitymatrix`: Referenced and highly efficient implementation of `tc.DMCircuit` class, with similar set API of `tc.Circuit` while simulating the noise in the full form of the density matrix.
- `tensorcircuit.noisemodel`: The global noise configuration and circuit noisy method APIs

ML Interfaces Related Modules:

- `tensorcircuit.interfaces`: Provide interfaces when quantum simulation backend is different from neural libraries. Currently include PyTorch and scipy optimizer interfaces.

- `tensorcircuit.keras`: Provide TensorFlow Keras layers, as well as wrappers of jitted function, save/load from tf side.
- `tensorcircuit.torchnn`: Provide PyTorch nn Modules.

MPS and MPO Utiliy Modules:

- `tensorcircuit.quantum`: Provide definition and classes for Matrix Product States as well as Matrix Product Operators, we also include various quantum physics and quantum information quantities in this module.

MPS Based Simulator Modules:

- `tensorcircuit.mps_base`: Customized and jit/AD compatible MPS class from TensorNetwork package.
- `tensorcircuit.mpscircuit`: `tensorcircuit.mpscircuit.MPS Circuit` class with similar (but subtly different) APIs as `tc.Circuit`, where the simulation engine is based on MPS TEBD.

Supplemental Modules:

- `tensorcircuit.simplify`: Provide tools and utility functions to simplify the tensornetworks before the real contractions.
- `tensorcircuit.experimental`: Experimental functions, long and stable support is not guaranteed.
- `tensorcircuit.utils`: Some general function tools that are not quantum at all.
- `tensorcircuit.vis`: Visualization code for circuit drawing.
- `tensorcircuit.translation`: Translate circuit object to circuit object in other quantum packages.

Processing and error mitigation on sample results:

- `tensorcircuit.results`: Provide tools to process count dict and to apply error mitigation

Shortcuts and Templates for Circuit Manipulation:

- `tensorcircuit.templates`: provide handy shortcuts functions for expectation or circuit building patterns.

Applications:

- `tensorcircuit.applications`: most code here is not maintained and deprecated, use at your own risk.

Note: Recommend reading order – only read the part of code you care about for your purpose. If you want to get an overview of the codebase, please read `tc.circuit` followed by `tc.cons` and `tc.gates`.

2.5.2 Relation between TensorCircuit and TensorNetwork

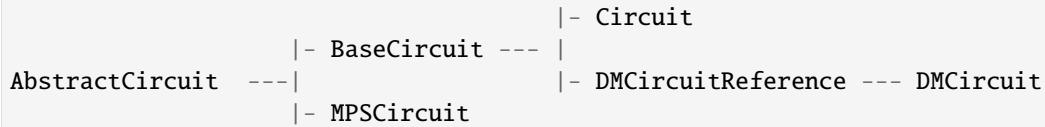
TensorCircuit has a strong connection with the [TensorNetwork package](#) released by Google. Since the TensorNetwork package has poor documentation and tutorials, most of the time, we need to delve into the codebase of TensorNetwork to figure out what happened. In other words, to read the TensorCircuit codebase, one may have to frequently refer to the TensorNetwork codebase.

Inside TensorCircuit, we heavily utilize TensorNetwork-related APIs from the TensorNetwork package and highly customized several modules from TensorNetwork by inheritance and rewriting:

- We implement our own /backends from TensorNetwork's /backends by adding much more APIs and fixing lots of bugs in TensorNetwork's implementations on certain backends via monkey patching. (The upstream is inactive and not that responsive anyhow.)
- We borrow TensorNetwork's code in /quantum to our `tc.quantum` module, since TensorNetwork has no `__init__.py` file to export these MPO and MPS related objects. Of course, we have made substantial improvements since then.

- We borrow the TensorNetwork's code in /matrixproductstates as `tc.mps_base` for bug fixing and jit/AD compatibility, so that we have better support for our MPS based quantum circuit simulator.

2.5.3 Relations of Circuit-like classes



2.5.4 QuOperator/QuVector and MPO/MPS

`tensorcircuit.quantum.QuOperator`, `tensorcircuit.quantum.QuVector` and `tensorcircuit.quantum.QuAdjointVector` are classes adopted from TensorNetwork package. They behave like a matrix/vector (column or row) when interacting with other ingredients while the inner structure is maintained by the tensor network for efficiency and compactness.

We use code examples and associated tensor diagrams to illustrate these object abstractions.

Note: `QuOperator` can express MPOs and `QuVector` can express MPSs, but they can express more than these fixed structured tensor networks.

```

import tensornetwork as tn

n1 = tn.Node(np.ones([2, 2, 2]))
n2 = tn.Node(np.ones([2, 2, 2]))
n3 = tn.Node(np.ones([2, 2]))
n1[2]^n2[2]
n2[1]^n3[0]

matrix = tc.quantum.QuOperator(out_edges=[n1[0], n2[0]], in_edges=[n1[1], n3[1]])

n4 = tn.Node(np.ones([2]))
n5 = tn.Node(np.ones([2]))

vector = tc.quantum.QuVector([n4[0], n5[0]])

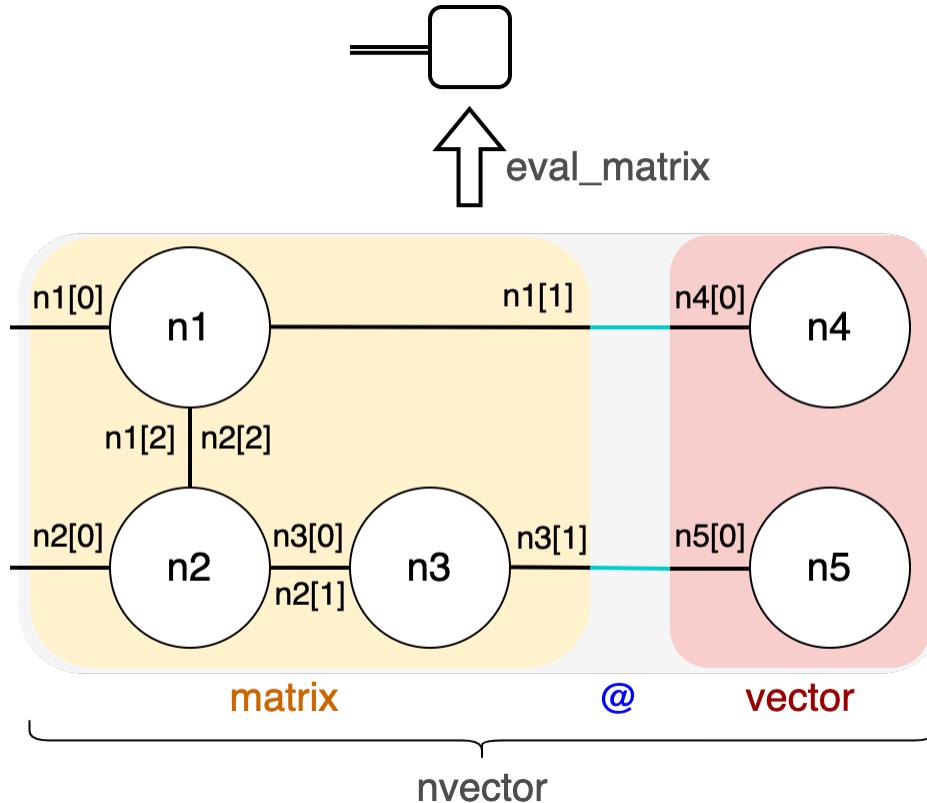
nvector = matrix @ vector

assert type(nvector) == tc.quantum.QuVector
nvector.eval_matrix()
# array([[16.], [16.], [16.], [16.]])

```

Note how in this example, `matrix` is not a typical MPO but still can be expressed as `QuOperator`. Indeed, any tensor network with two sets of dangling edges of the same dimension can be treated as `QuOperator`. `QuVector` is even more flexible since we can treat all dangling edges as the vector dimension.

Also, note how `^` is overloaded as `tn.connect` to connect edges between different nodes in TensorNetwork. And indexing the node gives the edges of the node, eg. `n1[0]` means the first edge of node `n1`.



The convention to define the QuOperator is firstly giving `out_edges` (left index or row index of the matrix) and then giving `in_edges` (right index or column index of the matrix). The edges list contains edge objects from the TensorNetwork library.

Such QuOperator/QuVector abstraction support various calculations only possible on matrix/vectors, such as `matmul` (@), adjoint (`.adjoint()`), scalar multiplication (*), tensor product (|), and partial trace (`.partial_trace(subsystems_to_trace_out)`). To extract the matrix information of these objects, we can use `.eval()` or `.eval_matrix()`, the former keeps the shape information of the tensor network while the latter gives the matrix representation with shape rank 2.

2.6 Guide for Contributors

We welcome everyone's contributions! The development of TensorCircuit is open-sourced and centered on [GitHub](#).

There are various ways to contribute:

- Answering questions on the discussions page or issue page.
- Raising issues such as bug reports or feature requests on the issue page.
- Improving the documentation (docstrings/tutorials) by pull requests.
- Contributing to the codebase by pull requests.

2.6.1 Pull Request Guidelines

We welcome pull requests from everyone. For large PRs involving feature enhancement or API changes, we ask that you first open a GitHub issue to discuss your proposal.

The following git workflow is recommended for contribution by PR:

- Configure your git username and email so that they match your GitHub account if you haven't.

```
git config user.name <GitHub name>
git config user.email <GitHub email>
```

- Fork the TensorCircuit repository by clicking the Fork button on GitHub. This will create an independent version of the codebase in your own GitHub account.
- Clone your forked repository and set up an upstream reference to the official TensorCircuit repository.

```
git clone <your-forked-repo-git-link>
cd tensorcircuit
git remote add upstream <official-repo-git-link>
```

- Configure the python environment locally for development. The following commands are recommended:

```
pip install -r requirements/requirements.txt
pip install -r requirements/requirements-dev.txt
```

Extra packages may be required for specific development tasks.

- Pip installing your fork from the source. This allows you to modify the code locally and immediately test it out.

```
python setup.py develop
```

- Create a feature branch where you can make modifications and developments. DON'T open PR from your master/main branch.

```
git checkout -b <name-of-change>
```

- Make sure your changes can pass all checks by running: `./check_all.sh`. (See the Checks section below for details)
- Once you are satisfied with your changes, create a commit as follows:

```
git add file1.py file2.py ...
git commit -m "Your commit message (should be brief and informative)"
```

- You should sync your code with the official repo:

```
git fetch upstream
git rebase upstream/master      # resolve conflicts if any
```

- Note that PRs typically comprise a single git commit, you should squash all your commits in the feature branch. Using `git rebase -i` for commits squash, see [instructions](#)
- Push your commit from your feature branch. This will create a remote branch in your forked repository on GitHub, from which you will raise a PR.

```
git push --set-upstream origin <name-of-change>
```

- Create a PR from the official TensorCircuit repository and send it for review. Some comments and remarks attached with the PR are recommended. If the PR is not finally finished, please add [WIP] at the beginning of the title of your PR.
- The PR will be reviewed by the developers and may get approved or change requested. In the latter case, you can further revise the PR according to suggestions and feedback from the code reviewers.
- The PR you opened can be automatically updated once you further push commits to your forked repository. Please remember to ping the code reviewers in the PR conversation soon.
- Please always include new docs and tests for your PR if possible and record your changes on CHANGELOG.

2.6.2 Checks

The simplest way to ensure the codebase is ok with checks and tests is to run one-in-all scripts `./check_all.sh` (you may need to `chmod +x check_all.sh` to grant permissions on this file).

The scripts include the following components:

- black
- mypy: configure file is `mypy.ini`, results strongly correlated with the version of numpy, we fix `numpy==1.21.5` as mypy standard in CI.
- pylint: configure file is `.pylintrc`
- pytest: see Pytest sections for details.
- sphinx doc builds: see Docs section for details.

Make sure the scripts check are successful by .

Similar tests and checks are also available via GitHub action as CI infrastructures.

Please also include corresponding changes for `CHANGELOG.md` and docs for the PR.

2.6.3 Pytest

For pytest, one can speed up the test by `pip install pytest-xdist`, and then run parallelly as `pytest -v -n [number of processes]`. We also have included some micro-benchmark tests, which work with `pip install pytest-benchmark`.

Fixtures:

There are some pytest fixtures defined in the `conftest` file, which are for customization on backends and dtype in function level. `highp` is a fixture for complex128 simulation. While `npb`, `tfb`, `jaxb` and `torchb` are fixtures for global numpy, tensorflow, jax and pytorch backends, respectively. To test different backends in one function, we need to use the parameterized fixture, which is enabled by `pip install pytest-lazy-fixture`. Namely, we have the following approach to test different backends in one function.

```
from pytest_lazyfixture import lazy_fixture as lf

@pytest.mark.parametrize("backend", [lf("npb"), lf("tfb"), lf("jaxb"), lf("torchb")])
def test_parameterized_backend(backend):
    print(tc.backend.name)
```

2.6.4 Docs

We use `sphinx` to manage the documentation.

The source files for docs are .rst file in docs/source.

For English docs, `sphinx-build source build/html` in docs dir is enough. The html version of the docs are in docs/build/html.

Auto Generation of API Docs:

We utilize a python script to generate/refresh all API docs rst files under /docs/source/api based on the codebase /tensorcircuit.

```
cd docs/source
python generate_rst.py
```

i18n:

For Chinese docs, we refer to the standard i18n workflow provided by sphinx, see [here](#).

To update the po file from updated English rst files, using

```
cd docs
make gettext
sphinx-intl update -p build/gettext -l zh
```

Edit these .po files to add translations (`poedit` recommended). These files are in docs/source/locale/zh/LC_MESSAGES.

To generate the Chinese version of the documentation: `sphinx-build source -D language="zh" build/html_cn` which is in the separate directory `.../build/html_cn/index.html`, whereas English version is in the directory `.../build/html/index.html`.

2.6.5 Releases

Firstly, ensure that the version numbers in `__init__.py` and `CHANGELOG` are correctly updated.

GitHub Release

```
git tag v0.x.y
git push origin v0.x.y
# assume origin is the upstream name
```

And from GitHub page choose draft a release from tag.

PyPI Release

```
python setup.py sdist bdist_wheel
export VERSION=0.x.y
twine upload dist/tensorcircuit-${VERSION}-py3-none-any.whl dist/tensorcircuit-${VERSION}
→.tar.gz
```

DockerHub Release

Make sure the DockerHub account is logged in via `docker login`.

```
sudo docker build . -f docker/Dockerfile -t tensorcircuit
sudo docker tag tensorcircuit:latest tensorcircuit/tensorcircuit:0.x.y
sudo docker push tensorcircuit/tensorcircuit:0.x.y
sudo docker tag tensorcircuit:latest tensorcircuit/tensorcircuit:latest
sudo docker push tensorcircuit/tensorcircuit:latest
```

Binder Release

One may need to update the tensorcirucit version for binder environment by pushing new commit in refraction-ray/tc-env repo with new version update in its `requriements.txt`. See [mybind setup](#) for speed up via nbgitpuller.

TUTORIALS

The following documentation sections include integrated examples in the form of Jupyter Notebook.

3.1 Jupyter Tutorials

3.1.1 Circuit Basics

Overview

In this note, we will learn about basic operations of the core object in TensorCircuit - `tc.Circuit` which supports both noiseless simulation and noisy simulation with the Monte Carlo trajectory-based method. More importantly, near all the operations on the `Circuit` object is differentiable and jittable, which is the key for successful and efficient variational quantum algorithm simulations.

[WIP note]

Setup

```
[1]: from functools import partial
import inspect
import sys
import numpy as np
import tensorflow as tf

import tensorcircuit as tc
```

Hello world example

```
[2]: def get_circuit(n):
    c = tc.Circuit(n) # initialize a circuit object with n qubits
    for i in range(n):
        c.H(i) # apply Hadamard gate on each qubit
    c.cnot(0, 1) # apply cnot with control qubit on 0-th qubit
    c.CNOT(n - 1, n - 2) # capitalized API also works
    return c
```

```
[3]: # print possible gates without parameters
print(tc.Circuit.sgates)

['i', 'x', 'y', 'z', 'h', 't', 's', 'td', 'sd', 'wroot', 'cnot', 'cz', 'swap', 'cy',
 ↪'iswap', 'ox', 'oy', 'oz', 'toffoli', 'fredkin']
```

```
[4]: # the corresponding matrix for these gates definition
for g in tc.Circuit.sgates:
    gf = getattr(tc.gates, g)
    print(g)
    print(tc.gates.matrix_for_gate(gf()))
```

```
i
[[1.+0.j 0.+0.j]
 [0.+0.j 1.+0.j]]
x
[[0.+0.j 1.+0.j]
 [1.+0.j 0.+0.j]]
y
[[0.+0.j 0.-1.j]
 [0.+1.j 0.+0.j]]
z
[[ 1.+0.j  0.+0.j]
 [ 0.+0.j -1.+0.j]]
h
[[ 0.70710677+0.j  0.70710677+0.j]
 [ 0.70710677+0.j -0.70710677+0.j]]
t
[[1.        +0.j      0.        +0.j      ]
 [0.        +0.j      0.70710677+0.70710677j]]
s
[[1.+0.j 0.+0.j]
 [0.+0.j 0.+1.j]]
td
[[1.        +0.j      0.        +0.j      ]
 [0.        +0.j      0.70710677-0.70710677j]]
sd
[[1.+0.j 0.+0.j]
 [0.+0.j 0.-1.j]]
wroot
[[ 0.70710677+0.j -0.5       -0.5j]
 [ 0.5       -0.5j   0.70710677+0.j ]]
cnot
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]]
cz
[[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j -1.+0.j]]
swap
```

(continues on next page)

(continued from previous page)

```
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
cy
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.-1.j]
 [0.+0.j 0.+0.j 0.+1.j 0.+0.j]]
iswap
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+1.j 0.+0.j]
 [0.+0.j 0.+1.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
ox
[[0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
oy
[[0.+0.j 0.-1.j 0.+0.j 0.+0.j]
 [0.+1.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
oz
[[ 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [ 0.+0.j -1.+0.j 0.+0.j 0.+0.j]
 [ 0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [ 0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
toffoli
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
fredkin
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j]]
```

```
[5]: c = get_circuit(3)
ir = c.to_qir() # intermediate representation of the circuit
ir
```

```
[5]: [{"gatef": h,
  "gate": Gate(
    name: 'h',
    tensor:
      array([[ 0.70710677+0.j,  0.70710677+0.j],
             [ 0.70710677+0.j, -0.70710677+0.j]], dtype=complex64),
  edges: [
    Edge('cnot')[2] -> 'h'[0] ),
    Edge('h')[1] -> 'qb-1'[0] )
  ]),
  'index': (0,),
  'name': 'h',
  'split': None,
  'mpo': False},
 {"gatef": h,
  "gate": Gate(
    name: 'h',
    tensor:
      array([[ 0.70710677+0.j,  0.70710677+0.j],
             [ 0.70710677+0.j, -0.70710677+0.j]], dtype=complex64),
  edges: [
    Edge('cnot')[3] -> 'h'[0] ),
    Edge('h')[1] -> 'qb-2'[0] )
  ]),
  'index': (1,),
  'name': 'h',
  'split': None,
  'mpo': False},
 {"gatef": h,
  "gate": Gate(
    name: 'h',
    tensor:
      array([[ 0.70710677+0.j,  0.70710677+0.j],
             [ 0.70710677+0.j, -0.70710677+0.j]], dtype=complex64),
  edges: [
    Edge('cnot')[2] -> 'h'[0] ),
    Edge('h')[1] -> 'qb-3'[0] )
  ]),
  'index': (2,),
  'name': 'h',
  'split': None,
  'mpo': False},
 {"gatef": cnot,
  "gate": Gate(
    name: 'cnot',
    tensor:
      array([[[[1.+0.j, 0.+0.j],
              [0.+0.j, 0.+0.j]],
             [[0.+0.j, 1.+0.j],
              [0.+0.j, 0.+0.j]]],
```

(continues on next page)

(continued from previous page)

```

[[[0.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]],

 [[0.+0.j, 0.+0.j],
 [1.+0.j, 0.+0.j]]], dtype=complex64),
edges: [
    Edge(Dangling Edge)[0],
    Edge('cnot'[3] -> 'cnot'[1]),
    Edge('cnot'[2] -> 'h'[0]),
    Edge('cnot'[3] -> 'h'[0])
],
'index': (0, 1),
'name': 'cnot',
'split': None,
'mpo': False},
{'gatef': cnot,
'gate': Gate(
    name: 'cnot',
    tensor:
        array([[[[1.+0.j, 0.+0.j],
 [0.+0.j, 0.+0.j]],

 [[0.+0.j, 1.+0.j],
 [0.+0.j, 0.+0.j]]],

 [[[0.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]],

 [[0.+0.j, 0.+0.j],
 [1.+0.j, 0.+0.j]]], dtype=complex64),
edges: [
    Edge(Dangling Edge)[0],
    Edge(Dangling Edge)[1],
    Edge('cnot'[2] -> 'h'[0]),
    Edge('cnot'[3] -> 'cnot'[1])
],
'index': (2, 1),
'name': 'cnot',
'split': None,
'mpo': False}]

```

[6]: `ir[0]["gatef"]().tensor, ir[-1]["gate"].tensor # the actually unitary for each gate`

[6]: `(array([[0.70710677+0.j, 0.70710677+0.j],
 [0.70710677+0.j, -0.70710677+0.j]], dtype=complex64),
array([[[[1.+0.j, 0.+0.j],
 [0.+0.j, 0.+0.j]],

 [[0.+0.j, 1.+0.j],
 [0.+0.j, 0.+0.j]]],`

(continues on next page)

(continued from previous page)

```
[[[0.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]],

 [[0.+0.j, 0.+0.j],
 [1.+0.j, 0.+0.j]]], dtype=complex64))
```

[7]: # compute the final output quantum state from the circuit
c.state()

[7]: array([0.35355335+0.j, 0.35355335+0.j, 0.35355335+0.j, 0.35355335+0.j,
 0.35355335+0.j, 0.35355335+0.j, 0.35355335+0.j, 0.35355335+0.j],
 dtype=complex64)

[8]: # compute some expectation values, say <X1>
x1 = c.expectation([tc.gates.x(), [1]])

or <Z1Z2>
z1z2 = c.expectation([tc.gates.z(), [1]], [tc.gates.z(), [2]])

print(x1, z1z2)
(0.9999998+0j) 0j

[9]: # make some random samples
for _ in range(10):
 print(c.perfect_sampling())

(array([0., 0., 0.], dtype=float32), 0.12499997764825821)
(array([1., 1., 0.], dtype=float32), 0.1249999776482098)
(array([1., 1., 0.], dtype=float32), 0.1249999776482098)
(array([0., 1., 0.], dtype=float32), 0.12499997764825821)
(array([1., 0., 0.], dtype=float32), 0.12499997019766829)
(array([0., 0., 1.], dtype=float32), 0.12499997764825821)
(array([1., 1., 1.], dtype=float32), 0.1250001713634208)
(array([1., 0., 0.], dtype=float32), 0.12499997019766829)
(array([0., 1., 1.], dtype=float32), 0.12499997764825821)
(array([1., 0., 1.], dtype=float32), 0.12499997019766829)

[10]: # we can easily switch simulation backends away from NumPy!

with tc.runtime_backend("tensorflow") as K:
 c = get_circuit(3)
 print(c.state())

with tc.runtime_backend("jax") as K:
 c = get_circuit(3)
 print(c.state())

with tc.runtime_backend("pytorch") as K:
 # best performance and full functionality are not guaranteed on pytorch backend
 c = get_circuit(3)
 print(c.state())

```

tf.Tensor(
[0.35355335+0.j 0.35355335+0.j 0.35355335+0.j 0.35355335+0.j
 0.35355335+0.j 0.35355335+0.j 0.35355335+0.j 0.35355335+0.j], shape=(8,),  

→dtype=complex64)

WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and  

→rerun for more info.)

[0.35355335+0.j 0.35355335+0.j 0.35355335+0.j 0.35355335+0.j
 0.35355335+0.j 0.35355335+0.j 0.35355335+0.j 0.35355335+0.j]
tensor([0.3536+0.j, 0.3536+0.j, 0.3536+0.j, 0.3536+0.j, 0.3536+0.j, 0.3536+0.j,  

→,
 0.3536+0.j])

```

Parameterized Quantum Circuit (PQC)

```
[11]: # circuit gates that accept parameters

print(tc.Circuit.vgates)

['r', 'cr', 'rx', 'ry', 'rz', 'crx', 'cry', 'crz', 'orx', 'ory', 'orz', 'any', 'exp',
→'exp1']
```

```
[12]: # see the keyword parameters (with type float) for each type of variable gate
for g in tc.Circuit.vgates:
    print(g, inspect.signature(getattr(tc.gates, g).f))

r (theta: float = 0, alpha: float = 0, phi: float = 0) -> tensorcircuit.gates.Gate
cr (theta: float = 0, alpha: float = 0, phi: float = 0) -> tensorcircuit.gates.Gate
rx (theta: float = 0) -> tensorcircuit.gates.Gate
ry (theta: float = 0) -> tensorcircuit.gates.Gate
rz (theta: float = 0) -> tensorcircuit.gates.Gate
crx (*args: Any, **kws: Any) -> Any
cry (*args: Any, **kws: Any) -> Any
crz (*args: Any, **kws: Any) -> Any
orx (*args: Any, **kws: Any) -> Any
ory (*args: Any, **kws: Any) -> Any
orz (*args: Any, **kws: Any) -> Any
any (unitary: Any, name: str = 'any') -> tensorcircuit.gates.Gate
exp (unitary: Any, theta: float, name: str = 'none') -> tensorcircuit.gates.Gate
exp1 (unitary: Any, theta: float, name: str = 'none') -> tensorcircuit.gates.Gate
```

```
[13]: def get_circuit(n, params):
    c = tc.Circuit(n) # initialize a circuit object with n qubits
    for i in range(n):
        c.rx(i, theta=params[i]) # apply rx gate
    c.cnot(0, 1)
    return c
```

```
[14]: K = tc.set_backend("tensorflow")
```

```
[15]: n = 3
params = K.ones([n])
c = get_circuit(n, params)
print(c.state())

tf.Tensor(
[ 0.6758712 +0.j      0.       -0.36923012j  0.       -0.36923015j
 -0.20171136-0.j     -0.20171136+0.j      0.       +0.11019541j
 0.       -0.36923015j -0.20171136-0.j      ], shape=(8,), dtype=complex64)
```



```
[16]: ir = c.to_qir()
ir
```



```
[16]: [{}'gatef': rx,
  'index': (0,),
  'name': 'rx',
  'split': None,
  'mpo': False,
  'parameters': {'theta': <tf.Tensor: shape=(), dtype=complex64, numpy=(1+0j)>},
  'gate': Gate(
    name: '__unnamed_node__',
    tensor:
      <tf.Tensor: shape=(2, 2), dtype=complex64, numpy=
      array([[0.87758255+0.j      , 0.       -0.47942555j],
             [0.       -0.47942555j, 0.87758255+0.j      ]], dtype=complex64)>,
    edges: [
      Edge('cnot'[2] -> '__unnamed_node__'[0]),
      Edge('__unnamed_node__'[1] -> 'qb-1'[0])
    ]),
  {}'gatef': rx,
  'index': (1,),
  'name': 'rx',
  'split': None,
  'mpo': False,
  'parameters': {'theta': <tf.Tensor: shape=(), dtype=complex64, numpy=(1+0j)>},
  'gate': Gate(
    name: '__unnamed_node__',
    tensor:
      <tf.Tensor: shape=(2, 2), dtype=complex64, numpy=
      array([[0.87758255+0.j      , 0.       -0.47942555j],
             [0.       -0.47942555j, 0.87758255+0.j      ]], dtype=complex64)>,
    edges: [
      Edge('cnot'[3] -> '__unnamed_node__'[0]),
      Edge('__unnamed_node__'[1] -> 'qb-2'[0])
    ]),
  {}'gatef': rx,
  'index': (2,),
  'name': 'rx',
  'split': None,
  'mpo': False,
  'parameters': {'theta': <tf.Tensor: shape=(), dtype=complex64, numpy=(1+0j)>},
  'gate': Gate(
    name: '__unnamed_node__',
```

(continues on next page)

(continued from previous page)

```

tensor:
<tf.Tensor: shape=(2, 2), dtype=complex64, numpy=
array([[0.87758255+0.j         , 0.           -0.47942555j],
       [0.           -0.47942555j, 0.87758255+0.j        ]], dtype=complex64)>,
edges: [
    Edge(Dangling Edge)[0],
    Edge('__unnamed_node__[1] -> 'qb-3'[0] )
],
{'gatef': cnot,
 'gate': Gate(
    name: 'cnot',
    tensor:
        <tf.Tensor: shape=(2, 2, 2, 2), dtype=complex64, numpy=
        array([[[[1.+0.j, 0.+0.j],
                 [0.+0.j, 0.+0.j]],
               [[0.+0.j, 1.+0.j],
                 [0.+0.j, 0.+0.j]]], [[[0.+0.j, 0.+0.j],
                 [0.+0.j, 1.+0.j]],
               [[0.+0.j, 0.+0.j],
                 [1.+0.j, 0.+0.j]]]], dtype=complex64)>,
    edges: [
        Edge(Dangling Edge)[0],
        Edge(Dangling Edge)[1],
        Edge('cnot'[2] -> '__unnamed_node__[0] '),
        Edge('cnot'[3] -> '__unnamed_node__[0] ')
    ],
    'index': (0, 1),
    'name': 'cnot',
    'split': None,
    'mpo': False}]

```

```
[17]: # see the gate unitary
ir[0]["gatef"](**ir[0]["parameters"]).tensor
```

```
[17]: <tf.Tensor: shape=(2, 2), dtype=complex64, numpy=
array([[0.87758255+0.j         , 0.           -0.47942555j],
       [0.           -0.47942555j, 0.87758255+0.j        ]], dtype=complex64)>
```

```
[18]: # let us compose a differentiable quantum function
```

```

def energy(params):
    c = get_circuit(n, params)
    return K.real(c.expectation([tc.gates.z(), [1]]))

energy_vag = K.value_and_grad(energy)

```

(continues on next page)

(continued from previous page)

```
print(energy_vag(params))

# once we have the gradient, we can run gradient-based descent for variational optimization

(<tf.Tensor: shape=(), dtype=float32, numpy=0.2919265>, <tf.Tensor: shape=(3,), dtype=complex64, numpy=
array([-4.5464873e-01+0.j, -4.5464873e-01+0.j, 2.2351742e-08+0.j],
      dtype=complex64>)
```

[19]: # and jit it for acceleration!

```
energy_vag_jit = K.jit(K.value_and_grad(energy))

print(energy_vag_jit(params))
# the first time to run a jitted function will be slow, but the following evaluation will be super fast

(<tf.Tensor: shape=(), dtype=float32, numpy=0.2919265>, <tf.Tensor: shape=(3,), dtype=complex64, numpy=
array([-4.5464873e-01+0.j, -4.5464873e-01+0.j, 2.2351742e-08+0.j],
      dtype=complex64>)
```

Advances for Circuit

Input State

We can replace the input state from the default $|0^n\rangle$

[20]:

```
input_state = K.ones([2**n])
input_state /= K.norm(input_state)

c = tc.Circuit(n, inputs=input_state)
c.H(0)
c.state()

[20]: <tf.Tensor: shape=(8,), dtype=complex64, numpy=
array([0.49999997+0.j, 0.49999997+0.j, 0.49999997+0.j, 0.49999997+0.j,
       0.          +0.j, 0.          +0.j, 0.          +0.j, 0.          +0.j],
      dtype=complex64>
```

Monte Carlo Noise Simulation

`tc.Circuit` supports noisy simulation using the Monte Carlo method, and it is also jittable! Besides, `tc.DMCircuit` supports noisy simulation using the full density matrix method.

```
[21]: c = tc.Circuit(n)
for i in range(n):
    c.H(i)
for i in range(n - 1):
    c.cnot(i, i + 1)
    c.depolarizing(i, px=0.1, py=0.1, pz=0.1)
    c.apply_general_kraus(tc.channels.phasedampingchannel(gamma=0.2), i + 1)
print(c.expectation([tc.gates.y(), [1]]))

tf.Tensor(0j, shape=(), dtype=complex64)
```

Apply Arbitrary Gate

Just directly using any API by feeding the corresponding unitary

```
[22]: c = tc.Circuit(n)
c.any(0, 1, unitary=K.ones([4, 4]) / K.norm(K.ones([4, 4])))
c.state()

[22]: <tf.Tensor: shape=(8,), dtype=complex64, numpy=
array([0.25+0.j, 0. -+0.j, 0.25+0.j, 0. -+0.j, 0.25+0.j, 0. -+0.j,
       0.25+0.j, 0. -+0.j], dtype=complex64)>
```

Exponential Gate

If we want to simulate gate as $e^{i\theta G}$ where $G^2 = 1$ is a matrix, we have a fast and efficient implementation for such gates as `exp1`

```
[23]: c = tc.Circuit(n)
for i in range(n):
    c.H(i)
for i in range(n - 1):
    c.exp1(i, i + 1, theta=K.ones([]), unitary=tc.gates._zz_matrix)
c.state()

[23]: <tf.Tensor: shape=(8,), dtype=complex64, numpy=
array([-0.14713009-3.2148516e-01j,  0.35355335+1.4901161e-08j,
       -0.14713009+3.2148516e-01j,  0.35355335-1.4901161e-08j,
       0.35355335-1.4901161e-08j, -0.14713009+3.2148516e-01j,
       0.35355335+1.4901161e-08j, -0.14713009-3.2148516e-01j],
      dtype=complex64)>
```

In the above example $G = Z \otimes Z$

```
[24]: print(tc.gates._zz_matrix)
```

```
[[ 1.  0.  0.  0.]
 [ 0. -1.  0. -0.]
 [ 0.  0. -1. -0.]
 [ 0. -0. -0.  1.]]
```

Common matrices in gates modules are listed below

```
[25]: for name in dir(tc.gates):
    if name.endswith("_matrix"):
        print(name, ":\n", getattr(tc.gates, name))

_cnot_matrix :
[[1.  0.  0.  0.]
 [0.  1.  0.  0.]
 [0.  0.  0.  1.]
 [0.  0.  1.  0.]]
_cy_matrix :
[[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j -0.-1.j]
 [ 0.+0.j  0.+0.j  0.+1.j  0.+0.j]]
_cz_matrix :
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0. -1.]]
_fredkin_matrix :
[[1.  0.  0.  0.  0.  0.  0.]
 [0.  1.  0.  0.  0.  0.  0.]
 [0.  0.  1.  0.  0.  0.  0.]
 [0.  0.  0.  1.  0.  0.  0.]
 [0.  0.  0.  0.  1.  0.  0.]
 [0.  0.  0.  0.  0.  1.  0.]
 [0.  0.  0.  0.  0.  0.  1.]]
_h_matrix :
[[ 0.70710678  0.70710678]
 [ 0.70710678 -0.70710678]]
_i_matrix :
[[1.  0.]
 [0.  1.]]
_ii_matrix :
[[1.  0.  0.  0.]
 [0.  1.  0.  0.]
 [0.  0.  1.  0.]
 [0.  0.  0.  1.]]
_s_matrix :
[[1.+0.j 0.+0.j]
 [0.+0.j 0.+1.j]]
_swap_matrix :
[[1.  0.  0.  0.]
 [0.  0.  1.  0.]
 [0.  1.  0.  0.]]
```

(continues on next page)

(continued from previous page)

```
[0. 0. 0. 1.]
_t_matrix :
[[1.         +0.j      0.         +0.j      ]
 [0.         +0.j      0.70710678+0.70710678j]]
_toffoli_matrix :
[[1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 1. 0.]]
_wroot_matrix :
[[ 0.70710678+0.j -0.5       -0.5j]
 [ 0.5       -0.5j  0.70710678+0.j ]]
_x_matrix :
[[0. 1.]
 [1. 0.]]
_xx_matrix :
[[0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]
 [1. 0. 0. 0.]]
_y_matrix :
[[ 0.+0.j -0.-1.j]
 [ 0.+1.j  0.+0.j]]
_yy_matrix :
[[ 0.+0.j  0.-0.j  0.-0.j -1.+0.j]
 [ 0.+0.j  0.+0.j  1.-0.j  0.-0.j]
 [ 0.+0.j  1.-0.j  0.+0.j  0.-0.j]
 [-1.+0.j  0.+0.j  0.+0.j  0.+0.j]]
_z_matrix :
[[ 1. 0.]
 [ 0. -1.]]
_zz_matrix :
[[ 1. 0. 0. 0.]
 [ 0. -1. 0. -0.]
 [ 0. 0. -1. -0.]
 [ 0. -0. -0. 1.]]
```

Non-unitary Gate

tc.Circuit automatically support non-unitary circuit simulation due to its TensorNetwork engine nature

```
[26]: c = tc.Circuit(n)
c.exp1(1, unitary=tc.gates._x_matrix, theta=K.ones([]) + 1.0j * K.ones([]))
c.state()
```

```
[26]: <tf.Tensor: shape=(8,), dtype=complex64, numpy=
array([0.83373 -0.9888977j, 0.           +0.j       ,
```

(continues on next page)

(continued from previous page)

```
0.63496387-1.2984576j, 0.        +0.j      ,
0.        +0.j      , 0.        +0.j      ,
0.        +0.j      , 0.        +0.j      ],
, dtype=complex64)>
```

Note in this case the final state is not normalized anymore

```
[27]: try:
    np.testing.assert_allclose(K.norm(c.state()), 1.0)
except AssertionError as e:
    print(e)
```

```
Not equal to tolerance rtol=1e-07, atol=0
```

```
Mismatched elements: 1 / 1 (100%)
Max absolute difference: 0.93963802
Max relative difference: 0.93963802
x: array(1.939638+0.j, dtype=complex64)
y: array(1.)
```

3.1.2 Quantum Approximation Optimization Algorithm (QAOA)

Overview

QAOA is a hybrid classical-quantum algorithm that combines quantum circuits, and classical optimization of those circuits. In this tutorial, we utilize QAOA to solve the maximum cut (MAX CUT) combinatorial optimization problem: Given a graph $G = (V, E)$ with nodes V and edges E , find a subset $S \in V$ such that the number of edges between S and $S \setminus V$ is maximized. And this problem can be reduced to that of finding the ground state of an antiferromagnetic Ising model whose Hamiltonian reads:

$$H_C = \frac{1}{2} \sum_{i,j \in E} C_{ij} \sigma_i^z \sigma_j^z,$$

where σ_i^z is the Pauli-z matrix on i th qubit and C_{ij} is the weight of the edge between nodes i and j . We set $C_{ij} = 1$ for simplicity. If $\sigma_i^z = \sigma_j^z$, $i, j \in S$ or $i, j \in S \setminus V$; if $\sigma_i^z = -\sigma_j^z$, $i \in S, j \in S \setminus V$ or $i \in S \setminus V, j \in S$. Obviously, the number of edges between S and $S \setminus V$ is maximized with the graph structure decoded from the ground state.

Setup

```
[1]: import tensorcircuit as tc
import tensorflow as tf
import networkx as nx

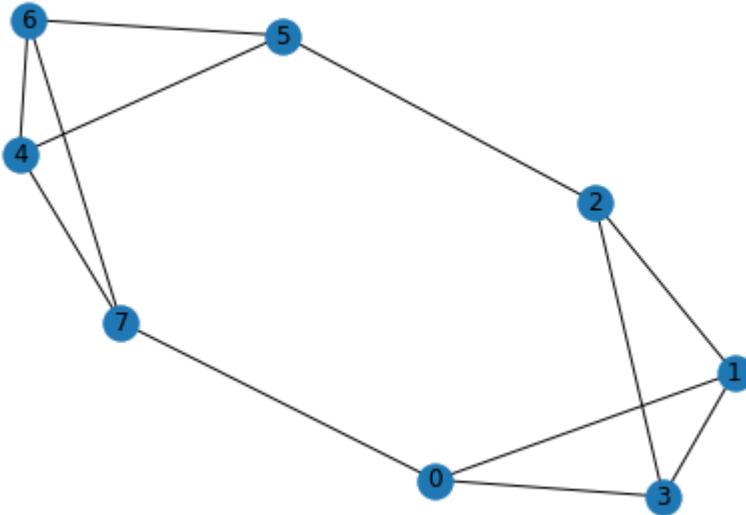
K = tc.set_backend("tensorflow")

nlayers = 3 # the number of layers
ncircuits = 2 # the number of circuits with different initial parameters
```

Define the Graph

```
[2]: def dict2graph(d):
    g = nx.to_networkx_graph(d)
    for e in g.edges:
        if not g[e[0]][e[1]].get("weight"):
            g[e[0]][e[1]]["weight"] = 1.0
    nx.draw(g, with_labels=True)
    return g

# a graph instance
# each node is connected to three nodes
# for example, node 0 is connected to nodes 1,7,3
example_graph_dict = {
    0: {1: {"weight": 1.0}, 7: {"weight": 1.0}, 3: {"weight": 1.0}},
    1: {0: {"weight": 1.0}, 2: {"weight": 1.0}, 3: {"weight": 1.0}},
    2: {1: {"weight": 1.0}, 3: {"weight": 1.0}, 5: {"weight": 1.0}},
    4: {7: {"weight": 1.0}, 6: {"weight": 1.0}, 5: {"weight": 1.0}},
    7: {4: {"weight": 1.0}, 6: {"weight": 1.0}, 0: {"weight": 1.0}},
    3: {1: {"weight": 1.0}, 2: {"weight": 1.0}, 0: {"weight": 1.0}},
    6: {7: {"weight": 1.0}, 4: {"weight": 1.0}, 5: {"weight": 1.0}},
    5: {6: {"weight": 1.0}, 4: {"weight": 1.0}, 2: {"weight": 1.0}},
}
example_graph = dict2graph(example_graph_dict)
```



Parameterized Quantum Circuit (PQC)

The PQC with p layers can be written as:

$$U(\vec{\beta}, \vec{\gamma}) = V_p U_p \cdots V_1 U_1,$$

where :math: 'U_j = e^{-i\gamma_j H_C}' and :math: 'V_j = e^{-i\beta_j \sum_k \sigma_k^x}'.

```
[3]: def QAOAansatz(params, g=example_graph):
    n = len(g.nodes) # the number of nodes
    c = tc.Circuit(n)
    for i in range(n):
        c.H(i)
    # PQC
    for j in range(nlayers):
        # U_j
        for e in g.edges:
            c.exp1(
                e[0],
                e[1],
                unitary=tc.gates._zz_matrix,
                theta=g[e[0]][e[1]].get("weight", 1.0) * params[2 * j],
            )
        # V_j
        for i in range(n):
            c.rx(i, theta=params[2 * j + 1])

    # calculate the loss function
    loss = 0.0
    for e in g.edges:
        loss += c.expectation_ps(z=[e[0], e[1]])

    return K.real(loss)
```

Main Optimization Loop

```
[4]: # use vvag to get the losses and gradients with different random circuit instances
QAOA_vvag = K.jit(tc.backend.vvag(QAOAansatz, argnums=0, vectorized_argnums=0))
```

```
[5]: params = K.implicit_randn(
    shape=[ncircuits, 2 * nlayers], stddev=0.1
) # initial parameters
opt = K.optimizer(tf.keras.optimizers.Adam(1e-2))

for i in range(50):
    loss, grads = QAOA_vvag(params, example_graph)
    print(K.numpy(loss))
    params = opt.update(grads, params) # gradient descent
[-0.23837963 -1.1651934 ]
[-0.5175445 -1.4539642]
[-0.7306818 -1.6646069]
```

(continues on next page)

(continued from previous page)

```
[-0.91530037 -1.8384367 ]
[-1.0832287 -1.9884492]
[-1.2398103 -2.120449 ]
[-1.3878661 -2.2374902]
[-1.5290209 -2.341291 ]
[-1.6642232 -2.4328852]
[-1.7940071 -2.5128942]
[-1.9186544 -2.5888019]
[-2.0382538 -2.6627793]
[-2.152771 -2.735217]
[-2.2620971 -2.8060198]
[-2.3657765 -2.8749723]
[-2.4635859 -2.942443 ]
[-2.5571456 -3.0074604]
[-2.6474872 -3.071116 ]
[-2.7343643 -3.1320357]
[-2.8174913 -3.1904984]
[-2.896546 -3.2464304]
[-2.971222 -3.298626]
[-3.0411685 -3.3485155]
[-3.1060221 -3.3945203]
[-3.1671162 -3.4365993]
[-3.2244647 -3.47741 ]
[-3.2800133 -3.51378 ]
[-3.328074 -3.5467668]
[-3.3779154 -3.5716858]
[-3.42378 -3.6026983]
[-3.4665916 -3.6264663]
[-3.5065007 -3.6452012]
[-3.5436964 -3.6676104]
[-3.5783873 -3.6827888]
[-3.6107998 -3.696251 ]
[-3.6411772 -3.710956 ]
[-3.6697989 -3.725151 ]
[-3.6969085 -3.739223 ]
[-3.7227716 -3.753837 ]
[-3.747642 -3.7637105]
[-3.7717733 -3.7778597]
[-3.7953677 -3.7897499]
[-3.8185773 -3.8026254]
[-3.8415692 -3.8186839]
[-3.864397 -3.8288355]
[-3.887118 -3.8470592]
[-3.9089546 -3.8578553]
[-3.9298224 -3.8789082]
[-3.9531326 -3.898365 ]
[-3.9759274 -3.9132624]
```

3.1.3 VQE on 1D TFIM

Overview

The main aim of this tutorial is not about the physics perspective of VQE, instead, we demonstrate the main ingredients of TensorCircuit by this simple VQE toy model.

Background

Basically, we train a parameterized quantum circuit with repetitions of $e^{i\theta}ZZ$ and $e^{i\theta X}$ layers as $U(\theta)$. And the objective to be minimized is this task is $\mathcal{L}(\theta) = \langle 0^n | U(\theta)^\dagger H U(\theta) | 0^n \rangle$. The Hamiltonian is from TFIM as $H = \sum_i Z_i Z_{i+1} - \sum_i X_i$.

Setup

```
[1]: from functools import partial
import numpy as np
import tensorflow as tf
import jax
from jax.config import config

config.update("jax_enable_x64", True)
from jax import numpy as jnp
from jax.experimental import optimizers
import tensorcircuit as tc
```

To enable automatic differentiation support, we should set the TensorCircuit backend beyond the default one “NumPy”. And we can also set the high precision complex128 for the simulation.

```
[2]: tc.set_backend("tensorflow")
tc.set_dtype("complex128")
```

```
[3]: print(
    "complex dtype of simulation:",
    tc.dtypestr,
    "\nreal dtype of simulation:",
    tc.rdtypestr,
    "\nbackend package of simulation:",
    tc.backend.name,
)

complex dtype of simulation: complex128
real dtype of simulation: float64
backend package of simulation: tensorflow
```

```
[4]: # zz gate matrix to be utilized
zz = np.kron(tc.gates._z_matrix, tc.gates._z_matrix)
print(zz)

[[ 1.  0.  0.  0.]
 [ 0. -1.  0. -0.]]
```

(continues on next page)

(continued from previous page)

```
[ 0.  0. -1. -0.]
[ 0. -0. -0.  1.]]
```

Higher-level API

We first design the Hamiltonian energy expectation function with the input as quantum circuit.

```
[5]: def tfi_energy(c: tc.Circuit, j: float = 1.0, h: float = -1.0):
    e = 0.0
    n = c._nqubits
    for i in range(n):
        e += h * c.expectation((tc.gates.x(), [i])) # <X_i>
    for i in range(n - 1): # OBC
        e += j * c.expectation(
            (tc.gates.z(), [i]), (tc.gates.z(), [(i + 1) % n])
        ) # <Z_iZ_{i+1}>
    return tc.backend.real(e)
```

Now we make the quantum function with θ as input and energy expectation \mathcal{L} as output.

```
[6]: def vqe_tfim(param, n, nlayers):
    c = tc.Circuit(n)
    paramc = tc.backend.cast(
        param, tc.dtypestr
    ) # We assume the input param with dtype float64
    for i in range(n):
        c.H(i)
    for j in range(nlayers):
        for i in range(n - 1):
            c.exp1(i, i + 1, unitary=zz, theta=paramc[2 * j, i])
        for i in range(n):
            c.rx(i, theta=paramc[2 * j + 1, i])
    e = tfi_energy(c)
    return e
```

To train the parameterized circuit, we should utilize the gradient information $\frac{\partial \mathcal{L}}{\partial \theta}$ with gradient descent. We also use `jit` to wrap the value and grad function for a substantial speedup. Note how (1, 2) args of `vqe_tfim` are labeled as static since they are just integers for qubit number and layer number instead of tensors.

```
[7]: vqe_tfim_vag = tc.backend.jit(
    tc.backend.value_and_grad(vqe_tfim), static_argnums=(1, 2)
)
```

```
[8]: def train_step_tf(n, nlayers, maxiter=10000):
    param = tf.Variable(
        initial_value=tf.random.normal(
            shape=[nlayers * 2, n], stddev=0.1, dtype=getattr(tf, tc.rdtypestr)
        )
    )
    opt = tf.keras.optimizers.Adam(1e-2)
    for i in range(maxiter):
```

(continues on next page)

(continued from previous page)

```

e, grad = vqe_tfim_vvag(param, n, nlayers)
opt.apply_gradients([(grad, param)])
if i % 200 == 0:
    print(e)
return e

train_step_tf(6, 3, 2000)

tf.Tensor(-5.2044235531710905, shape=(), dtype=float64)
tf.Tensor(-7.168460907768175, shape=(), dtype=float64)
tf.Tensor(-7.229007202330065, shape=(), dtype=float64)
tf.Tensor(-7.2368387790165105, shape=(), dtype=float64)
tf.Tensor(-7.246179597523659, shape=(), dtype=float64)
tf.Tensor(-7.262673966580785, shape=(), dtype=float64)
tf.Tensor(-7.286129364991173, shape=(), dtype=float64)
tf.Tensor(-7.291252895716095, shape=(), dtype=float64)
tf.Tensor(-7.2930457160020765, shape=(), dtype=float64)
tf.Tensor(-7.293225326335964, shape=(), dtype=float64)

[8]: <tf.Tensor: shape=(), dtype=float64, numpy=-7.293297606006469>

```

Batched VQE Example

We can even run a batched version of VQE optimization, namely, we simultaneously optimize parameterized circuits for different random initializations, so that we can try our best to avoid local minimums and locate the best of the converged energies.

```
[9]: vqe_tfim_vvag = tc.backend.jit(
    tc.backend.vectorized_value_and_grad(vqe_tfim), static_argnums=(1, 2)
)
```

```
[10]: def batched_train_step_tf(batch, n, nlayers, maxiter=10000):
    param = tf.Variable(
        initial_value=tf.random.normal(
            shape=[batch, nlayers * 2, n], stddev=0.1, dtype=getattr(tf, tc.rdtypestr)
        )
    )
    opt = tf.keras.optimizers.Adam(1e-2)
    for i in range(maxiter):
        e, grad = vqe_tfim_vvag(param, n, nlayers)
        opt.apply_gradients([(grad, param)])
        if i % 200 == 0:
            print(e)
    return e

batched_train_step_tf(16, 6, 3, 2000)

tf.Tensor
[-4.56780182 -5.32411397 -5.34948039 -5.49728838 -5.51974631 -4.89464895
 -5.23113926 -5.70097167 -5.4384308 -5.27898261 -4.73926061 -5.43748391
```

(continues on next page)

(continued from previous page)

```

-5.02246224 -4.46749643 -5.34320604 -5.29828815], shape=(16,), dtype=float64)
tf.Tensor(
[-7.15906597 -7.20867528 -7.16615816 -7.16164269 -7.15427498 -7.17176534
 -7.15677645 -7.19769858 -7.1876547 -7.17160745 -7.14313137 -7.16458417
 -7.12556993 -7.1043696 -7.17233218 -7.17955502], shape=(16,), dtype=float64)
tf.Tensor(
[-7.22332735 -7.28775096 -7.22854626 -7.28800389 -7.22006811 -7.2773814
 -7.22241623 -7.23446324 -7.23115651 -7.23081143 -7.25399986 -7.26564648
 -7.16463543 -7.27854832 -7.23574558 -7.28935649], shape=(16,), dtype=float64)
tf.Tensor(
[-7.23956454 -7.29093555 -7.23464822 -7.2914774 -7.22326999 -7.29014637
 -7.24891067 -7.2505597 -7.23879431 -7.23826618 -7.28737831 -7.29193732
 -7.22649018 -7.29136679 -7.25276205 -7.29214669], shape=(16,), dtype=float64)
tf.Tensor(
[-7.24561853 -7.29413883 -7.23950499 -7.29230127 -7.22749993 -7.29051998
 -7.28702174 -7.289441 -7.25016979 -7.26370483 -7.29320874 -7.29451577
 -7.22882824 -7.29213765 -7.27040912 -7.29358236], shape=(16,), dtype=float64)
tf.Tensor(
[-7.24997971 -7.294748 -7.25420008 -7.29271584 -7.24577837 -7.29082466
 -7.29171805 -7.29016935 -7.28645108 -7.29170429 -7.29499124 -7.29520514
 -7.23115011 -7.29305292 -7.28793637 -7.2949226 ], shape=(16,), dtype=float64)
tf.Tensor(
[-7.25300306 -7.29512508 -7.28240557 -7.29287622 -7.28264095 -7.29125472
 -7.29399162 -7.29066326 -7.29233232 -7.29290676 -7.29521188 -7.29530935
 -7.23475933 -7.29429836 -7.29053038 -7.29559969], shape=(16,), dtype=float64)
tf.Tensor(
[-7.25706762 -7.29527205 -7.29168082 -7.29292216 -7.29221412 -7.29183888
 -7.29474989 -7.29119684 -7.29306204 -7.29300514 -7.29525079 -7.29538907
 -7.24226472 -7.29544118 -7.29086393 -7.29576418], shape=(16,), dtype=float64)
tf.Tensor(
[-7.26218683 -7.29529443 -7.29438024 -7.29294969 -7.29426403 -7.29243576
 -7.29491073 -7.29171828 -7.29384631 -7.29301505 -7.29527304 -7.29545727
 -7.25772904 -7.29581457 -7.2910381 -7.29582192], shape=(16,), dtype=float64)
tf.Tensor(
[-7.26654138 -7.29529938 -7.29515899 -7.29297612 -7.29499748 -7.29281659
 -7.29500531 -7.29227399 -7.29455691 -7.29302087 -7.29529063 -7.29551808
 -7.2892432 -7.29593328 -7.29120759 -7.29585771], shape=(16,), dtype=float64)

[10]: <tf.Tensor: shape=(16,), dtype=float64, numpy=
array([-7.29011428, -7.29530356, -7.29549915, -7.29300424, -7.29529224,
       -7.29296047, -7.29508027, -7.29270021, -7.29499648, -7.29302725,
       -7.29530574, -7.2955733 , -7.29593608, -7.29604964, -7.29138482,
       -7.29589095])>

```

Different Backends

We can change the backends at runtime without even changing one line of the code!

However, in normal user cases, we strongly recommend the users stick to one backend in one jupyter or python script. One can enjoy the facility provided by other backends by changing the `set_backend` line and running the same script again. This approach is much safer than using multiple backends in the same file unless you know the lower-level details of TensorCircuit enough.

```
[11]: tc.set_backend("jax") # change to jax backend

[12]: vqe_tfim_vvag = tc.backend.jit(
        tc.backend.vectorized_value_and_grad(vqe_tfim), static_argnums=(1, 2)
    )

def batched_train_step_jax(batch, n, nlayers, maxiter=10000):

    key = jax.random.PRNGKey(42)
    param = jax.random.normal(key, shape=[batch, nlayers * 2, n]) * 0.1
    opt_init, opt_update, get_params = optimizers.adam(step_size=1e-2)
    opt_state = opt_init(param)

    def update(i, opt_state):
        param = get_params(opt_state)
        (value, gradient) = vqe_tfim_vvag(param, n, nlayers)
        return value, opt_update(i, gradient, opt_state)

    for i in range(maxiter):
        value, opt_state = update(i, opt_state)
        param = get_params(opt_state)
        if i % 200 == 0:
            print(value)
    return value

batched_train_step_jax(16, 6, 3, 2000)
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
         --rerun for more info.)
```

[-5.67575948 -5.44768444 -5.7821556 -5.36699503 -5.00485098 -5.59416181
-5.13421084 -5.70462279 -5.73699416 -5.25819658 -4.70729299 -5.82823766
-5.69154358 -5.51112311 -5.46091316 -5.31649863]
[-7.16831387 -7.17873365 -7.21905991 -7.17714641 -7.21910053 -7.17729778
-7.23594046 -7.1978075 -7.2311691 -7.18566164 -7.15141273 -7.1760751
-7.20727055 -7.22174427 -7.15227955 -7.15343225]
[-7.24047827 -7.23486717 -7.26382185 -7.25267406 -7.23938877 -7.24135079
-7.28655961 -7.24413064 -7.28070556 -7.24825735 -7.23400189 -7.25234153
-7.25756263 -7.2505181 -7.22647645 -7.2589444]
[-7.28642159 -7.23707926 -7.28988032 -7.28627451 -7.28716418 -7.25068739
-7.29122589 -7.2510777 -7.2906953 -7.25976327 -7.23891735 -7.29227009
-7.28973637 -7.26238069 -7.245065 -7.29155041]
[-7.29198674 -7.24196434 -7.29188725 -7.29243688 -7.2926968 -7.26254168]

(continues on next page)

(continued from previous page)

```
-7.29233808 -7.26729904 -7.29277165 -7.28066403 -7.24315235 -7.29344766
-7.2920645 -7.26717433 -7.26959622 -7.29307748]
[-7.29320541 -7.27162341 -7.29245991 -7.2934821 -7.29360574 -7.27103573
-7.29311302 -7.29213505 -7.29356392 -7.29162927 -7.24981922 -7.29384423
-7.2930642 -7.27323089 -7.29156252 -7.293684 ]
[-7.29384132 -7.29130123 -7.29333191 -7.29442748 -7.29396942 -7.29011734
-7.2936549 -7.2929216 -7.29430937 -7.29288465 -7.27758063 -7.29446428
-7.2939575 -7.27958774 -7.29260011 -7.29408982]
[-7.29433271 -7.29273603 -7.29467579 -7.29507754 -7.29438517 -7.29298445
-7.2940647 -7.29297896 -7.29494014 -7.29348052 -7.29142477 -7.29524322
-7.29455267 -7.28904449 -7.29319333 -7.29437335]
[-7.29480095 -7.29294578 -7.29507043 -7.29530148 -7.29479745 -7.29390014
-7.29439439 -7.29304737 -7.29520739 -7.29397645 -7.29276852 -7.2955135
-7.29492853 -7.29145529 -7.29390729 -7.29462868]
[-7.29530498 -7.29299558 -7.29520643 -7.29537379 -7.29505127 -7.2945667
-7.29466477 -7.29312996 -7.29529573 -7.29447499 -7.29327567 -7.29560796
-7.29520605 -7.29179062 -7.29484071 -7.29489978]
```

[12]: DeviceArray([-7.29575181, -7.29302602, -7.29529976, -7.29541094,
-7.29517778, -7.29488194, -7.29487651, -7.29323608,
-7.29532772, -7.29494698, -7.29369784, -7.29567791,
-7.29534388, -7.29187906, -7.29536221, -7.29516005],
→ dtype=float64)

Lower-level API

The higher-level API under the namespace of `TensorCircuit` provides a unified framework to do linear algebra and automatic differentiation which is backend agnostic.

One may also use the related APIs (ops, AD-related, jit-related) directly provided by TensorFlow or Jax, as long as one is ok to stick with one fixed backend. See the tensorflow backend example below.

[13]: `tc.set_backend("tensorflow")`

```
def tfi_energy(c: tc.Circuit, j: float = 1.0, h: float = -1.0):
    e = 0.0
    n = c._nqubits
    for i in range(n):
        e += h * c.expectation((tc.gates.x(), [i])) # <X_i>
    for i in range(n - 1): # OBC
        e += j * c.expectation(
            (tc.gates.z(), [i]), (tc.gates.z(), [(i + 1) % n])
        ) # <Z_iZ_{i+1}>
    return tf.math.real(e)

def vqe_tfim(param, n, nlayers):
    c = tc.Circuit(n)
    paramc = tf.cast(param, tf.complex128)
    for i in range(n):
        c.H(i)
    for j in range(nlayers):
```

(continues on next page)

(continued from previous page)

```

for i in range(n - 1):
    c.exp1(i, i + 1, unitary=zz, theta=paramc[2 * j, i])
for i in range(n):
    c.rx(i, theta=paramc[2 * j + 1, i])
e = tfi_energy(c)
return e

@tf.function
def vqe_tfim_vag(param, n, nlayers):
    with tf.GradientTape() as tape:
        tape.watch(param)
        v = vqe_tfim(param, n, nlayers)
    grad = tape.gradient(v, param)
    return v, grad

```

[15]: train_step_tf(6, 3, 2000)

```

tf.Tensor(-5.5454151788179376, shape=(), dtype=float64)
tf.Tensor(-7.167693061786028, shape=(), dtype=float64)
tf.Tensor(-7.254761404891117, shape=(), dtype=float64)
tf.Tensor(-7.290050014550046, shape=(), dtype=float64)
tf.Tensor(-7.29133881232428, shape=(), dtype=float64)
tf.Tensor(-7.2918048286324915, shape=(), dtype=float64)
tf.Tensor(-7.292590929769901, shape=(), dtype=float64)
tf.Tensor(-7.294195015132205, shape=(), dtype=float64)
tf.Tensor(-7.295013538531699, shape=(), dtype=float64)
tf.Tensor(-7.2951174084838835, shape=(), dtype=float64)

```

[15]: <tf.Tensor: shape=(), dtype=float64, numpy=-7.295170441938537>

3.1.4 QML on MNIST Classification

Overview

The aim of this tutorial is not about the machine learning perspective on better design of QML method for MNIST classification. Instead, we use a simple parameterized circuit and demonstrate the QML-related technical ingredients of `tensorcircuit`. Nevertheless, this note is by no means a good practice on QML.

[WIP note]

Setup

```

[1]: from functools import partial
import numpy as np
import tensorflow as tf
import jax
from jax.config import config

config.update("jax_enable_x64", True)
from jax import numpy as jnp

```

(continues on next page)

(continued from previous page)

```
import optax
import tensorcircuit as tc
```

[2]: tc.set_backend("tensorflow")
tc.set_dtype("complex128")

[2]: ('complex128', 'float64')

Data Processing

We utilize MNIST data and resize them to 3*3 to fit into a 9-qubit circuit. The testbed we use is a binary classification task, digit 1 vs. 5. And since this tutorial is not about good practice on QML, we leave the validation set away. And we only collect 100 data points for a small demo.

```
# numpy data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train = x_train[..., np.newaxis] / 255.0

def filter_pair(x, y, a, b):
    keep = (y == a) | (y == b)
    x, y = x[keep], y[keep]
    y = y == a
    return x, y

x_train, y_train = filter_pair(x_train, y_train, 1, 5)
x_train_small = tf.image.resize(x_train, (3, 3)).numpy()
x_train_bin = np.array(x_train_small > 0.5, dtype=np.float32)
x_train_bin = np.squeeze(x_train_bin)[:100]
```

```
# tensorflow data
x_train_tf = tf.reshape(tf.constant(x_train_bin, dtype=tf.float64), [-1, 9])
y_train_tf = tf.constant(y_train[:100], dtype=tf.float64)

# jax data
x_train_jax = jnp.array(x_train_bin, dtype=np.float64).reshape([100, -1])
y_train_jax = jnp.array(y_train[:100], dtype=np.float64).reshape([100])

WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
rerun for more info.)
```

Using vectorized_value_and_grad API

```
[5]: nlayers = 3

def qml_loss(x, y, weights, nlayers):
    n = 9
    weights = tc.backend.cast(weights, "complex128")
    x = tc.backend.cast(x, "complex128")
    c = tc.Circuit(n)
    for i in range(n):
        c.rx(i, theta=x[i])
    for j in range(nlayers):
        for i in range(n - 1):
            c.cnot(i, i + 1)
        for i in range(n):
            c.rx(i, theta=weights[2 * j, i])
            c.ry(i, theta=weights[2 * j + 1, i])
    ypred = c.expectation([tc.gates.z(), (4,)])
    ypred = tc.backend.real(ypred)
    ypred = (tc.backend.real(ypred) + 1) / 2.0
    return -y * tc.backend.log(ypred) - (1 - y) * tc.backend.log(1 - ypred), ypred
```

```
[6]: def get_qml_vvag():
    qml_vvag = tc.backend.vectorized_value_and_grad(
        qml_loss, argnums=(2,), vectorized_argnums=(0, 1), has_aux=True
    )
    qml_vvag = tc.backend.jit(qml_vvag, static_argnums=(3,))
    return qml_vvag

qml_vvag = get_qml_vvag()
qml_vvag(x_train_tf, y_train_tf, tf.ones([nlayers * 2, 9], dtype=tf.float64), nlayers)
```

```
[6]: ((<tf.Tensor: shape=(100,), dtype=float64, numpy=
array([0.8433698 , 0.56257199, 0.54653163, 0.56257199, 0.82036163,
       0.56257199, 0.56257199, 0.58030506, 0.82036163, 0.56257199,
       0.82036163, 0.56257199, 0.82036163, 0.56257199, 0.54653163,
       0.54653163, 0.56257199, 0.56257199, 0.58030506, 0.82036163,
       0.54653163, 0.56257199, 0.56257199, 0.56257199, 0.56257199,
       0.56257199, 0.56257199, 0.85182866, 0.56257199, 0.82036163,
       0.82036163, 0.56257199, 0.8433698 , 0.56257199, 0.8433698 ,
       0.56257199, 0.85182866, 0.56257199, 0.82036163, 0.54653163,
       0.56257199, 0.56257199, 0.56257199, 0.56257199, 0.8433698 ,
       0.58030506, 0.56257199, 0.82036163, 0.8433698 , 0.8433698 ,
       0.54653163, 0.56257199, 0.82036163, 0.86501404, 0.56257199,
       0.56257199, 0.8433698 , 0.56257199, 0.85182866, 0.82036163,
       0.82036163, 0.56257199, 0.82036163, 0.56257199, 0.56257199,
       0.56257199, 0.82036163, 0.8433698 , 0.8433698 , 0.82036163,
       0.56257199, 0.56257199, 0.56257199, 0.56257199, 0.56257199,
       0.54653163, 0.86501404, 0.54653163, 0.54653163, 0.82036163,
       0.56257199, 0.54653163, 0.8433698 , 0.54653163, 0.8433698 ,
       0.56257199, 0.56257199, 0.8433698 , 0.82036163, 0.8433698 ,
```

(continues on next page)

(continued from previous page)

```
[7]: # %timeit qml_vvag(x_train_tf, y_train_tf, tf.ones([nlayers*2, 9], dtype=tf.float64),  
#                         nlayers)
```

Jax Backend Compatibility

```
[8]: tc.set_backend("jax")
[8]: <tensorcircuit.backends.jax_backend.JaxBackend at 0x7ffb04a71820>

[9]: qml_vvag = get_qml_vvag()
      qml_vvag(
          x_train_jax, y_train_jax, jnp.ones([nlayers * 2, 9], dtype=np.float64), nlayers
      )
[9]: ((DeviceArray([0.8433698 , 0.56257199, 0.54653163, 0.56257199, 0.82036163,
                  0.56257199, 0.56257199, 0.58030506, 0.82036163, 0.56257199,
                  0.82036163, 0.56257199, 0.82036163, 0.56257199, 0.54653163,
                  0.54653163, 0.56257199, 0.56257199, 0.58030506, 0.82036163,
                  0.54653163, 0.56257199, 0.56257199, 0.56257199, 0.56257199,
                  0.56257199, 0.56257199, 0.85182866, 0.56257199, 0.82036163,
                  0.82036163, 0.56257199, 0.8433698 , 0.56257199, 0.8433698 ,
                  0.56257199, 0.85182866, 0.56257199, 0.82036163, 0.54653163,
                  0.56257199, 0.56257199, 0.56257199, 0.56257199, 0.8433698 ,
                  0.58030506, 0.56257199, 0.82036163, 0.8433698 , 0.8433698 ,
                  0.54653163, 0.56257199, 0.82036163, 0.86501404, 0.56257199,
                  0.56257199, 0.8433698 , 0.56257199, 0.85182866, 0.82036163,
                  0.82036163, 0.56257199, 0.82036163, 0.56257199, 0.56257199,
                  0.56257199, 0.82036163, 0.8433698 , 0.8433698 , 0.82036163,
                  0.56257199, 0.56257199, 0.56257199, 0.56257199, 0.56257199,
                  0.54653163, 0.86501404, 0.54653163, 0.54653163, 0.82036163,
                  0.56257199, 0.54653163, 0.8433698 , 0.54653163, 0.8433698 ,
                  0.56257199, 0.56257199, 0.8433698 , 0.82036163, 0.8433698 ,
                  0.56257199, 0.56257199, 0.56257199, 0.56257199, 0.56257199,
                  0.82036163, 0.56257199, 0.58030506, 0.8433698 , 0.56257199],
      dtype=float64),
      DeviceArray([0.56974181, 0.56974181, 0.57895436, 0.56974181, 0.55972759,
                  0.56974181, 0.56974181, 0.55972759, 0.55972759, 0.56974181,
                  0.55972759, 0.56974181, 0.55972759, 0.56974181, 0.57895436,
                  0.57895436, 0.56974181, 0.56974181, 0.55972759, 0.55972759,
                  0.57895436, 0.56974181, 0.56974181, 0.56974181, 0.56974181,
                  0.56974181, 0.56974181, 0.57336595, 0.56974181, 0.55972759,
                  0.55972759, 0.56974181, 0.56974181, 0.56974181, 0.56974181,
                  0.56974181, 0.56974181, 0.55972759, 0.56974181, 0.56974181,
                  0.56974181, 0.56974181, 0.56974181, 0.56974181, 0.56974181,
                  0.56974181, 0.57336595, 0.56974181, 0.55972759, 0.57895436,
                  0.56974181, 0.56974181, 0.56974181, 0.56974181, 0.56974181,
                  0.55972759, 0.56974181, 0.55972759, 0.56974181, 0.56974181,
                  0.55972759, 0.56974181, 0.55972759, 0.57895436, 0.56974181,
                  0.56974181, 0.56974181, 0.56974181, 0.57336595, 0.55972759,
                  0.55972759, 0.56974181, 0.55972759, 0.56974181, 0.56974181,
                  0.56974181, 0.55972759, 0.56974181, 0.56974181, 0.55972759,
                  0.56974181, 0.56974181, 0.56974181, 0.56974181, 0.56974181,
                  0.56974181, 0.56974181, 0.56974181, 0.56974181, 0.56974181,
                  0.56974181, 0.56974181, 0.56974181, 0.56974181, 0.56974181,
                  0.56974181, 0.56974181, 0.55972759, 0.56974181, 0.56974181,
                  0.57895436, 0.57895436, 0.57895436, 0.57895436, 0.55972759,
                  0.56974181, 0.57895436, 0.56974181, 0.57895436, 0.56974181,
                  0.56974181, 0.56974181, 0.56974181, 0.55972759, 0.56974181,
                  0.56974181, 0.56974181, 0.56974181, 0.56974181, 0.56974181,
                  0.55972759, 0.56974181, 0.55972759, 0.56974181, 0.56974181]),
      dtype=float64)),
```

(continues on next page)

(continued from previous page)

```
(DeviceArray([[ 5.79464357e-02,  1.12182823e-01,  8.13605755e-02,
              1.52611620e-01,  1.13641690e+00, -1.41695736e+00,
              7.62883290e-01,  1.01307851e-15, -1.03389519e-15],
             [ 3.57155554e-02,  1.61488509e-01,  7.62331819e-02,
              1.50335863e-01, -1.10363460e-01, -3.23606686e-01,
              -3.14523756e-01, -4.09394740e-16, -1.65145675e-15],
             [-3.04959149e-02,  6.03271869e-02,  2.47760477e-02,
              -7.61859417e-02,  1.72064441e+00,  1.66891120e+00,
              6.24500451e-16,  9.71445147e-16, -8.39606162e-16],
             [-8.26503466e-03, -6.90338030e-02, -1.07589110e-01,
              1.88650816e-01, -2.68228700e+00, -2.41159987e+00,
              -6.66133815e-16,  5.27355937e-16, -7.56339436e-16],
             [-1.08246745e-15,  1.65839564e-15, -2.77555756e-16,
              1.38777878e-17,  4.02608813e-01, -7.14706072e-16,
              -4.16333634e-17, -9.02056208e-16, -9.57567359e-16],
             [-1.08940634e-15,  3.26128013e-16, -5.34294831e-16,
              6.93889390e-18, -1.11152817e+00,  3.19189120e-16,
              -1.68615122e-15,  1.249000090e-16,  1.79717352e-15]], ),
             dtype=float64), ))
```

```
[10]: # %timeit qml_vvag(x_train_jax, y_train_jax, jnp.ones([nlayers * 2, 9], dtype=np.
             ↪float64), nlayers)
```

Training Using tf.data

```
[11]: # switch back to tensorflow
tc.set_backend("tensorflow")
qml_vvag = get_qml_vvag()
qml_vvag = tc.backend.jit(qml_vvag, static_argnums=(3,))
```

```
[12]: mnist_data = (
    tf.data.Dataset.from_tensor_slices((x_train_tf, y_train_tf))
    .repeat(200)
    .shuffle(100)
    .batch(32)
)
```

```
[13]: opt = tf.keras.optimizers.Adam(1e-2)
w = tf.Variable(
    initial_value=tf.random.normal(shape=(2 * nlayers, 9), stddev=0.5, dtype=tf.float64)
)
for i, (xs, ys) in zip(range(2000), mnist_data):
    (losses, ypreds), grad = qml_vvag(xs, ys, w, nlayers)
    if i % 20 == 0:
        print(tf.reduce_mean(losses))
        opt.apply_gradients([(grad[0], w)])
tf.Tensor(0.689301607482696, shape=(), dtype=float64)
tf.Tensor(0.6825438352666904, shape=(), dtype=float64)
tf.Tensor(0.6815497367036047, shape=(), dtype=float64)
```

(continues on next page)

(continued from previous page)

```
tf.Tensor(0.6632433448327015, shape=(), dtype=float64)
tf.Tensor(0.6641348270253142, shape=(), dtype=float64)
tf.Tensor(0.6779914200102861, shape=(), dtype=float64)
tf.Tensor(0.6550256969249619, shape=(), dtype=float64)
tf.Tensor(0.6801325087248677, shape=(), dtype=float64)
tf.Tensor(0.6190616725052769, shape=(), dtype=float64)
tf.Tensor(0.6711760566099414, shape=(), dtype=float64)
tf.Tensor(0.6965496746836946, shape=(), dtype=float64)
tf.Tensor(0.6443036572691725, shape=(), dtype=float64)
tf.Tensor(0.6060956714527996, shape=(), dtype=float64)
tf.Tensor(0.6728839286340991, shape=(), dtype=float64)
tf.Tensor(0.6584085272471567, shape=(), dtype=float64)
tf.Tensor(0.6600981577311038, shape=(), dtype=float64)
tf.Tensor(0.6581071758186605, shape=(), dtype=float64)
tf.Tensor(0.6609348320181809, shape=(), dtype=float64)
tf.Tensor(0.5919640703180435, shape=(), dtype=float64)
tf.Tensor(0.6362392080775805, shape=(), dtype=float64)
tf.Tensor(0.6844038809425064, shape=(), dtype=float64)
tf.Tensor(0.6924617230085226, shape=(), dtype=float64)
tf.Tensor(0.6594653043250199, shape=(), dtype=float64)
tf.Tensor(0.7076707818117074, shape=(), dtype=float64)
tf.Tensor(0.6730725215608222, shape=(), dtype=float64)
tf.Tensor(0.6565711271336594, shape=(), dtype=float64)
tf.Tensor(0.6665226844123278, shape=(), dtype=float64)
tf.Tensor(0.6368469891760338, shape=(), dtype=float64)
tf.Tensor(0.6499572506552256, shape=(), dtype=float64)
tf.Tensor(0.6110576844713855, shape=(), dtype=float64)
tf.Tensor(0.6312147945757532, shape=(), dtype=float64)
tf.Tensor(0.6013772883771527, shape=(), dtype=float64)
```

Using `tf.keras` API

[14]: `from tensorcircuit import keras`

```
def qml_y(x, weights, nlayers):
    n = 9
    weights = tc.backend.cast(weights, "complex128")
    x = tc.backend.cast(x, "complex128")
    c = tc.Circuit(n)
    for i in range(n):
        c.rx(i, theta=x[i])
    for j in range(nlayers):
        for i in range(n - 1):
            c.cnot(i, i + 1)
        for i in range(n):
            c.rx(i, theta=weights[2 * j, i])
            c.ry(i, theta=weights[2 * j + 1, i])
    ypred = c.expectation([tc.gates.z(), (4,)])
    ypred = tc.backend.real(ypred)
```

(continues on next page)

(continued from previous page)

```
ypred = (tc.backend.real(ypred) + 1) / 2.0
return ypred
```

```
ql = keras.QuantumLayer(partial(qml_y, nlayers=nlayers), [(2 * nlayers, 9)])
```

[15]: # keras interface with value and grad paradigm

```
@tf.function
def my_vvag(xs, ys):
    with tf.GradientTape() as tape:
        ypred = ql(xs)
        loss = tf.keras.losses.BinaryCrossentropy()(ys, ypred)
    return loss, tape.gradient(loss, ql.variables)
```

```
my_vvag(x_train_tf, y_train_tf)
```

[15]:

```
<tf.Tensor: shape=(), dtype=float64, numpy=0.7179324626922607>,
[<tf.Tensor: shape=(6, 9), dtype=float64, numpy=
 array([[[-1.97741333e-02, -3.24903196e-03, -1.19449484e-02,
          1.34411790e-02, -2.29378194e-03,  9.24968875e-04,
          3.41827505e-04,  1.38777878e-17, -6.93889390e-18],
         [-1.85390086e-02,  3.81940052e-03, -3.05341288e-02,
          -1.79981829e-03, -5.77913396e-02, -3.71762005e-03,
          -5.10097165e-03, -1.71303943e-17, -1.73472348e-18],
         [ 5.04193508e-03, -1.77846516e-02,  2.26429668e-02,
          -1.41076421e-02, -3.13874407e-02,  1.37515418e-03,
          2.08166817e-17,  2.42861287e-17, -1.73472348e-18],
         [ 2.67860892e-02,  1.92311176e-02, -2.44580361e-02,
          -5.08346256e-02, -1.15289797e-02, -8.99461139e-03,
          3.46944695e-18, -5.20417043e-18, -6.93889390e-18],
         [ 9.54097912e-18, -3.46944695e-18,  1.04083409e-17,
          -1.73472348e-18, -2.53960212e-03,  1.31188463e-17,
          5.20417043e-18, -1.38777878e-17, -1.04083409e-17],
         [-2.60208521e-18,  5.20417043e-18, -5.20417043e-18,
          5.20417043e-18, -3.82010017e-03,  1.38777878e-17,
          1.73472348e-18, -2.08166817e-17,  6.93889390e-18]])>])
```

[16]: # %timeit my_vvag(x_train_tf, y_train_tf)

[17]: # keras interface with keras training paradigm

```
model = tf.keras.Sequential([ql])

model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(0.01),
    metrics=[tf.keras.metrics.BinaryAccuracy()],
)
```

(continues on next page)

(continued from previous page)

```
model.fit(x_train_tf, y_train_tf, batch_size=32, epochs=100)

Epoch 1/100
4/4 [=====] - 21s 8ms/step - loss: 0.7221 - binary_accuracy: 0.
↪6016
Epoch 2/100
4/4 [=====] - 0s 7ms/step - loss: 0.7123 - binary_accuracy: 0.
↪6016
Epoch 3/100
4/4 [=====] - 0s 8ms/step - loss: 0.7039 - binary_accuracy: 0.
↪6562
Epoch 4/100
4/4 [=====] - 0s 7ms/step - loss: 0.7009 - binary_accuracy: 0.
↪6562
Epoch 5/100
4/4 [=====] - 0s 7ms/step - loss: 0.6979 - binary_accuracy: 0.
↪6562
Epoch 6/100
4/4 [=====] - 0s 7ms/step - loss: 0.6957 - binary_accuracy: 0.
↪6016
Epoch 7/100
4/4 [=====] - 0s 7ms/step - loss: 0.6935 - binary_accuracy: 0.
↪4922
Epoch 8/100
4/4 [=====] - 0s 7ms/step - loss: 0.6918 - binary_accuracy: 0.
↪6562
Epoch 9/100
4/4 [=====] - 0s 7ms/step - loss: 0.6910 - binary_accuracy: 0.
↪7109
Epoch 10/100
4/4 [=====] - 0s 8ms/step - loss: 0.6901 - binary_accuracy: 0.
↪5469
Epoch 11/100
4/4 [=====] - 0s 7ms/step - loss: 0.6893 - binary_accuracy: 0.
↪6016
Epoch 12/100
4/4 [=====] - 0s 8ms/step - loss: 0.6883 - binary_accuracy: 0.
↪6562
Epoch 13/100
4/4 [=====] - 0s 8ms/step - loss: 0.6876 - binary_accuracy: 0.
↪6016
Epoch 14/100
4/4 [=====] - 0s 7ms/step - loss: 0.6869 - binary_accuracy: 0.
↪5469
Epoch 15/100
4/4 [=====] - 0s 7ms/step - loss: 0.6865 - binary_accuracy: 0.
↪7109
Epoch 16/100
4/4 [=====] - 0s 7ms/step - loss: 0.6858 - binary_accuracy: 0.
↪6562
Epoch 17/100
4/4 [=====] - 0s 7ms/step - loss: 0.6853 - binary_accuracy: 0.
↪5469
```

(continues on next page)

(continued from previous page)

```

Epoch 18/100
4/4 [=====] - 0s 9ms/step - loss: 0.6847 - binary_accuracy: 0.
˓→6016
Epoch 19/100
4/4 [=====] - 0s 9ms/step - loss: 0.6844 - binary_accuracy: 0.
˓→6016
Epoch 20/100
4/4 [=====] - 0s 9ms/step - loss: 0.6842 - binary_accuracy: 0.
˓→5469
Epoch 21/100
4/4 [=====] - 0s 9ms/step - loss: 0.6841 - binary_accuracy: 0.
˓→6016
Epoch 22/100
4/4 [=====] - 0s 7ms/step - loss: 0.6839 - binary_accuracy: 0.
˓→7109
Epoch 23/100
4/4 [=====] - 0s 7ms/step - loss: 0.6835 - binary_accuracy: 0.
˓→6562
Epoch 24/100
4/4 [=====] - 0s 7ms/step - loss: 0.6829 - binary_accuracy: 0.
˓→6016
Epoch 25/100
4/4 [=====] - 0s 7ms/step - loss: 0.6823 - binary_accuracy: 0.
˓→7109
Epoch 26/100
4/4 [=====] - 0s 7ms/step - loss: 0.6816 - binary_accuracy: 0.
˓→6016
Epoch 27/100
4/4 [=====] - 0s 7ms/step - loss: 0.6811 - binary_accuracy: 0.
˓→5469
Epoch 28/100
4/4 [=====] - 0s 7ms/step - loss: 0.6805 - binary_accuracy: 0.
˓→4922
Epoch 29/100
4/4 [=====] - 0s 7ms/step - loss: 0.6803 - binary_accuracy: 0.
˓→6562
Epoch 30/100
4/4 [=====] - 0s 7ms/step - loss: 0.6799 - binary_accuracy: 0.
˓→5469
Epoch 31/100
4/4 [=====] - 0s 6ms/step - loss: 0.6795 - binary_accuracy: 0.
˓→6016
Epoch 32/100
4/4 [=====] - 0s 6ms/step - loss: 0.6793 - binary_accuracy: 0.
˓→5469
Epoch 33/100
4/4 [=====] - 0s 8ms/step - loss: 0.6789 - binary_accuracy: 0.
˓→6016
Epoch 34/100
4/4 [=====] - 0s 7ms/step - loss: 0.6785 - binary_accuracy: 0.
˓→5469
Epoch 35/100

```

(continues on next page)

(continued from previous page)

```
4/4 [=====] - 0s 7ms/step - loss: 0.6781 - binary_accuracy: 0.  
˓→6562  
Epoch 36/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6775 - binary_accuracy: 0.  
˓→5469  
Epoch 37/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6762 - binary_accuracy: 0.  
˓→6016  
Epoch 38/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6752 - binary_accuracy: 0.  
˓→6562  
Epoch 39/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6736 - binary_accuracy: 0.  
˓→6016  
Epoch 40/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6714 - binary_accuracy: 0.  
˓→6562  
Epoch 41/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6690 - binary_accuracy: 0.  
˓→6562  
Epoch 42/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6658 - binary_accuracy: 0.  
˓→6016  
Epoch 43/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6637 - binary_accuracy: 0.  
˓→6016  
Epoch 44/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6617 - binary_accuracy: 0.  
˓→6562  
Epoch 45/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6596 - binary_accuracy: 0.  
˓→6016  
Epoch 46/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6586 - binary_accuracy: 0.  
˓→6016  
Epoch 47/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6571 - binary_accuracy: 0.  
˓→6016  
Epoch 48/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6561 - binary_accuracy: 0.  
˓→6562  
Epoch 49/100  
4/4 [=====] - 0s 8ms/step - loss: 0.6549 - binary_accuracy: 0.  
˓→6562  
Epoch 50/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6536 - binary_accuracy: 0.  
˓→6562  
Epoch 51/100  
4/4 [=====] - 0s 8ms/step - loss: 0.6536 - binary_accuracy: 0.  
˓→6562  
Epoch 52/100  
4/4 [=====] - 0s 8ms/step - loss: 0.6519 - binary_accuracy: 0.  
˓→6016
```

(continues on next page)

(continued from previous page)

```

Epoch 53/100
4/4 [=====] - 0s 7ms/step - loss: 0.6516 - binary_accuracy: 0.
↪7109
Epoch 54/100
4/4 [=====] - 0s 7ms/step - loss: 0.6504 - binary_accuracy: 0.
↪6016
Epoch 55/100
4/4 [=====] - 0s 8ms/step - loss: 0.6500 - binary_accuracy: 0.
↪6016
Epoch 56/100
4/4 [=====] - 0s 7ms/step - loss: 0.6486 - binary_accuracy: 0.
↪5469
Epoch 57/100
4/4 [=====] - 0s 7ms/step - loss: 0.6468 - binary_accuracy: 0.
↪6016
Epoch 58/100
4/4 [=====] - 0s 8ms/step - loss: 0.6466 - binary_accuracy: 0.
↪7109
Epoch 59/100
4/4 [=====] - 0s 8ms/step - loss: 0.6456 - binary_accuracy: 0.
↪6562
Epoch 60/100
4/4 [=====] - 0s 7ms/step - loss: 0.6446 - binary_accuracy: 0.
↪7109
Epoch 61/100
4/4 [=====] - 0s 7ms/step - loss: 0.6435 - binary_accuracy: 0.
↪6016
Epoch 62/100
4/4 [=====] - 0s 6ms/step - loss: 0.6429 - binary_accuracy: 0.
↪7109
Epoch 63/100
4/4 [=====] - 0s 6ms/step - loss: 0.6417 - binary_accuracy: 0.
↪7109
Epoch 64/100
4/4 [=====] - 0s 6ms/step - loss: 0.6432 - binary_accuracy: 0.
↪5469
Epoch 65/100
4/4 [=====] - 0s 7ms/step - loss: 0.6439 - binary_accuracy: 0.
↪7109
Epoch 66/100
4/4 [=====] - 0s 7ms/step - loss: 0.6430 - binary_accuracy: 0.
↪6016
Epoch 67/100
4/4 [=====] - 0s 6ms/step - loss: 0.6415 - binary_accuracy: 0.
↪5469
Epoch 68/100
4/4 [=====] - 0s 7ms/step - loss: 0.6391 - binary_accuracy: 0.
↪6562
Epoch 69/100
4/4 [=====] - 0s 6ms/step - loss: 0.6375 - binary_accuracy: 0.
↪6016
Epoch 70/100

```

(continues on next page)

(continued from previous page)

```
4/4 [=====] - 0s 7ms/step - loss: 0.6372 - binary_accuracy: 0.  
↳ 6016  
Epoch 71/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6369 - binary_accuracy: 0.  
↳ 5469  
Epoch 72/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6367 - binary_accuracy: 0.  
↳ 6016  
Epoch 73/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6380 - binary_accuracy: 0.  
↳ 7109  
Epoch 74/100  
4/4 [=====] - 0s 8ms/step - loss: 0.6377 - binary_accuracy: 0.  
↳ 6562  
Epoch 75/100  
4/4 [=====] - 0s 8ms/step - loss: 0.6365 - binary_accuracy: 0.  
↳ 6562  
Epoch 76/100  
4/4 [=====] - 0s 8ms/step - loss: 0.6350 - binary_accuracy: 0.  
↳ 6562  
Epoch 77/100  
4/4 [=====] - 0s 8ms/step - loss: 0.6331 - binary_accuracy: 0.  
↳ 6016  
Epoch 78/100  
4/4 [=====] - 0s 8ms/step - loss: 0.6331 - binary_accuracy: 0.  
↳ 6562  
Epoch 79/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6337 - binary_accuracy: 0.  
↳ 5469  
Epoch 80/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6319 - binary_accuracy: 0.  
↳ 6562  
Epoch 81/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6317 - binary_accuracy: 0.  
↳ 7109  
Epoch 82/100  
4/4 [=====] - 0s 6ms/step - loss: 0.6312 - binary_accuracy: 0.  
↳ 6562  
Epoch 83/100  
4/4 [=====] - 0s 6ms/step - loss: 0.6307 - binary_accuracy: 0.  
↳ 6562  
Epoch 84/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6326 - binary_accuracy: 0.  
↳ 6016  
Epoch 85/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6307 - binary_accuracy: 0.  
↳ 6016  
Epoch 86/100  
4/4 [=====] - 0s 6ms/step - loss: 0.6299 - binary_accuracy: 0.  
↳ 6016  
Epoch 87/100  
4/4 [=====] - 0s 7ms/step - loss: 0.6288 - binary_accuracy: 0.  
↳ 6016
```

(continues on next page)

(continued from previous page)

```

Epoch 88/100
4/4 [=====] - 0s 7ms/step - loss: 0.6288 - binary_accuracy: 0.
    ↵7109
Epoch 89/100
4/4 [=====] - 0s 6ms/step - loss: 0.6289 - binary_accuracy: 0.
    ↵6562
Epoch 90/100
4/4 [=====] - 0s 8ms/step - loss: 0.6273 - binary_accuracy: 0.
    ↵6562
Epoch 91/100
4/4 [=====] - 0s 7ms/step - loss: 0.6275 - binary_accuracy: 0.
    ↵5469
Epoch 92/100
4/4 [=====] - 0s 7ms/step - loss: 0.6269 - binary_accuracy: 0.
    ↵7109
Epoch 93/100
4/4 [=====] - 0s 7ms/step - loss: 0.6269 - binary_accuracy: 0.
    ↵6016
Epoch 94/100
4/4 [=====] - 0s 7ms/step - loss: 0.6263 - binary_accuracy: 0.
    ↵6016
Epoch 95/100
4/4 [=====] - 0s 6ms/step - loss: 0.6258 - binary_accuracy: 0.
    ↵6016
Epoch 96/100
4/4 [=====] - 0s 7ms/step - loss: 0.6256 - binary_accuracy: 0.
    ↵6562
Epoch 97/100
4/4 [=====] - 0s 7ms/step - loss: 0.6250 - binary_accuracy: 0.
    ↵6562
Epoch 98/100
4/4 [=====] - 0s 6ms/step - loss: 0.6246 - binary_accuracy: 0.
    ↵7109
Epoch 99/100
4/4 [=====] - 0s 6ms/step - loss: 0.6240 - binary_accuracy: 0.
    ↵6562
Epoch 100/100
4/4 [=====] - 0s 6ms/step - loss: 0.6251 - binary_accuracy: 0.
    ↵5469

```

[17]: <keras.callbacks.History at 0x7ffaf92858b0>

Quantum-Classical Hybrid Model in Keras

```
[18]: def qml_ys(x, weights, nlayers):
    n = 9
    weights = tc.backend.cast(weights, "complex128")
    x = tc.backend.cast(x, "complex128")
    c = tc.Circuit(n)
    for i in range(n):
        c.rx(i, theta=x[i])
```

(continues on next page)

(continued from previous page)

```

for j in range(nlayers):
    for i in range(n - 1):
        c.cnot(i, i + 1)
    for i in range(n):
        c.rx(i, theta=weights[2 * j, i])
        c.ry(i, theta=weights[2 * j + 1, i])
ypreds = []
for i in range(n):
    ypred = c.expectation([tc.gates.z(), (i,)])
    ypred = tc.backend.real(ypred)
    ypred = (tc.backend.real(ypred) + 1) / 2.0
    ypreds.append(ypred)
return tc.backend.stack(ypreds)

```

[19]: ql = tc.keras.QuantumLayer(partial(qml_ys, nlayers=nlayers), [(2 * nlayers, 9)])
model = tf.keras.Sequential([ql, tf.keras.layers.Dense(1, activation="sigmoid")])

[20]: model.compile(
 loss=tf.keras.losses.BinaryCrossentropy(),
 optimizer=tf.keras.optimizers.Adam(0.01),
 metrics=[tf.keras.metrics.BinaryAccuracy()],
)

 model.fit(x_train_tf, y_train_tf, batch_size=32, epochs=100)
Epoch 1/100
4/4 [=====] - 24s 14ms/step - loss: 0.9307 - binary_accuracy: 0.
 ˓→3700
Epoch 2/100
4/4 [=====] - 0s 14ms/step - loss: 0.8286 - binary_accuracy: 0.
 ˓→3700
Epoch 3/100
4/4 [=====] - 0s 15ms/step - loss: 0.7538 - binary_accuracy: 0.
 ˓→3700
Epoch 4/100
4/4 [=====] - 0s 14ms/step - loss: 0.7044 - binary_accuracy: 0.
 ˓→3700
Epoch 5/100
4/4 [=====] - 0s 14ms/step - loss: 0.6796 - binary_accuracy: 0.
 ˓→6300
Epoch 6/100
4/4 [=====] - 0s 14ms/step - loss: 0.6599 - binary_accuracy: 0.
 ˓→6300
Epoch 7/100
4/4 [=====] - 0s 13ms/step - loss: 0.6543 - binary_accuracy: 0.
 ˓→6300
Epoch 8/100
4/4 [=====] - 0s 15ms/step - loss: 0.6559 - binary_accuracy: 0.
 ˓→6300
Epoch 9/100
4/4 [=====] - 0s 14ms/step - loss: 0.6575 - binary_accuracy: 0.
 ˓→6300

(continues on next page)

(continued from previous page)

```

Epoch 10/100
4/4 [=====] - 0s 14ms/step - loss: 0.6588 - binary_accuracy: 0.
˓→6300
Epoch 11/100
4/4 [=====] - 0s 14ms/step - loss: 0.6587 - binary_accuracy: 0.
˓→6300
Epoch 12/100
4/4 [=====] - 0s 14ms/step - loss: 0.6567 - binary_accuracy: 0.
˓→6300
Epoch 13/100
4/4 [=====] - 0s 14ms/step - loss: 0.6551 - binary_accuracy: 0.
˓→6300
Epoch 14/100
4/4 [=====] - 0s 14ms/step - loss: 0.6540 - binary_accuracy: 0.
˓→6300
Epoch 15/100
4/4 [=====] - 0s 14ms/step - loss: 0.6528 - binary_accuracy: 0.
˓→6300
Epoch 16/100
4/4 [=====] - 0s 15ms/step - loss: 0.6533 - binary_accuracy: 0.
˓→6300
Epoch 17/100
4/4 [=====] - 0s 13ms/step - loss: 0.6540 - binary_accuracy: 0.
˓→6300
Epoch 18/100
4/4 [=====] - 0s 13ms/step - loss: 0.6550 - binary_accuracy: 0.
˓→6300
Epoch 19/100
4/4 [=====] - 0s 15ms/step - loss: 0.6546 - binary_accuracy: 0.
˓→6300
Epoch 20/100
4/4 [=====] - 0s 14ms/step - loss: 0.6538 - binary_accuracy: 0.
˓→6300
Epoch 21/100
4/4 [=====] - 0s 15ms/step - loss: 0.6513 - binary_accuracy: 0.
˓→6300
Epoch 22/100
4/4 [=====] - 0s 15ms/step - loss: 0.6499 - binary_accuracy: 0.
˓→6300
Epoch 23/100
4/4 [=====] - 0s 14ms/step - loss: 0.6497 - binary_accuracy: 0.
˓→6300
Epoch 24/100
4/4 [=====] - 0s 15ms/step - loss: 0.6491 - binary_accuracy: 0.
˓→6300
Epoch 25/100
4/4 [=====] - 0s 14ms/step - loss: 0.6492 - binary_accuracy: 0.
˓→6300
Epoch 26/100
4/4 [=====] - 0s 14ms/step - loss: 0.6493 - binary_accuracy: 0.
˓→6300
Epoch 27/100

```

(continues on next page)

(continued from previous page)

```
4/4 [=====] - 0s 15ms/step - loss: 0.6487 - binary_accuracy: 0.  
˓→6300  
Epoch 28/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6473 - binary_accuracy: 0.  
˓→6300  
Epoch 29/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6470 - binary_accuracy: 0.  
˓→6300  
Epoch 30/100  
4/4 [=====] - 0s 16ms/step - loss: 0.6462 - binary_accuracy: 0.  
˓→6300  
Epoch 31/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6451 - binary_accuracy: 0.  
˓→6300  
Epoch 32/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6439 - binary_accuracy: 0.  
˓→6300  
Epoch 33/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6433 - binary_accuracy: 0.  
˓→6300  
Epoch 34/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6438 - binary_accuracy: 0.  
˓→6300  
Epoch 35/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6413 - binary_accuracy: 0.  
˓→6300  
Epoch 36/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6403 - binary_accuracy: 0.  
˓→6300  
Epoch 37/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6391 - binary_accuracy: 0.  
˓→6300  
Epoch 38/100  
4/4 [=====] - 0s 16ms/step - loss: 0.6388 - binary_accuracy: 0.  
˓→6300  
Epoch 39/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6379 - binary_accuracy: 0.  
˓→6300  
Epoch 40/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6365 - binary_accuracy: 0.  
˓→6300  
Epoch 41/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6352 - binary_accuracy: 0.  
˓→6300  
Epoch 42/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6336 - binary_accuracy: 0.  
˓→6300  
Epoch 43/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6338 - binary_accuracy: 0.  
˓→6300  
Epoch 44/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6358 - binary_accuracy: 0.  
˓→6300
```

(continues on next page)

(continued from previous page)

```

Epoch 45/100
4/4 [=====] - 0s 14ms/step - loss: 0.6367 - binary_accuracy: 0.
˓→6300
Epoch 46/100
4/4 [=====] - 0s 13ms/step - loss: 0.6345 - binary_accuracy: 0.
˓→6300
Epoch 47/100
4/4 [=====] - 0s 15ms/step - loss: 0.6303 - binary_accuracy: 0.
˓→6300
Epoch 48/100
4/4 [=====] - 0s 14ms/step - loss: 0.6298 - binary_accuracy: 0.
˓→6300
Epoch 49/100
4/4 [=====] - 0s 14ms/step - loss: 0.6285 - binary_accuracy: 0.
˓→6300
Epoch 50/100
4/4 [=====] - 0s 14ms/step - loss: 0.6280 - binary_accuracy: 0.
˓→6300
Epoch 51/100
4/4 [=====] - 0s 14ms/step - loss: 0.6274 - binary_accuracy: 0.
˓→6300
Epoch 52/100
4/4 [=====] - 0s 15ms/step - loss: 0.6268 - binary_accuracy: 0.
˓→6300
Epoch 53/100
4/4 [=====] - 0s 14ms/step - loss: 0.6262 - binary_accuracy: 0.
˓→6300
Epoch 54/100
4/4 [=====] - 0s 14ms/step - loss: 0.6246 - binary_accuracy: 0.
˓→6300
Epoch 55/100
4/4 [=====] - 0s 15ms/step - loss: 0.6231 - binary_accuracy: 0.
˓→6300
Epoch 56/100
4/4 [=====] - 0s 14ms/step - loss: 0.6228 - binary_accuracy: 0.
˓→6300
Epoch 57/100
4/4 [=====] - 0s 14ms/step - loss: 0.6226 - binary_accuracy: 0.
˓→6300
Epoch 58/100
4/4 [=====] - 0s 13ms/step - loss: 0.6224 - binary_accuracy: 0.
˓→6300
Epoch 59/100
4/4 [=====] - 0s 14ms/step - loss: 0.6228 - binary_accuracy: 0.
˓→6900
Epoch 60/100
4/4 [=====] - 0s 15ms/step - loss: 0.6224 - binary_accuracy: 0.
˓→7200
Epoch 61/100
4/4 [=====] - 0s 14ms/step - loss: 0.6214 - binary_accuracy: 0.
˓→7200
Epoch 62/100

```

(continues on next page)

(continued from previous page)

```
4/4 [=====] - 0s 14ms/step - loss: 0.6183 - binary_accuracy: 0.  
˓→6300  
Epoch 63/100  
4/4 [=====] - 0s 17ms/step - loss: 0.6161 - binary_accuracy: 0.  
˓→6300  
Epoch 64/100  
4/4 [=====] - 0s 13ms/step - loss: 0.6142 - binary_accuracy: 0.  
˓→6300  
Epoch 65/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6131 - binary_accuracy: 0.  
˓→6300  
Epoch 66/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6124 - binary_accuracy: 0.  
˓→6300  
Epoch 67/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6101 - binary_accuracy: 0.  
˓→6300  
Epoch 68/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6117 - binary_accuracy: 0.  
˓→6600  
Epoch 69/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6099 - binary_accuracy: 0.  
˓→7200  
Epoch 70/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6085 - binary_accuracy: 0.  
˓→7200  
Epoch 71/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6070 - binary_accuracy: 0.  
˓→7200  
Epoch 72/100  
4/4 [=====] - 0s 13ms/step - loss: 0.6069 - binary_accuracy: 0.  
˓→7200  
Epoch 73/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6060 - binary_accuracy: 0.  
˓→7200  
Epoch 74/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6040 - binary_accuracy: 0.  
˓→7200  
Epoch 75/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6041 - binary_accuracy: 0.  
˓→7200  
Epoch 76/100  
4/4 [=====] - 0s 15ms/step - loss: 0.6011 - binary_accuracy: 0.  
˓→6900  
Epoch 77/100  
4/4 [=====] - 0s 14ms/step - loss: 0.6005 - binary_accuracy: 0.  
˓→6300  
Epoch 78/100  
4/4 [=====] - 0s 14ms/step - loss: 0.5993 - binary_accuracy: 0.  
˓→6300  
Epoch 79/100  
4/4 [=====] - 0s 15ms/step - loss: 0.5987 - binary_accuracy: 0.  
˓→6300
```

(continues on next page)

(continued from previous page)

```

Epoch 80/100
4/4 [=====] - 0s 14ms/step - loss: 0.5982 - binary_accuracy: 0.
↪6300
Epoch 81/100
4/4 [=====] - 0s 15ms/step - loss: 0.5970 - binary_accuracy: 0.
↪6300
Epoch 82/100
4/4 [=====] - 0s 15ms/step - loss: 0.5960 - binary_accuracy: 0.
↪6900
Epoch 83/100
4/4 [=====] - 0s 14ms/step - loss: 0.5937 - binary_accuracy: 0.
↪7200
Epoch 84/100
4/4 [=====] - 0s 15ms/step - loss: 0.5932 - binary_accuracy: 0.
↪7200
Epoch 85/100
4/4 [=====] - 0s 15ms/step - loss: 0.5913 - binary_accuracy: 0.
↪7200
Epoch 86/100
4/4 [=====] - 0s 14ms/step - loss: 0.5904 - binary_accuracy: 0.
↪7200
Epoch 87/100
4/4 [=====] - 0s 15ms/step - loss: 0.5895 - binary_accuracy: 0.
↪7200
Epoch 88/100
4/4 [=====] - 0s 14ms/step - loss: 0.5882 - binary_accuracy: 0.
↪7200
Epoch 89/100
4/4 [=====] - 0s 13ms/step - loss: 0.5873 - binary_accuracy: 0.
↪7200
Epoch 90/100
4/4 [=====] - 0s 15ms/step - loss: 0.5858 - binary_accuracy: 0.
↪7200
Epoch 91/100
4/4 [=====] - 0s 14ms/step - loss: 0.5848 - binary_accuracy: 0.
↪7200
Epoch 92/100
4/4 [=====] - 0s 16ms/step - loss: 0.5839 - binary_accuracy: 0.
↪7200
Epoch 93/100
4/4 [=====] - 0s 15ms/step - loss: 0.5832 - binary_accuracy: 0.
↪7200
Epoch 94/100
4/4 [=====] - 0s 14ms/step - loss: 0.5836 - binary_accuracy: 0.
↪7200
Epoch 95/100
4/4 [=====] - 0s 15ms/step - loss: 0.5848 - binary_accuracy: 0.
↪7200
Epoch 96/100
4/4 [=====] - 0s 15ms/step - loss: 0.5836 - binary_accuracy: 0.
↪7200
Epoch 97/100

```

(continues on next page)

(continued from previous page)

```
4/4 [=====] - 0s 14ms/step - loss: 0.5812 - binary_accuracy: 0.
    ↵7200
Epoch 98/100
4/4 [=====] - 0s 14ms/step - loss: 0.5795 - binary_accuracy: 0.
    ↵7200
Epoch 99/100
4/4 [=====] - 0s 14ms/step - loss: 0.5780 - binary_accuracy: 0.
    ↵7200
Epoch 100/100
4/4 [=====] - 0s 14ms/step - loss: 0.5768 - binary_accuracy: 0.
    ↵7200
```

[20]: <keras.callbacks.History at 0x7ffac3142e20>

Hybrid Model in Jax

[21]: `tc.set_backend("jax")`

[21]: <tensorcircuit.backends.jax_backend.JaxBackend at 0x7ffb04a71820>

```
[22]: key = jax.random.PRNGKey(42)
key, *subkeys = jax.random.split(key, num=4)
params = {
    "qweights": jax.random.normal(subkeys[0], shape=[nlayers * 2, 9]),
    "cweights:w": jax.random.normal(subkeys[1], shape=[9]),
    "cweights:b": jax.random.normal(subkeys[2], shape=[1]),
}
```

```
[23]: def qml_hybrid_loss(x, y, params, nlayers):
    weights = params["qweights"]
    w = params["cweights:w"]
    b = params["cweights:b"]
    ypred = qml_ys(x, weights, nlayers)
    ypred = tc.backend.reshape(ypred, [-1, 1])
    ypred = w @ ypred + b
    ypred = jax.nn.sigmoid(ypred)
    ypred = ypred[0]
    loss = -y * tc.backend.log(ypred) - (1 - y) * tc.backend.log(1 - ypred)
    return loss
```

```
[24]: qml_hybrid_loss_vag = tc.backend.jit(
    tc.backend.vvag(qml_hybrid_loss, vectorized_argnums=(0, 1), argnums=2),
    static_argnums=3,
)
```

[25]: `qml_hybrid_loss_vag(x_train_jax, y_train_jax, params, nlayers)`

```
[25]: (DeviceArray([3.73282398, 0.02421603, 0.02899787, 0.02421603, 4.08996787,
    0.03069481, 0.02421603, 0.01688146, 4.08996787, 0.03069481,
    4.08996787, 0.02421603, 4.08996787, 0.02421603, 0.02899787,
```

(continues on next page)

(continued from previous page)

```

0.03354042, 0.02421603, 0.02421603, 0.01688146, 4.08996787,
0.03354042, 0.02421603, 0.02421603, 0.03069481, 0.02421603,
0.02421603, 0.03069481, 3.73798651, 0.02421603, 3.68810189,
4.08996787, 0.03069481, 3.73282398, 0.03069481, 3.73282398,
0.02421603, 3.49674264, 0.02421603, 4.08996787, 0.02899787,
0.02421603, 0.02421603, 0.03069481, 0.03069481, 3.73282398,
0.02533775, 0.03069481, 3.68810189, 3.73282398, 3.49896983,
0.02899787, 0.03069481, 4.08996787, 3.41172721, 0.02421603,
0.02421603, 3.73282398, 0.02421603, 3.73798651, 3.68810189,
4.08996787, 0.03069481, 4.08996787, 0.02421603, 0.03069481,
0.02421603, 3.68810189, 3.49896983, 3.49896983, 4.08996787,
0.02421603, 0.02421603, 0.02421603, 0.02421603, 0.03069481,
0.02899787, 3.41172721, 0.03354042, 0.02899787, 3.68810189,
0.02421603, 0.03354042, 3.73282398, 0.02899787, 3.73282398,
0.03069481, 0.02421603, 3.73282398, 3.68810189, 3.73282398,
0.02421603, 0.02421603, 0.03069481, 0.03069481, 0.02421603,
4.08996787, 0.02421603, 0.01688146, 3.73282398, 0.02421603],
dtype=float64),
{'cweights:b': DeviceArray([34.49476789], dtype=float64),
 'cweights:w': DeviceArray([16.81782277, 15.05718878, 15.02498328, 23.18351696,
    17.01897109, 16.13466029, 16.26046722, 23.54180309,
    12.0721068 ], dtype=float64),
 'qweights': DeviceArray([[-1.16993912e+01, -6.74730815e+00, -2.27227872e+00,
    -1.08703899e+00, 2.56625721e+00, 1.69462223e+00,
    -4.89847061e+00, 1.62487935e+00, 1.02424785e+01],
   [ 3.29984130e+00, -5.90635608e-01, 2.11407610e+00,
    3.67096431e-02, 3.32526833e+00, -1.06468920e+00,
    -4.12299772e-01, -7.78105081e+00, -3.38506241e+00],
   [-3.59434442e+00, 3.84548015e+00, 8.50409406e-01,
    -2.66504333e+00, 1.47559967e+00, 1.38536529e+00,
    -1.47291602e-01, -7.32213541e+00, 5.17021200e+00],
   [-1.30975045e+00, 1.83003338e+00, 1.51443252e+00,
    3.15082430e+00, -4.41767236e+00, 6.25968228e+00,
    5.96980281e+00, 9.67198061e+00, -1.63091455e+01],
   [-2.24757712e+00, -5.66276080e-01, -1.67376432e+00,
    1.75249049e-01, 2.77917505e-01, 3.84402979e-02,
    1.03434679e-01, -4.05760762e-02, -3.33671956e-03],
   [ 3.13599600e+00, 3.85470136e+00, 3.17986238e-01,
    1.72308312e-01, 5.09749793e+00, 2.90706770e-02,
    -5.59919189e-01, 1.96734688e+00, -6.96372626e-01]],

dtype=float64)})

```

```
[26]: optimizer = optax.adam(5e-3)
opt_state = optimizer.init(params)
for i, (xs, ys) in zip(range(2000), mnist_data): # using tf data loader here
    xs = xs.numpy()
    ys = ys.numpy()
    v, grads = qml_hybrid_loss_vag(xs, ys, params, nlayers)
    updates, opt_state = optimizer.update(grads, opt_state)
    params = optax.apply_updates(params, updates)
    if i % 30 == 0:
        print(jnp.mean(v))
```

```
1.2979572281332594
0.8331012068009501
0.6805939758448183
0.5897353928152392
0.6460840124038746
0.6093143713632384
0.6671721223530598
0.5863347320393952
0.5465362554431986
0.5594138744621404
0.5493311423294576
0.5228166702417829
0.6176455570797168
0.5256494465741394
0.5359881696740493
0.5787532611935906
0.49082340457493323
0.4062487079116086
0.5802733401377229
0.4762524476616207
0.5404245247888219
```

3.1.5 QML in PyTorch

Overview

In this tutorial, we show the MNIST binary classification QML example with the same setup as *mnist_qml*. This time, we use the PyTorch machine learning pipeline to build the QML model. Again, this note is not about the best QML practice or the best PyTorch pipeline practice, instead, it is just a demonstration of the integration between PyTorch and TensorCircuit.

Setup

```
[1]: import time
import numpy as np
import tensorflow as tf
import torch

import tensorcircuit as tc

K = tc.set_backend("tensorflow")

# Use TensorFlow as the backend, while wrapping the quantum function in the PyTorch_
# interface
```



```
[2]: # We load and preprocess the dataset as the previous notebook using tensorflow and jax_
# backend

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train = x_train[..., np.newaxis] / 255.0
```

(continues on next page)

(continued from previous page)

```
def filter_pair(x, y, a, b):
    keep = (y == a) | (y == b)
    x, y = x[keep], y[keep]
    y = y == a
    return x, y

x_train, y_train = filter_pair(x_train, y_train, 1, 5)
x_train_small = tf.image.resize(x_train, (3, 3)).numpy()
x_train_bin = np.array(x_train_small > 0.5, dtype=np.float32)
x_train_bin = np.squeeze(x_train_bin).reshape([-1, 9])
y_train_torch = torch.tensor(y_train, dtype=torch.float32)
x_train_torch = torch.tensor(x_train_bin)
x_train_torch.shape, y_train_torch.shape
```

[2]: (torch.Size([12163, 9]), torch.Size([12163]))

Wrap Quantum Function Using torch_interface

```
[3]: n = 9
nlayers = 3

# We define the quantum function,
# note how this function is running on tensorflow

def qpred(x, weights):
    c = tc.Circuit(n)
    for i in range(n):
        c.rx(i, theta=x[i])
    for j in range(nlayers):
        for i in range(n - 1):
            c.cnot(i, i + 1)
        for i in range(n):
            c.rx(i, theta=weights[2 * j, i])
            c.ry(i, theta=weights[2 * j + 1, i])
    ypred = c.expectation_ps(z=[n // 2])
    ypred = K.real(ypred)
    return K.sigmoid(ypred)

# Wrap the function into pytorch form but with tensorflow speed!
qpred_torch = tc.interfaces.torch_interface(qpred, jit=True)
```

After we have the AD aware function in PyTorch format, we can further wrap it as a torch Module (network layer).

```
[4]: class QuantumNet(torch.nn.Module):
    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
    self.q_weights = torch.nn.Parameter(torch.randn([2 * nlayers, n]))  
  
def forward(self, inputs):  
    ypred = qpred_torch(inputs, self.q_weights)  
    return ypred
```

[5]: net = QuantumNet()
net(x_train_torch[0])

[5]: tensor(0.4539, grad_fn=<FunBackward>)

[6]: criterion = torch.nn.BCELoss()
opt = torch.optim.Adam(net.parameters(), lr=1e-2)
nepochs = 500
batch_size = 32
times = []

for epoch in range(nepochs):
 index = np.random.randint(low=0, high=100, size=batch_size)
 # index = np.arange(batch_size)
 inputs, labels = x_train_torch[index], y_train_torch[index]
 opt.zero_grad()

 with torch.set_grad_enabled(True):
 time0 = time.time()
 yps = []
 for i in range(batch_size):
 yp = net(inputs[i])
 yps.append(yp)
 yps = torch.stack(yps)
 loss = criterion(
 torch.reshape(yps, [batch_size, 1]), torch.reshape(labels, [batch_size, 1]))
)
 loss.backward()
 if epoch % 100 == 0:
 print(loss)
 opt.step()
 time1 = time.time()
 times.append(time1 - time0)

print("training time per step: ", np.mean(time1 - time0))

tensor(0.7287, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.5947, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.5804, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.6358, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.6503, grad_fn=<BinaryCrossEntropyBackward0>)
training time per step: 0.12587213516235352

Batched Version

Now let's try vectorized version to speed up the batch input processing. Note how intrinsically, `tf.vectorized_map` helps in the batch pipeline.

```
[7]: qpred_vmap = K.vmap(qpred, vectorized_argnums=0)

# `qpred_vmap` is a tensorflow function with vectorization capacity

qpred_batch = tc.interfaces.torch_interface(qpred_vmap, jit=True)

# We further wrap the function as a PyTorch one
```

```
[8]: # Test the AD capacity of the PyTorch function

w = torch.ones([2 * nlayers, n])
w.requires_grad_()
with torch.set_grad_enabled(True):
    yps = qpred_batch(x_train_torch[:3], w)
    loss = torch.sum(yps)
    loss.backward()
print(w.grad)

tensor([[ -6.2068e-03, -3.0100e-05, -1.0997e-02, -1.8381e-02, -9.1800e-02,
         1.2481e-01, -6.5200e-02,  1.1176e-08,  7.4506e-09],
        [-3.2353e-03,  3.4989e-03, -1.1344e-02, -1.6136e-02,  1.9075e-02,
         2.1119e-02,  2.6881e-02, -1.1176e-08,  0.0000e+00],
        [-1.1777e-02, -1.1572e-03, -5.0570e-03,  6.4838e-03, -5.5077e-02,
         -3.4250e-02, -7.4506e-09, -1.1176e-08,  3.7253e-09],
        [-1.4748e-02, -2.3818e-02, -4.3567e-02, -4.7879e-02,  1.2331e-01,
         1.4314e-01,  3.7253e-09,  1.1176e-08,  3.7253e-09],
        [-3.7253e-09,  3.7253e-09,  0.0000e+00,  0.0000e+00, -2.7574e-02,
         7.4506e-09,  7.4506e-09, -1.1176e-08,  0.0000e+00],
        [ 3.7253e-09,  3.7253e-09,  1.4901e-08, -7.4506e-09,  7.1655e-02,
        -7.4506e-09,  3.7253e-09,  1.4901e-08,  0.0000e+00]])
```

```
[9]: class QuantumNetV2(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.q_weights = torch.nn.Parameter(torch.randn([2 * nlayers, n]))

    def forward(self, inputs):
        ypred = qpred_batch(inputs, self.q_weights)
        return ypred
```

```
[10]: net2 = QuantumNetV2()
net2(x_train_torch[:3])

[10]: tensor([0.4706, 0.4706, 0.4767], grad_fn=<FunBackward>)
```

With the help of vmap infrastructure borrowed from TensorFlow, the performance of training is greatly improved!

```
[11]: criterion = torch.nn.BCELoss()
opt = torch.optim.Adam(net2.parameters(), lr=1e-2)
```

(continues on next page)

(continued from previous page)

```

nepochs = 500
nbatch = 32
times = []
for epoch in range(nepochs):
    index = np.random.randint(low=0, high=100, size=nbatch)
    # index = np.arange(nbatch)
    inputs, labels = x_train_torch[index], y_train_torch[index]
    opt.zero_grad()

    with torch.set_grad_enabled(True):
        time0 = time.time()
        yps = net2(inputs)
        loss = criterion(
            torch.reshape(yps, [nbatch, 1]), torch.reshape(labels, [nbatch, 1]))
    )
    loss.backward()
    if epoch % 100 == 0:
        print(loss)
    opt.step()
    time1 = time.time()
    times.append(time1 - time0)
print("training time per step: ", np.mean(times[1:]))

tensor(0.6973, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.6421, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.6419, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.6498, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.6466, grad_fn=<BinaryCrossEntropyBackward0>)
training time per step:  0.009107916531916371

```

Hybrid Model with Classical Post-processing

We now build a quantum-classical hybrid machine learning model pipeline where the output measurement results are further fed into a classical fully connected layer.

```
[12]: def qpreds(x, weights):
    c = tc.Circuit(n)
    for i in range(n):
        c.rx(i, theta=x[i])
    for j in range(nlayers):
        for i in range(n - 1):
            c.cnot(i, i + 1)
        for i in range(n):
            c.rx(i, theta=weights[2 * j, i])
            c.ry(i, theta=weights[2 * j + 1, i])

    return K.stack([K.real(c.expectation_ps(z=[i])) for i in range(n)])
```

```

qpreds_vmap = K.vmap(qpreds, vectorized_argnums=0)
qpreds_batch = tc.interfaces.torch_interface(qpreds_vmap, jit=True)

```

(continues on next page)

(continued from previous page)

```
[12]: qpreds_batch(x_train_torch[:2], torch.ones([2 * nlayers, n]))
tensor([[ 0.2839,  0.3786,  0.0158,  0.1512,  0.1395,  0.1364,  0.1403,  0.1423,
         -0.1285],
        [ 0.2839,  0.3786,  0.0158,  0.1512,  0.1395,  0.1364,  0.1403,  0.1423,
         -0.1285]])
```

```
[13]: class QuantumNetV3(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.q_weights = torch.nn.Parameter(torch.randn([2 * nlayers, n]))

    def forward(self, inputs):
        ypred = qpreds_batch(inputs, self.q_weights)
        return ypred
```

```
[14]: net3 = QuantumNetV3()
net3(x_train_bin[:2])
tensor([[ 0.2931,  0.5393, -0.0369, -0.0450,  0.0511, -0.0121,  0.0156, -0.0406,
         -0.1330],
        [ 0.2931,  0.5393, -0.0369, -0.0450,  0.0511, -0.0121,  0.0156, -0.0406,
         -0.1330]], grad_fn=<FunBackward>)
```

We now build a hybrid model with the quantum layer `net3` and append a Linear layer behind it.

```
[15]: model = torch.nn.Sequential(QuantumNetV3(), torch.nn.Linear(9, 1), torch.nn.Sigmoid())
model(x_train_bin[:2])
tensor([[0.5500],
       [0.5500]], grad_fn=<SigmoidBackward0>)
```

```
[16]: criterion = torch.nn.BCELoss()
opt = torch.optim.Adam(model.parameters(), lr=1e-2)
nepochs = 500
batch_size = 32
times = []
for epoch in range(nepochs):
    index = np.random.randint(low=0, high=100, size=batch_size)
    # index = np.arange(batch_size)
    inputs, labels = x_train_torch[index], y_train_torch[index]
    opt.zero_grad()

    with torch.set_grad_enabled(True):
        time0 = time.time()
        yps = model(inputs)
        loss = criterion(
            torch.reshape(yps, [batch_size, 1]), torch.reshape(labels, [batch_size, 1]))
    )
    loss.backward()
    if epoch % 100 == 0:
```

(continues on next page)

(continued from previous page)

```

    print(loss)
    opt.step()
    time1 = time.time()
    times.append(time1 - time0)
print("training time per step: ", np.mean(times[1:]))

tensor(0.6460, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.6086, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.5199, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.5697, grad_fn=<BinaryCrossEntropyBackward0>)
tensor(0.5248, grad_fn=<BinaryCrossEntropyBackward0>)
training time per step:  0.020270218113381304

```

3.1.6 Quantum Machine Learning for Classification Task

Demonstrations on some common setups and techniques

Overview

We use the fashion-MNIST dataset to set up a binary classification task, we will try different encoding schemes for the inputs and apply possible classical post-processing on the quantum outputs to enhance the classification accuracy. In this tutorial, we stick to the TensorFlow backend and try to consistently use the **Keras interface** provided by tensorcircuit for quantum functions, where we can magically turn a quantum function into a Keras layer.

```
[1]: from matplotlib import pyplot as plt
from sklearn.decomposition import PCA
import tensorflow as tf
import tensorcircuit as tc

K = tc.set_backend("tensorflow")
```

Dataset and Pre-processing

We first load the fashion-mnist dataset and differentiation between T-shirt (0) and Trouser (1).

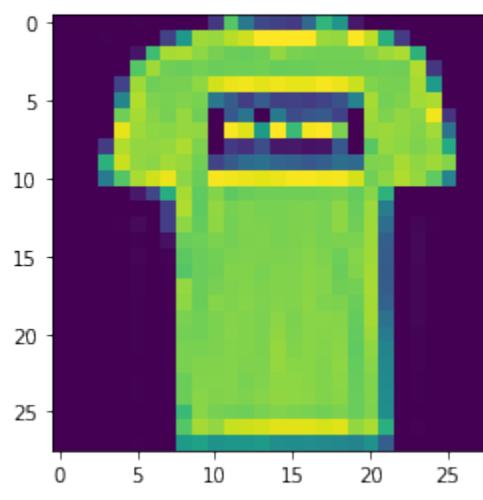
```
[2]: (x_train, y_train), (x_test, y_test) = tc.templates.dataset.mnist_pair_data(
    0, 1, loader=tf.keras.datasets.fashion_mnist
)
```

```
[3]: x_train.shape, y_train.shape, x_test.shape, y_test.shape
```

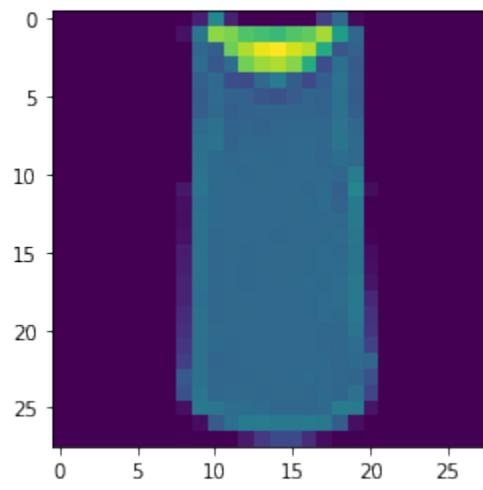
```
[3]: ((12000, 28, 28, 1), (12000,), (2000, 28, 28, 1), (2000,))
```

```
[4]: plt.imshow(x_train[0])
```

```
[4]: <matplotlib.image.AxesImage at 0x7f9147029610>
```



```
[5]: plt.imshow(x_train[1])  
[5]: <matplotlib.image.AxesImage at 0x7f91473b6c10>
```



Amplitude Encoding

```
[6]: x_train = tf.image.pad_to_bounding_box(x_train, 2, 2, 32, 32)  
x_test = tf.image.pad_to_bounding_box(x_test, 2, 2, 32, 32)
```

```
[7]: batched_ae = K.vmap(tc.templates.dataset.amplitude_encoding, vectorized_argnums=0)
```

```
[8]: x_train_q = batched_ae(x_train, 10)  
x_test_q = batched_ae(x_test, 10)
```

```
[9]: n = 10  
blocks = 3
```

(continues on next page)

(continued from previous page)

```

def qml(x, weights):
    c = tc.Circuit(n, inputs=x)
    for j in range(blocks):
        for i in range(n):
            c.rx(i, theta=weights[j, i, 0])
            c.rz(i, theta=weights[j, i, 1])
        for i in range(n - 1):
            c.exp1(i, i + 1, theta=weights[j, i, 2], unitary=tc.gates._zz_matrix)
    outputs = K.stack([
        [K.real(c.expectation([tc.gates.z(), [i]])) for i in range(n)]
        + [K.real(c.expectation([tc.gates.x(), [i]])) for i in range(n)]
    ])
    outputs = K.reshape(outputs, [-1])
    return K.sigmoid(K.sum(outputs))

qml_layer = tc.keras.QuantumLayer(qml, weights_shape=[blocks, n, 3])

```

```

[10]: model = tf.keras.Sequential([qml_layer])
model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(0.01),
    metrics=[tf.keras.metrics.BinaryAccuracy()],
)

model.fit(x_train_q, y_train, batch_size=32, epochs=3, validation_split=0.8)

Epoch 1/3
75/75 [=====] - 85s 559ms/step - loss: 0.6217 - binary_accuracy: 0.7667 - val_loss: 0.3990 - val_binary_accuracy: 0.9620
Epoch 2/3
75/75 [=====] - 14s 185ms/step - loss: 0.3701 - binary_accuracy: 0.9571 - val_loss: 0.3421 - val_binary_accuracy: 0.9507
Epoch 3/3
75/75 [=====] - 14s 185ms/step - loss: 0.3252 - binary_accuracy: 0.9542 - val_loss: 0.3030 - val_binary_accuracy: 0.9540
[10]: <keras.callbacks.History at 0x7f9147416e50>

```

Classical Post-processing

We attached one Linear layer after the quantum outputs to enhance the capacity of the machine learning model as a quantum-neural hybrid machine learning approach.

```

[11]: def qml(x, weights):
    c = tc.Circuit(n, inputs=x)
    for j in range(blocks):
        for i in range(n):
            c.rx(i, theta=weights[j, i, 0])
            c.rz(i, theta=weights[j, i, 1])
        for i in range(n - 1):
            c.exp1(i, i + 1, theta=weights[j, i, 2], unitary=tc.gates._zz_matrix)

```

(continues on next page)

(continued from previous page)

```

outputs = K.stack(
    [K.real(c.expectation([tc.gates.z(), [i]])) for i in range(n)]
    + [K.real(c.expectation([tc.gates.x(), [i]])) for i in range(n)])
)
outputs = K.reshape(outputs, [-1])
return outputs

qml_layer = tc.keras.QuantumLayer(qml, weights_shape=[blocks, n, 3])

inputs = tf.keras.Input(shape=(2**n), dtype=tf.complex64)
measurements = qml_layer(inputs)
output = tf.keras.layers.Dense(1, activation="sigmoid")(measurements)
model = tf.keras.Model(inputs=inputs, outputs=output)

```

```

[12]: model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(0.01),
    metrics=[tf.keras.metrics.BinaryAccuracy()],
)

model.fit(x_train_q, y_train, batch_size=32, epochs=3, validation_split=0.8)

Epoch 1/3
75/75 [=====] - 71s 508ms/step - loss: 0.5140 - binary_accuracy: 0.8841 - val_loss: 0.3617 - val_binary_accuracy: 0.9521
Epoch 2/3
75/75 [=====] - 14s 182ms/step - loss: 0.2803 - binary_accuracy: 0.9421 - val_loss: 0.2093 - val_binary_accuracy: 0.9506
Epoch 3/3
75/75 [=====] - 15s 200ms/step - loss: 0.2057 - binary_accuracy: 0.9437 - val_loss: 0.1795 - val_binary_accuracy: 0.9483
[12]: <keras.callbacks.History at 0x7f913d4ee520>

```

PCA Embedding

Amplitude encoding is difficult to implement on real quantum hardware, we here instead consider another way for data input, where only a single qubit rotation is involved. To compress the input data such that it can fit into a small circuit, PCA dimension reduction is required.

```

[13]: x_train_r = PCA(10).fit_transform(x_train.numpy().reshape([-1, 32 * 32]))

[14]: x_train_r.shape # we now has 10-d vector compression for each figure
[14]: (12000, 10)

[15]: def qml(x, weights):
    c = tc.Circuit(n)
    for i in range(10):
        c.rx(i, theta=x[i]) # loading the data
    for j in range(blocks):

```

(continues on next page)

(continued from previous page)

```

for i in range(n):
    c.rx(i, theta=weights[j, i, 0])
    c.rz(i, theta=weights[j, i, 1])
for i in range(n - 1):
    c.exp1(i, i + 1, theta=weights[j, i, 2], unitary=tc.gates._zz_matrix)
outputs = K.stack(
    [K.real(c.expectation([tc.gates.z(), [i]])) for i in range(n)]
    + [K.real(c.expectation([tc.gates.x(), [i]])) for i in range(n)])
)
outputs = K.reshape(outputs, [-1])
return K.sigmoid(K.sum(outputs))

qml_layer = tc.keras.QuantumLayer(qml, weights_shape=[blocks, n, 3])

```

```
[16]: model = tf.keras.Sequential([qml_layer])
model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(0.01),
    metrics=[tf.keras.metrics.BinaryAccuracy()],
)

model.fit(x_train_r, y_train, batch_size=32, epochs=3, validation_split=0.8)

Epoch 1/3
75/75 [=====] - 71s 447ms/step - loss: 0.7993 - binary_accuracy: 0.6996 - val_loss: 0.3026 - val_binary_accuracy: 0.8829
Epoch 2/3
75/75 [=====] - 6s 80ms/step - loss: 0.2745 - binary_accuracy: 0.8983 - val_loss: 0.2559 - val_binary_accuracy: 0.9087
Epoch 3/3
75/75 [=====] - 6s 83ms/step - loss: 0.2513 - binary_accuracy: 0.9167 - val_loss: 0.2385 - val_binary_accuracy: 0.9187
[16]: <keras.callbacks.History at 0x7f91204f1430>
```

Data Re-uploading

By loading the PCA embedding data multiple times in the VQA, we may further increase the accuracy of the model.

```
[17]: def qml(x, weights):
    c = tc.Circuit(n)
    for j in range(blocks):
        for i in range(10):
            c.ry(i, theta=x[i]) # loading the data repetitively
        for i in range(n):
            c.rx(i, theta=weights[j, i, 0])
            c.rz(i, theta=weights[j, i, 1])
        for i in range(n - 1):
            c.exp1(i, i + 1, theta=weights[j, i, 2], unitary=tc.gates._zz_matrix)
    outputs = K.stack(
        [K.real(c.expectation([tc.gates.z(), [i]])) for i in range(n)]
```

(continues on next page)

(continued from previous page)

```

    + [K.real(c.expectation([tc.gates.x(), [i]])) for i in range(n)]
)
outputs = K.reshape(outputs, [-1])
return K.sigmoid(K.sum(outputs))

qml_layer = tc.keras.QuantumLayer(qml, weights_shape=[blocks, n, 3])

```

```
[19]: model = tf.keras.Sequential([qml_layer])
model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(0.01),
    metrics=[tf.keras.metrics.BinaryAccuracy()],
)

model.fit(x_train_r, y_train, batch_size=32, epochs=3, validation_split=0.8)

Epoch 1/3
75/75 [=====] - 20s 146ms/step - loss: 0.2506 - binary_accuracy: 0.9070 - val_loss: 0.2271 - val_binary_accuracy: 0.9177
Epoch 2/3
75/75 [=====] - 7s 95ms/step - loss: 0.2191 - binary_accuracy: 0.9246 - val_loss: 0.2071 - val_binary_accuracy: 0.9287
Epoch 3/3
75/75 [=====] - 7s 95ms/step - loss: 0.2049 - binary_accuracy: 0.9287 - val_loss: 0.2016 - val_binary_accuracy: 0.9305
[19]: <keras.callbacks.History at 0x7f90a813b460>
```

3.1.7 Variational Quantum Eigensolver (VQE) on Molecules

Overview

VQE is a variational algorithm for calculating the ground state of some given hamiltonian H which we call it ψ_g that satisfied $H|\psi_g\rangle = E_g|\psi_g\rangle$. For an arbitrary normalized wavefunction ψ_f , the expectation value $\langle\psi_f|H|\psi_f\rangle$ is always not lower than the ground state energy unless $\psi_f = \psi_g$ to some phase factor (here we assume there is no degeneracy in ground state). Based on that fact, if we use a parameterized wavefunction ψ_θ , e.g. given by a parameterized quantum circuit (PQC) with parameters θ , we can give an approximation for the ground state enery and wavefunction by minimizing the expectation value of H . In practical quantum hardware, this algorithm can be realized in a quantum-neural hybrid paradigm with the gradient calculated using finite difference or paremeter shift in quantum hardware and the optimization using gradient descent method in classical computer. While in a numerical simulation, we can just calculate the gradients using automatic differentiation.

Calculating the ground state energy for molecules is often important for quantum chemistry tasks since it can be used to find out the atom structure of the molecules. In the simulation of molecules, we do not consider the motion of nuclei which means we fix the nuclear coordinates of its constituent atoms. We only consider the electrons in the molecules since the nuclei are way heavier than the electrons and thus the energy carried by phonons is negligible or can be reconsidered using Born-Oppenheimer approximation. Strictly speaking, the eletrons lives in continuous space, thus the Hilbert space is of infinite dimensions. To conduct a practical calculation, we only preserve some important single-particle basis, e.g. the low energy atomic orbitals. In the second quantization formalism, we can represent these atomic orbitals as $c_i^\dagger|0\rangle$. By considering the interactions of nuclei and electrons as background and also the electron-electron interaction, a molecules hamiltonian can in generally be represented as $H = \sum_{i,j} h_{i,j} c_i^\dagger c_j +$

$\sum_{i,j,k,l} \alpha_{i,j,k,l} c_i^\dagger c_j^\dagger c_k c_l$. Notice that the spin index is also absorbed into the orbital index. There are many softwares that can give these parameters in H such as `pyscf` which we will use later in this tutorial. Now we have a fermionic description for molecules. By using a mapping from fermionic operators to spin operators such as Jordan-Wigner transformation or Bravyi-Kitaev transformation, we can map the fermionic hamiltonian to a spin hamiltonian which is more compatible with quantum computer. For a spin hamiltonian, we can easily use a PQC to construct a trial wavefunction and conduct the VQE algorithm. In the following part of this tutorial, we will demonstrate a complete example of how to use `TensorCircuit` to simulate VQE algorithm on Molecules.

Setup

We should first pip install `openfermion openfermionpyscf` to generate fermionic and qubit Hamiltonian of H₂O molecule based on quantum chemistry computation provided by `openfermion` and `pyscf`.

```
[1]: import numpy as np
from openfermion.chem import MolecularData
from openfermion.transforms import (
    get_fermion_operator,
    jordan_wigner,
    binary_code_transform,
    checksum_code,
    reorder,
)
from openfermion.chem import geometry_from_pubchem
from openfermion.utils import up_then_down
from openfermion.linalg import LinearQubitOperator
from openfermionpyscf import run_pyscf
import tensorflow as tf

import tensorcircuit as tc

K = tc.set_backend("tensorflow")
```

Generate Hamiltonian

- Get molecule energy info and molecule orbitals

```
[2]: multiplicity = 1
basis = "sto-3g"
# 14 spin orbitals for H2O
geometry = geometry_from_pubchem("h2o")
description = "h2o"
molecule = MolecularData(geometry, basis, multiplicity, description=description)
molecule = run_pyscf(molecule, run_mp2=True, run_cisd=True, run_ccsd=True, run_fci=True)
print(molecule.fci_energy, molecule.ccsl_energy, molecule.hf_energy)

-75.0155301894916 -75.01540899923558 -74.96444758276998
```

- Get Fermionic Hamiltonian

```
[3]: mh = molecule.get_molecular_hamiltonian()
```

```
[4]: fh = get_fermion_operator(mh)
```

```
[5]: print(fh.terms[((0, 1), (0, 0))]) # coefficient of C0^\dagger C_0
-32.68991541360029
```

- Transform into qubit Hamiltonian

```
[6]: # The normal transformation such as JW or BK requires 14 qubits for H2O's 14 orbitals

a = jordan_wigner(fh)
LinearQubitOperator(a).n_qubits
```

[6]: 14

We can use binary code to save two further qubits, as the number of spin up and spin down filling is both 5 (5/odd electrons in 7 orbitals)

```
[7]: b = binary_code_transform(reorder(fh, up_then_down), 2 * checksum_code(7, 1))
# 7 is 7 spin polarized orbitals, and 1 is for odd occupation
LinearQubitOperator(b).n_qubits
```

[7]: 12

```
[8]: print(b.terms[((0, "Z"),)]) # coefficient of Z_0 Pauli-string
12.412562749393349
```

- Transform the qubit Hamiltonian in openfermion to the format in TensorCircuit

```
[9]: lsb, wb = tc.templates.chems.get_ps(b, 12)
lsa, wa = tc.templates.chems.get_ps(a, 14)
```

- Inspect Hamiltonian in matrix form

```
[10]: ma = tc.quantum.PauliStringSum2COO_numpy(lsa, wa)
```

```
[11]: mb = tc.quantum.PauliStringSum2COO_numpy(lsb, wb)
```

```
[12]: mad, mbd = ma.todense(), mb.todense()
```

The corresponding Hartree Fock product state in these two types of Hamiltonian

```
[13]: bin(np.argmin(np.diag(mad)))
```

[13]: '0b11111111110000'

```
[14]: bin(np.argmin(np.diag(mbd)))
```

[14]: '0b11110111110'

VQE Setup

We can in principle evaluate each Pauli string of the Hamiltonian as an expectation measurement, but it costs lots of simulation time, instead we fuse them as a Hamiltonian matrix as shown above to run the VQE.

- Using dense matrix expectation

```
[15]: n = 12
depth = 4
mbd_tf = tc.array_to_tensor(mbd)

def vqe(param):
    c = tc.Circuit(n)
    for i in [0, 1, 2, 3, 4, 6, 7, 8, 9, 10]:
        c.X(i)
    for j in range(depth):
        for i in range(n - 1):
            c.exp1(i, i + 1, unitary=tc.gates._xx_matrix, theta=param[j, i, 0])
        for i in range(n):
            c.rx(i, theta=param[j, i, 1])
        for i in range(n):
            c.ry(i, theta=param[j, i, 2])
        for i in range(n):
            c.rz(i, theta=param[j, i, 3])
    return tc.templates.measurements.operator_expectation(c, mbd_tf)
```

```
[16]: vags = tc.backend.jit(tc.backend.value_and_grad(vqe))
lr = tf.keras.optimizers.schedules.ExponentialDecay(
    decay_rate=0.5, decay_steps=300, initial_learning_rate=0.5e-2
)
opt = tc.backend.optimizer(tf.keras.optimizers.Adam(lr))

param = tc.backend.implicit_randn(shape=[depth, n, 4], stddev=0.02, dtype="float32")
for i in range(600):
    e, g = vags(param)
    param = opt.update(g, param)
    if i % 100 == 0:
        print(e)

tf.Tensor(-74.76671, shape=(), dtype=float32)
tf.Tensor(-74.95493, shape=(), dtype=float32)
tf.Tensor(-74.95319, shape=(), dtype=float32)
tf.Tensor(-74.954315, shape=(), dtype=float32)
tf.Tensor(-74.956116, shape=(), dtype=float32)
tf.Tensor(-74.95809, shape=(), dtype=float32)
```

- Using sparse matrix expectation

We can also use the sparse Hamiltonian matrix for circuit expectation evaluation, the only difference is to replace `mbd_tf` with `mb_tf`

```
[17]: mb_tf = tc.backend.coo_sparse_matrix(
    np.transpose(np.stack([mb.row, mb.col])), mb.data, shape=(2**n, 2**n)
)
```

A micro-benchmark between sparse matrix evaluation and dense matrix evaluation for expectation in terms of time, sparse always wins in terms of space, of course.

```
[18]: def dense_expt(param):
    c = tc.Circuit(n)
    for i in range(n):
        c.H(i)
        c.rx(i, theta=param[i])
    return tc.templates.measurements.operator_expectation(c, mbd_tf)
```

```
def sparse_expt(param):
    c = tc.Circuit(n)
    for i in range(n):
        c.H(i)
        c.rx(i, theta=param[i])
    return tc.templates.measurements.operator_expectation(c, mb_tf)
```

```
[19]: dense_vag = tc.backend.jit(tc.backend.value_and_grad(dense_expt))
sparse_vag = tc.backend.jit(tc.backend.value_and_grad(sparse_expt))

v0, g0 = dense_vag(tc.backend.ones([n]))
v1, g1 = sparse_vag(tc.backend.ones([n]))

# consistency check

np.testing.assert_allclose(v0, v1, atol=1e-5)
np.testing.assert_allclose(g0, g1, atol=1e-5)
```

```
[20]: %timeit dense_vag(tc.backend.ones([n]))
30.7 ms ± 1.45 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[21]: %timeit sparse_vag(tc.backend.ones([n]))
3.6 ms ± 63 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Therefore, sparse matrix evaluation also saves time apart from space, which is always recommended.

3.1.8 VQE on 1D TFIM with Different Hamiltonian Representation

Overview

For the ground state preparation of a Hamiltonian H in VQE, we need to calculate the expectation value of Hamiltonian H , i.e., $\langle 0^N | U^\dagger(\theta) H U(\theta) | 0^N \rangle$ and update the parameters θ in $U(\theta)$ based on gradient descent. In this tutorial, we will show four ways supported in TensorCircuit to calculate $\langle H \rangle$:

- 1, $\langle H \rangle = \sum_i \langle h_i \rangle$, where h_i are the Pauli-string operators;
- 2, $\langle H \rangle$ where H is a sparse matrix;
- 3, $\langle H \rangle$ where H is a dense matrix;
- 4, expectation value of the Matrix Product Operator (MPO).

We consider transverse field ising model (TFIM) here, which reads

$$H = \sum_i \sigma_i^x \sigma_{i+1}^x - \sum_i \sigma_i^z,$$

where :math: ‘ σ_i^x, z ’ are Pauli matrices of the :math: ‘ i ’ – th qubit.

Setup

```
[1]: import numpy as np
import tensorflow as tf
import tensorcircuit as tc
import tensornetwork as tn
from tensorcircuit.templates.measurements import operator_expectation
from tensorcircuit.quantum import quantum_constructor

tc.set_backend("tensorflow")
tc.set_dtype("complex128")
dtype = np.complex128

xx = tc.gates._xx_matrix # xx gate matrix to be utilized
```

Parameters

```
[2]: n = 4 # The number of qubits
nlayers = 2 # The number of circuit layers
ntrials = 2 # The number of random circuit instances
```

Parameterized Quantum Circuits

```
[3]: def tfi_circuit(param, n, nlayers):
    c = tc.Circuit(n)
    for j in range(nlayers):
        for i in range(n - 1):
            c.exp1(i, i + 1, unitary=xx, theta=param[2 * j, i])
        for i in range(n):
            c.rz(i, theta=param[2 * j + 1, i])
    return c
```

Pauli-string Operators

Energy

```
[4]: def tfi_energy(c: tc.Circuit, j: float = 1.0, h: float = -1.0):
    e = 0.0
    n = c._nqubits
    for i in range(n):
        e += h * c.expectation((tc.gates.z(), [i])) # <Z_i>
```

(continues on next page)

(continued from previous page)

```
for i in range(n - 1): # OBC
    e += j * c.expectation(
        (tc.gates.x(), [i]), (tc.gates.x(), [(i + 1) % n])
    ) # <X_iX_{i+1}>
return tc.backend.real(e)
```

[5]:

```
def vqe_tfirm_paulistring(param, n, nlayers):
    c = tfi_circuit(param, n, nlayers)
    e = tfirm_energy(c)
    return e
```

[6]:

```
vqe_tfirm_paulistring_vvag = tc.backend.jit(
    tc.backend.vectorized_value_and_grad(vqe_tfirm_paulistring)
) # use vvag to get losses and gradients of different random circuit instances
```

Main Optimization Loop

[7]:

```
def batched_train_step_paulistring_tf(batch, n, nlayers, maxiter=10000):
    param = tf.Variable(
        initial_value=tf.random.normal(
            shape=[batch, nlayers * 2, n], stddev=0.1, dtype=getattr(tf, tc.rdtypestr)
        )
    ) # initial parameters
    opt = tf.keras.optimizers.Adam(1e-2)
    for i in range(maxiter):
        e, grad = vqe_tfirm_paulistring_vvag(
            param.value(), n, nlayers
        ) # energy and gradients
        opt.apply_gradients([(grad, param)])
        if i % 200 == 0:
            print(e)
    return e
```

```
batched_train_step_paulistring_tf(ntrials, n, nlayers, 400)
```

2022-03-16 14:09:12.304188: I tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX2 FMA To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.

```
tf.Tensor([-4.00557571 -3.97372412], shape=(2,), dtype=float64)
tf.Tensor([-4.68208061 -4.684804], shape=(2,), dtype=float64)
```

[7]:

```
<tf.Tensor: shape=(2,), dtype=float64, numpy=array([-4.75683202, -4.73689914])>
```

Sparse Matrix, Dense Matrix, and MPO

Hamiltonian

```
[8]: def tfi_hamiltonian():
    h = []
    w = []

    ### Z
    for i in range(n):
        h.append([])
        w.append(-1.0) # weight
        for j in range(n):
            if j == i:
                h[i].append(3)
            else:
                h[i].append(0)

    ### XX
    for i in range(n - 1):
        h.append([])
        w.append(1.0) # weight
        for j in range(n):
            if j == (i + 1) % n or j == i:
                h[i + n].append(1)
            else:
                h[i + n].append(0)

    hamiltonian_sparse = tc.quantum.PauliStringSum2COO(
        tf.constant(h, dtype=tf.complex128), tf.constant(w, dtype=tf.complex128)
    ) # sparse matrix

    hamiltonian_dense = tc.quantum.PauliStringSum2Dense(
        tf.constant(h, dtype=tf.complex128), tf.constant(w, dtype=tf.complex128)
    ) # dense matrix
    return hamiltonian_sparse, hamiltonian_dense
```

Generate QuOperator

```
[9]: def quoperator_mpo(tfi_mpo):
    tfi_mpo = tfi_mpo.tensors

    mpo = []
    for i in range(n):
        mpo.append(tn.Node(tfi_mpo[i]))

    for i in range(n - 1):
        tn.connect(mpo[i][1], mpo[i + 1][0])

    tfi_mpo = quantum_constructor(
        [mpo[i][-1] for i in range(n)], # out_edges
```

(continues on next page)

(continued from previous page)

```
[mpo[i][-2] for i in range(n)], # in_edges
[],
[mpo[0][0], mpo[-1][1]], # ignore_edges
)
return tfi_mpo
```

Energy

```
[10]: def vqe_tfir(param, n, nlayers, hamiltonian):
    c = tfi_circuit(param, n, nlayers)
    e = operator_expectation(
        c, hamiltonian
    ) # in operator_expectation, the "hamiltonian" can be sparse matrix, dense matrix,
    ↪ or mpo
    return e
```

```
[11]: vqe_tfir_vvag = tc.backend.jit(
    tc.backend.vectorized_value_and_grad(vqe_tfir)
) # use vvag to get losses and gradients of different random circuit instances
```

Main Optimization Loop

```
[12]: def batched_train_step_tf(batch, n, nlayers, hamiltonian, maxiter=10000):
    param = tf.Variable(
        initial_value=tf.random.normal(
            shape=[batch, nlayers * 2, n], stddev=0.1, dtype=getattr(tf, tc.rdtypestr)
        )
    ) # initial parameters

    opt = tf.keras.optimizers.Adam(1e-2)
    for i in range(maxiter):
        e, grad = vqe_tfir_vvag(
            param.value(), n, nlayers, hamiltonian
        ) # energy and gradients
        opt.apply_gradients([(grad, param)])
        if i % 200 == 0:
            print(e)
    return e
```

Sparse Matrix, Dense Matrix, and MPO

```
[13]: (
    hamiltonian_sparse,
    hamiltonian_dense,
) = tfi_hamiltonian() # hamiltonian: sparse matrix, dense matrix

Jx = np.array([1.0 for _ in range(n - 1)]) # strength of xx interaction (OBC)
Bz = np.array([1.0 for _ in range(n)]) # strength of transverse field
hamiltonian_mpo = tn.matrixproductstates.mpo.FiniteTFI(
    Jx, Bz, dtype=dtype
) # matrix product operator
hamiltonian_mpo = quoperator_mpo(hamiltonian_mpo) # generate QuOperator from mpo

2022-03-16 14:09:30.874680: I tensorflow/compiler/xla/service/service.cc:171] XLA_
˓→service 0x7fd94503abc0 initialized for platform Host (this does not guarantee that XLA_
˓→will be used). Devices:
2022-03-16 14:09:30.874726: I tensorflow/compiler/xla/service/service.cc:179] ˓
˓→StreamExecutor device (0): Host, Default Version
2022-03-16 14:09:31.014341: I tensorflow/compiler/jit/xla_compilation_cache.cc:351] ˓
˓→Compiled cluster using XLA! This line is logged at most once for the lifetime of the_
˓→process.
```



```
[14]: batched_train_step_tf(ntrials, n, nlayers, hamiltonian_sparse, 400) # sparse matrix
WARNING:tensorflow:Using a while_loop for converting SparseTensorDenseMatMul
tf.Tensor([-4.04418884 -3.22012342], shape=(2,), dtype=float64)
tf.Tensor([-4.67668625 -4.66761143], shape=(2,), dtype=float64)

[14]: <tf.Tensor: shape=(2,), dtype=float64, numpy=array([-4.74512239, -4.69965641])>
```



```
[15]: batched_train_step_tf(ntrials, n, nlayers, hamiltonian_dense, 400) # dense matrix
tf.Tensor([-3.72705324 -3.99225849], shape=(2,), dtype=float64)
tf.Tensor([-4.70773521 -4.7330719 ], shape=(2,), dtype=float64)

[15]: <tf.Tensor: shape=(2,), dtype=float64, numpy=array([-4.74236986, -4.7559722 ])>
```



```
[16]: batched_train_step_tf(ntrials, n, nlayers, hamiltonian_mpo, 400) # mpo
tf.Tensor([-3.9129593 -3.44283879], shape=(2,), dtype=float64)
tf.Tensor([-4.68271695 -4.67584305], shape=(2,), dtype=float64)

[16]: <tf.Tensor: shape=(2,), dtype=float64, numpy=array([-4.75283209, -4.75535872])>
```

3.1.9 MERA

Overview

In this tutorial, we'll show you how to implement MERA (multi-scale entangled renormalization ansatz) with TensorCircuit, but not in physics perspective.

Background

MERA is a kind of VQE starts from only one qubit in the 0 state, and progressively enlarges the Hilbert space by tensoring on new qubits. In the MERA we are going to train (denoted by $U(\theta)$), we use parameterized quantum gates $e^{i\theta XX}$, $e^{i\theta ZZ}$ as two-qubit gates and $e^{i\theta X}$, $e^{i\theta Z}$ as single-qubit gates. The Hamiltonian we choose as the example is from TFIM as $\hat{H}_{Ising} = J \sum_i Z_i Z_{i+1} - B_x \sum_i X_i$. And the loss function to be minimized in this task is $\mathcal{L}_{MERA}(\theta) = \langle 0^n | U(\theta)^\dagger \hat{H} U(\theta) | 0^n \rangle$.

Setup

```
[1]: import numpy as np
import tensorflow as tf
import tensorcircuit as tc

tc.set_backend("tensorflow")
tc.set_dtype("complex128")

[1]: ('complex128', 'float64')
```

Energy

We first design the Hamiltonian energy expectation function as loss.

$$\hat{H}_{Ising} = J \sum_i Z_i Z_{i+1} - B_x \sum_i X_i$$

```
[2]: def energy(c: tc.Circuit, j: float = 1.0, hx: float = 1.0):
    e = 0.0
    n = c._nqubits
    # <Z_i Z_{i+1}>
    for i in range(n - 1):
        e += j * c.expectation((tc.gates.z(), [i]), (tc.gates.z(), [i + 1]))
    # <X_i>
    for i in range(n):
        e -= hx * c.expectation((tc.gates.x(), [i]))
    return tc.backend.real(e)
```

MERA circuit

Now we design the circuit. We use θ as input.

```
[5]: def MERA(params, n):
    params = tc.backend.cast(params, "complex128")
    c = tc.Circuit(n)

    idx = 0 # index of params

    for i in range(n):
        c.rx(i, theta=params[2 * i])
```

(continues on next page)

(continued from previous page)

```

    c.rz(i, theta=params[2 * i + 1])
    idx += 2 * n

    for n_layer in range(1, int(np.log2(n)) + 1):
        n_qubit = 2**n_layer # number of qubits involving
        step = int(n / n_qubit)

        # even
        for i in range(step, n - step, 2 * step):
            c.exp1(i, i + step, theta=params[idx], unitary=tc.gates._xx_matrix)
            c.exp1(i, i + step, theta=params[idx + 1], unitary=tc.gates._zz_matrix)
            idx += 2

        # odd
        for i in range(0, n, 2 * step):
            c.exp1(i, i + step, theta=params[idx], unitary=tc.gates._xx_matrix)
            c.exp1(i, i + step, theta=params[idx + 1], unitary=tc.gates._zz_matrix)
            idx += 2

        # single qubit
        for i in range(0, n, step):
            c.rx(i, theta=params[idx])
            c.rz(i, theta=params[idx + 1])
            idx += 2

    # measure
    e = energy(c)
    return e
    # return c, idx

```

We can visualize the MERA circuit.

Hint: Please change return to `return c, idx`, which will only be used here. After visualization, don't forget to restore the return and run the code block above again.

```
[4]: n = 8
cirq, idx = MERA(np.zeros(1000), n)
print("The number of parameters is", idx)
cirq.draw()
```

The number of parameters is 66

[4]:

Train

Now we can train the MERA circuit with tensorflow.

```
[6]: MERA_tfir_vvag = tc.backend.jit(tc.backend.vectorized_value_and_grad(MERA))

def batched_train(n, batch=10, maxiter=10000, lr=0.005):
    params = tf.Variable(
        initial_value=tf.random.normal(
```

(continues on next page)

(continued from previous page)

```

        shape=[batch, idx], stddev=1, dtype=getattr(tf, tc.rdtypestr)
    )
)
opt = tf.keras.optimizers.Adam(lr)
lowest_energy = 1e5
for i in range(maxiter):
    e, grad = MERA_tfir_vvag(params, n)
    opt.apply_gradients([(grad, params)])
    if tf.reduce_min(e) < lowest_energy:
        lowest_energy = tf.reduce_min(e)
    if i % 200 == 0:
        print(e)
return lowest_energy

n = 8
lowest_energy = batched_train(n, batch=5, maxiter=2000, lr=0.007)

tf.Tensor([-0.6449017  0.14083987  0.17227418  1.42731099  0.93767164], shape=(5,),  

˓→dtype=float64)  

tf.Tensor([-9.57952648 -9.15354269 -9.53415983 -9.55291257 -9.46880555], shape=(5,),  

˓→dtype=float64)  

tf.Tensor([-9.63166728 -9.60922826 -9.59883555 -9.66639936 -9.60174669], shape=(5,),  

˓→dtype=float64)  

tf.Tensor([-9.65441326 -9.61830383 -9.6219077 -9.68289435 -9.61427165], shape=(5,),  

˓→dtype=float64)  

tf.Tensor([-9.66991104 -9.6307931 -9.64993901 -9.71396225 -9.63848947], shape=(5,),  

˓→dtype=float64)  

tf.Tensor([-9.67960751 -9.64303661 -9.67696885 -9.76317346 -9.6507455 ], shape=(5,),  

˓→dtype=float64)  

tf.Tensor([-9.68303361 -9.6575349 -9.70118521 -9.7740601 -9.65751254], shape=(5,),  

˓→dtype=float64)  

tf.Tensor([-9.68481667 -9.67473162 -9.71392119 -9.78200161 -9.66880068], shape=(5,),  

˓→dtype=float64)  

tf.Tensor([-9.6864865 -9.67835678 -9.73033137 -9.79128949 -9.68317883], shape=(5,),  

˓→dtype=float64)  

tf.Tensor([-9.68762425 -9.67928153 -9.77502182 -9.79465957 -9.69252806], shape=(5,),  

˓→dtype=float64)

```

Compare

We can compare the ground energy we get by MERA with DMRG.

```
[7]: # DMRG
import quimb

h = quimb.tensor.tensor_gen.MPO_ham_ising(n, j=4.0, bx=2.0, S=0.5, cyclic=False)
dmrg = quimb.tensor.tensor_dmrg.DMRG(
    h, bond_dims=[10, 20, 100, 100, 200], cutoffs=1e-13
)
dmrg.solve(tol=1e-9, verbosity=0)
```

(continues on next page)

(continued from previous page)

```
energy_DMRG = dmrg.energy

# Compare
print("DMRG solution: ", energy_DMRG)
print("MERA solution: ", lowest_energy.numpy())

DMRG solution: -9.837951447459426
MERA solution: -9.795198473308487
```

3.1.10 Gradient Evaluation Efficiency Comparison

Overview

In this tutorial, we compare the efficiency of gradient and gradient-like object (such as quantum Fisher information) evaluation via automatical differentiation framework provided by TensorCircuit and the traditional parameter shift framework provided by Qiskit

Setup

We import necessary packages and modules from Qiskit and TensorCircuit

```
[1]: import time
import numpy as np
from functools import reduce
from operator import xor

from qiskit.opflow import I, X, Z, StateFn, CircuitStateFn, SummedOp
from qiskit.circuit import QuantumCircuit, ParameterVector
from scipy.optimize import minimize
from qiskit.opflow.gradients import Gradient, QFI, Hessian

import tensorcircuit as tc
from tensorcircuit import experimental
```

Qiskit Gradient Framework Benchmark

Since Qiskit is **TOO** slow in terms of gradient evaluation, we use small systems to do the benchmark in Jupyter to save time, for larger size and deep circuits, the efficiency difference will become more evident.

The three gradient like tasks are Gradient, Quantum Fisher Information(QFI), and Hessian evaluation.

```
[2]: def benchmark(f, *args, trials=10):
    time0 = time.time()
    r = f(*args)
    time1 = time.time()
    for _ in range(trials):
        r = f(*args)
    time2 = time.time()
    if trials > 0:
        time21 = (time2 - time1) / trials
```

(continues on next page)

(continued from previous page)

```

else:
    time21 = 0
ts = (time1 - time0, time21)
print("staging time: %.6f s" % ts[0])
if trials > 0:
    print("running time: %.6f s" % ts[1])
return r, ts

def grad_qiskit(n, l, trials=2):
    hamiltonian = reduce(xor, [X for _ in range(n)])
    wavefunction = QuantumCircuit(n)
    params = ParameterVector("theta", length=3 * n * l)
    for j in range(l):
        for i in range(n - 1):
            wavefunction.cnot(i, i + 1)
        for i in range(n):
            wavefunction.rx(params[3 * n * j + i], i)
        for i in range(n):
            wavefunction.rz(params[3 * n * j + i + n], i)
        for i in range(n):
            wavefunction.rx(params[3 * n * j + i + 2 * n], i)

    # Define the expectation value corresponding to the energy
    op = ~StateFn(hamiltonian) @ StateFn(wavefunction)
    grad = Gradient().convert(operator=op, params=params)

    def get_grad_qiskit(values):
        value_dict = {params: values}
        grad_result = grad.assign_parameters(value_dict).eval()
        return grad_result

    return benchmark(get_grad_qiskit, np.ones([3 * n * l]), trials=trials)

def qfi_qiskit(n, l, trials=0):
    hamiltonian = reduce(xor, [X for _ in range(n)])
    wavefunction = QuantumCircuit(n)
    params = ParameterVector("theta", length=3 * n * l)
    for j in range(l):
        for i in range(n - 1):
            wavefunction.cnot(i, i + 1)
        for i in range(n):
            wavefunction.rx(params[3 * n * j + i], i)
        for i in range(n):
            wavefunction.rz(params[3 * n * j + i + n], i)
        for i in range(n):
            wavefunction.rx(params[3 * n * j + i + 2 * n], i)

    nat_grad = QFI().convert(operator=StateFn(wavefunction), params=params)

    def get_qfi_qiskit(values):

```

(continues on next page)

(continued from previous page)

```

value_dict = {params: values}
grad_result = nat_grad.assign_parameters(value_dict).eval()
return grad_result

return benchmark(get_qfi_qiskit, np.ones([3 * n * 1]), trials=trials)

def hessian_qiskit(n, l, trials=0):
    hamiltonian = reduce(xor, [X for _ in range(n)])
    wavefunction = QuantumCircuit(n)
    params = ParameterVector("theta", length=3 * n * l)
    for j in range(l):
        for i in range(n - 1):
            wavefunction.cnot(i, i + 1)
        for i in range(n):
            wavefunction.rx(params[3 * n * j + i], i)
        for i in range(n):
            wavefunction.rz(params[3 * n * j + i + n], i)
        for i in range(n):
            wavefunction.rx(params[3 * n * j + i + 2 * n], i)

    # Define the expectation value corresponding to the energy
    op = ~StateFn(hamiltonian) @ StateFn(wavefunction)
    grad = Hessian().convert(operator=op, params=params)

    def get_hs_qiskit(values):
        value_dict = {params: values}
        grad_result = grad.assign_parameters(value_dict).eval()
        return grad_result

    return benchmark(get_hs_qiskit, np.ones([3 * n * 1]), trials=trials)

```

[3]: g0, _ = grad_qiskit(6, 3) # gradient

staging time: 1.665786 s
running time: 1.474930 s

[4]: qfi0, _ = qfi_qiskit(6, 3) # QFI

staging time: 47.131374 s

[5]: hs0, _ = hessian_qiskit(6, 3) # Hessian

staging time: 80.495983 s

TensorCircuit Automatic Differentiation Benchmark

We benchmark on the same problems defined in the Qiskit part above, and we can see the speed boost! In fact, for a moderate 10-qubit 4-blocks system, QFI evaluation is accelerated more than 10^6 times! (Note how staging time for jit can be amortized and only running time relevant. In the Qiskit case, there is no jit and thus the running time is the same as the staging time.)

```
[6]: def grad_tc(n, l, trials=10):
    def f(params):
        c = tc.Circuit(n)
        for j in range(l):
            for i in range(n - 1):
                c.cnot(i, i + 1)
        for i in range(n):
            c.rx(i, theta=params[3 * n * j + i])
        for i in range(n):
            c.rz(i, theta=params[3 * n * j + i + n])
        for i in range(n):
            c.rx(i, theta=params[3 * n * j + i + 2 * n])
        return tc.backend.real(c.expectation(*[[tc.gates.x(), [i]] for i in range(n)]))

    get_grad_tc = tc.backend.jit(tc.backend.grad(f))
    return benchmark(get_grad_tc, tc.backend.ones([3 * n * l], dtype="float32"))

def qfi_tc(n, l, trials=10):
    def s(params):
        c = tc.Circuit(n)
        for j in range(l):
            for i in range(n - 1):
                c.cnot(i, i + 1)
            for i in range(n):
                c.rx(i, theta=params[3 * n * j + i])
            for i in range(n):
                c.rz(i, theta=params[3 * n * j + i + n])
            for i in range(n):
                c.rx(i, theta=params[3 * n * j + i + 2 * n])
        return c.state()

    get_qfi_tc = tc.backend.jit(experimental.qng(s, mode="fwd"))
    return benchmark(get_qfi_tc, tc.backend.ones([3 * n * l], dtype="float32"))

def hessian_tc(n, l, trials=10):
    def f(params):
        c = tc.Circuit(n)
        for j in range(l):
            for i in range(n - 1):
                c.cnot(i, i + 1)
            for i in range(n):
                c.rx(i, theta=params[3 * n * j + i])
            for i in range(n):
                c.rz(i, theta=params[3 * n * j + i + n])
```

(continues on next page)

(continued from previous page)

```

for i in range(n):
    c.rx(i, theta=params[3 * n * j + i + 2 * n])
return tc.backend.real(c.expectation(*[[tc.gates.x(), [i]] for i in range(n)]))

get_hs_tc = tc.backend.jit(tc.backend.hessian(f))
return benchmark(get_hs_tc, tc.backend.ones([3 * n * 1], dtype="float32"))

```

```
[7]: for k in ["tensorflow", "jax"]:
    with tc.runtime_backend(k):
        print("-----")
        print("%s backend" % k)
        print("gradient")
        g1, _ = grad_tc(6, 3)
        print("quantum Fisher information")
        qfi1, _ = qfi_tc(6, 3)
        print("Hessian")
        hs1, _ = hessian_tc(6, 3)

-----
tensorflow backend
gradient
staging time: 15.889095 s
running time: 0.001126 s
quantum Fisher information
WARNING:tensorflow:The dtype of the watched primal must be floating (e.g. tf.float32), ↴
          got tf.complex64
staging time: 53.973453 s
running time: 0.002332 s
Hessian
staging time: 96.066412 s
running time: 0.004685 s

WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and ↴
          rerun for more info.)

-----
jax backend
gradient
staging time: 4.696845 s
running time: 0.000105 s
quantum Fisher information
staging time: 4.618631 s
running time: 0.000386 s
Hessian
staging time: 23.591966 s
running time: 0.001681 s

```

The results obtained from the two methods are consistent by the following checks

- Gradient

```
[8]: np.testing.assert_allclose(g0, g1, atol=1e-4)
```

- Quantum Fisher Information(QFI)

```
[9]: np.testing.assert_allclose(qfi0, 4.0 * qfi1, atol=1e-3)
```

- Hessian

```
[10]: np.testing.assert_allclose(hs0, hs1, atol=1e-4)
```

3.1.11 The usage of contractor

Overview

In this tutorial, we will demonstrate how to utilize different types of TensorNetwork contractors for the circuit simulation to achieve a better space-time tradeoff. The customization of the contractor is the main highlight for the TensorCircuit package since a better contractor can make better use of the power of the TensorNetwok simulation engine. [WIP]

Setup

```
[1]: import tensorcircuit as tc
import numpy as np
import cotengra as ctg
import opt_einsum as oem

K = tc.set_backend("tensorflow")
```

Testbed System

We provide tensor networks for two circuits, and test the contraction efficiency for these two systems, the first system is small while the second one is large.

```
[2]: # get state for small system
def small_tn():
    n = 10
    d = 4
    param = K.ones([2 * d, n])
    c = tc.Circuit(n)
    c = tc.templates.blocks.example_block(c, param, nlayers=d)
    return c.state()
```

```
[3]: # get expectation for extra large system
def large_tn():
    n = 60
    d = 8
    param = K.ones([2 * d, n])
    c = tc.Circuit(n)
    c = tc.templates.blocks.example_block(c, param, nlayers=d, is_split=True)
    # the two qubit gate is split and truncated with SVD decomposition
    return c.expectation([tc.gates.z(), [n // 2]], reuse=False)
```

Opt-einsum

There are several contractor optimizers provided by opt-einsum and shipped with the TensorNetwork package. Since TensorCircuit is built on top of TensorNetwork, we can use these simple contractor optimizers. Though for any moderate system, only greedy optimizer works, other optimizers come with exponential scaling and fail in circuit simulation scenarios.

We always set `contraction_info=True` (default `False`) for the contractor system, which will print contraction information summary including contraction size, flops, and writes. For the definition of these metrics, also refer to cotengra docs.

```
[4]: # if we set nothing, the default optimizer is greedy, i.e.
tc.set_contractor("greedy", debug_level=2, contraction_info=True)
# We set debug_level=2 to not really run the contraction computation
# i.e. by set debug_level>0, only contraction information and the return shape is
# correct, the result is wrong

[4]: functools.partial(<function custom at 0x7fd5a0a3d430>, optimizer=<function contraction_
# info_decorator.<locals>.new_algorithm at 0x7fd588e281f0>, memory_limit=None, debug_
# level=2)

[5]: small_tn()
-----
contraction cost summary -----
log10[FLOPs]: 5.132 log2[SIZE]: 11 log2[WRITE]: 13.083

[5]: <tf.Tensor: shape=(1024,), dtype=complex64, numpy=
array([0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
      dtype=complex64)>

[6]: large_tn()
-----
contraction cost summary -----
log10[FLOPs]: 17.766 log2[SIZE]: 44 log2[WRITE]: 49.636

[6]: <tf.Tensor: shape=(), dtype=complex64, numpy=0j>

[7]: # we can use more fancy contractor in opt-einsum but not in tensornetwork
# custom_stateful is used for contraction path finder which has a life cycle of one-time
# path solver
tc.set_contractor(
    "custom_stateful",
    optimizer=oem.RandomGreedy,
    max_time=60,
    max_repeats=128,
    minimize="size",
    debug_level=2,
    contraction_info=True,
)
<functools.partial object at 0x7fd5a0a3d4c0>

[7]: functools.partial(<function custom_stateful at 0x7fd5a0a3d4c0>, optimizer=<class 'opt_einsum.path_random.RandomGreedy'>, opt_conf=None, contraction_info=True, debug_level=2,
# max_time=60, max_repeats=128, minimize='size')

[8]: small_tn()
```

```
----- contraction cost summary -----
log10[FLOPs]: 4.925 log2[SIZE]: 10 log2[WRITE]: 12.531
```

[8]: <tf.Tensor: shape=(1024,), dtype=complex64, numpy=array([0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j], dtype=complex64)>

[9]: large_tn()

```
----- contraction cost summary -----
log10[FLOPs]: 11.199 log2[SIZE]: 26 log2[WRITE]: 28.183
```

[9]: <tf.Tensor: shape=(), dtype=complex64, numpy=0j>

Cotengra

for more advanced contractors, we ask help for the sota contractor optimizer on the market: cotengra

[10]: opt = ctg.ReusableHyperOptimizer(
 methods=["greedy", "kahypar"],
 parallel=True,
 minimize="write",
 max_time=120,
 max_repeats=1024,
 progbar=True,
)
tc.set_contractor(
 "custom", optimizer=opt, preprocessing=True, contraction_info=True, debug_level=2
)
for more setup on cotengra optimizers, see the reference in
https://cotengra.readthedocs.io/en/latest/advanced.html
preprocessing=True will merge all single-qubit gates into neighboring two-qubit gates

[10]: functools.partial(<function custom at 0x7fd5a0a3d430>, optimizer=<function contraction_info_decorator.<locals>.new_algorithm at 0x7fd588e28ee0>, memory_limit=None, debug_level=2, preprocessing=True)

[11]: small_tn()

```
log2[SIZE]: 10.00 log10[FLOPs]: 4.90: 100%|| 1024/1024 [00:28<00:00, 35.45it/s]
```

```
----- contraction cost summary -----
```

```
log10[FLOPs]: 4.900 log2[SIZE]: 10 log2[WRITE]: 12.255
```

[11]: <tf.Tensor: shape=(1024,), dtype=complex64, numpy=array([0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j], dtype=complex64)>

[12]: large_tn()

```
log2[SIZE]: 20.00 log10[FLOPs]: 9.50: 4% | 43/
→ 1024 [02:09<49:22, 3.02s/it]
```

```
----- contraction cost summary -----
log10[FLOPs]:  9.501  log2[SIZE]:  20  log2[WRITE]:  24.090
[12]: <tf.Tensor: shape=(), dtype=complex64, numpy=0j>
```

We can also apply subtree reconf after cotengra find the path, which in general further (and usually greatly) improve flops and write for the contraction. Indeed, the subtree reconf postprocessing is in general more important than increasing the search time or repeats for the optimizer.

```
[13]: opt = ctg.ReusableHyperOptimizer(
    minimize="combo",
    max_repeats=1024,
    max_time=240,
    progbar=True,
)

def opt_reconf(inputs, output, size, **kws):
    tree = opt.search(inputs, output, size)
    tree_r = tree.subtree_reconfigure_forest(
        progbar=True, num_trees=10, num_restarts=20, subtree_weight_what=("size",)
    )
    return tree_r.get_path()

tc.set_contractor(
    "custom",
    optimizer=opt_reconf,
    contraction_info=True,
    preprocessing=True,
    debug_level=2,
)
[13]: functools.partial(<function custom at 0x7fd5a0a3d430>, optimizer=<function contraction_
    ↪info_decorator.<locals>.new_algorithm at 0x7fd58c832700>, memory_limit=None, debug_
    ↪level=2, preprocessing=True)
```

```
[14]: small_tn()

log2[SIZE]: 10.00 log10[FLOPs]: 4.87: 100%|| 1024/1024 [02:29<00:00,  6.86it/s]
log2[SIZE]: 10.00 log10[FLOPs]: 4.86: 100%|| 20/20 [00:31<00:00,  1.57s/it]

----- contraction cost summary -----
log10[FLOPs]:  4.859  log2[SIZE]:  10  log2[WRITE]:  12.583
```

```
[14]: <tf.Tensor: shape=(1024,), dtype=complex64, numpy=
array([0.+0.j, 0.+0.j, 0.+0.j, ..., 0.+0.j, 0.+0.j, 0.+0.j],
      dtype=complex64)>
```

```
[15]: large_tn()

log2[SIZE]: 21.00 log10[FLOPs]: 9.62:   9%| 93/
    ↪1024 [04:04<40:49,  2.63s/it]
log2[SIZE]: 17.00 log10[FLOPs]: 8.66: 100%|| 20/20 [03:08<00:00,  9.44s/it]
```

```
----- contraction cost summary -----
log10[FLOPs]: 8.657 log2[SIZE]: 17 log2[WRITE]: 25.035
[15]: <tf.Tensor: shape=(), dtype=complex64, numpy=0j>
```

We can also extract the tensornetwork directly for the circuit or observable calculations and we can do the contraction by ourselves using any method we like. Moreover, all these contractors or our customized external contractions can still be compatible with jit, automatic differentiation. Specifically, the contraction path solver, though taking some time overhead, is only evaluated once due to the jit infrastructure (note for demonstration usage, we don't decorate our contraction function with K.jit here).

3.1.12 Operator spreading

Overview

In this tutorial, we will introduce the operator spreading that serves as a diagnostic of the chaotic dynamics and information scrambling. We will examine operator spreading as a function of the circuit depth L that can be regarded as the time t in discrete quantum systems. The operator spreading coefficient considered here reads:

where :math : \sigma_y^{(j)} is the Pauli - y matrix at the :math : j - th qubit. :math : O(0) is the Pauli - x matrix located at :math :

A physical picture for the operator spreading is that if the Heisenberg operator growth $O(t)$ does not reach the site j , $[O(t), \sigma_y^{(j)}] = 0$, while the equality will break down when sites $N/2, j$ become correlated inside the causal region.

Setup

```
[1]: import numpy as np
import tensorflow as tf
import tensorcircuit as tc
import matplotlib.pyplot as plt

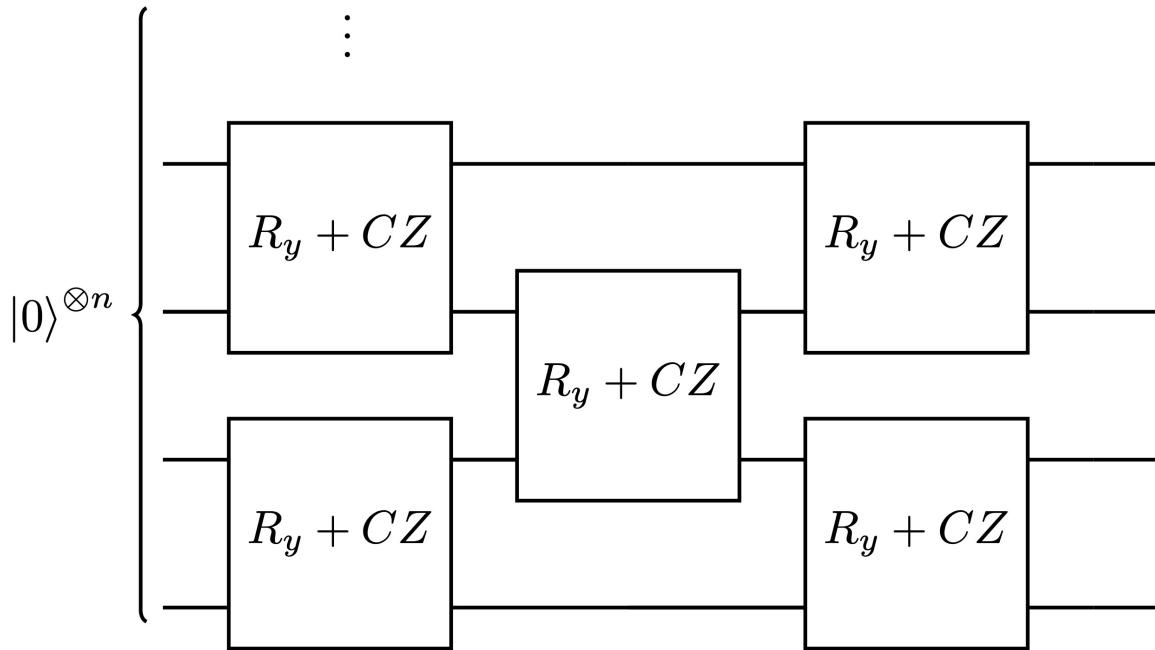
tc.set_backend("tensorflow")
tc.set_dtype("complex128")
dtype = np.complex128
```

Parameters

```
[2]: N = 6 # The number of qubits
L = 2 # The number of circuit layers
num_trial = 1 # The number of random circuit instances
```

The Unitary Matrix $U(t)$ of the Quantum Circuit

The circuit architecture is shown below, each block includes two Pauli rotation gates along the y-axis (R_y) followed by the CZ gate. We can get the unitary matrix $U(t)$ of the circuit by setting the input state as an identity matrix.



```
[3]: @tf.function()
def get_unitary_of_circuit(v):
    layers = len(v)
    c = tc.Circuit(
        N, inputs=np.eye(2**N, dtype=dtype)
    ) # when we choose inputs=np.eye(2**N,...), the output wavefunction
    # is the unitary matrix U(t) of the quantum circuit

    # Ry+Cz
    for layer in range(layers):
        if layer % 2 == 0:
            for i in range(0, N, 2):
                c.ry(i, theta=v[layer, i])
                c.ry(i + 1, theta=v[layer, i + 1])
                c.cz(i, i + 1)
        else:
            for i in range(1, N, 2):
                c.ry(i, theta=v[layer, i])
                c.ry((i + 1) % N, theta=v[layer, (i + 1) % N])
                c.cz(i, (i + 1) % N)
    U = c.wavefunction() # the output of wavefunction is a vector with dim=2^(2N)
    U = tf.reshape(U, [2**N, 2**N]) # reshape vector to matrix
    return U
```

Operator at $t = 0$

```
[4]: # j means the operator locates at the j-th qubit
# a means the type of operator, a=1 (2,3) is the Pauli-x (y,z) matrix
def get_operator(j, a):
    I = tc.gates._i_matrix
    if a == 1:
        = tc.gates._x_matrix # Pauli-x matrix
    elif a == 2:
        = tc.gates._y_matrix # Pauli-y matrix
    elif a == 3:
        = tc.gates._z_matrix # Pauli-z matrix
    h = []
    for i in range(N):
        if i == j:
            h.append()
        else:
            h.append(I)

    # Pauli-a matrix locates at j-th qubit
    operator = h[0]
    for i in range(N - 1):
        operator = np.kron(operator, h[i + 1])
    return tf.cast(operator, dtype=dtype)
```

Operator Spreading

The operator spreading $C_y(j, t) = \frac{1}{2}\text{Tr}([O(t), \sigma_y^{(j)}]^\dagger [O(t), \sigma_y^{(j)}])$ can be written as:

$$C_y(j, t) = 1 - \frac{\text{Re}(\text{Tr}(\sigma_y^{(j)} O(t)^\dagger \sigma_y^{(j)} O(t))))}{2^N}.$$

```
[5]: @tf.function()
def get_spreading(v, _y):
    U = get_unitary_of_circuit(v)
    U_dagger = tf.transpose(U, conjugate=True)
    _x = get_operator(int(N / 2), 1) # O(0)
    _x_t = U_dagger @ _x @ U # O(t)
    _x_t_dagger = tf.transpose(_x_t, conjugate=True)
    C = tf.linalg.trace(_y @ _x_t_dagger @ _y @ _x_t)
    C = 1 - tf.math.real(C) / (2**N)
    return C
```

```
[6]: def main(layers=1):
    # use vmap to get operator spreading coefficients of different random circuit
    ↪instances
    # num_trial: the number of random circuit instances
    # layers: the number of circuit layers
    get_spreading_vmap = tc.backend.vmap(get_spreading, vectorized_argnums=0)
```

(continues on next page)

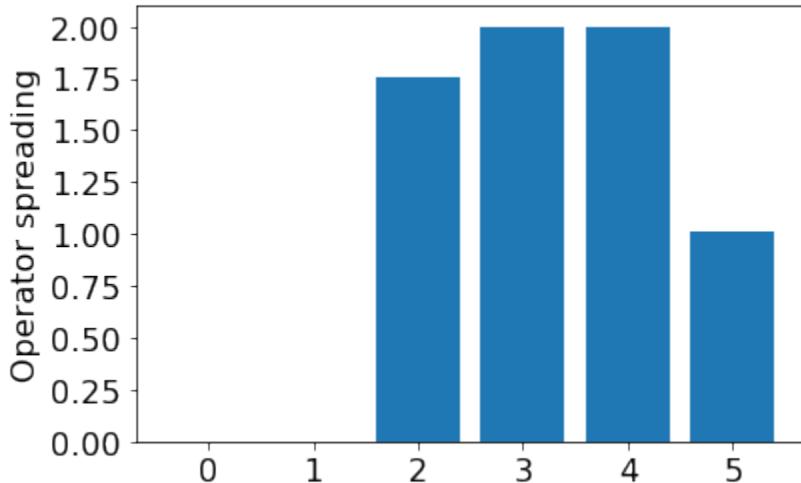
(continued from previous page)

```
C = get_spreading_vmap(
    tf.random.uniform(
        [num_trial, layers, N], minval=0.0, maxval=2 * np.pi, dtype=tf.float64
    ),
    [[get_operator(j, 2) for j in range(N)] for _ in range(num_trial)],
)
return tf.reduce_mean(C, 0)[0] # averaged over different random circuit instances
```

[7]: C = main(L) # operator spreading averaged over different random circuit instances

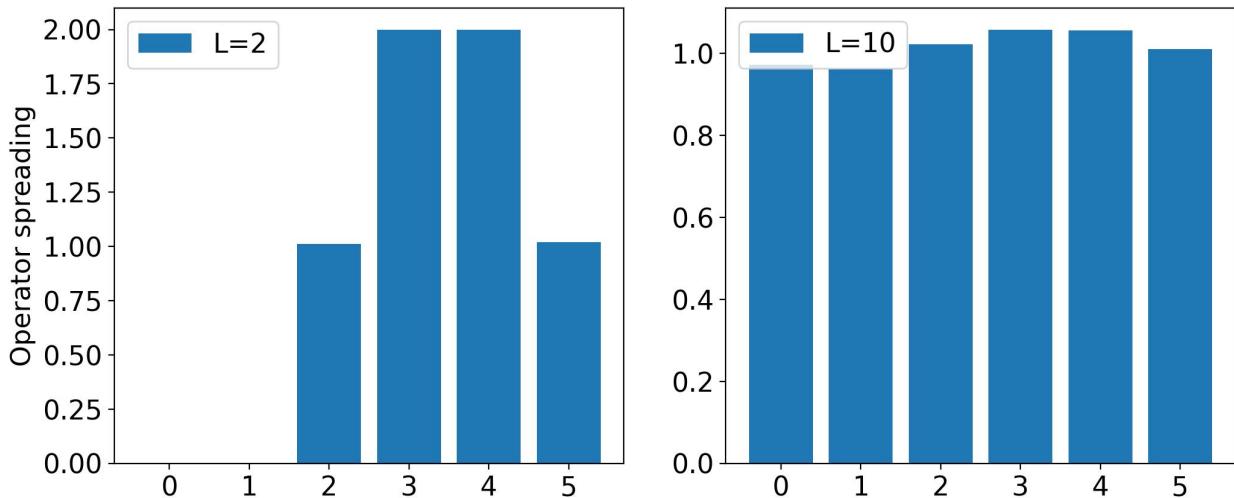
```
plt.bar([i for i in range(N)], C)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.ylabel("Operator spreading", fontsize=16)
```

[7]: Text(0, 0.5, 'Operator spreading')



Results

The operator spreading for a system of $n = 6$ qubits as a function of $1jn = 6$ for a different number of layers L , averaged over 20 random circuit instances. When $L = 2$, the Heisenberg operator growth of $O(t)$ does not reach the site 0, 1, $C = 0$; when $L = 10$, the equality will break down.



Reference

[1] <https://arxiv.org/pdf/2201.01452.pdf>.

3.1.13 Optimization vs. expressibility of the circuit

Overview

In this tutorial, we will show the relationship between circuit parameter optimization and circuit expressibility. Utilizing the variational quantum eigensolver (VQE) algorithm, we can get the ground state energy of the quantum many-body system. Although the random parameterized circuit will have larger expressibility with more layers, since the entanglement entropy of the ground state satisfies “area law”, the accurate ground state energy can not be obtained because of the entanglement made by the random circuit.

The model considered here reads:

$$H = \sum_{i=1}^n \sigma_i^z \sigma_i^z$$

where :math: \sigma_i^x, \sigma_i^z are Pauli matrices of the :math: i-th qubit. The expressibility is measured through the Renyi entanglement

$$R_A^2 = -\log(\text{tr}_A \rho_A^2),$$

where :math: \rho_A is obtained through partially tracing out subsystem :math: A.

Setup

```
[1]: import numpy as np
import tensorflow as tf
import tensorcircuit as tc
import quSpin

tc.set_backend("tensorflow")
tc.set_dtype("complex128")
dtype = np.complex128
```

Parameters

```
[2]: N = 4 # The number of qubits
NA = [i for i in range(int(N / 2))] # Subsystem A
L = 2 # The number of circuit layers
num_trial = 2 # The number of random circuit instances
```

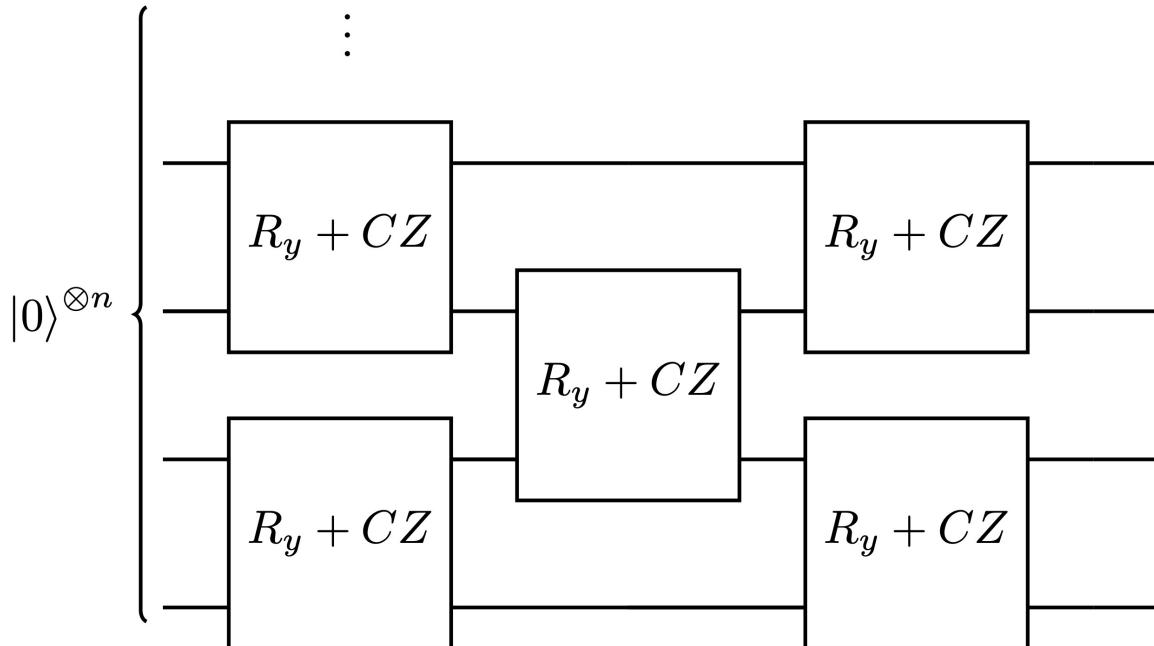
Exact Diagonalization

```
[3]: basis = quspin.basis.spin_basis_1d(N)
J_zz = [[1.0, i, (i + 1) % N] for i in range(N)]
J_x = [[1.0, i] for i in range(N)]
H_TFIM = quspin.operators.hamiltonian(
    [{"zz": J_zz}, {"x": J_x}], [], basis=basis, check_symm=False, dtype=np.float64
)
E, V = H_TFIM.eigh()
E0 = E[0]
print("The ground state energy: ", E0)

Hermiticity check passed!
The ground state energy: -5.226251859505491
```

The Density Matrix of the Output State

The circuit architecture is shown below, each block includes two Pauli rotation gates along the y-axis (R_y) followed by the CZ gate.



```
[4]: @tf.function
def get_state(v):
    layers = len(v)
    c = tc.Circuit(N)
    # Ry+Cz
    for layer in range(layers):
        if layer % 2 == 0:
            for i in range(0, N, 2):
                c.ry(i, theta=v[layer, i])
                c.ry(i + 1, theta=v[layer, i + 1])
                c.cz(i, i + 1)
        else:
            for i in range(1, N, 2):
                c.ry(i, theta=v[layer, i])
                c.ry((i + 1) % N, theta=v[layer, (i + 1) % N])
                c.cz(i, (i + 1) % N)
    = c.wavefunction() # the output state
    return
```

Hamiltonian

```
[5]: def get_hamiltonian():
    h = []
    w = []
    ###ZZ
    for i in range(N):
        h.append([])
        w.append(1.0) # weight
        for j in range(N):
            if j == (i + 1) % N or j == i:
                h[i].append(3)
            else:
                h[i].append(0)
    ###potential
    for i in range(N):
        h.append([])
        w.append(1.0)
        for j in range(N):
            if j == i:
                h[i + N].append(1)
            else:
                h[i + N].append(0)

    hamiltonian = tc.quantum.PauliString2Dense(
        tf.constant(h, dtype=tf.complex128), tf.constant(w, dtype=tf.complex128)
    )
    return hamiltonian
```

Loss Function

$$C(\theta) = \langle H \rangle.$$

```
[6]: @tf.function
def energy_loss(v, hamiltonian):
    = get_state(v)
    = tf.reshape(, [2**N, 1])
    loss = tf.transpose(, conjugate=True) @ hamiltonian @
    loss = tc.backend.real(loss)
    return loss
```

Rényi Entanglement Entropy

$$R_A^2 = -\log(\text{tr}_A \rho_A^2),$$

where :math: '\rho_A' is obtained through partially tracing out subsystem :math: 'A'.

```
[7]: @tf.function
def entropy(v):
    = get_state(v)
    _reduced = tc.quantum.reduced_density_matrix(
        , NA
    ) # reduced density matrix obtained through partially tracing out subsystem A
    S = tc.quantum.renyi_entropy(_reduced) # renyi entanglement entropy
    return S
```

Main Optimization Loop

```
[8]: def opt_main(v):
    opt = tc.backend.optimizer(tf.keras.optimizers.Adam(1e-2))

    hamiltonian = get_hamiltonian()

    loss_and_grad = tc.backend.jit(tc.backend.value_and_grad(energy_loss, argnums=0))
    loss_and_grad_vag = tc.backend.jit(
        tc.backend.vvag(loss_and_grad, argnums=0, vectorized_argnums=0)
    ) # use vvag to get losses and gradients of different random circuit instances
    entropy_vag = tc.backend.jit(
        tc.backend.vmap(entropy, vectorized_argnums=0)
    ) # use vmap to get renyi entanglement entropy of different random circuit instances

    maxiter = 100
    for i in range(maxiter):
```

(continues on next page)

(continued from previous page)

```

loss, gr = loss_and_grad_vag(v, hamiltonian)

if i == 0:
    E_initial_avg = tf.reduce_mean(loss[0]) # energy of the initial state
    S_initial = entropy_vag(v)
    S_initial_avg = tf.reduce_mean(
        S_initial[0]
    ) # renyi entanglement entropy of the initial state
elif i == maxiter - 1:
    E_final_avg = tf.reduce_mean(loss[0]) # energy of the final state
    S_final = entropy_vag(v)
    S_final_avg = tf.reduce_mean(
        S_final[0]
    ) # renyi entanglement entropy of the final state

v = opt.update(gr, v)

return (
    E_initial_avg.numpy(),
    E_final_avg.numpy(),
    S_initial_avg.numpy(),
    S_final_avg.numpy(),
)

```

[9]:

```
v = tf.random.uniform([num_trial, L, N], minval=0.0, maxval=2 * np.pi, dtype=tf.float64)
E_initial, E_final, S_initial, S_final = opt_main(v)

2022-02-18 15:28:57.668768: I tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2022-02-18 15:28:58.399077: I tensorflow/compiler/xla/service/service.cc:171] XLA service 0x7fe206d450b0 initialized for platform Host (this does not guarantee that XLA will be used). Devices:
2022-02-18 15:28:58.399104: I tensorflow/compiler/xla/service/service.cc:179] StreamExecutor device (0): Host, Default Version
2022-02-18 15:28:58.581897: I tensorflow/compiler/jit/xla_compilation_cache.cc:351] Compiled cluster using XLA! This line is logged at most once for the lifetime of the process.
```

[10]:

```
print("Number of layers = ", L, ";")
print(
    "E (initial) = ",
    E_initial - E0,
    ";",
    "E (final) = ",
    E_final - E0,
)
print(
    "S (initial) = ",
    S_initial,
```

(continues on next page)

(continued from previous page)

```

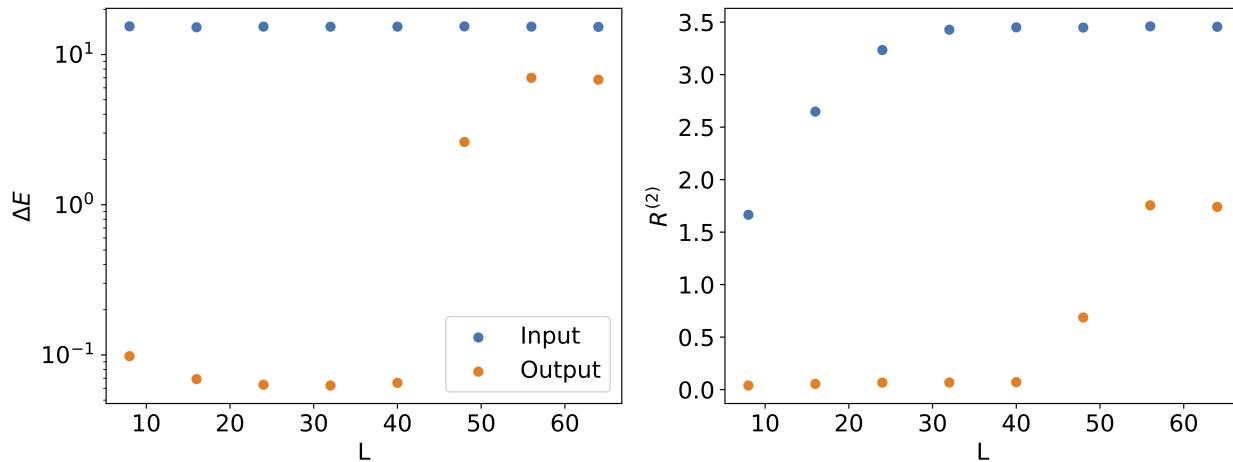
";",
"S (final) = ",
S_final,
)

Number of layers = 2 ;
E (initial) = 6.496137029102603 ; E (final) = 3.666036635713118
S (initial) = 0.561303369826885 ; S (final) = 0.07905923157886821

```

Results

The energy and Rényi entropy for a system of $n = 12$ qubits and a different number of layers L averaged over 13 random circuit instances. Per each independent random instance, we will allow enough time for convergence towards the ground state by waiting for 1000 steps of the parameter update. We can see that Rényi entanglement entropy at random parameters saturating around its maximum possible value, has an adverse effect in reaching the ground state of the VQE Hamiltonian.



Reference

Reference paper: <https://arxiv.org/pdf/2201.01452.pdf>.

3.1.14 Probing Many-body Localization by Excited-state VQE

Overview

This tutorial introduces a new method to probe many-body localization (MBL) by excited-state VQE. The model hosting MBL transition considered here is the interacting Aubry-André model which reads:

where :math : \sigma_i^{x,y,z} are Pauli matrices of the :math : i-th qubit, :math : L is the number of total qubits, :math : V_0 is the

Setup

```
[1]: import time
import numpy as np
import tensorflow as tf
import tensorcircuit as tc
```

You can define different gates easily in TensorCircuit.

```
[2]: tc.set_backend("tensorflow")
tc.set_dtype("complex128")
dtype = np.complex128

ii = tc.gates._ii_matrix
xx = tc.gates._xx_matrix
yy = tc.gates._yy_matrix
zz = tc.gates._zz_matrix
```

Note: The implementation in this tutorial is fixed to the TensorFlow backend. One can use the APIs of TensorCircuit and the APIs of TensorFlow seamlessly and freely, and the automatic differential and the JIT compilation will not be influenced.

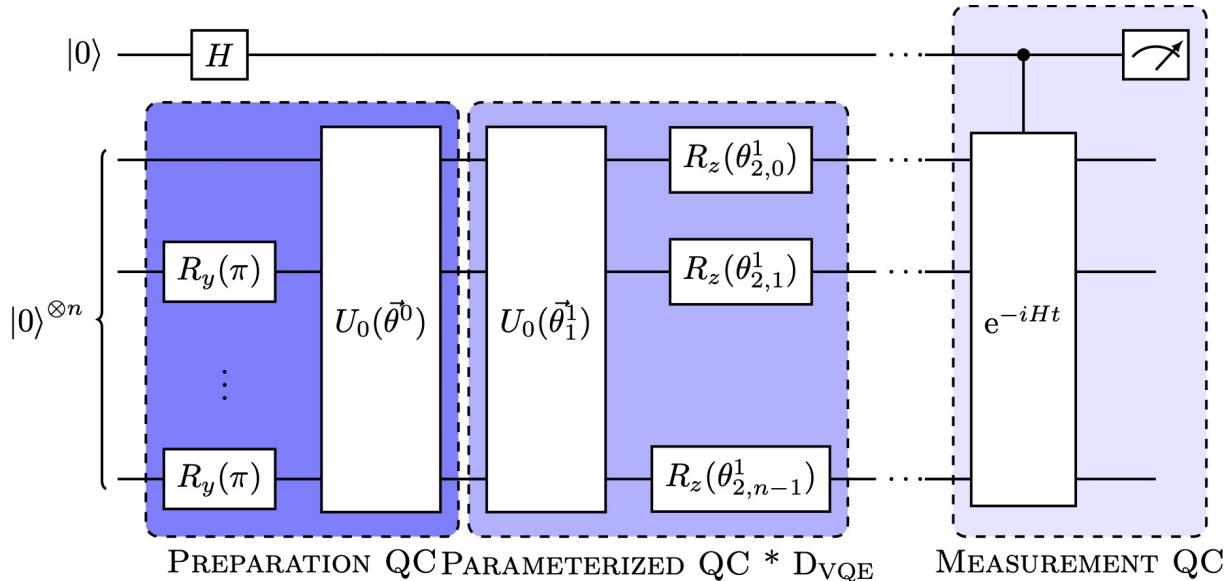
Parameters

```
[3]: L = 4
v0 = 0.5
w0 = [1.0, 10.0] # W0=1.0: thermal phase; W0=10.0: MBL phase
= (np.sqrt(5) - 1) / 2
= 0.1

num_trial = 5 # The number of independent trials
depth = 1 # The depth of the PQC
theta = np.pi / 5.0 # The rotation angle of the input state
```

Construct Circuit

The circuit structure for the excited-state VQE and eigenstate witness measurement. Here $U_0(\theta) = \exp\left(\frac{i\theta}{2}(\sigma^x \otimes \sigma^x + \sigma^y \otimes \sigma^y)\right)$.



```
[4]: @tf.function
def get_circuit(v, epochs=2):
    c = tc.DMCircuit(L)

    ### Part I: the input-state preparation quantum circuit
    ### prepare an antiferromagnetic (AF) state ( $/0101\dots>$ ) from the initial
    state  $/0000\dots>$ 
    ### and the system will stay in  $Mz=0$  sector due to  $U(1)$  symmetry
    for i in range(L):
        if i % 2 == 1:
            c.ry(i, theta=tf.cast(np.pi, dtype=dtype))

    ### prepare input state
    for i in range(L):
        c.any(
            i,
            (i + 1) % L,
            unitary=@.5 * tf.math.cos(tf.cast(0, dtype=dtype)) * (ii - zz)
            + @.5 * (ii + zz)
            + @.5 * tc.backend.i() * tf.math.sin(tf.cast(0, dtype=dtype)) * (xx + yy),
        ) ###  $U_{\{0\}}$ 

    ### Part II: the parameterized quantum circuit (PQC): optimize the parameters based
    on gradient descent
    for epoch in range(epochs):
        for i in range(L):
            c.any(
                i,
                (i + 1) % L,
                unitary=@.5
                * tf.math.cos(tf.cast(v[2 * epoch, i], dtype=dtype))
                * (ii - zz)
                + @.5 * (ii + zz)
                + @.5
                * tc.backend.i()
```

(continues on next page)

(continued from previous page)

```

        * tf.math.sin(tf.cast(v[2 * epoch, i], dtype=dtype))
        * (xx + yy),
    ) ### U_{\theta}()

for i in range(L):
    c.rz(i, theta=tf.cast(v[2 * epoch + 1, i], dtype=dtype))
= c.densitymatrix()
return

```

Construct Hamiltonian

```
[5]: def get_hamiltonian(W0):
    h = []
    w = []
    ###XX
    for i in range(L):
        h.append([])
        w.append(1.0)
        for j in range(L):
            if j == (i + 1) % L or j == i:
                h[i].append(1)
            else:
                h[i].append(0)
    ###YY
    for i in range(L):
        h.append([])
        w.append(1.0)
        for j in range(L):
            if j == (i + 1) % L or j == i:
                h[i + L].append(2)
            else:
                h[i + L].append(0)
    ###ZZ
    for i in range(L):
        h.append([])
        w.append(W0)
        for j in range(L):
            if j == (i + 1) % L or j == i:
                h[i + 2 * L].append(3)
            else:
                h[i + 2 * L].append(0)
    ###potential
    for i in range(L):
        h.append([])
        w.append(W0 * np.cos(2 * np.pi * i))
        for j in range(L):
            if j == i:
                h[i + 3 * L].append(3)
            else:
                h[i + 3 * L].append(0)

```

(continues on next page)

(continued from previous page)

```
hamiltonian = tc.quantum.PauliStringSum2Dense(
    tf.constant(h, dtype=tf.complex128), tf.constant(w, dtype=tf.complex128)
)
return hamiltonian
```

Loss Function

The loss function used here is the energy variance that vanishes for eigenstates:

$$C(\theta) = \langle H^2 \rangle - \langle H \rangle^2$$

```
[6]: @tf.function
def variance_loss(hamiltonian, v, epochs=2):
    with tf.GradientTape() as tape:
        tape.watch(v)
        = get_circuit(v, epochs)
        hexp = tf.linalg.trace(@ hamiltonian)
        loss = tf.linalg.trace(@ hamiltonian @ hamiltonian) - hexp**2
        loss = tf.math.real(loss)
    gr = tape.gradient(loss, v)
    return loss, gr,
```

Eigenstate Witness

Under the excited-state VQE, the output states will converge to some highly excited states, and the MBL phase has a higher eigenstate witness.

```
[7]: @tf.function
def get_eigenstate_witness(hamiltonian, evolution_time):
    evolution_p = tf.linalg.expm(
        tc.backend.i() * tf.cast(hamiltonian, dtype=dtype) * evolution_time
    )
    evolution_m = tf.linalg.expm(
        tc.backend.i() * tf.cast(hamiltonian, dtype=dtype) * (-evolution_time)
    )

    _00 = tf.linalg.trace()
    _01 = tf.linalg.trace(@ evolution_p)
    _10 = tf.linalg.trace(evolution_m @ )
    _11 = tf.linalg.trace(evolution_m @ @ evolution_p)

    _cq_up = tf.stack([_00, _01], axis=0)
    _cq_dn = tf.stack([-_10, _11], axis=0)
    _cq = 0.5 * tf.stack([_cq_up, _cq_dn], axis=1)
    r = tf.linalg.trace(_cq @ _cq)
    return tf.math.real(r)
```

```
[8]: def opt_main(hamiltonian, epochs=2, maxiter=200):
    v = tf.Variable(
        initial_value=tf.random.normal(
            shape=[2 * epochs, L], mean=0.0, stddev=0.03, dtype=tf.float64
        )
    ) # initialize parameters
    opt = tf.keras.optimizers.Adam(learning_rate=0.02)
    print("loop begins")
    for i in range(maxiter):
        loss, gr, = variance_loss(hamiltonian, v.value(), epochs)
        opt.apply_gradients([(tf.math.real(gr), v)])
        if i == maxiter - 1:
            print(loss.numpy())
    return
```

```
[9]: start = time.time()
_list = []
r_list = []
for i in range(len(W0)):
    hamiltonian = get_hamiltonian(W0[i])
    evolution_time = 10.0 / W0[i]
    _list.append([])
    r_list.append([])
    for j in range(num_trial):
        = opt_main(hamiltonian, depth)
        _list[i].append()
        r = get_eigenstate_witness(, hamiltonian, evolution_time)
        r_list[i].append(r)
endtime = time.time()
print("Run time = ", endtime - start)
print(
    "Eigenstate Witness:",
    np.mean(r_list[0]),
    "(Thermal Phase);",
    np.mean(r_list[1]),
    "(MBL Phase)",
)
loop begins
0.18502121452649556
loop begins
0.3647578794270281
loop begins
1.1563841018010805
loop begins
1.1626719640361998
loop begins
0.20636721189392704
loop begins
0.40186462162307635
loop begins
0.39881574091884886
loop begins
```

(continues on next page)

(continued from previous page)

```
0.41128232341327475
loop begins
0.4055560658837294
loop begins
0.4046518612466343
Run time = 16.519852876663208
Eigenstate Witness: 0.8140801354126028 (Thermal Phase); 0.9994227942365521 (MBL Phase)
```

Research Project

Reference paper: <https://arxiv.org/pdf/2111.13719.pdf>.

3.1.15 Differentiable Quantum Architecture Search

Overview

This tutorial demonstrates how to utilize the advanced computational features provided by TensorCircuit such as `jit` and `vmap` to super efficiently simulate the differentiable quantum architecture search (DQAS) algorithm, where an ensemble of quantum circuits with different structures can be compiled to simulate at the same time.

[WIP note]

Setup

```
[1]: import numpy as np
import tensorcircuit as tc
import tensorflow as tf
```

```
[2]: K = tc.set_backend("tensorflow")
ctype, rtype = tc.set_dtype("complex128")
```

Problem Description

The task is to find the state preparation circuit for GHZ state $|GHZ_N\rangle = \frac{1}{\sqrt{2}}(|0^N\rangle + |1^N\rangle)$. We prepare a gate pool with `rx0`, `rx1`, `ry0`, `ry1`, `rz0`, `rz1`, `cnot01`, `cnot10` for the $N = 2$ demo. Amongst the eight gates, six are parameterized.

```
[3]: def rx0(theta):
    return K.kron(
        K.cos(theta) * K.eye(2) + 1.0j * K.sin(theta) * tc.gates._x_matrix, K.eye(2)
    )

def rx1(theta):
    return K.kron(
        K.eye(2), K.cos(theta) * K.eye(2) + 1.0j * K.sin(theta) * tc.gates._x_matrix
    )
```

(continues on next page)

(continued from previous page)

```

def ry0(theta):
    return K.kron(
        K.cos(theta) * K.eye(2) + 1.0j * K.sin(theta) * tc.gates._y_matrix, K.eye(2)
    )

def ry1(theta):
    return K.kron(
        K.eye(2), K.cos(theta) * K.eye(2) + 1.0j * K.sin(theta) * tc.gates._y_matrix
    )

def rz0(theta):
    return K.kron(
        K.cos(theta) * K.eye(2) + 1.0j * K.sin(theta) * tc.gates._z_matrix, K.eye(2)
    )

def rz1(theta):
    return K.kron(
        K.eye(2), K.cos(theta) * K.eye(2) + 1.0j * K.sin(theta) * tc.gates._z_matrix
    )

def cnot01():
    return K.cast(K.convert_to_tensor(tc.gates._cnot_matrix), ctype)

def cnot10():
    return K.cast(
        K.convert_to_tensor(
            np.array([[0, 1, 0, 0], [1, 0, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])
        ),
        ctype,
    )

ops_repr = ["rx0", "rx1", "ry0", "ry1", "rz0", "rz1", "cnot01", "cnot10"]

```

```

[4]: n, p, ch = 2, 3, 8
# number of qubits, number of layers, size of operation pool

target = tc.array_to_tensor(np.array([1, 0, 0, 1.0]) / np.sqrt(2.0))
# target wavefunction, we here use GHZ2 state as the objective target

def ansatz(params, structures):
    c = tc.Circuit(n)
    params = K.cast(params, ctype)
    structures = K.cast(structures, ctype)
    for i in range(p):
        c.any(

```

(continues on next page)

(continued from previous page)

```

    0,
    1,
    unitary=structures[i, 0] * rx0(params[i, 0])
    + structures[i, 1] * rx1(params[i, 1])
    + structures[i, 2] * ry0(params[i, 2])
    + structures[i, 3] * ry1(params[i, 3])
    + structures[i, 4] * rz0(params[i, 4])
    + structures[i, 5] * rz1(params[i, 5])
    + structures[i, 6] * cnot01()
    + structures[i, 7] * cnot10(),
)
s = c.state()
loss = K.sum(K.abs(target - s))
return loss

vag1 = K.jit(K.vvag(ansatz, argnums=0, vectorized_argnums=1))

```

Probability Ensemble Approach

This approach is more practical and experimental relevant and is the same algorithm described in Ref.1, though we here use advanced vmap to accelerate the simulation of circuits with different structures.

```

[5]: def sampling_from_structure(structures, batch=1):
    prob = K.softmax(K.real(structures), axis=-1)
    return np.array([np.random.choice(ch, p=K.numpy(prob[i])) for i in range(p)])

@K.jit
def best_from_structure(structures):
    return K.argmax(structures, axis=-1)

@K.jit
def nmf_gradient(structures, oh):
    """
    compute the Monte Carlo gradient with respect of naive mean-field probabilistic model
    """
    choice = K.argmax(oh, axis=-1)
    prob = K.softmax(K.real(structures), axis=-1)
    indices = K.transpose(K.stack([K.cast(tf.range(p), "int64"), choice]))
    prob = tf.gather_nd(prob, indices)
    prob = K.reshape(prob, [-1, 1])
    prob = K.tile(prob, [1, ch])

    return tf.tensor_scatter_nd_add(
        tf.cast(-prob, dtype=ctype),
        indices,
        tf.ones([p], dtype=ctype),
    )

```

(continues on next page)

(continued from previous page)

```
nmf_gradient_vmap = K.vmap(nmf_gradient, vectorized_argnums=1)
```

```
[6]: verbose = False
epochs = 400
batch = 256
lr = tf.keras.optimizers.schedules.ExponentialDecay(0.06, 100, 0.5)
structure_opt = tc.backend.optimizer(tf.keras.optimizers.Adam(0.12))
network_opt = tc.backend.optimizer(tf.keras.optimizers.Adam(lr))
nnp = K.implicit_randn(stddev=0.02, shape=[p, 6], dtype=rtype)
stp = K.implicit_randn(stddev=0.02, shape=[p, 8], dtype=rtype)
avcost1 = 0
for epoch in range(epochs): # iteration to update strcuture param
    avcost2 = avcost1
    costl = []
    batched_stuctures = K.onehot(
        np.stack([sampling_from_structure(stp) for _ in range(batch)]), num=8
    )
    infd, gnnp = vag1(nnp, batched_stuctures)
    gs = nmf_gradient_vmap(stp, batched_stuctures) # \nabla lnp
    gstp = [K.cast((infd[i] - avcost2), ctype) * gs[i] for i in range(infd.shape[0])]
    gstp = K.real(K.sum(gstp, axis=0) / infd.shape[0])
    avcost1 = K.sum(infd) / infd.shape[0]
    nnp = network_opt.update(gnnp, nnp)
    stp = structure_opt.update(gstp, stp)

    if epoch % 40 == 0 or epoch == epochs - 1:
        print("-----epoch %s-----" % epoch)
        print(
            "batched average loss: ",
            np.mean(avcost1),
        )

        if verbose:
            print(
                "strcuture parameter: \n",
                stp.numpy(),
                "\n network parameter: \n",
                nnp.numpy(),
            )

    cand_preset = best_from_structure(stp)
    print("best candidates so far:", [ops_repr[i] for i in cand_preset])
    print(
        "corresponding weights for each gate:",
        [K.numpy(nnp[j, i]) if i < 6 else 0.0 for j, i in enumerate(cand_preset)],
    )

WARNING:tensorflow:Using a while_loop for converting GatherNd
WARNING:tensorflow:Using a while_loop for converting TensorScatterAdd
-----epoch 0-----
batched average loss: 1.438692604002888
```

(continues on next page)

(continued from previous page)

```
best candidates so far: ['cnot01', 'rx0', 'rx1']
corresponding weights for each gate: [0.0, -0.049711242696246834, 0.0456804722145847]
-----epoch 40-----
batched average loss: 1.0024311791127296
best candidates so far: ['cnot01', 'ry0', 'cnot01']
corresponding weights for each gate: [0.0, -0.1351106165832465, 0.0]
-----epoch 80-----
batched average loss: 0.09550699673720528
best candidates so far: ['ry0', 'ry0', 'cnot01']
corresponding weights for each gate: [-0.06370593607560585, -0.7355997299177472, 0.0]
-----epoch 120-----
batched average loss: 0.0672150785213724
best candidates so far: ['ry0', 'ry0', 'cnot01']
corresponding weights for each gate: [-0.062430880135008346, -0.7343246757666638, 0.0]
-----epoch 160-----
batched average loss: 0.07052086338808516
best candidates so far: ['ry0', 'ry0', 'cnot01']
corresponding weights for each gate: [-0.060554804305087445, -0.7324486014485383, 0.0]
-----epoch 200-----
batched average loss: 0.06819711768556835
best candidates so far: ['ry0', 'ry0', 'cnot01']
corresponding weights for each gate: [-0.05860750144346523, -0.7305012995010937, 0.0]
-----epoch 240-----
batched average loss: 0.05454652406620351
best candidates so far: ['ry0', 'ry0', 'cnot01']
corresponding weights for each gate: [-0.05680703664615186, -0.728700835323507, 0.0]
-----epoch 280-----
batched average loss: 0.047745385543626825
best candidates so far: ['ry0', 'ry0', 'cnot01']
corresponding weights for each gate: [-0.05680097807715014, -0.7286947772784904, 0.0]
-----epoch 320-----
batched average loss: 0.039626618064439574
best candidates so far: ['ry0', 'ry0', 'cnot01']
corresponding weights for each gate: [-0.05679499116702013, -0.7286887907723886, 0.0]
-----epoch 360-----
batched average loss: 0.036450806118657045
best candidates so far: ['ry0', 'ry0', 'cnot01']
corresponding weights for each gate: [-0.056789547021157315, -0.7286833469676062, 0.0]
-----epoch 399-----
batched average loss: 0.012538933640035648
best candidates so far: ['ry0', 'ry0', 'cnot01']
corresponding weights for each gate: [-0.06360204206353537, -0.7354958422632526, 0.0]
```

Directly Optimize the Structure Parameters

Since we are using numerical simulation anyway, we can directly optimize the structure parameter and omit whether the super circuit is unitary or not, this approach can be faster and more reliable for some scenarios.

```
[7]: def ansatz2(params, structures):
    c = tc.Circuit(n)
    params = K.cast(params, ctype)
    structures = K.softmax(structures, axis=-1)
    structures = K.cast(structures, ctype)
    for i in range(p):
        c.any(
            0,
            1,
            unitary=structures[i, 0] * rx0(params[i, 0])
            + structures[i, 1] * rx1(params[i, 1])
            + structures[i, 2] * ry0(params[i, 2])
            + structures[i, 3] * ry1(params[i, 3])
            + structures[i, 4] * rz0(params[i, 4])
            + structures[i, 5] * rz1(params[i, 5])
            + structures[i, 6] * cnot01()
            + structures[i, 7] * cnot10(),
        )
    s = c.state()
    s /= K.norm(s)
    loss = K.sum(K.abs(target - s))
    return loss

vag2 = K.jit(K.value_and_grad(ansatz2, argnums=(0, 1)))
```

```
[8]: verbose = True
epochs = 700
lr = tf.keras.optimizers.schedules.ExponentialDecay(0.05, 200, 0.5)
structure_opt = tc.backend.optimizer(tf.keras.optimizers.Adam(0.04))
network_opt = tc.backend.optimizer(tf.keras.optimizers.Adam(lr))
nnp = K.implicit_randn(stddev=0.02, shape=[p, 6], dtype=rtype)
stp = K.implicit_randn(stddev=0.02, shape=[p, 8], dtype=rtype)
for epoch in range(epochs):

    infd, (gnnp, gstp) = vag2(nnp, stp)

    nnp = network_opt.update(gnnp, nnp)
    stp = structure_opt.update(gstp, stp)
    if epoch % 70 == 0 or epoch == epochs - 1:
        print("-----epoch %s-----" % epoch)
        print(
            "batched average loss: ",
            np.mean(infd),
        )
    if verbose:
        print(
            "strcuture parameter: \n",
```

(continues on next page)

(continued from previous page)

```

        stp.numpy(),
        "\n network parameter: \n",
        nnp.numpy(),
    )

    cand_preset = best_from_structure(stp)
    print("best candidates so far:", [ops_repr[i] for i in cand_preset])
    print(
        "corresponding weights for each gate:",
        [K.numpy(nnp[j, i]) if i < 6 else 0.0 for j, i in enumerate(cand_preset)],
    )
}

-----epoch 0-----
batched average loss: 1.3046788213442395
strcuture parameter:
[[ 0.07621179  0.04934165  0.04669995  0.04737751  0.02036102  0.01170415
  0.03786593 -0.05644197]
 [ 0.01168381  0.0561013   0.02979136  0.03134415  0.03763557  0.03739249
  0.03408754 -0.05335854]
 [ 0.03540374  0.03219197  0.01680129  0.02014464  0.06939972  0.02393527
  0.04619596 -0.01844729]]
network parameter:
[[-0.0584098   0.04281717  0.0642035   0.06008445  0.0357175   0.05512457]
 [-0.07067937  0.04410743  0.03608519  0.03465959  0.02446072  0.06917318]
 [-0.01337738  0.04776898  0.04278249  0.04917169  0.0495427   0.01059102]]
best candidates so far: ['rx0', 'rx1', 'rz0']
corresponding weights for each gate: [-0.058409803714939854, 0.04410743113093344, 0.
 04954270315507654]
-----epoch 70-----
batched average loss: 1.0081966098666586
strcuture parameter:
[[ -0.91750096  0.35057522  0.32585577  0.37681816  1.77239369  1.7734987
  1.80143958 -0.38591221]
 [ 0.30087524  0.28764993  0.36971695  0.36078872  1.79887933  1.47542633
  1.79490296 -0.38283427]
 [ 0.29950339  0.32101711 -0.07372448  0.34959339  1.83486426  1.78887106
  1.81320642 -0.34792317]]
network parameter:
[[ 0.01163284 -0.02749067 -0.00602475  0.46422017 -0.03365732 -0.01443091]
 [-0.00057541 -0.02624807 -0.03408587  0.43879875 -0.04520759 -0.00055711]
 [ 0.05673025  0.03099979 -0.02736317  0.45331194 -0.02026327 -0.00559595]]
best candidates so far: ['cnot01', 'rz0', 'rz0']
corresponding weights for each gate: [0.0, -0.045207589104267296, -0.02026326781055693]
-----epoch 140-----
batched average loss: 0.8049806880722175
strcuture parameter:
[[ -3.20900567 -2.18126972  1.96173331  0.3704988   0.75310085  2.01979348
  2.47701794 -0.37965676]
 [-0.78487034 -1.05072503  0.83960507  0.35409074  1.49913186  0.4284363
  4.58858068 -0.37664102]
 [ 0.72348068  0.29661214  0.82121041  0.34328667  4.57946006  3.79373413
  2.24252671 -0.3416766 ]]

```

(continues on next page)

(continued from previous page)

```

network parameter:
[[-5.93268249e-04 -4.03543595e-02 -1.13260135e+00  4.62883177e-01
 -3.47753230e-02 -1.57096245e-02]
[-1.38210543e-03 -5.03624409e-02  1.02006945e+00  4.37465879e-01
 -4.64645263e-02 -1.16956649e-03]
[-7.80346264e-02  1.90816551e-02  1.09724554e+00  4.51972627e-01
 -2.15345680e-02 -6.84665987e-03]]
best candidates so far: ['cnot01', 'cnot01', 'rz0']
corresponding weights for each gate: [0.0, 0.0, -0.02153456797370576]
-----epoch 210-----
batched average loss: 0.041816948869616476
strcuture parameter:
[[-3.86458991 -2.84058112  2.62327171  0.26992388  0.09167012  1.35827717
 1.81549048 -0.56415243]
[-1.16314411 -1.7698344  1.49466411 -0.30614419  0.75064439 -0.31409853
 5.25000534 -0.65623059]
[ 1.4704075  0.89799938  2.01474589  2.50046978  4.8946084   4.44647834
 1.49549043 -0.52420796]]
network parameter:
[[ 0.00716229  0.0950563  -1.62490102  0.60459966 -0.033863  -0.01472524]
 [ 0.41329341  0.02296645  1.58326833  0.57927215 -0.04604745 -0.05234586]
 [-0.07409766  0.08796055 -0.2881097  -0.52346262 -0.02053635 -0.00585734]]
best candidates so far: ['ry0', 'cnot01', 'rz0']
corresponding weights for each gate: [-1.624901021386579, 0.0, -0.020536353581361112]
-----epoch 280-----
batched average loss: 0.04771541732805661
strcuture parameter:
[[-3.86686803 -2.8436989  2.62698311  0.26657346  0.0879849  1.35457084
 1.81178177 -0.67868932]
[-1.33926046 -1.75120967  1.4909566  -0.18080598  0.75530859 -0.30731494
 5.25370236 -0.67797002]
[ 1.47961761  0.93984054  2.12875762  2.49693907  4.78860574  4.56031727
 1.50105437 -3.90111934]]
network parameter:
[[ 0.01376387  0.10062571 -1.62306954  0.60290458 -0.0321182  -0.01292187]
 [ 0.41014329  0.10543278  1.58481867  0.66429872 -0.04488787 -0.0548457 ]
 [-0.07315826  0.09047629 -0.35272068 -0.52529957 -0.01871782 -0.00404166]]
best candidates so far: ['ry0', 'cnot01', 'rz0']
corresponding weights for each gate: [-1.6230695444527814, 0.0, -0.018717818357922293]
-----epoch 350-----
batched average loss: 0.0484244468649333
strcuture parameter:
[[-3.86889078 -2.84635759  2.63008504  0.26367828  0.08490232  1.35147264
 1.80868181 -0.68180282]
[-1.61872015 -1.73606594  1.48784963 -0.16422792  0.75890549 -0.30250011
 5.25680704 -0.70078063]
[ 1.48953381  0.96377125  2.13183873  2.49380904  4.79193079  4.5635497
 1.5054478  -5.6630325 ]]
network parameter:
[[ 0.02014668  0.10330406 -1.62102814  0.60086296 -0.03016668 -0.01090957]
 [ 0.40649692  0.12322429  1.58685415  0.67753112 -0.04356077 -0.05752607]
 [-0.0730366  0.09201651 -0.35067966 -0.52733454 -0.01668937 -0.00201604]]

```

(continues on next page)

(continued from previous page)

```

best candidates so far: ['ry0', 'cnot01', 'rz0']
corresponding weights for each gate: [-1.6210281393366774, 0.0, -0.01668936553578817]
-----epoch 420-----
batched average loss: 0.0490371292665724
strcuture parameter:
[[ -3.8707214 -2.84868677 2.63274431 0.26116667 0.08225763 1.34881617
  1.80602401 -0.68447817]
 [ -2.17422677 -1.72638998 1.48517837 -0.14718937 0.76130497 -0.30044208
  5.25948165 -0.72889795]
 [ 1.50031397 0.986817 2.13447811 2.49111713 4.79485557 4.56643982
  1.50815833 -6.42399688]]
network parameter:
[[ 2.66201104e-02 1.04232454e-01 -1.61904956e+00 5.98884773e-01
  -2.82809595e-02 -8.96136472e-03]
 [ 4.01114366e-01 1.42266730e-01 1.58916732e+00 6.88026030e-01
  -4.23682835e-02 -6.01395817e-02]
 [ -7.30273413e-02 9.34775776e-02 -3.48701372e-01 -5.29307121e-01
  -1.47240949e-02 -5.37945560e-05]]
best candidates so far: ['ry0', 'cnot01', 'rz0']
corresponding weights for each gate: [-1.6190495562985034, 0.0, -0.014724094945736954]
-----epoch 490-----
batched average loss: 0.04976228840362948
strcuture parameter:
[[ -3.87241212 -2.85077089 2.63506255 0.25896325 0.07995022 1.3465
  1.80370686 -0.68681685]
 [ -2.94364254 -1.72476469 1.48284202 -0.13112958 0.76233067 -0.30191621
  5.26182547 -0.76243791]
 [ 1.5130618 1.0086917 2.13677828 2.4887622 4.79747567 4.56906223
  1.50813331 -6.85905145]]
network parameter:
[[ 0.03313986 0.1034762 -1.6172524 0.59708835 -0.02657844 -0.00719578]
 [ 0.39208856 0.1630666 1.59170906 0.69656645 -0.04144895 -0.06259256]
 [ -0.07305937 0.09493765 -0.34690445 -0.53109883 -0.01294077 0.00172629]]
best candidates so far: ['ry0', 'cnot01', 'rz0']
corresponding weights for each gate: [-1.617252401393024, 0.0, -0.012940768995802935]
-----epoch 560-----
batched average loss: 0.05065420046477945
strcuture parameter:
[[ -3.8739842 -2.85266121 2.6371075 0.25701232 0.0779132 1.34445655
  1.80166269 -0.68888562]
 [ -3.87431144 -1.73479083 1.48077359 -0.1180455 0.76187864 -0.30734926
  5.26390441 -0.7996279 ]
 [ 1.52854329 1.02822466 2.13880714 2.48667702 4.79985267 4.57146409
  1.50375206 -7.1518221 ]]
network parameter:
[[ 0.03952625 0.09952938 -1.6156793 0.59551608 -0.02510117 -0.0056556 ]
 [ 0.37625454 0.18679017 1.59450355 0.70332975 -0.04084047 -0.0651834 ]
 [ -0.07341076 0.09584067 -0.3453315 -0.53266713 -0.01138222 0.00328135]]
best candidates so far: ['ry0', 'cnot01', 'rz0']
corresponding weights for each gate: [-1.6156792973551577, 0.0, -0.01138222012800648]
-----epoch 630-----
batched average loss: 0.05169300354431061

```

(continues on next page)

(continued from previous page)

```

strcuture parameter:
[[ -3.87544608 -2.85438213  2.63892729  0.25527243  0.0760992   1.34263784
  1.79984346 -0.69073143]
 [-4.81271904 -1.75890412  1.47892571 -0.1106167   0.75992863 -0.31691451
  5.26576503 -0.83861308]
 [ 1.54468553  1.04577833  2.14061233  2.48481415  4.80202942  4.57367936
  1.49248445 -7.36638329]]
network parameter:
[[ 0.04556425  0.08358514 -1.61433443  0.59417203 -0.02385289 -0.00434529]
 [ 0.35393466  0.2231968   1.59768193  0.70856295 -0.04056008 -0.06806482]
 [-0.07428592  0.0956737  -0.3439867  -0.53400786 -0.01005294  0.00460687]]
best candidates so far: ['ry0', 'cnot01', 'rz0']
corresponding weights for each gate: [-1.614334431628148, 0.0, -0.010052938889800701]
-----epoch 699-----
batched average loss: 0.10256012079470897
strcuture parameter:
[[ -3.787228  -2.7662122   2.55075055  0.34350642  0.16427612  1.43081451
  1.88802021 -0.60259055]
 [-5.6073258  -2.03961941  1.56706312 -0.37366319  0.66712936 -0.41905248
  5.17764218 -0.96685202]
 [ 1.46844149  0.80497766  2.05243854  2.57293907  4.71424946  4.48595762
  1.38208153 -7.52953678]]
network parameter:
[[ 4.11010856e-02  9.05029846e-03 -1.60328084e+00  5.83119412e-01
 -1.28978327e-02  6.67130794e-03]
 [ 3.37676048e-01  5.40067470e-01  1.61148200e+00  7.22459586e-01
 -3.43741776e-02 -6.15320200e-02]
 [-6.44290250e-02  8.86161369e-02 -3.32933747e-01 -5.45057551e-01
  9.83673124e-04  1.56395047e-02]]
best candidates so far: ['ry0', 'cnot01', 'rz0']
corresponding weights for each gate: [-1.6032808418929712, 0.0, 0.0009836731240883082]

```

Final Fine-tune

For the obtained circuit layout we can further adjust the circuit weights to make the objective more close to zero.

```
[9]: chosen_structure = K.onehot(np.array([2, 4, 6]), num=8)
chosen_structure = K.reshape(chosen_structure, [1, p, ch])
chosen_structure
```

```
[9]: <tf.Tensor: shape=(1, 3, 8), dtype=float32, numpy=
array([[[0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0.]]], dtype=float32)>
```

```
[10]: network_opt = tc.backend.optimizer(tf.keras.optimizers.Adam(1e-3))
nnp = K.implicit_rndn(stddev=0.02, shape=[p, 6], dtype=rtype)
verbose = True
epochs = 600
for epoch in range(epochs):
    infd, gnnp = vag1(nnp, chosen_structure)
```

(continues on next page)

(continued from previous page)

```

nnp = network_opt.update(gnnp, nnp)
if epoch % 60 == 0 or epoch == epochs - 1:
    print(epoch, "loss: ", K.numpy(infd[0]))

0 loss:  0.9827084438054802
60 loss:  0.9449745688150044
120 loss:  0.8850948396917335
180 loss:  0.8048454837720991
240 loss:  0.706158632509899
300 loss:  0.5901794549931197
360 loss:  0.45808014651166296
420 loss:  0.3113751664914397
480 loss:  0.1520672098147883
540 loss:  0.0014714944860031312
599 loss:  0.0032763286672428237

```

References

1. <https://arxiv.org/pdf/2010.08561.pdf>

3.1.16 Barren Plateaus

Overview

Barren plateaus are the greatest difficulties in the gradient-based optimization for a large family of random parameterized quantum circuits (PQC). The gradients vanish almost everywhere. In this example, we will show barren plateaus in quantum neural networks (QNNs).

Setup

```
[1]: import numpy as np
import tensorflow as tf
import tensorcircuit as tc

tc.set_backend("tensorflow")
tc.set_dtype("complex64")

Rx = tc.gates.rx
Ry = tc.gates.ry
Rz = tc.gates.rz
```

Parameters

```
[2]: n = 4 # The number of qubits
nlayers = 1 # The number of circuit layers
ncircuits = 3 # The number of circuits with different initial parameters
ntrials = 2 # The number of random circuits with different structures
```

Generating QNN

```
[3]: def op_expectation(params, seed, n, nlayers):
    paramsc = tc.backend.cast(params, dtype="float32") # parameters of gates
    seedc = tc.backend.cast(seed, dtype="float32") # parameters of circuit structure

    c = tc.Circuit(n)
    for i in range(n):
        c.ry(i, theta=np.pi / 4)
    for l in range(nlayers):
        for i in range(n):
            # choose one gate from Rx, Ry, and Rz gates with equal prob=1/3; status is
            # the seed.
            c.unitary_kraus(
                [Rx(paramsc[i, l]), Ry(paramsc[i, l]), Rz(paramsc[i, l])],
                i,
                prob=[1 / 3, 1 / 3, 1 / 3],
                status=seedc[i, l],
            )
        for i in range(n - 1):
            c.cz(i, i + 1)

    return tc.backend.real(
        c.expectation((tc.gates.z(), [0]), (tc.gates.z(), [1]))
    ) # expectations of <Z_0Z_1>
```

```
[4]: # use vmap and vvag to get the expectations of ZZ observable and gradients of different
# random circuit instances
op_expectation_vmap_vvag = tc.backend.jit(
    tc.backend.vmap(
        tc.backend.vvag(op_expectation, argnums=0, vectorized_argnums=0),
        vectorized_argnums=1,
    )
)
```

Batch Variance Computation

```
[5]: seed = tc.array_to_tensor(
    np.random.uniform(low=0.0, high=1.0, size=[ntrials, n, nlayers]), dtype="float32"
)
params = tc.array_to_tensor(
    np.random.uniform(low=0.0, high=2 * np.pi, size=[ncircuits, n, nlayers]),
    dtype="float32",
)

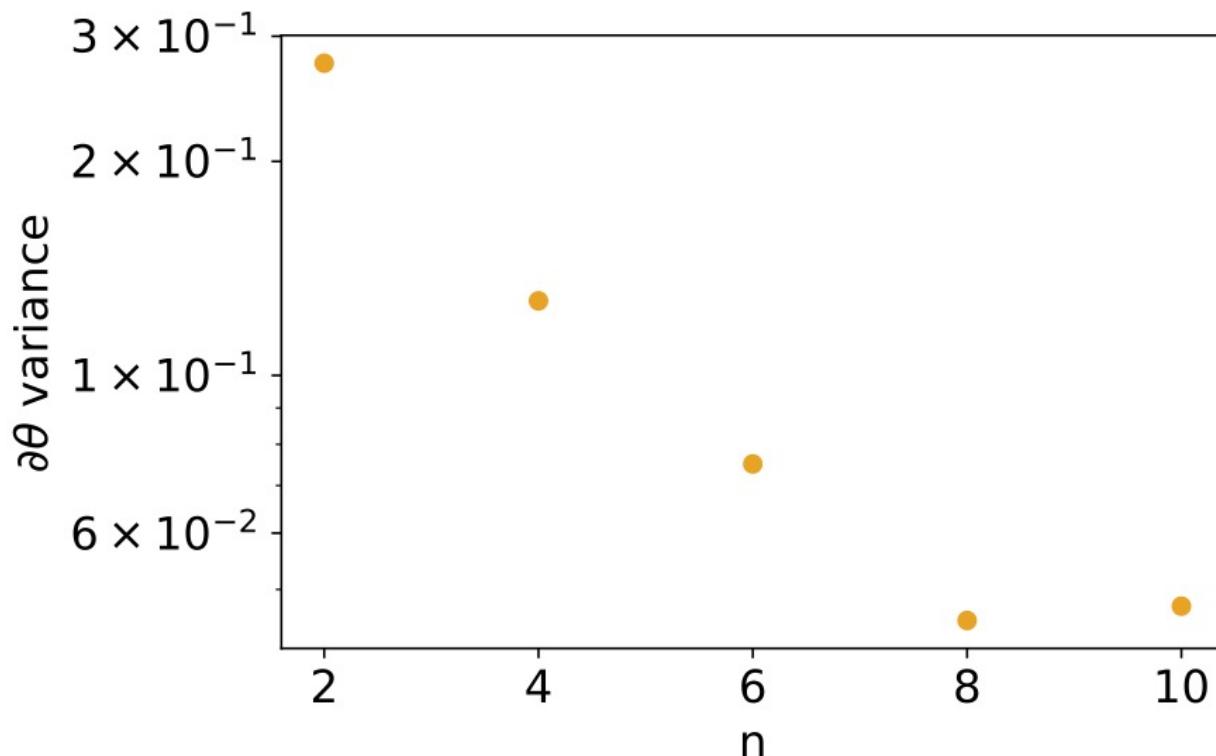
e, grad = op_expectation_vmap_vvag(
    params, seed, n, nlayers
) # the expectations of ZZ observable and gradients of different random circuits

grad_var = tf.math.reduce_std(tf.math.reduce_std(grad, axis=0), axis=0)[
    0, 0
] # the gradient variance of the first parameter
print("The variance of the gradients is:", grad_var.numpy())
```

The variance of the gradients is: 0.19805922

Results

The gradient variances in QNNs ($n_{layers} = 50$, $n_{trials} = 20$, $n_{circuits} = 20$). The landscape become exponentially barren with increasing qubit number.



3.2 Whitepaper Tutorials

3.2.1 Circuits and Gates

Overview

In TensorCircuit, a quantum circuit on n qubits – which supports both noiseless and noisy simulations via Monte Carlo trajectory methods – is created by the `tc.Circuit(n)` API. Here we show how to create basic circuits, apply gates to them, and compute various outputs.

Setup

```
[1]: import inspect
import numpy as np
import tensorcircuit as tc

K = tc.set_backend("tensorflow")
```

In TensorCircuit the default runtime datatype is `complex64`, but if higher precision is required this can be set as follows

```
[2]: tc.set_dtype("complex128")
[2]: ('complex128', 'float64')
```

Basic Circuits and Outputs

Quantum circuits can be constructed as follows.

```
[3]: c = tc.Circuit(2)
c.h(0)
c.cnot(0, 1)
c.rx(1, theta=0.2)
```

Output: state

From this, various outputs can be computed.

The full wavefunction can be obtained via

```
[4]: c.state()
[4]: <tf.Tensor: shape=(4,), dtype=complex128, numpy=
array([0.70357418+0.j         , 0.           -0.07059289j,
       0.           -0.07059289j, 0.70357418+0.j         ])>
```

The full wavefunction can also be used to generate the reduced density matrix of a subset of the qubits

```
[5]: # reduced density matrix for qubit 1
s = c.state()
tc.quantum.reduced_density_matrix(s, cut=[0]) # cut: list of qubit indices to trace out
[5]: <tf.Tensor: shape=(2, 2), dtype=complex128, numpy=
array([[0.5+0.j, 0. +0.j],
       [0. +0.j, 0.5+0.j]])>
```

Amplitudes of individual basis vectors are computed by passing the corresponding bit-string value to the `amplitude` function. For example, the amplitude of the $|10\rangle$ basis vector is computed by

```
[6]: c.amplitude("10")
[6]: <tf.Tensor: shape=(), dtype=complex128, numpy=-0.0705928857402556j>
```

The unitary matrix corresponding to the entire quantum circuit can also be output.

```
[7]: c.matrix()
[7]: <tf.Tensor: shape=(4, 4), dtype=complex128, numpy=
array([[ 0.70357418+0.j      ,  0.        -0.07059289j,
       0.70357418+0.j      ,  0.        -0.07059289j],
       [ 0.        -0.07059289j,  0.70357418+0.j      ,
       0.        -0.07059289j,  0.70357418+0.j      ],
       [ 0.        -0.07059289j,  0.70357418+0.j      ,
       0.        +0.07059289j, -0.70357418+0.j      ],
       [ 0.70357418+0.j      ,  0.        -0.07059289j,
      -0.70357418+0.j      ,  0.        +0.07059289j]])>
```

Output: measurement

Random samples corresponding to Z -measurements on all qubits can be generated using the following API, which will output a (bitstring, probability) tuple, comprising a binary string corresponding to the measurement outcomes of a Z measurement on all the qubits and the associated probability of obtaining that outcome. Z measurements on a subset of qubits can be performed with the `measure` command

```
[8]: c.sample()
[8]: (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([1., 1.])>,
       <tf.Tensor: shape=(), dtype=float64, numpy=0.4950166615971341>)
```

```
[9]: c.measure(0, with_prob=True)
[9]: (<tf.Tensor: shape=(1,), dtype=float64, numpy=array([1.])>,
       <tf.Tensor: shape=(), dtype=float64, numpy=0.5000000171142709>)
```

```
[10]: c.measure(0, 1, with_prob=True)
[10]: (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([1., 1.])>,
       <tf.Tensor: shape=(), dtype=float64, numpy=0.4950166615971341>)
```

Output: expectation

Expectation values such as $\langle X_0 \rangle$, $\langle X_1 + Z_1 \rangle$ or $\langle Z_0 Z_1 \rangle$ can be computed via the `expectation` method of a circuit object

```
[11]: print(c.expectation([tc.gates.x(), [0]])) # <X0>
print(c.expectation([tc.gates.x() + tc.gates.z(), [1]])) # <X1 + Z1>
print(c.expectation([tc.gates.z(), [0]], [tc.gates.z(), [1]])) # <Z0 Z1>
tf.Tensor(0j, shape=(), dtype=complex128)
tf.Tensor(0j, shape=(), dtype=complex128)
tf.Tensor((0.9800665437029109+0j), shape=(), dtype=complex128)
```

```
[12]: # user-defined operator
c.expectation([np.array([[3, 2], [2, -3]]), [0]])
```

```
[12]: <tf.Tensor: shape=(), dtype=complex128, numpy=0j>
```

While expectations of products of Pauli operators, e.g. $\langle Z_0 X_1 \rangle$ can be computed using `c.expectation` as above, TensorCircuit provides another way of computing such expressions which may be more convenient for longer Pauli strings, and longer Pauli strings can similarly be computed by providing lists of indices corresponding to the qubits that the X, Y, Z operators act on.

```
[13]: c.expectation_ps(x=[1], z=[0])
```

```
[13]: <tf.Tensor: shape=(), dtype=complex128, numpy=0j>
```

Built-in Gates

TensorCircuit provides support for a wide variety of commonly encountered quantum gates. The full list is as below.

```
[14]: for g in tc.Circuit.sgates:
    gf = getattr(tc.gates, g)
    print(g)
    print(tc.gates.matrix_for_gate(gf()))

i
[[1.+0.j 0.+0.j]
 [0.+0.j 1.+0.j]]
x
[[0.+0.j 1.+0.j]
 [1.+0.j 0.+0.j]]
y
[[0.+0.j 0.-1.j]
 [0.+1.j 0.+0.j]]
z
[[ 1.+0.j  0.+0.j]
 [ 0.+0.j -1.+0.j]]
h
[[ 0.70710678+0.j  0.70710678+0.j]
 [ 0.70710678+0.j -0.70710678+0.j]]
t
[[1.          +0.j           0.          +0.j           ]
 [0.          +0.j           0.70710678+0.70710678j]]
s
[[1.+0.j 0.+0.j]
 [0.+0.j 0.+1.j]]
td
[[1.          +0.j           0.          +0.j           ]
 [0.          +0.j           0.70710677-0.70710677j]]
sd
[[1.+0.j 0.+0.j]
 [0.+0.j 0.-1.j]]
wroot
[[ 0.70710678+0.j -0.5       -0.5j]
 [ 0.5       -0.5j  0.70710678+0.j ]]
cnot
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j]]
```

(continues on next page)

(continued from previous page)

```
[0.+0.j 0.+0.j 0.+0.j 1.+0.j]
[0.+0.j 0.+0.j 1.+0.j 0.+0.j]]
cz
[[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j -1.+0.j]]
swap
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
cy
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.-1.j]
 [0.+0.j 0.+0.j 0.+1.j 0.+0.j]]
iswap
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+1.j 0.+0.j]
 [0.+0.j 0.+1.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
ox
[[0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
oy
[[0.+0.j 0.-1.j 0.+0.j 0.+0.j]
 [0.+1.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
oz
[[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j -1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j]]
toffoli
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
fredkin
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j]]
```

(continues on next page)

(continued from previous page)

```
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j]
```

```
[15]: for g in tc.Circuit.vgates:
    print(g, inspect.signature(getattr(tc.gates, g).f))

r (theta: float = 0, alpha: float = 0, phi: float = 0) -> tensorcircuit.gates.Gate
cr (theta: float = 0, alpha: float = 0, phi: float = 0) -> tensorcircuit.gates.Gate
rx (theta: float = 0) -> tensorcircuit.gates.Gate
ry (theta: float = 0) -> tensorcircuit.gates.Gate
rz (theta: float = 0) -> tensorcircuit.gates.Gate
crx (*args: Any, **kws: Any) -> Any
cry (*args: Any, **kws: Any) -> Any
crz (*args: Any, **kws: Any) -> Any
orx (*args: Any, **kws: Any) -> Any
ory (*args: Any, **kws: Any) -> Any
orz (*args: Any, **kws: Any) -> Any
any (unitary: Any, name: str = 'any') -> tensorcircuit.gates.Gate
exp (unitary: Any, theta: float, name: str = 'none') -> tensorcircuit.gates.Gate
exp1 (unitary: Any, theta: float, name: str = 'none') -> tensorcircuit.gates.Gate
```

Also, we have built-in matrices as

```
[16]: for name in dir(tc.gates):
    if name.endswith("_matrix"):
        print(name, ":\n", getattr(tc.gates, name))

_cnot_matrix :
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]]

_cy_matrix :
[[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j -0.-1.j]
 [ 0.+0.j  0.+0.j  0.+1.j  0.+0.j]]

_cz_matrix :
[[ 1. 0. 0. 0.]
 [ 0. 1. 0. 0.]
 [ 0. 0. 1. 0.]
 [ 0. 0. 0. -1.]]]

_fredkin_matrix :
[[1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1.]]]

_h_matrix :
```

(continues on next page)

(continued from previous page)

```
[[ 0.70710678  0.70710678]
 [ 0.70710678 -0.70710678]]
_i_matrix :
[[1. 0.]
 [0. 1.]]
_i_i_matrix :
[[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]]
_S_matrix :
[[[1.+0.j 0.+0.j]
 [0.+0.j 0.+1.j]]]
_swap_matrix :
[[[1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]]]
_t_matrix :
[[[1.          +0.j        0.          +0.j        ]
 [0.          +0.j        0.70710678+0.70710678j]]]
_toffoli_matrix :
[[[1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 1. 0.]]]
_wroot_matrix :
[[[ 0.70710678+0.j -0.5      -0.5j]
 [ 0.5       -0.5j  0.70710678+0.j ]]]
_x_matrix :
[[[0. 1.]
 [1. 0.]]]
_xx_matrix :
[[[0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]
 [1. 0. 0. 0.]]]
_y_matrix :
[[[ 0.+0.j -0.-1.j]
 [ 0.+1.j  0.+0.j]]]
_yy_matrix :
[[[ 0.+0.j  0.-0.j  0.-0.j -1.+0.j]
 [ 0.+0.j  0.+0.j  1.-0.j  0.-0.j]
 [ 0.+0.j  1.-0.j  0.+0.j  0.-0.j]
 [-1.+0.j  0.+0.j  0.+0.j  0.+0.j]]]
_z_matrix :
[[[ 1. 0.]
 [ 0. -1.]]]
```

(continues on next page)

(continued from previous page)

```
_zz_matrix :  
[[ 1.  0.  0.  0.]  
[ 0. -1.  0. -0.]  
[ 0.  0. -1. -0.]  
[ 0. -0. -0.  1.]]
```

Arbitrary unitaries. User-defined unitary gates may be implemented by specifying their matrix elements as an array. As an example, the unitary $S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$ – which can also directly be added by calling `c.s()` – can be implemented as

```
[17]: c.unitary(0, unitary=np.array([[1, 0], [0, 1j]]), name="S")  
  
# the optional name argument specifies how this gate is displayed when the circuit is  
# output to \LaTeX
```

Exponential gates. Gates of the form $e^{i\theta G}$ where matrix G satisfies $G^2 = I$ admit a fast implementation via the `exp1` command, e.g., the two-qubit gate $e^{i\theta Z \otimes Z}$ acting on qubits 0 and 1

```
[18]: c.exp1(0, 1, theta=0.2, unitary=tc.gates._zz_matrix)
```

General exponential gates, where $G^2 \neq I$ can be implemented via the `exp` command:

```
[19]: c.exp(0, theta=0.2, unitary=np.array([[2, 0], [0, 1]]))
```

Non-unitary gates. TensorCircuit also supports the application of non-unitary gates, either by providing a non-unitary matrix as the argument to `c.unitary` or by supplying a complex angle θ to an exponential gate.

```
[20]: c.unitary(0, unitary=np.array([[1, 2], [2, 3]]), name="non_unitary")  
c.exp1(0, theta=0.2 + 1j, unitary=tc.gates._x_matrix)
```

Note that the non-unitary gates will lead to an output state that is no longer normalized since normalization is often unnecessary and takes extra time which can be avoided.

Specifying the Input State and Composing Circuits

By default, quantum circuits are applied to the initial all zero product state. Arbitrary initial states can be set by passing an array containing the input state amplitudes to the optional `inputs` argument of `tc.Circuit`. For example, the maximally entangled state $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$ can be input as follows.

```
[21]: c1 = tc.Circuit(2, inputs=np.array([1, 0, 0, 1] / np.sqrt(2)))
```

Circuits that act on the same number of qubits can be composed together via the `c.append()` or `c.prepend()` commands. With `c1` defined as above, we can create a new circuit `c2` and then compose them together:

```
[22]: c2 = tc.Circuit(2)  
c2.cnot(1, 0)  
  
c3 = c1.append(c2)  
c3.state()  
  
[22]: <tf.Tensor: shape=(4,), dtype=complex128, numpy=array([0.70710678+0.j, 0.70710678+0.j, 0.  
#          +0.j, 0.          +0.j])>
```

Circuit Transformation and Visualization

`tc.Circuit` objects can be converted to and from Qiskit `QuantumCircuit` objects.

```
[23]: c = tc.Circuit(2)
c.H(0)
c.cnot(1, 0)
cq = c.to_qiskit()
```

```
[24]: c1 = tc.Circuit.from_qiskit(cq)
```

```
[25]: # print the quantum circuit intermediate representation
```

```
c1.to_qir()

[25]: [{'gatef': h,
        'gate': Gate(
            name: 'h',
            tensor:
                <tf.Tensor: shape=(2, 2), dtype=complex128, numpy=
                    array([[ 0.70710677+0.j,  0.70710677+0.j],
                           [ 0.70710677+0.j, -0.70710677+0.j]])>,
            edges: [
                Edge('cnot'[3] -> 'h'[0]),
                Edge('h'[1] -> 'qb-1'[0])
            ],
            'index': (0,),
            'name': 'h',
            'split': None,
            'mpo': False},
       {'gatef': cnot,
        'gate': Gate(
            name: 'cnot',
            tensor:
                <tf.Tensor: shape=(2, 2, 2, 2), dtype=complex128, numpy=
                    array([[[[1.+0.j, 0.+0.j],
                            [0.+0.j, 0.+0.j]],
                           [[0.+0.j, 1.+0.j],
                            [0.+0.j, 0.+0.j]]],
                           [[[0.+0.j, 0.+0.j],
                            [0.+0.j, 1.+0.j]],
                           [[[0.+0.j, 0.+0.j],
                            [1.+0.j, 0.+0.j]]]])>,
            edges: [
                Edge(Dangling Edge)[0],
                Edge(Dangling Edge)[1],
                Edge('cnot'[2] -> 'qb-2'[0]),
                Edge('cnot'[3] -> 'h'[0])
            ],
            'index': (1, 2),
            'name': 'cnot',
            'split': None,
            'mpo': False}]]
```

(continues on next page)

(continued from previous page)

```
'index': (1, 0),
'name': 'cnot',
'split': None,
'mpo': False}]
```

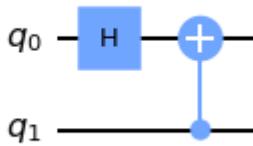
There are two ways to visualize quantum circuits generated in TensorCircuit. The first is to use `c.tex()` to get :nbsphinx-math:`\Latex` quantikz commands.

```
[26]: c.tex()
[26]: '\\begin{quantikz}\n\\lstick{\\ket{0}} & \\gate{h} & \\targ{} & \\qw \\\\\n\\lstick{\\ket{0}} & \\qw & \\ctrl{-1} & \\qw\n\\end{quantikz}'
```

The second method uses the `draw` function from `qiskit`.

```
[27]: c.draw(output="mpl")
```

```
[27]:
```



3.2.2 Gradient and Variational Optimization

Overview

TensorCircuit is designed to make optimization of parameterized quantum gates easy, fast, and convenient. In this note, we review how to obtain circuit gradients and run variational optimization.

Setup

```
[1]: import numpy as np
import scipy.optimize as optimize
import tensorflow as tf
import tensorcircuit as tc

K = tc.set_backend("tensorflow")
```

PQC

Consider a variational circuit acting on n qubits, and consisting of k layers, where each layer comprises parameterized $e^{i\theta X \otimes X}$ gates between neighboring qubits followed by a sequence of single qubit parameterized Z and X rotations. We now show how to implement such circuits in TensorCircuit, and how to use one of the machine learning backends to compute cost functions and gradients easily and efficiently.

The circuit for general n, k and set of parameters can be defined as follows:

```
[2]: def qcircuit(n, k, params):
    c = tc.Circuit(n)
    for j in range(k):
        for i in range(n - 1):
            c.exp1(
                i, i + 1, theta=params[j * (3 * n - 1) + i], unitary=tc.gates._xx_matrix
            )
        for i in range(n):
            c.rz(i, theta=params[j * (3 * n - 1) + n - 1 + i])
            c.rx(i, theta=params[j * (3 * n - 1) + 2 * n - 1 + i])
    return c
```

As an example, we take $n = 3, k = 2$, set TensorFlow as our backend, and define an energy cost function to minimize

$$E = \langle X_0 X_1 \rangle_\theta + \langle X_1 X_2 \rangle_\theta.$$

```
[3]: n = 3
k = 2

def energy(params):
    c = qcircuit(n, k, params)
    e = c.expectation_ps(x=[0, 1]) + c.expectation_ps(x=[1, 2])
    return K.real(e)
```

Grad and JIT

Using the ML backend support for automatic differentiation, we can now quickly compute both the energy and the gradient of the energy with respect to the parameters.

```
[4]: energy_val_grad = K.value_and_grad(energy)
```

This creates a function that given a set of parameters as input, returns both the energy and the gradient of the energy. If only the gradient is desired, then this can be computed by `K.grad(energy)`. While we could run the above code directly on a set of parameters, if multiple evaluations of the energy will be performed, significant time savings can be had by using a just-in-time compiled version of the function.

```
[5]: energy_val_grad_jit = K.jit(energy_val_grad)
```

With `K.jit`, the initial evaluation of the energy and gradient may take longer, but subsequent evaluations will be noticeably faster than non-jitted code. We recommend always using `jit` as long as the function is “tensor-in, tensor-out”, and we have worked hard to make all aspects of the circuit simulator compatible with JIT.

Optimization via ML Backend

With the energy function and gradients available, optimization of the parameters is straightforward. Below is an example of how to do this via stochastic gradient descent.

```
[6]: learning_rate = 2e-2
opt = K.optimizer(tf.keras.optimizers.SGD(learning_rate))

def grad_descent(params, i):
    val, grad = energy_val_grad_jit(params)
    params = opt.update(grad, params)
    if i % 10 == 0:
        print(f"i={i}, energy={val}")
    return params

params = K.implicit_randn(k * (3 * n - 1))
for i in range(100):
    params = grad_descent(params, i)

i=0, energy=0.11897378414869308
i=10, energy=-0.3692811131477356
i=20, energy=-0.7194114923477173
i=30, energy=-0.904697597026825
i=40, energy=-1.013866662979126
i=50, energy=-1.1042678356170654
i=60, energy=-1.1998062133789062
i=70, energy=-1.308410406112671
i=80, energy=-1.4276418685913086
i=90, energy=-1.5474387407302856
```

Optimization via Scipy Interface

An alternative to using the machine learning backends for the optimization is to use SciPy. This can be done via the `scipy_interface` API call and allows for gradient-based (e.g. BFGS) and non-gradient-based (e.g. COBYLA) optimizers to be used, which are not available via the ML backends.

```
[7]: f_scipy = tc.interfaces.scipy_interface(energy, shape=[k * (3 * n - 1)], jit=True)
params = K.implicit_randn(k * (3 * n - 1))
r = optimize.minimize(f_scipy, params, method="L-BFGS-B", jac=True)
r

/Users/shixin/Cloud/newwork/quantum-information/codebases/tensorcircuit/tensorcircuit/
└─interfaces.py:237: ComplexWarning: Casting complex values to real discards the
  imaginary part
    scipy_gs = scipy_gs.astype(np.float64)

[7]: fun: -2.000000476837158
hess_inv: <16x16 LbfgsInvHessProduct with dtype=float64>
jac: array([ 2.43186951e-04, -1.50322914e-04,  8.94665718e-05,  1.18807920e-05,
           2.95639038e-05,   1.19209290e-07, -5.96046448e-08, -2.98023224e-08,
           0.00000000e+00, -1.19209290e-07,  3.90738450e-07,  9.34305717e-07,
          -8.22039729e-05,   1.19209290e-07,  0.00000000e+00,  0.00000000e+00])
```

(continues on next page)

(continued from previous page)

```
message: 'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
    nfev: 60
    nit: 19
    njev: 60
    status: 0
    success: True
        x: array([ 2.35625520e+00,  7.85409154e-01,  1.57088576e+00,  2.10625989e-05,
       -1.57088425e+00, -1.70256902e+00, -5.33743572e-01,  3.11436816e-01,
       1.26543793e+00,  1.91663337e+00, -1.15901008e-07, -1.76623396e-05,
      -1.59972887e-04, -8.97072367e-01,  1.79929630e+00, -9.67278961e-01])
```

The first line above specifies the shape of the parameters to be supplied to the function to be minimized, which here is the energy function. The `jit=True` argument automatically takes care of jitting the energy function. Gradient-free optimization can similarly be performed efficiently by supplying the `gradient=False` argument to `scipy_interface`.

```
[8]: f_scipy = tc.interfaces.scipy_interface(
    energy, shape=[k * (3 * n - 1)], jit=True, gradient=False
)
params = K.implicit_randn(k * (3 * n - 1))
r = optimize.minimize(f_scipy, params, method="COBYLA")
r

[8]: fun: -1.9999911785125732
      maxcv: 0.0
      message: 'Optimization terminated successfully.'
      nfev: 386
      status: 1
      success: True
      x: array([ 7.87597857e-01, -5.14158452e-01, -1.56560250e+00, -3.15230777e-04,
       9.91532990e-01,  5.95588091e-01,  1.38523058e+00, -3.59642968e-04,
      -3.23365306e-01, -4.16465772e-01, -7.32259085e-03,  6.53997758e-05,
      7.71203778e-01,  2.46256921e+00,  8.78602039e-01, -3.51989842e-01])
```

3.2.3 Density Matrix and Mixed State Evolution

Overview

TensorCircuit provides two methods of simulating noisy, mixed state quantum evolution. Full density matrix simulation of n qubits is provided by using `tc.DMCircuit(n)`, and then adding quantum operations – both unitary gates as well as general quantum operations specified by Kraus operators – to the circuit. Relative to pure state simulation of n qubits via `tc.Circuit`, full density matrix simulation is twice as memory-intensive, and thus the maximum system size simulatable will be half of what can be simulated in the pure state case. A less memory intensive option is to use the standard `tc.Circuit(n)` object and stochastically simulate open system evolution via Monte Carlo trajectory methods.

Setup

```
[1]: import numpy as np
import tensorcircuit as tc

K = tc.set_backend("tensorflow")
```

Density Matrix Simulation with `tc.DMCircuit`

We illustrate this method below, by considering a simple circuit on a single qubit, which takes as input the mixed state corresponding to a probabilistic mixture of the $|0\rangle$ state and the maximally mixed state $\rho(\alpha) = \alpha|0\rangle\langle 0| + (1 - \alpha)I/2$.

This state is then passed through a circuit that applies an X gate, followed by a quantum operation corresponding to an amplitude damping channel \mathcal{E}_γ with parameter γ . This has Kraus operators $K_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}$, $K_1 = \begin{pmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{pmatrix}$

This circuit thus causes the evolution $\rho(\alpha) \xrightarrow{X} X\rho(\alpha)X \xrightarrow{\mathcal{E}_\gamma} \sum_{i=0}^1 K_i X \rho(\alpha) X K_i^\dagger$

To simulate this in TensorCircuit, we first create a `tc.DMCircuit` (density matrix circuit) object and set the input state using the `dminputs` optional argument (note that if a pure state input is provided to `tc.DMCircuit`, this should be done via the `inputs` optional argument).

$\rho(\alpha)$ has matrix form $\rho(\alpha) = \begin{pmatrix} \frac{1+\alpha}{2} & \frac{1-\alpha}{2} \\ \frac{1-\alpha}{2} & \frac{1-\alpha}{2} \end{pmatrix}$, and thus (taking $\alpha = 0.6$) we initialize the density matrix circuit as follows.

To implement a general quantum operation such as the amplitude damping channel, we use `general_kraus`, supplied with the corresponding list of Kraus operators.

```
[2]: def rho(alpha):
    return np.array([[[(1 + alpha) / 2, 0], [0, (1 - alpha) / 2]]])

input_state = rho(0.6)
dmc = tc.DMCircuit(1, dminputs=input_state)

dmc.x(0)

def amp_damp_kraus(gamma):
    K0 = np.array([[1, 0], [0, np.sqrt(1 - gamma)]])
    K1 = np.array([[0, np.sqrt(gamma)], [0, 0]])
    return K0, K1

K0, K1 = amp_damp_kraus(0.3)
dmc.general_kraus([K0, K1], 0) # apply channel with Kraus operators [K0,K1] to qubit 0
```

```
[3]: # get the output density matrix
dmc.state()

[3]: <tf.Tensor: shape=(2, 2), dtype=complex64, numpy=
array([[0.44+0.j, 0. +0.j],
       [0. +0.j, 0.56+0.j]], dtype=complex64)>
```

```
[4]: # evaluate the expectation as a circuit object
print(dmc.expectation_ps(z=[0]), dmc.measure(0))

tf.Tensor((-0.11999999+0j), shape=(), dtype=complex64) (<tf.Tensor: shape=(1,), 
→dtype=float32, numpy=array([1.], dtype=float32)>, -1.0)
```

In the above example, we input the Kraus operators for the amplitude damping channel manually, in order to illustrate the general approach to implementing quantum channels. In fact, TensorCircuit includes built-in methods for returning the Kraus operators for a number of common channels, including the amplitude damping, depolarizing, phase damping, and reset channels.

```
[5]: # a set of built-in quantum channels
```

```
for k in dir(tc.channels):
    if k.endswith("channel"):
        print(k)

amplitudedampingchannel
depolarizingchannel
phasedampingchannel
resetchannel
```

```
[6]: dmc = tc.DMCircuit(2)
dmc.h(0)
gamma = 0.2
K0, K1 = tc.channels.phasedampingchannel(gamma)
dmc.general_kraus([K0, K1], 0)
dmc.state()
```

```
[6]: <tf.Tensor: shape=(4, 4), dtype=complex64, numpy=
array([[0.49999997+0.j, 0.          +0.j, 0.4472136 +0.j, 0.          +0.j],
       [0.          +0.j, 0.          +0.j, 0.          +0.j, 0.          +0.j],
       [0.4472136 +0.j, 0.          +0.j, 0.49999994+0.j, 0.          +0.j],
       [0.          +0.j, 0.          +0.j, 0.          +0.j, 0.          +0.j]], 
      dtype=complex64)>
```

```
[7]: # or we can directly use the following API for shorthand
```

```
dmc = tc.DMCircuit(2)
dmc.h(0)
gamma = 0.2
dmc.phasedamping(0, gamma=0.2)
dmc.state()
```

```
[7]: <tf.Tensor: shape=(4, 4), dtype=complex64, numpy=
array([[0.49999997+0.j, 0.          +0.j, 0.4472136 +0.j, 0.          +0.j],
       [0.          +0.j, 0.          +0.j, 0.          +0.j, 0.          +0.j],
       [0.4472136 +0.j, 0.          +0.j, 0.49999994+0.j, 0.          +0.j],
       [0.          +0.j, 0.          +0.j, 0.          +0.j, 0.          +0.j]], 
      dtype=complex64)>
```

AD and JIT Compatibility

`tc.DMCircuit`, like `tc.Circuit` is also compatible with ML paradigms such as AD, jit, and vmap. See the example below.

```
[8]: n = 3
nbatch = 2

def loss(params, noise):
    c = tc.DMCircuit(n)
    for i in range(n):
        c.rx(i, theta=params[i])
    for i in range(n):
        c.depolarizing(i, px=noise, py=noise, pz=noise)
    return K.real(K.sum([c.expectation_ps(z=[i]) for i in range(n)]))

loss_vvg = K.jit(
    K.vectorized_value_and_grad(loss, argnums=(0, 1), vectorized_argnums=(0))
)

[9]: vs, (gparams, gnoise) = loss_vvg(0.1 * K.ones([nbatch, n]), 0.1 * K.ones([]))

[10]: vs.shape, gparams.shape, gnoise.shape
[10]: (TensorShape([2]), TensorShape([2, 3]), TensorShape([]))
```

Note how the noise parameter can also be differentiated and jitted!

Monte Carlo Noise Simulation with `tc.Circuit`

For pure state inputs, Monte Carlo methods can be used to sample noisy quantum evolution using `tc.Circuit` instead of `tc.DMCircuit` where the mixed state is effectively simulated with an ensemble of pure states.

As for density matrix simulation, quantum channels \mathcal{E} can be added to a circuit object by providing a list of their associated Kraus operators $\{K_i\}$. The API is the same as for the full density matrix simulation.

```
[11]: input_state = np.array([1, 1] / np.sqrt(2))
c = tc.Circuit(1, inputs=input_state)
c.general_kraus(tc.channels.phasedampingchannel(0.5), 0)
c.state()

[11]: <tf.Tensor: shape=(2,), dtype=complex64, numpy=array([0.+0.j, 1.+0.j], dtype=complex64)>
```

In this framework though, the output of a channel acting on $|\psi\rangle$, i.e. $\mathcal{E}(|\psi\rangle\langle\psi|) = \sum_i K_i |\psi\rangle\langle\psi| K_i^\dagger$ is viewed as an ensemble of states $\frac{K_i |\psi\rangle}{\sqrt{\langle\psi| K_i^\dagger K_i |\psi\rangle}}$ that each occur with probability $p_i = \langle\psi| K_i^\dagger K_i |\psi\rangle$. Thus, the code above stochastically produces the output of a single qubit initialized in state $|\psi\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ being passed through a phase damping channel with parameter $\gamma = 0.5$.

The Monte Carlo simulation of channels where the Kraus operators are all unitary matrices (up to a constant factor) can be handled with additional efficiency by using `unitary_kraus` instead of `general_kraus`.

```
[12]: px, py, pz = 0.1, 0.2, 0.3
c.unitary_kraus(tc.channels.depolarizingchannel(px, py, pz), 0)

[12]: <tf.Tensor: shape=(), dtype=int32, numpy=3>
```

Note the int tensor returned above indicates in this trajectory, which operator is applied on the circuit.

Externalizing the Randomness

The `general_kraus` and `unitary\kraus` examples above both handle randomness generation from inside the respective methods. That is, when the list $[K_0, K_1, \dots, K_{m-1}]$ of Kraus operators is supplied to `general_kraus` or `unitary_kraus`, the method partitions the interval $[0, 1]$ into m contiguous intervals $[0, 1] = I_0 \cup I_1 \cup \dots \cup I_{m-1}$ where the length of I_i is equal to the relative probability of obtaining outcome i . Then a uniformly random variable x in $[0, 1]$ is generated from within the method, and outcome i selected based on which interval x lies in.

In TensorCircuit, we have full backend agnostic infrastructure for random number generation and management. However, the interplay between jit, random number, and backend switch is often subtle if we rely on the random number generation inside these methods. See [advance.html#randoms-jit-backend-agnostic-and-their-interplay](#) for details.

In some situations, it may be preferable to first generate the random variable from outside the method, and then pass the value generated into `general_kraus` or `unitary_kraus`. This can be done via the optional `status` argument:

```
[13]: px, py, pz = 0.1, 0.2, 0.3
x = 0.5
print(c.unitary_kraus(tc.channels.depolarizingchannel(px, py, pz), 0, status=x))
x = 0.8
print(c.unitary_kraus(tc.channels.depolarizingchannel(px, py, pz), 0, status=x))

tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(3, shape=(), dtype=int32)
```

This is useful, for instance, when one wishes to use `vmap` to batch compute multiple runs of a Monte Carlo simulation. This is illustrated in the example below, where `vmap` is used to compute 10 runs of the simulation in parallel.

```
[14]: def f(x):
    c = tc.Circuit(1)
    c.h(0)
    c.unitary_kraus(tc.channels.depolarizingchannel(0.1, 0.2, 0.3), 0, status=x)
    return c.expectation_ps(x=[0])

f_vmap = K.vmap(f, vectorized_argnums=0)
X = K.implicit_randn(10)
f_vmap(X)

[14]: <tf.Tensor: shape=(10,), dtype=complex64, numpy=
array([ 0.99999994+0.j,  0.99999994+0.j,  0.99999994+0.j, -0.99999994+0.j,
       0.99999994+0.j,  0.99999994+0.j,  0.99999994+0.j,  0.99999994+0.j,
      -0.99999994+0.j,  0.99999994+0.j], dtype=complex64)>
```

3.2.4 Different Types of Measurement API

Overview

TensorCircuit allows for two kinds of operations to be performed that are related to the outcomes of measurements. These are (i) conditional measurements, the outcomes of which can be used to control downstream conditional quantum gates, and (ii) post-selection, which allows the user to select the post-measurement state corresponding to a particular measurement outcome.

Setup

```
[1]: import tensorcircuit as tc
import numpy as np

K = tc.set_backend("tensorflow")
```

Conditional Measurements

The `cond_measure` command is used to simulate the process of performing a Z measurement on a qubit, generating a measurement outcome with probability given by the Born rule, and then collapsing the wavefunction in accordance with the measured outcome. The classical measurement outcome obtained can then act as a control for a subsequent quantum operation via the `conditional_gate` API and can be used, for instance, to implement the canonical teleportation circuit.

```
[2]: # quantum teleportation of state |psi> = a|0> + sqrt(1-a^2)|1>
a = 0.3
input_state = np.kron(np.array([a, np.sqrt(1 - a**2)]), np.array([1, 0, 0, 0]))

c = tc.Circuit(3, inputs=input_state)
c.h(2)
c.cnot(2, 1)
c.cnot(0, 1)
c.h(0)

# mid-circuit measurements
z = c.cond_measure(0)
x = c.cond_measure(1)

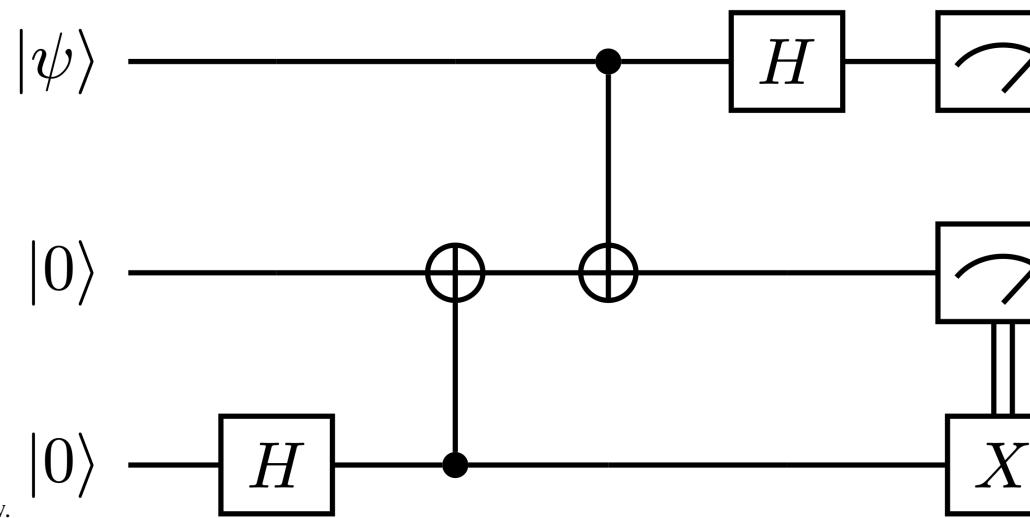
# if x = 0 apply I, if x = 1 apply X (to qubit 2)
c.conditional_gate(x, [tc.gates.i(), tc.gates.x()], 2)

# if z = 0 apply I, if z = 1 apply Z (to qubit 2)
c.conditional_gate(z, [tc.gates.i(), tc.gates.z()], 2)
```

```
[3]: # we indeed recover the state at the third qubit.
```

```
c.measure(2, with_prob=True), a**2, 1 - a**2
```

```
[3]: ((<tf.Tensor: shape=(1,), dtype=float32, numpy=array([1.], dtype=float32)>,
      <tf.Tensor: shape=(), dtype=float32, numpy=0.90999997>),
      0.09,
      0.91)
```



The teleportation circuit is shown below.

Post Selection

Post-selection is enabled in TensorCircuit via the `post_select` method. This allows the user to select the post- Z -measurement state of a qubit via the `keep` argument. Unlike `cond_measure`, the state returned by `post_select` is collapsed but not normalized.

```
[4]: c = tc.Circuit(2, inputs=np.array([1, 0, 0, 1] / np.sqrt(2)))
c.post_select(0, keep=1) # measure qubit 0, post-select on outcome 1
c.state()

[4]: <tf.Tensor: shape=(4,), dtype=complex64, numpy=
array([0.           +0.j, 0.           +0.j, 0.           +0.j, 0.70710677+0.j],
      dtype=complex64)>
```

This example initialize a 2-qubit maximally entangled state $|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$. The first qubit (q_0) is then measured in the Z -basis, and the unnormalized state $|11\rangle/\sqrt{2}$ corresponding to measurement outcome 1 is post-selected.

This post-selection scheme with unnormalized states is fast and can, for instance, be used to explore various quantum algorithms and nontrivial quantum physics such as measurement-induced entanglement phase transitions.

Plain Measurements

```
[5]: c = tc.Circuit(3)
c.H(0)
print(c.measure(0, with_prob=True))
print(c.measure(0, 1, with_prob=True))

(<tf.Tensor: shape=(1,), dtype=float32, numpy=array([1.], dtype=float32>, <tf.Tensor: shape=(), dtype=float32, numpy=0.5>)
 (<tf.Tensor: shape=(2,), dtype=float32, numpy=array([1., 0.], dtype=float32>, <tf.Tensor: shape=(), dtype=float32, numpy=0.49999997>)
```

Note how the plain measure API is virtual in the sense that the state is not collapsed after measurement.

```
[6]: for _ in range(5):
    print(c.measure(0, with_prob=True))
```

```
(<tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.], dtype=float32>, <tf.Tensor: shape=(1,), dtype=float32, numpy=0.49999997>)
(<tf.Tensor: shape=(1,), dtype=float32, numpy=array([1.], dtype=float32>, <tf.Tensor: shape=(1,), dtype=float32, numpy=0.5>)
(<tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.], dtype=float32>, <tf.Tensor: shape=(1,), dtype=float32, numpy=0.49999997>)
(<tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.], dtype=float32>, <tf.Tensor: shape=(1,), dtype=float32, numpy=0.49999997>)
(<tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.], dtype=float32>, <tf.Tensor: shape=(1,), dtype=float32, numpy=0.49999997>)
```

Let's jit measure! (with careful random number manipulation)

(continues on next page)

(continued from previous page)

```
tf.Tensor([0.  0.  0.], shape=(3,), dtype=float32)
tf.Tensor([0.  0.  0.], shape=(3,), dtype=float32)
tf.Tensor([0.  1.  0.], shape=(3,), dtype=float32)
tf.Tensor([0.  0.  0.], shape=(3,), dtype=float32)
tf.Tensor([1.  0.  1.], shape=(3,), dtype=float32)
tf.Tensor([0.  0.  0.], shape=(3,), dtype=float32)
tf.Tensor([0.  0.  0.], shape=(3,), dtype=float32)
tf.Tensor([0.  0.  0.], shape=(3,), dtype=float32)
tf.Tensor([0.  1.  0.], shape=(3,), dtype=float32)
```

For a summary of the differences between plain `measure` and the two types of measurement we mentioned here, please see [FAQ documentation](#).

3.2.5 Evaluation on Pauli String Sum

Overview

We need to evaluate the sum of many Pauli string terms on the circuit in various quantum algorithms, the ground state preparation of a Hamiltonian H in VQE is a typical example. We need to calculate the expectation value of Hamiltonian H , i.e., $\langle 0^N | U^\dagger(\theta) H U(\theta) | 0^N \rangle$ and update the parameters θ in $U(\theta)$ based on gradient descent in VQE workflow. In this tutorial, we will demonstrate five approaches supported in `TensorCircuit` to calculate $\langle H \rangle$:

1. $\langle H \rangle = \sum_i \langle h_i \rangle$, where h_i are the Pauli-string operators;
2. Similar to 1, but we evaluate the sum of Pauli string via `vmap`;
3. $\langle H \rangle$ where H is a sparse matrix;
4. $\langle H \rangle$ where H is a dense matrix;
5. the expectation value of the Matrix Product Operator (MPO) for H .

We consider the transverse field Ising model (TFIM) as the example, which reads $H = \sum_i \sigma_i^x \sigma_{i+1}^x + \sum_i \sigma_i^z \sigma_{i+1}^z$ are Pauli matrices of the i -th qubit.

Setup

```
[1]: import time
from functools import partial
import numpy as np
import tensorflow as tf
import tensornetwork as tn
import optax
import tensorcircuit as tc

K = tc.set_backend("tensorflow")

xx = tc.gates._xx_matrix # xx gate matrix to be utilized
```



```
[2]: n = 10 # The number of qubits
nlayers = 4 # The number of circuit layers
```

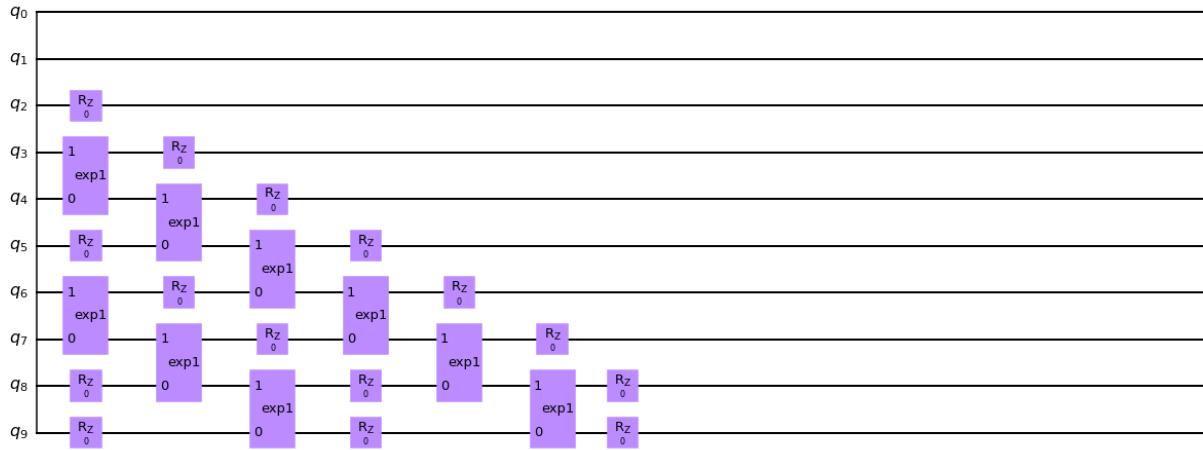
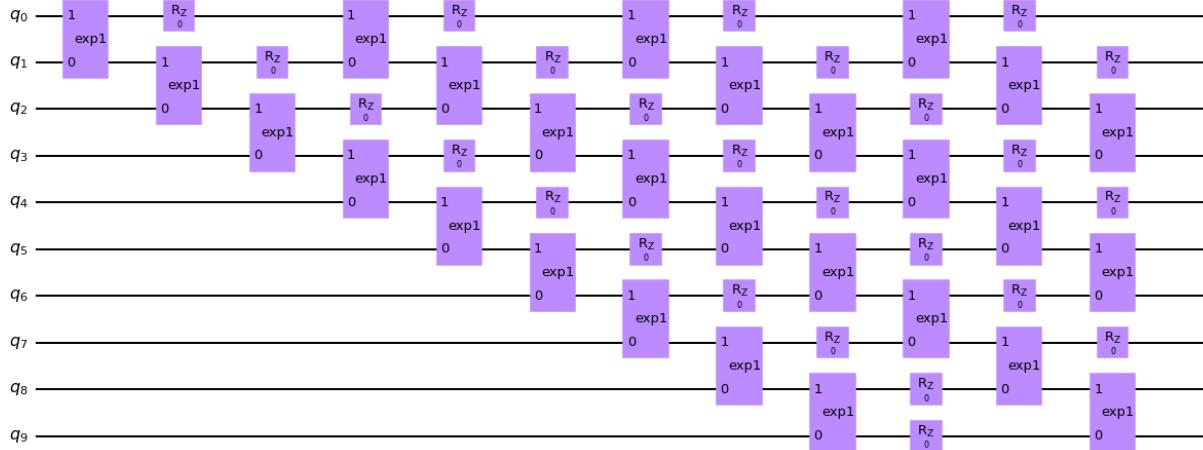
Parameterized Quantum Circuits

```
[3]: # Define the ansatz circuit to evaluate the Hamiltonian expectation
def tfim_circuit(param):
    c = tc.Circuit(n)
    for j in range(nlayers):
        for i in range(n - 1):
            c.exp1(i, i + 1, unitary=xx, theta=param[2 * j, i])
        for i in range(n):
            c.rz(i, theta=param[2 * j + 1, i])
    return c
```

[4]: # the circuit ansatz we utilized is shown as below, the two-qubit gate is in ladder_layout

```
tfim_circuit(K.zeros([2 * nlayers, n])).draw(output="mpl")
```

[4]:



Main Optimization Loop

VQE train loop (value and grad function agnostic) is defined below.

```
[5]: # We define the universal train_step function for different approaches evaluating the ↴Pauli string sum
def train_step(vf, maxiter=400):
    param = K.implicit_randn(shape=[2 * nlayers, n], stddev=0.01)
    if K.name == "tensorflow":
        opt = K.optimizer(tf.keras.optimizers.Adam(1e-2))
    else: # jax
        opt = K.optimizer(optax.adam(1e-2))
    vgf = tc.backend.jit(tc.backend.value_and_grad(vf))
    time0 = time.time()
    _ = vgf(param)
    time1 = time.time()
    print("staging time: ", time1 - time0)
    times = []
    for i in range(maxiter):
        time0 = time.time()
        e, grad = vgf(param) # energy and gradients
        time1 = time.time()
        times.append(time1 - time0)
        param = opt.update(grad, param)
        if i % 200 == 0:
            print(e)
    print("running time: ", np.mean(times))
    return e
```

1. Pauli-string Operators Sum

```
[6]: # Sum the Pauli string expectation in a plain way
def tfim_energy(c, j=1.0, h=-1.0):
    e = 0.0
    n = c._nqubits
    for i in range(n):
        e += h * c.expectation_ps(z=[i]) # <Z_i>
    for i in range(n - 1): # OBC
        e += j * c.expectation_ps(x=[i, i + 1]) # <X_iX_{i+1}>
    return K.real(e)
```

```
[7]: def vqe_tfim_v1(param):
    c = tfim_circuit(param)
    e = tfim_energy(c)
    return e
```

```
[8]: train_step(vqe_tfim_v1)

staging time: 29.903974294662476
tf.Tensor(-9.9926815, shape=(), dtype=float32)
tf.Tensor(-11.69182, shape=(), dtype=float32)
running time: 0.004392582178115845
```

[8]: <tf.Tensor: shape=(), dtype=float32, numpy=-11.658401>

2. Vmap the Pauli-string Operators Sum

A string of Pauli operators acting on n qubits can be represented as a length n vector $v \in \{0, 1, 2, 3\}^n$, where the value of $v_i = j$ corresponds to σ_i^j , i.e. Pauli operator σ^j acting on qubit i (with $\sigma^0 = I, \sigma^1 = X, \sigma^2 = Y, \sigma^3 = Z$). For example, in this notation, if $n = 3$ the term X_1X_2 corresponds to $v = [0, 1, 1]$. We refer to such a vector representation of a Pauli string as a **structure**, and a list of structures, one for each Pauli string term in the Hamiltonian, is used as the input to compute sums of expectation values in a number of ways.

If each structure has an associated weight, e.g. the term X_iX_{i+1} has weight J_i in TFIM Hamiltonian, then we define a corresponding tensor of weights.

```
[9]: def measurement(s, structure):
    c = tc.Circuit(n, inputs=s)
    return tc.templates.measurements.parameterized_measurements(
        c, structure, onehot=True
    )

measurement = K.jit(K.vmap(measurement, vectorized_argnums=1))

structures = []
for i in range(n - 1):
    s = [0 for _ in range(n)]
    s[i] = 1
    s[i + 1] = 1
    structures.append(s)
for i in range(n):
    s = [0 for _ in range(n)]
    s[i] = 3
    structures.append(s)

structures = tc.array_to_tensor(structures)
weights = tc.array_to_tensor(
    np.array([1.0 for _ in range(n - 1)] + [-1.0 for _ in range(n)])
)

print(K.numpy(structures))
print(K.numpy(weights))

def vqe_tfim_v2(param):
    c = tfim_circuit(param)
    s = c.state()
    ms = measurement(s, structures)
    return K.sum(ms * K.real(weights))

[[1.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]]
```

(continues on next page)

(continued from previous page)

```
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 1.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j 1.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
[3.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
[0.+0.j 3.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 3.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 3.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 3.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 3.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 3.+0.j 0.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 3.+0.j 0.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 3.+0.j 0.+0.j]
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 3.+0.j]]
[ 1.+0.j 1.+0.j 1.+0.j 1.+0.j 1.+0.j 1.+0.j 1.+0.j 1.+0.j 1.+0.j 1.+0.j
 -1.+0.j -1.+0.j -1.+0.j -1.+0.j -1.+0.j -1.+0.j -1.+0.j -1.+0.j -1.+0.j
 -1.+0.j]
```

```
[10]: train_step(vqe_tfim_v2)

staging time: 18.228847980499268
tf.Tensor(-9.991278, shape=(), dtype=float32)
tf.Tensor(-11.868717, shape=(), dtype=float32)
running time: 0.00716902494430542

[10]: <tf.Tensor: shape=(), dtype=float32, numpy=-11.842572>
```

3. Sparse Matrix

The significant computational advantage in terms of space and time can be obtained if the Hamiltonian is sparse, in which case a sparse representation of the operator is preferable. This can be provided in a backend agnostic way by converting from a list of Pauli structures in a two-stage process. First, we convert it to a sparse numpy matrix in COO (COOrdinate) format, and then we convert it to coo sparse tensor on a given ML backend.

```
[11]: def vqe_tfim_template(param, op):
    c = tfim_circuit(param)
    e = tc.templates.measurements.operator_expectation(
        c, op
    ) # in operator_expectation, the "hamiltonian" can be a sparse matrix, dense matrix
    ↪ or mpo
    return e
```

```
[12]: # We first generate the Hamiltonian matrix as follows

hamiltonian_sparse_numpy = tc.quantum.PauliStringSum2COO_numpy(structures, weights)
hamiltonian_sparse_numpy

[12]: <1024x1024 sparse matrix of type '<class 'numpy.complex64'>' with 9988 stored elements in COOrdinate format>
```

```
[13]: hamiltonian_sparse = K.coo_sparse_matrix(
    np.transpose(
```

(continues on next page)

(continued from previous page)

```

        np.stack([hamiltonian_sparse_numpy.row, hamiltonian_sparse_numpy.col])
),
hamiltonian_sparse_numpy.data,
shape=(2**n, 2**n),
)

```

[14]: vqe_tfim_v3 = partial(vqe_tfim_template, op=hamiltonian_sparse)

[15]: train_step(vqe_tfim_v3)

```

staging time: 13.532029867172241
tf.Tensor(-9.976083, shape=(), dtype=float32)
tf.Tensor(-11.907323, shape=(), dtype=float32)
running time: 0.002562386989593506

```

[15]: <tf.Tensor: shape=(), dtype=float32, numpy=-11.875302>

4. Dense Matrix

[16]: hamiltonian_dense = K.to_dense(hamiltonian_sparse)

[17]: vqe_tfim_v4 = partial(vqe_tfim_template, op=hamiltonian_dense)

[18]: train_step(vqe_tfim_v4)

```

staging time: 15.174713850021362
tf.Tensor(-9.981074, shape=(), dtype=float32)
tf.Tensor(-11.836525, shape=(), dtype=float32)
running time: 0.0039513856172561646

```

[18]: <tf.Tensor: shape=(), dtype=float32, numpy=-11.799629>

5. MPO

The TFIM Hamiltonian, as a short-ranged spin Hamiltonian, admits an efficient Matrix Product Operator representation. Again this is a two-stage process using TensorCircuit. We first convert the Hamiltonian into an MPO representation via the TensorNetwork or Quimb package.

[19]: # generate the corresponding MPO by converting the MPO in tensornetwork package

```

Jx = np.array([1.0 for _ in range(n - 1)]) # strength of xx interaction (OBC)
Bz = np.array([1.0 for _ in range(n)]) # strength of transverse field
# Note the convention for the sign of Bz
hamiltonian_mpo = tn.matrixproductstates.mpo.FiniteTFI(
    Jx, Bz, dtype=np.complex64
) # matrix product operator in TensorNetwork
hamiltonian_mpo = tc.quantum.tn2qop(hamiltonian_mpo) # QuOperator in TensorCircuit

```

[20]: vqe_tfim_v5 = partial(vqe_tfim_template, op=hamiltonian_mpo)

```
[21]: train_step(vqe_tfim_v5)

staging time: 26.719671964645386
tf.Tensor(-9.985788, shape=(), dtype=float32)
tf.Tensor(-11.857734, shape=(), dtype=float32)
running time: 0.00492021381855011

[21]: <tf.Tensor: shape=(), dtype=float32, numpy=-11.828297>
```

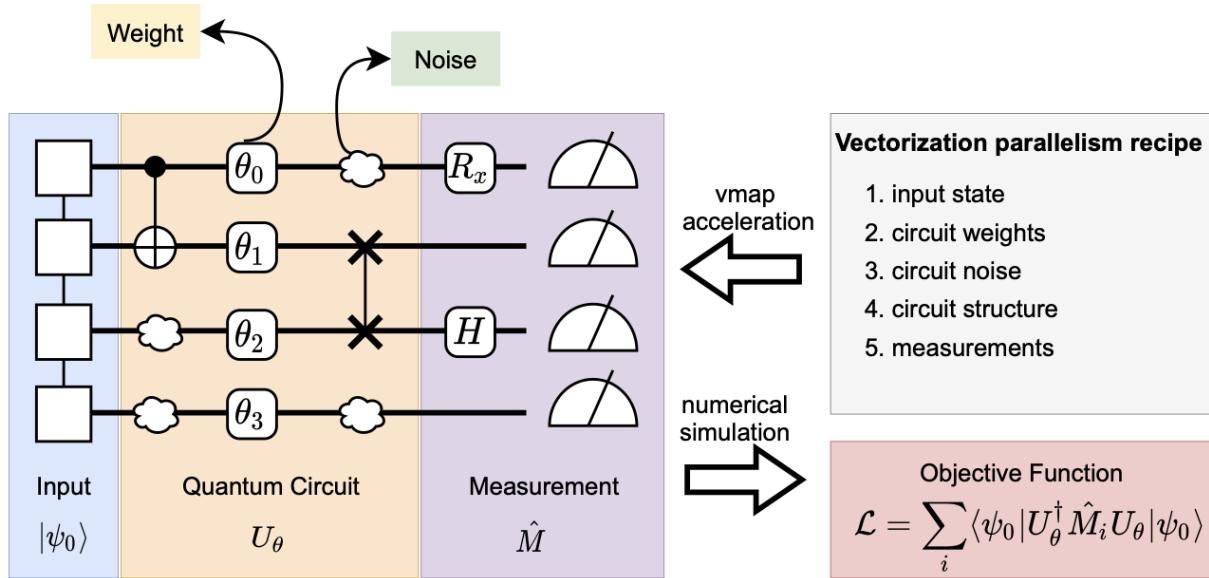
3.2.6 Utilizing vmap in Quantum Circuit Simulations

Overview

We introduce vmap, the advanced feature of the modern machine learning library, to quantum circuit simulations. By vmapping different ingredients of quantum circuit simulation, we can implement variational quantum algorithms with high efficiency.

It is worth noting that in the following use cases, vmap is supported together with jit and AD which renders highly efficient differentiable simulation.

The ingredients that support vmap paradigm are shown in the following figure.



We have two different types of APIs for vmap, the first one is `vmap` while the second one is `vectorized_value_and_grad`, aka, `vvag`. The latter can also return the gradient information over a batch of the different circuits.

If batch evaluation of gradients as well as function values is required, then this can be done via `vectorized_value_and_grad`. In the simplest case, consider a function $f(x, y)$ where $x \in R^p, y \in R^q$ are both vectors, and one wishes to evaluate both $f(x, y)$ and $\sum_x \nabla_y f(x, y) = \sum_x \left(\frac{\partial f(x, y_1)}{\partial y_1}, \dots, \frac{\partial f(x, y_q)}{\partial y_q} \right)^\top$ over a batch x_1, x_2, \dots, x_k of inputs x . This is achieved by creating a new, vectorized value-and-gradient function :

```
[1]: %%latex
\begin{equation}
f_{\text{vvg}} \left( \begin{pmatrix} \xrightarrow{} x_1 \xrightarrow{} \vdots \xrightarrow{} x_k \xrightarrow{} \end{pmatrix}, y \right) =
```

(continues on next page)

(continued from previous page)

```
\begin{pmatrix} \begin{pmatrix} f(x_1, y) \\ \vdots \\ f(x_k, y) \end{pmatrix}, \sum_{i=1}^k \nabla_y f(x_i, y) \end{pmatrix}
```

$$f_{vvg} \left(\begin{pmatrix} \leftarrow x_1 \rightarrow \\ \vdots \\ \leftarrow x_k \rightarrow \end{pmatrix}, y \right) = \left(\begin{pmatrix} f(x_1, y) \\ \vdots \\ f(x_k, y) \end{pmatrix}, \sum_{i=1}^k \nabla_y f(x_i, y) \right) \quad (3.1)$$

which takes as zeroth argument the batched inputs expressed as a $k \times p$ tensor, and as first argument the variables we wish to differentiate with respect to. The outputs are a vector of function values evaluated at all points (x_i, y) , and the gradient averaged over all those points.

Setup

```
[2]: import numpy as np
import tensorcircuit as tc

tc.set_backend("tensorflow")
print(tc.__version__)

nwires = 5
nlayers = 2
batch = 6

0.0.220509
```

vmap the Input States

Use case: batch processing of input states in quantum machine learning task.

For applications of batched input state processing, please see [MNIST QML tutorial](#).

Minimal Example

```
[3]: def f(inputs, weights):
    c = tc.Circuit(nwires, inputs=inputs)
    c = tc.templates.blocks.example_block(c, weights, nlayers=nlayers)
    loss = c.expectation([tc.gates.z(), [2]])
    loss = tc.backend.real(loss)
    return loss

f_vg = tc.backend.jit(tc.backend.vvag(f, argnums=1, vectorized_argnums=0))
f_vg(tc.backend.ones([batch, 2**nwires]), tc.backend.ones([2 * nlayers, nwires]))
```



```
[3]: (<tf.Tensor: shape=(6,), dtype=float32, numpy=
array([10.88678, 10.88678, 10.88678, 10.88678, 10.88678, 10.88678],
      dtype=float32)>,
<tf.Tensor: shape=(4, 5), dtype=complex64, numpy=
```

(continues on next page)

(continued from previous page)

```
array([[ 0.0000000e+00+1.3064140e+02j, -1.1444092e-05+1.3064142e+02j,
       0.0000000e+00+1.3064140e+02j,  0.0000000e+00+1.3064139e+02j,
       0.0000000e+00+0.0000000e+00j],
       [-1.9073486e-06-5.1765751e-06j, -5.1105431e+01-5.7347143e-07j,
        -8.1339760e+01-6.6063179e+01j, -5.1105446e+01+3.3477118e-06j,
        -7.6293945e-06+1.5500746e-07j],
       [ 0.0000000e+00+8.4607742e+01j, -1.3292285e+02+1.1209973e+02j,
        -1.3292284e+02+1.1209971e+02j,  1.5258789e-05+8.4607750e+01j,
        0.0000000e+00+0.0000000e+00j],
       [ 1.9073486e-06+5.9908474e+01j, -1.5258789e-05-1.9285599e+01j,
        -8.1339752e+01+3.8049275e-06j,  3.8146973e-06-1.9285591e+01j,
        -9.5367432e-06+5.9908482e+01j]], dtype=complex64)>)
```

vmap the Circuit Weights

Use case: batched VQE, where different random initialization parameters are optimized simultaneously.

For application on batched VQE, please refer [TFIM VQE tutorial](#).

Minimal Example

```
[4]: def f(weights):
    c = tc.Circuit(nwires)
    c = tc.templates.blocks.example_block(c, weights, nlayers=nlayers)
    loss = c.expectation([tc.gates.z(), [2]])
    loss = tc.backend.real(loss)
    return loss

f_vg = tc.backend.jit(tc.backend.vvag(f, argnums=0, vectorized_argnums=0))
f_vg(tc.backend.ones([batch, 2 * nlayers, nwires]))
```



```
[4]: (<tf.Tensor: shape=(6,), dtype=float32, numpy=
array([-2.9802322e-08, -2.9802322e-08, -2.9802322e-08, -2.9802322e-08,
       -2.9802322e-08, -2.9802322e-08], dtype=float32)>,
<tf.Tensor: shape=(6, 4, 5), dtype=complex64, numpy=
array([[[[ 1.1614500e-08+2.1480869e-08j, -9.2439478e-10-1.8808342e-08j,
            2.6397275e-08-8.0511313e-09j,  2.7981415e-08-1.6564460e-08j,
            0.0000000e+00+0.0000000e+00j],
           [ 4.1470027e-09-1.9918247e-08j, -7.7494953e-09+9.5806874e-09j,
            0.0000000e+00-1.3076999e-08j,  1.2109957e-09+3.2571617e-08j,
            -1.0110498e-08+1.6951747e-08j],
           [ 1.1614500e-08-1.0295013e-08j, -1.6102263e-08+2.5077789e-08j,
            -3.2204525e-08+5.0155577e-08j,  1.8346144e-08-3.5633683e-09j,
            0.0000000e+00+0.0000000e+00j],
           [-7.1439974e-09-3.3070933e-09j,  0.0000000e+00+6.8412485e-09j,
            1.4287995e-08-9.5050003e-09j, -7.1439974e-09-6.5384995e-09j,
            -2.3792996e-08-7.8779987e-09j]],
          [[ 1.1614500e-08+2.1480869e-08j, -9.2439478e-10-1.8808342e-08j,
```

(continues on next page)

(continued from previous page)

$$[2.6397275e-08-8.0511313e-09j, 2.7981415e-08-1.6564460e-08j,$$

$$0.0000000e+00+0.0000000e+00j],$$

$$[4.1470027e-09-1.9918247e-08j, -7.7494953e-09+9.5806874e-09j,$$

$$0.0000000e+00-1.3076999e-08j, 1.2109957e-09+3.2571617e-08j,$$

$$-1.0110498e-08+1.6951747e-08j],$$

$$[1.1614500e-08-1.0295013e-08j, -1.6102263e-08+2.5077789e-08j,$$

$$-3.2204525e-08+5.0155577e-08j, 1.8346144e-08-3.5633683e-09j,$$

$$0.0000000e+00+0.0000000e+00j],$$

$$[-7.1439974e-09-3.3070933e-09j, 0.0000000e+00+6.8412485e-09j,$$

$$1.4287995e-08-9.5050003e-09j, -7.1439974e-09-6.5384995e-09j,$$

$$-2.3792996e-08-7.8779987e-09j]],$$

$$[[1.1614500e-08+2.1480869e-08j, -9.2439478e-10-1.8808342e-08j,$$

$$2.6397275e-08-8.0511313e-09j, 2.7981415e-08-1.6564460e-08j,$$

$$0.0000000e+00+0.0000000e+00j],$$

$$[4.1470027e-09-1.9918247e-08j, -7.7494953e-09+9.5806874e-09j,$$

$$0.0000000e+00-1.3076999e-08j, 1.2109957e-09+3.2571617e-08j,$$

$$-1.0110498e-08+1.6951747e-08j],$$

$$[1.1614500e-08-1.0295013e-08j, -1.6102263e-08+2.5077789e-08j,$$

$$-3.2204525e-08+5.0155577e-08j, 1.8346144e-08-3.5633683e-09j,$$

$$0.0000000e+00+0.0000000e+00j],$$

$$[-7.1439974e-09-3.3070933e-09j, 0.0000000e+00+6.8412485e-09j,$$

$$1.4287995e-08-9.5050003e-09j, -7.1439974e-09-6.5384995e-09j,$$

$$-2.3792996e-08-7.8779987e-09j]],$$

$$[[1.1614500e-08+2.1480869e-08j, -9.2439478e-10-1.8808342e-08j,$$

$$2.6397275e-08-8.0511313e-09j, 2.7981415e-08-1.6564460e-08j,$$

$$0.0000000e+00+0.0000000e+00j],$$

$$[4.1470027e-09-1.9918247e-08j, -7.7494953e-09+9.5806874e-09j,$$

$$0.0000000e+00-1.3076999e-08j, 1.2109957e-09+3.2571617e-08j,$$

$$-1.0110498e-08+1.6951747e-08j],$$

$$[1.1614500e-08-1.0295013e-08j, -1.6102263e-08+2.5077789e-08j,$$

$$-3.2204525e-08+5.0155577e-08j, 1.8346144e-08-3.5633683e-09j,$$

$$0.0000000e+00+0.0000000e+00j],$$

$$[-7.1439974e-09-3.3070933e-09j, 0.0000000e+00+6.8412485e-09j,$$

$$1.4287995e-08-9.5050003e-09j, -7.1439974e-09-6.5384995e-09j,$$

$$-2.3792996e-08-7.8779987e-09j]],$$

$$[[1.1614500e-08+2.1480869e-08j, -9.2439478e-10-1.8808342e-08j,$$

$$2.6397275e-08-8.0511313e-09j, 2.7981415e-08-1.6564460e-08j,$$

$$0.0000000e+00+0.0000000e+00j],$$

$$[4.1470027e-09-1.9918247e-08j, -7.7494953e-09+9.5806874e-09j,$$

$$0.0000000e+00-1.3076999e-08j, 1.2109957e-09+3.2571617e-08j,$$

$$-1.0110498e-08+1.6951747e-08j],$$

$$[1.1614500e-08-1.0295013e-08j, -1.6102263e-08+2.5077789e-08j,$$

$$-3.2204525e-08+5.0155577e-08j, 1.8346144e-08-3.5633683e-09j,$$

$$0.0000000e+00+0.0000000e+00j],$$

$$[-7.1439974e-09-3.3070933e-09j, 0.0000000e+00+6.8412485e-09j,$$

$$1.4287995e-08-9.5050003e-09j, -7.1439974e-09-6.5384995e-09j,$$

$$-2.3792996e-08-7.8779987e-09j]],$$

$$[[1.1614500e-08+2.1480869e-08j, -9.2439478e-10-1.8808342e-08j,$$

$$2.6397275e-08-8.0511313e-09j, 2.7981415e-08-1.6564460e-08j,$$

$$0.0000000e+00+0.0000000e+00j],$$

(continues on next page)

(continued from previous page)

```

2.6397275e-08-8.0511313e-09j, 2.7981415e-08-1.6564460e-08j,
0.0000000e+00+0.0000000e+00j],
[ 4.1470027e-09-1.9918247e-08j, -7.7494953e-09+9.5806874e-09j,
0.0000000e+00-1.3076999e-08j, 1.2109957e-09+3.2571617e-08j,
-1.0110498e-08+1.6951747e-08j],
[ 1.1614500e-08-1.0295013e-08j, -1.6102263e-08+2.5077789e-08j,
-3.2204525e-08+5.0155577e-08j, 1.8346144e-08-3.5633683e-09j,
0.0000000e+00+0.0000000e+00j],
[-7.1439974e-09-3.3070933e-09j, 0.0000000e+00+6.8412485e-09j,
1.4287995e-08-9.5050003e-09j, -7.1439974e-09-6.5384995e-09j,
-2.3792996e-08-7.8779987e-09j]], dtype=complex64)>)

```

vmap the Quantum Noise

Use case: parallel Monte Carlo noise simulation.

For applications that combine vmapped Monte Carlo noise simulation and quantum machine learning task, please see [noisy QML script](#).

Minimal Example

```
[5]: def f(weights, status):
    c = tc.Circuit(nwires)
    c = tc.templates.blocks.example_block(c, weights, nlayers=nlayers)
    for i in range(nwires):
        c.depolarizing(i, px=0.2, py=0.2, pz=0.2, status=status[i])
    loss = c.expectation([tc.gates.x(), [2]])
    loss = tc.backend.real(loss)
    return loss

f_vg = tc.backend.jit(tc.backend.vvag(f, argnums=0, vectorized_argnums=1))

def g(weights):
    status = tc.backend.implicit_randu(shape=[batch, nwires])
    return f_vg(weights, status)

g(tc.backend.ones([2 * nlayers, nwires]))

[5]: (<tf.Tensor: shape=(6,), dtype=float32, numpy=
array([ 0.34873545, -0.34873545, -0.34873545, -0.34873545, -0.34873545,
       0.34873545], dtype=float32>,
<tf.Tensor: shape=(4, 5), dtype=complex64, numpy=
array([[ -8.8614023e-01-1.7026657e-08j,   5.7763958e-01+2.3834804e-01j,
         5.7763910e-01+2.3834780e-01j,  -8.8614047e-01+1.2538894e-07j,
         0.0000000e+00+0.0000000e+00j],
       [ 0.0000000e+00+6.9313288e-01j,   3.6122650e-01-1.1496974e-02j,
       -5.2079970e-01-1.7869800e-01j,   3.6122644e-01-1.1496985e-02j,
```

(continues on next page)

(continued from previous page)

```
-2.9802322e-08+6.9313288e-01j],
[-5.9604645e-08-1.0189922e+00j, -3.0850098e-01-1.4861794e-07j,
-3.0850050e-01-2.2304604e-08j, 5.9604645e-08-1.0189921e+00j,
0.0000000e+00+0.000000e+00j],
[ 0.0000000e+00+3.1868588e-02j, -8.9406967e-08+2.5656950e-01j,
-2.9802322e-08-1.9999983e+00j, -2.9802322e-08+2.5656945e-01j,
-1.1920929e-07+3.1868652e-02j]], dtype=complex64)>)
```

vmap the Circuit Structure

Use case: differentiable quantum architecture search (DQAS).

For more detail on DQAS application, see [DQAS tutorial](#).

Minimal Example

```
[6]: eye = tc.gates.i().tensor
x = tc.gates.x().tensor
y = tc.gates.y().tensor
z = tc.gates.z().tensor

def f(params, structures):
    c = tc.Circuit(nwires)
    for i in range(nwires):
        c.H(i)
    for j in range(nlayers):
        for i in range(nwires - 1):
            c.cz(i, i + 1)
        for i in range(nwires):
            c.unitary(
                i,
                unitary=structures[i, j, 0]
            *
            (
                tc.backend.cos(params[i, j, 0]) * eye
                + tc.backend.sin(params[i, j, 0]) * x
            )
            +
            structures[i, j, 1]
            *
            (
                tc.backend.cos(params[i, j, 1]) * eye
                + tc.backend.sin(params[i, j, 1]) * y
            )
            +
            structures[i, j, 2]
            *
            (
                tc.backend.cos(params[i, j, 2]) * eye
                + tc.backend.sin(params[i, j, 2]) * z
            ),
        )
    loss = c.expectation([tc.gates.z(), (2,)])
    return tc.backend.real(loss)
```

(continues on next page)

(continued from previous page)

```

structures = tc.backend.ones([batch, nwires, nlayers, 3])
params = tc.backend.ones([nwires, nlayers, 3])
f_vg = tc.backend.jit(tc.backend.vvag(f, argnums=0, vectorized_argnums=1))
f_vg(params, structures)

[6]: (<tf.Tensor: shape=(6,), dtype=float32, numpy=
array([2.4917054e+08, 2.4917054e+08, 2.4917054e+08, 2.4917054e+08,
       2.4917054e+08, 2.4917054e+08], dtype=float32)>,
<tf.Tensor: shape=(5, 2, 3), dtype=complex64, numpy=
array([[[-4.8252989e+08+2.3603376e+07j, -6.4132224e+08+1.1064736e+08j,
        -4.5701562e+08-7.4987272e+07j],
       [-5.4175347e+08+5.2096408e+07j, -5.5254317e+08-4.6495180e+07j,
        -4.5219101e+08-5.6013205e+06j]],

      [[-7.1430163e+08-1.2090212e+08j, -6.2410163e+08-4.1363908e+07j,
        -3.9189485e+08+4.0016840e+06j],
       [-5.8365677e+08+9.4236816e+07j, -5.7693280e+08-9.7727496e+07j,
        -3.9540646e+08+3.4906362e+06j]],

      [[-5.9637555e+08+8.9477632e+07j, -7.6615610e+08+1.1949610e+08j,
        -3.8039136e+08-4.7556400e+07j],
       [-1.1637092e+09-4.0144461e+08j, -1.1735478e+09+4.5104198e+08j,
        -1.5947418e+08+1.5322706e+07j]],

      [[-7.1430170e+08-1.2090210e+08j, -6.2410170e+08-4.1363864e+07j,
        -3.9189485e+08+4.0016840e+06j],
       [-5.8365658e+08+9.4236840e+07j, -5.7693261e+08-9.7727496e+07j,
        -3.9540637e+08+3.4906552e+06j]],

      [[-4.8253002e+08+2.3603400e+07j, -6.4132237e+08+1.1064734e+08j,
        -4.5701565e+08-7.4987248e+07j],
       [-5.4175334e+08+5.2096460e+07j, -5.5254304e+08-4.6495116e+07j,
        -4.5219091e+08-5.6013120e+06j]]], dtype=complex64)>)

```

vmap the Circuit Measurements

Use case: accelerating evaluation of Pauli string sum by parallel the parameterized measurement.

For applications on evaluation of parameterized measurements via vmap on large-scale systems, see [large-scale vqe example script](#).

Minimal Example

```
[7]: def f(params, structures):
    c = tc.Circuit(nwires)
    c = tc.templates.blocks.example_block(c, params, nlayers=nlayers)
    loss = tc.templates.measurements.parameterized_measurements(
        c, structures, onehot=True
    )
    return loss

# measure X0 to X3
structures = tc.backend.eye(nwires)
f_vvag = tc.backend.jit(tc.backend.vvag(f, vectorized_argnums=1, argnums=0))
f_vvag(tc.backend.ones([2 * nlayers, nwires]), structures)

WARNING:tensorflow:5 out of the last 5 calls to <function TensorFlowBackend.vectorized_
->value_and_grad.<locals>.wrapper at 0x7fe6cbed1af0> triggered tf.function retracing.
->Tracing is expensive and the excessive number of tracings could be due to (1) creating
->@tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3)
->passing Python objects instead of tensors. For (1), please define your @tf.function
->outside of the loop. For (2), @tf.function has experimental_relax_shapes=True option
->that relaxes argument shapes that can avoid unnecessary retracing. For (3), please
->refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://
->www.tensorflow.org/api_docs/python/tf/function for more details.
```

```
[7]: (<tf.Tensor: shape=(5,), dtype=float32, numpy=
array([-0.3118263 ,  0.00371493,  0.3487355 ,  0.00371514, -0.31182614],
      dtype=float32)>,
<tf.Tensor: shape=(4, 5), dtype=complex64, numpy=
array([[ 1.6707865e+00-0.40178323j, -1.1992662e+00-0.23834792j,
       -1.1992660e+00-0.2383478j ,  1.6707866e+00-0.40178335j,
       0.0000000e+00+0.j        ],
      [-1.8267021e-01-0.6483071j ,  7.7729575e-02+0.58401704j,
       -1.0082662e-01-0.52953976j,  7.7729806e-02+0.58401704j,
       -1.8267024e-01-0.6483072j ],
      [ 1.6707866e+00+0.19420199j, -1.1992658e+00+0.50487465j,
       -1.1992657e+00+0.504875j ,  1.6707867e+00+0.19420168j,
       0.0000000e+00+0.j        ],
      [ 7.4505806e-09+0.99540246j,  1.4901161e-08+0.7925009j ,
       -7.4505806e-09+0.71156096j, -7.4505806e-09+0.7925008j ,
       2.2351742e-08+0.9954027j ]], dtype=complex64)>)
```

3.2.7 QuOperator in TensorCircuit

Overview

`tensorcircuit.quantum.QuOperator`, `tensorcircuit.quantum.QuVector` and `tensorcircuit.quantum.QuAdjointVector` are classes adopted from the TensorNetwork package. They behave like a matrix/vector (column or row) when interacting with other ingredients while the inner structure is maintained by the TensorNetwork for efficiency and compactness.

Typical tensor network structures for a QuOperator/QuVector correspond to Matrix Product Operators (MPO) / Matrix

Product States (MPS). The former represents a matrix as: $M_{i_1, i_2, \dots, i_n; j_1, j_2, \dots, j_n} = \prod_k T_k^{i_k, j_k}$, i.e., a product of $d \times d$ matrices $T_k^{i_k, j_k}$, where d is known as the bond dimension. Similarly, an MPS represents a vector as: $V_{i_1, \dots, i_n} = \prod_k T_k^{i_k}$, where the $T_k^{i_k}$ are, again, $d \times d$ matrices. MPS and MPO often occur in computational quantum physics contexts, as they give compact representations for certain types of quantum states and operators.

QuOperator/QuVector objects can represent any MPO/MPS, but they can additionally express more flexible tensor network structures. Indeed, any tensor network with two sets of dangling edges of the same dimension (i.e., for each k , the set $\{T_k^{i_k, j_k}\}_{i_k, j_k}$ of matrices has i_k and j_k running over the same index set) can be treated as a QuOperator. A general QuVector is even more flexible, in that the dangling edge dimensions can be chosen freely, and thus arbitrary tensor products of vectors can be represented.

In this note, we will show how such tensor network backend matrix/vector data structure is more efficient and compact in several scenarios and how these structures integrated with quantum circuit simulation tasks seamlessly as different circuit ingredients.

Setup

```
[1]: import numpy as np
import tensornetwork as tn
import tensorcircuit as tc

print(tc.__version__)
0.0.220509
```

Introduction to QuOperator/QuVector

```
[2]: n1 = tc.gates.Gate(np.ones([2, 2, 2]), name="n1")
n2 = tc.gates.Gate(np.ones([2, 2, 2]), name="n2")
n3 = tc.gates.Gate(np.ones([2, 2]), name="n3")
# name is only for debug and visualization, can be omitted
n1[2] ^ n2[2]
n2[1] ^ n3[0]

# initialize a QuOperator by giving two sets of dangling edges for row and col index
matrix = tc.quantum.QuOperator(out_edges=[n1[0], n2[0]], in_edges=[n1[1], n3[1]])
tn.to_graphviz(matrix.nodes)
```

[2]:

```
[3]: n4 = tc.gates.Gate(np.ones([2]), name="n4")
n5 = tc.gates.Gate(np.ones([2]), name="n5")

# initialize a QuVector by giving dangling edges
vector = tc.quantum.QuVector([n4[0], n5[0]])
tn.to_graphviz(vector.nodes)
```

[3]:

```
[4]: nvector = matrix @ vector
tn.to_graphviz(nvector.nodes)
# nvector has two dangling edges
```

[4]:

```
[5]: assert type(nvector) == tc.quantum.QuVector

[6]: nvector.eval_matrix()
[6]: array([[16.],
           [16.],
           [16.],
           [16.]])
```



```
[7]: # or we can have more matrix/vector-like operation
(3 * nvector).eval_matrix()
[7]: array([[48.],
           [48.],
           [48.],
           [48.]])
```



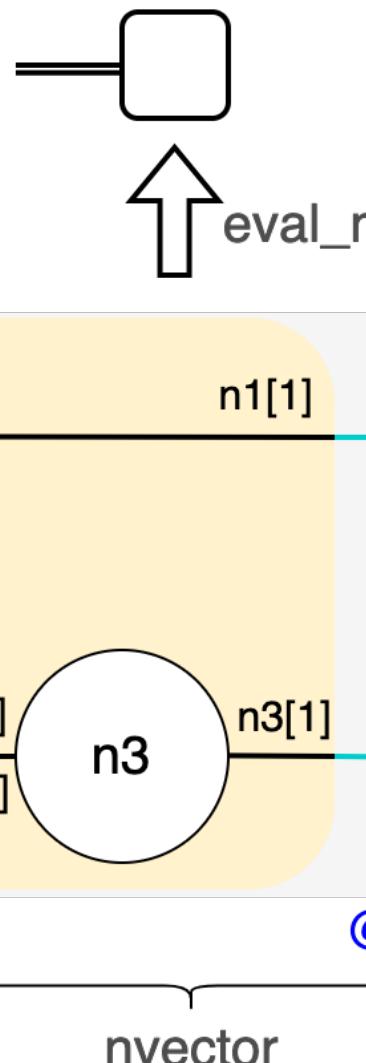
```
[8]: matrix.partial_trace([0]).eval_matrix()
[8]: array([[8., 8.],
           [8., 8.]])
```

Note how in this example, `matrix` is not a typical MPO but still can be expressed as `QuOperator`. Indeed, any tensor network with two sets of dangling edges of the same dimension can be treated as `QuOperator`. `QuVector` is even more flexible since we can treat all dangling edges as the vector dimension.

Also, note how `^` is overloaded as `tn.connect` to connect edges between different nodes in `TensorNetwork`. And indexing the node gives the edges of the node, eg. `n1[0]` means the first edge of node `n1`.

The convention to define the `QuOperator` is firstly giving `out_edges` (left index or row index of the matrix) and then giving `in_edges` (right index or column index of the matrix). The edges list contains edge objects from the `TensorNetwork` library.

Such `QuOperator/QuVector` abstraction support various calculations only possible on matrix/vectors, such as `matmul (@)`, `adjoint (.adjoint())`, scalar multiplication `(*)`, tensor product `(|)`, and partial trace `(.partial_trace(subsystems_to_trace_out))`. To extract the matrix information of these objects, we can use `.eval()` or `.eval_matrix()`, the former keeps the shape information of the tensor network while the latter gives the matrix representation with shape rank 2.



The workflow here can also be summarized and visualized as

QuVector as the Input State for the Circuit

Since QuVector behaves like a real vector with a more compact representation, we can feed the circuit input states in the form of QuVector instead of a plain numpy array vector.

```
[9]: # This examples shows how we feed a |111> state into the circuit

n = 3
nodes = [tc.gates.Gate(np.array([0.0, 1.0])) for _ in range(n)]
mps = tc.quantum.QuVector([nd[0] for nd in nodes])
c = tc.Circuit(n, mps_inputs=mps)
c.x(0)
c.expectation_ps(z=[0])

[9]: array(1.+0.j)
```

QuVector as the Output State of the Circuit

The tensor network representation of the circuit can be regarded as a QuVector, namely we can manipulate the circuit as a vector before the real contraction. This is also how we do circuit composition internally.

[10]: # Circuit composition example

```
n = 3
c1 = tc.Circuit(n)
c1.X(0)
c1.cnot(0, 1)
mps = c1.quvector()
c2 = tc.Circuit(n, mps_inputs=mps)
c2.X(2)
c2.X(1)
c2.cz(1, 2)
c2.expectation_ps(z=[1])
tn.to_graphviz(c2.get_quvector().nodes)
```

[10]:

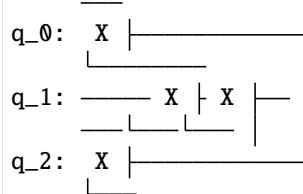
[11]: # The above is the core internal mechanism for circuit composition
the user API is as follows

```
n = 3
c1 = tc.Circuit(n)
c1.X(0)
c1.cnot(0, 1)

c2 = tc.Circuit(n)
c2.X(2)
c2.X(1)
c2.cz(1, 2)
c1.append(c2)

c1.draw()
```

[11]:



QuOperator as Operator to be Evaluated on the Circuit

The matrix to be evaluated over the output state of the circuit can also be represented by QuOperator, which is very powerful and efficient for some lattices model Hamiltonian.

```
[12]: # Here we show the simplest model, where we measure <Z_0Z_1>
```

```
z0, z1 = tc.gates.z(), tc.gates.z()
mpo = tc.quantum.QuOperator([z0[0], z1[0]], [z0[1], z1[1]])
c = tc.Circuit(2)
c.X(0)
tc.templates.measurements.mpo_expectation(c, mpo)
# the mpo expectation API
```

```
[12]: -1.0
```

QuOperator as the Quantum Gate Applied on the Circuit

Since quantum gates are also unitary matrices, we can also use QuOperator for quantum gates. In some cases, QuOperator representation for quantum gates is much more compact, such as in multi-control gate case, where the bond dimension can be reduced to 2 for neighboring qubits.

```
[13]: # The general mpo gate API is just ``Circuit.mpo()``
```

```
x0, x1 = tc.gates.Gate(np.ones([2, 2, 3]), name="x0"), tc.gates.Gate(
    np.ones([2, 2, 3]), name="x1")
)
x0[2] ^ x1[2]
mpo = tc.quantum.QuOperator([x0[0], x1[0]], [x0[1], x1[1]])
c = tc.Circuit(2)
c.mpo(0, 1, mpo=mpo)
tn.to_graphviz(c._nodes)
```

```
[13]:
```

```
[14]: # the built-in multi-control gate is used as follows:
```

```
c = tc.Circuit(3)
c.multicontrol(0, 1, 2, ctrl=[0, 1], unitary=tc.gates.x())
tn.to_graphviz(c._nodes)
```

```
[14]:
```

```
[15]: c.to_qir()[0]["gate"].nodes
```

```
[15]: {Node
```

```
(
    name : '__unnamed_node__',
    tensor :
        array([[[[0.+0.j, 1.+0.j],
                 [0.+0.j, 0.+0.j]],
               [
                   [[0.+0.j, 0.+0.j],
                    [1.+0.j, 0.+0.j]]],
```

(continues on next page)

(continued from previous page)

```

[[[0.+0.j, 1.+0.j],
 [0.+0.j, 0.+0.j]],

 [[0.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]]], dtype=complex64),
edges :
[
Edge('__unnamed_node__[2] -> '__unnamed_node__[0] )
,
Edge(Dangling Edge)[1]
,
Edge(Dangling Edge)[2]
,
Edge('__unnamed_node__[3] -> '__unnamed_node__[0] )
]
),
Node
(
name : '__unnamed_node__',
tensor :
array([[[1.+0.j, 0.+0.j],
 [0.+0.j, 0.+0.j]],

 [[0.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]]], dtype=complex64),
edges :
[
Edge(Dangling Edge)[0]
,
Edge(Dangling Edge)[1]
,
Edge('__unnamed_node__[2] -> '__unnamed_node__[0] )
]
),
Node
(
name : '__unnamed_node__',
tensor :
array([[[0.+0.j, 1.+0.j],
 [1.+0.j, 0.+0.j]],

 [[1.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]]], dtype=complex64),
edges :
[
Edge('__unnamed_node__[3] -> '__unnamed_node__[0] )
,
Edge(Dangling Edge)[1]
,
Edge(Dangling Edge)[2]
]
)
}

```

```
[16]: c.to_qir()[0]["gate"].eval_matrix()

[16]: array([[1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
           [0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
           [0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
           [0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
           [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
           [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j],
           [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j],
           [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j]],
          dtype=complex64)
```

3.2.8 Customized Contraction

Overview

If the simulated circuit has large qubit counts, we recommend users try a customized contraction setup instead of the default one, which is greedy.

Setup

Please refer to the [installation documentation](#) for cotengra, which cannot simply be obtained by pip install since it is not uploaded to PyPI. The easiest way for installation is `pip install -U git+https://github.com/jcmgray/cotengra.git`.

```
[3]: import tensorcircuit as tc
import numpy as np
import cotengra as ctg
```

We use the following example as a testbed for the contraction, the real contraction is invoked for `Circuit.expectation` API, and there are two stages for the contraction. The first one is contraction path searching which is used to find a better contraction path in terms of space and time. The second stage is the real contraction, where matrix multiplication is called using ML backend API. In this note, we focus on the performance of the first stage. And the contraction path solver can be customized with any type of [opt-einsum compatible path solver](#).

```
[4]: def testbed():
    n = 40
    d = 6
    param = K.ones([2 * d, n])
    c = tc.Circuit(n)
    c = tc.templates.blocks.example_block(c, param, nlayers=d, is_split=True)
    # the two-qubit gate is split and truncated with SVD decomposition
    return c.expectation_ps(z=[n // 2], reuse=False)
```

There are several contractor optimizers provided by opt-einsum and shipped with the TensorNetwork package. Since TensorCircuit is built on top of TensorNetwork, we can use these simple contractor optimizers. Though for any moderate system, only a greedy optimizer works, other optimizers come with exponential scaling and fail in circuit simulation scenarios.

We always set `contraction_info=True` (default is `False`) for the contractor system in this note, which will print contraction information summary including contraction size, flops, and write. For the definition of these metrics, also refer to cotengra docs and [the corresponding paper](#).

Metrics that measure the quality of a contraction path include

- **FLOPs**: the total number of computational operations required for all matrix multiplications involved when contracting the tensor network via the given path. This metric characterizes the total simulation time.
- **WRITE**: the total size (the number of elements) of all tensors – including intermediate tensors – computed during the contraction.
- **SIZE**: the size of the largest intermediate tensor stored in memory.

Since simulations in TensorCircuit are AD-enabled, where all intermediate results need to be cached and traced, the more relevant spatial cost metric is writes instead of size.

Also, we will enable `debug_level=2` in `set_contractor` (never use this option in real computation!) By enabling this, the second stage of the contraction, i.e. the real contraction, will not happen. We can focus on the contraction path information, which demonstrates the difference between different customized contractors.

```
[5]: tc.set_contractor("greedy", debug_level=2, contraction_info=True)
# the default contractor
testbed()

----- contraction cost summary -----
log10[FLOPs]: 12.393 log2[SIZE]: 30 log2[WRITE]: 35.125

[5]: <tf.Tensor: shape=(), dtype=complex64, numpy=0j>
```

cotengra optimizer: for hyperparameters tuning, see the documentation.

```
[7]: opt = ctg.ReusableHyperOptimizer(
    methods=["greedy", "kahypar"],
    parallel=True,
    minimize="write",
    max_time=120,
    max_repeats=1024,
    progbar=True,
)
# Caution: for now, parallel only works for "ray" in newer versions of python
tc.set_contractor(
    "custom", optimizer=opt, preprocessing=True, contraction_info=True, debug_level=2
)
# the opt-einsum compatible function interface is passed as the argument of optimizer\
# Also note how preprocessing=True merges the single qubits gate into the neighbor two-
→qubit gate
testbed()

log2[SIZE]: 15.00 log10[FLOPs]: 7.56: 45% | 458/1024 [02:03<02:
→32, 3.70it/s]

----- contraction cost summary -----
log10[FLOPs]: 7.565 log2[SIZE]: 15 log2[WRITE]: 19.192

[7]: <tf.Tensor: shape=(), dtype=complex64, numpy=0j>
```

We can even include contraction reconfigure after path searching, which further greatly boosts the space efficiency for the contraction path.

```
[8]: opt = ctg.ReusableHyperOptimizer(
    minimize="combo",
    max_repeats=1024,
```

(continues on next page)

(continued from previous page)

```

    max_time=120,
    progbar=True,
)

def opt_reconf(inputs, output, size, **kws):
    tree = opt.search(inputs, output, size)
    tree_r = tree.subtree_reconfigure_forest(
        progbar=True, num_trees=10, num_restarts=20, subtree_weight_what=("size",)
    )
    return tree_r.get_path()

# there is also a default parallel=True option for subtree_reconfigure_forest,
# this can only be set as "ray" for newer version python as above
# note how different versions of cotengra have breaking APIs in the last line: get_path_
# or path
# the user may need to change the API to make the example work

tc.set_contractor(
    "custom",
    optimizer=opt_reconf,
    contraction_info=True,
    preprocessing=True,
    debug_level=2,
)
testbed()

log2[SIZE]: 15.00 log10[FLOPs]: 7.46: 32%| 329/1024 [02:00
˓→<04:13, 2.74it/s]
log2[SIZE]: 14.00 log10[FLOPs]: 7.02: 100%|| 20/20 [01:05<00:00, 3.30s/it]
----- contraction cost summary -----
log10[FLOPs]: 7.021 log2[SIZE]: 14 log2[WRITE]: 19.953
[8]: <tf.Tensor: shape=(), dtype=complex64, numpy=0j>

```

3.2.9 Advanced Automatic Differentiation

Overview

In this section, we review some advanced AD tricks, especially their application to circuit simulations. With these advanced AD tricks, we can evaluate some quantum quantities more efficiently.

The advanced AD is possible in TensorCircuit, as we have implemented several AD-related API in a backend agnostic way, the implementation of them closely follows the design philosophy of [jax AD implementation](#).

Setup

```
[1]: import numpy as np  
import tensorcircuit as tc
```

```
[2]: n = 6  
      nlayers = 3
```

Backend agnostic AD related APIs include the following:

```
[3]: help(K.grad)
      help(K.value_and_grad)
      help(K.vectorized_value_and_grad)
      help(K.vjp)
      help(K.jvp)
      help(K.jacfwd)
      help(K.jacrev)
      help(K.stop_gradient)
      help(K.hessian)
```

Help on method grad in module tensorflow_backend:

```
grad(f: Callable[..., Any], argnums: Union[int, Sequence[int]] = 0, has_aux: bool = False) -> Callable[..., Any] method of tensorflowcircuit.backends.tensorflow_backend.TensorFlowBackend instance
    Return the function which is the grad function of input ``f``.
```

:Example:

```
>>> f = lambda x,y: x**2+2*y
>>> g = tc.backend.grad(f)
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
2
>>> g = tc.backend.grad(f, argnums=(0,1))
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
[2, 2]
```

:param f: the function to be differentiated

```
:type f: Callable[..., Any]
```

:param argnums: the position of args in ``f`` that are to be differentiated,
→ defaults to be 0

:type argnums:

:return: the grad function of ``f`` with the same

`return` the `gsl` function or `None` with the same set of arguments as `func`.

Type: ~~Callable[...]~~, ~~Any~~

help on method `value_and_grad` in module `tensorflow_circuit.backends.tensorflow_backend`.

Return the function which returns the value and grad of ``f``.

Return the function which returns the value and grad of γ .

(continues on next page)

(continued from previous page)

:Example:

```
>>> f = lambda x,y: x**2+2*y
>>> g = tc.backend.value_and_grad(f)
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
5, 2
>>> g = tc.backend.value_and_grad(f, argnums=(0,1))
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
5, [2, 2]

:param f: the function to be differentiated
:type f: Callable[..., Any]
:param argnums: the position of args in ``f`` that are to be differentiated, ↴
defaults to be 0
:type argnums: Union[int, Sequence[int]], optional
:return: the value and grad function of ``f`` with the same set of arguments as ``f``
:rtype: Callable[..., Tuple[Any, Any]]
```

Help on method vectorized_value_and_grad in module tensorcircuit.backends.tensorflow_backend:

```
vectorized_value_and_grad(f: Callable[..., Any], argnums: Union[int, Sequence[int]] = 0, ↴
vectorized_argnums: Union[int, Sequence[int]] = 0, has_aux: bool = False) -> Callable[..., Tuple[Any, Any]] method of tensorflow_backend.TensorFlowBackend instance
    Return the VVAG function of ``f``. The inputs for ``f`` is (args[0], args[1], ↴
args[2], ...),
    and the output of ``f`` is a scalar. Suppose VVAG(f) is a function with inputs in ↴
the form
    (vargs[0], args[1], args[2], ...), where vagrs[0] has one extra dimension than ↴
args[0] in the first axis
    and consistent with args[0] in shape for remaining dimensions, i.e. shape(vargs[0]) ↴
= [batch] + shape(args[0]).
    (We only cover cases where ``vectorized_argnums`` defaults to 0 here for ↴
demonstration).
    VVAG(f) returns a tuple as a value tensor with shape [batch, 1] and a gradient tuple ↴
with shape:
    ([batch]+shape(args[argnum]) for argnum in argnums). The gradient for argnums=k is ↴
defined as

... math::

g^k = \frac{\partial \sum_{i \in \text{batch}} f(vargs[0][i], args[1], \dots)}{\partial args[k]}
```

Therefore, if argnums=0, the gradient is reduced to

.. math::

$$g^0_i = \frac{\partial f(vargs[0][i])}{\partial vagrs[0][i]}$$

(continues on next page)

(continued from previous page)

, which is specifically suitable for batched VQE optimization, where `args[0]` is the circuit parameters.

And if `argnums=1`, the gradient is like

```
.. math::
g^1_i = \frac{\partial \sum_j f(\text{vargs}[0][j], \text{args}[1])}{\partial \text{args}[1][i]}

, which is suitable for quantum machine learning scenarios, where ``f`` is the loss function,
args[0] corresponds to the input data and args[1] corresponds to the weights in the QML model.
```

```
:param f: [description]
:type f: Callable[..., Any]
:param argnums: [description], defaults to 0
:type argnums: Union[int, Sequence[int]], optional
:param vectorized_argnums: the args to be vectorized, these arguments should share the same batch shape
    in the first dimension
:type vectorized_argnums: Union[int, Sequence[int]], defaults to 0
:return: [description]
:rtype: Callable[..., Tuple[Any, Any]]
```

Help on method `vjp` in module `tensorcircuit.backends.tensorflow_backend`:

```
vjp(f: Callable[..., Any], inputs: Union[Any, Sequence[Any]], v: Union[Any, Sequence[Any]]) -> Tuple[Union[Any, Sequence[Any]], Union[Any, Sequence[Any]]] method of tensorcircuit.backends.tensorflow_backend.TensorFlowBackend instance
Function that computes the dot product between a vector v and the Jacobian of the given function at the point given by the inputs. (reverse mode AD relevant)
Strictly speaking, this function is value_and_vjp.
```

```
:param f: the function to carry out vjp calculation
:type f: Callable[..., Any]
:param inputs: input for ``f``
:type inputs: Union[Tensor, Sequence[Tensor]]
:param v: value vector or gradient from downstream in reverse mode AD
    the same shape as return of function ``f``
:type v: Union[Tensor, Sequence[Tensor]]
:return: (`f(*inputs)`, vjp_tensor), where vjp_tensor is the same shape as inputs
:rtype: Tuple[Union[Tensor, Sequence[Tensor]], Union[Tensor, Sequence[Tensor]]]
```

Help on method `jvp` in module `tensorcircuit.backends.tensorflow_backend`:

```
jvp(f: Callable[..., Any], inputs: Union[Any, Sequence[Any]], v: Union[Any, Sequence[Any]]) -> Tuple[Union[Any, Sequence[Any]], Union[Any, Sequence[Any]]] method of tensorcircuit.backends.tensorflow_backend.TensorFlowBackend instance
Function that computes a (forward-mode) Jacobian-vector product of ``f``.
Strictly speaking, this function is value_and_jvp.
```

```
:param f: The function to compute jvp
```

(continues on next page)

(continued from previous page)

```
:type f: Callable[..., Any]
:param inputs: input for ``f``
:type inputs: Union[Tensor, Sequence[Tensor]]
:param v: tangents
:type v: Union[Tensor, Sequence[Tensor]]
:return: (`f(*inputs)`, jvp_tensor), where jvp_tensor is the same shape as the
output of ``f``
:rtype: Tuple[Union[Tensor, Sequence[Tensor]], Union[Tensor, Sequence[Tensor]]]
```

Help on method jacfwd in module tensorcircuit.backends.abstract_backend:

```
jacfwd(f: Callable[..., Any], argnums: Union[int, Sequence[int]] = 0) -> Any method of
tensorcircuit.backends.tensorflow_backend.TensorFlowBackend instance
Compute the Jacobian of ``f`` using the forward mode AD.
```

```
:param f: the function whose Jacobian is required
:type f: Callable[..., Any]
:param argnums: the position of the arg as Jacobian input, defaults to 0
:type argnums: Union[int, Sequence[int]], optional
:return: outer tuple for input args, inner tuple for outputs
:rtype: Tensor
```

Help on method jacrev in module tensorcircuit.backends.abstract_backend:

```
jacrev(f: Callable[..., Any], argnums: Union[int, Sequence[int]] = 0) -> Any method of
tensorcircuit.backends.tensorflow_backend.TensorFlowBackend instance
Compute the Jacobian of ``f`` using reverse mode AD.
```

```
:param f: The function whose Jacobian is required
:type f: Callable[..., Any]
:param argnums: the position of the arg as Jacobian input, defaults to 0
:type argnums: Union[int, Sequence[int]], optional
:return: outer tuple for output, inner tuple for input args
:rtype: Tensor
```

Help on method stop_gradient in module tensorcircuit.backends.tensorflow_backend:

```
stop_gradient(a: Any) -> Any method of tensorcircuit.backends.tensorflow_backend.
TensorFlowBackend instance
Stop backpropagation from ``a``.
```

```
:param a: [description]
:type a: Tensor
:return: [description]
:rtype: Tensor
```

Help on method hessian in module tensorcircuit.backends.abstract_backend:

```
hessian(f: Callable[..., Any], argnums: Union[int, Sequence[int]] = 0) -> Any method of
tensorcircuit.backends.tensorflow_backend.TensorFlowBackend instance
```

Forward AD

Using the Jacobian vector product (jvp), we can compute the circuit gradient in the forward AD mode, which is more suitable when the number of output elements is much larger than the input.

Suppose we are going to evaluate $\partial|\psi(\theta)\rangle$, where $\psi(\theta) = U(\theta)|0\rangle$ is the output state of some parameterized quantum circuit.

```
[4]: def ansatz(thetas):
    c = tc.Circuit(n)
    for j in range(nlayers):
        for i in range(n):
            c.rx(i, theta=thetas[j])
        for i in range(n - 1):
            c.cnot(i, i + 1)
    return c

def psi(thetas):
    c = ansatz(thetas)
    return c.state()
```

```
[5]: state, partial_psi_partial_theta0 = K.jvp(
    psi,
    K.implicit_randn([nlayers]),
    tc.array_to_tensor(np.array([1.0, 0, 0]), dtype="float32"),
)
```

We thus obtain $\frac{\partial\psi}{\partial\theta_0}$, since the tangent takes one in the first place and zero in other positions.

```
[6]: state.shape, partial_psi_partial_theta0.shape
[6]: (TensorShape([64]), TensorShape([64]))
```

Jacobian

We can compute the Jacobian row by row or col by col using vmap together with reverse mode or forward mode AD.

We still use the above example, to calculate Jacobian $J_{ij} = \frac{\partial\psi_i}{\partial\theta_j}$.

```
[7]: thetas = K.implicit_randn([nlayers])

jac_fun = K.jit(K.jacfwd(psi))

jac_value = jac_fun(thetas)
```

```
[8]: %timeit jac_fun(thetas)
601 µs ± 36.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
[9]: jac_value.shape
[9]: TensorShape([64, 3])
```

We can also use reverse mode AD to obtain Jacobian.

```
[10]: jac_fun2 = K.jit(K.jacrev(psi))

jac_value2 = jac_fun2(thetas)
```

```
[11]: %timeit jac_fun2(thetas)
843 µs ± 9.95 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
[12]: jac_value2.shape
```

```
[12]: TensorShape([64, 3])
```

```
[13]: np.testing.assert_allclose(np.real(jac_value), jac_value2, atol=1e-5)
```

It is worth noting that forward mode AD Jacobian is faster since the result Jacobian is a tall matrix.

Quantum Fisher Information

Quantum Fisher Information is a very important quantity in quantum computation, which can be utilized in so-called quantum natural gradient descent optimization as well as variational quantum dynamics. See [reference paper](#) for more details.

There are several variants of QFI like object, and the core to evaluate is all related to $\langle \partial\psi|\partial\psi \rangle - \langle \partial\psi|\psi \rangle \langle \psi|\partial\psi \rangle$. Such quantity is easily obtained with an advanced AD framework by first getting the Jacobian for the state and then vmap the inner product over Jacobian rows. The detailed implementation can be found in the codebase `tensorcircuit/experimental.py`. We directly call the corresponding API in this note.

```
[14]: from tensorcircuit.experimental import qng
```

```
[15]: qfi_fun = K.jit(qng(psi))
qfi_value = qfi_fun(thetas)
qfi_value.shape
```

WARNING:tensorflow:The dtype of the watched primal must be floating (e.g. tf.float32),
↳ got tf.complex64

```
[15]: TensorShape([3, 3])
```

```
[16]: %timeit qfi_fun(thetas) # the speed is comparable with a simple Jacobian computation
```

609 µs ± 14.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

Hessian

Hessian is defined as $\partial_{ij} \langle \psi(\theta)|H|\psi(\theta) \rangle$, where ij is shorthand for $\theta_i\theta_j$.

In the following examples, we use $H = Z_0$ for simplicity.

```
[17]: def h(thetas):
    c = ansatz(thetas)
    return K.real(c.expectation_ps(z=[0]))
```

```
hess_f = K.jit(K.hessian(h))
```

```
[18]: hess_value = hess_f(thetas)
hess_value.shape
```

```
[18]: TensorShape([3, 3])
```

$$\langle \psi | H | \partial \psi \rangle$$

This quantity is very common as the RHS of the variational quantum dynamics equation. And there is no good way to compute this quantity besides constructing a corresponding Hadamard test circuit.

However, we can easily obtain this in the AD framework, as long as the `stop_gradint` API exists, which is the case for TensorCircuit. Namely, this quantity is obtained as $\partial(\langle \psi | H | \perp(\psi) \rangle)$, where the outside ∂ is automatically implemented by AD and \perp is for `stop_gradient` op which stop the backpropagation.

```
[19]: z0 = tc.quantum.PauliString2Dense([[3, 0, 0, 0, 0, 0]])
```

```
def h(thetas):
    w = psi(thetas)
    wr = K.stop_gradient(w)
    wl = K.conj(w)
    wl = K.reshape(wl, [1, -1])
    wr = K.reshape(wr, [-1, 1])
    e = wl @ z0 @ wr
    return K.real(e)[0, 0]
```

```
[20]: psi_h_partial_psi = K.grad(h)(thetas)
psi_h_partial_psi.shape
```

```
[20]: TensorShape([3])
```


API REFERENCES

4.1 tensorcircuit

4.1.1 tensorcircuit.abstractcircuit

Methods for abstract circuits independent of nodes, edges and contractions

class tensorcircuit.abstractcircuit.**AbstractCircuit**

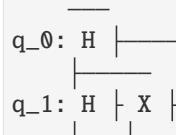
Bases: object

append(*c*: tensorcircuit.abstractcircuit.AbstractCircuit, *indices*: *Optional[List[int]]* = None) → tensorcircuit.abstractcircuit.AbstractCircuit
append circuit *c* before

Example

```
>>> c1 = tc.Circuit(2)
>>> c1.H(0)
>>> c1.H(1)
>>> c2 = tc.Circuit(2)
>>> c2.cnot(0, 1)
>>> c1.append(c2)
<tensorcircuit.circuit.Circuit object at 0x7f8402968970>
>>> c1.draw()

```



Parameters

- **c** (*BaseCircuit*) – The other circuit to be appended
- **indices** (*Optional[List[int]]*, *optional*) – the qubit indices to which *c* is appended on. Defaults to None, which means plain concatenation.

Returns The composed circuit

Return type *BaseCircuit*

append_from_qir(*qir*: *List[Dict[str, Any]]*) → None

Apply the ciurict in form of quantum intermediate representation after the current cirucit.

Example

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.to_qir()
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None, 'mpo': False}]
>>> c2 = tc.Circuit(3)
>>> c2.CNOT(0, 1)
>>> c2.to_qir()
[{'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None, 'mpo': False}]
>>> c.append_from_qir(c2.to_qir())
>>> c.to_qir()
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None, 'mpo': False},
 {'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None, 'mpo': False}]
```

Parameters `qir` (`List[Dict[str, Any]]`) – The quantum intermediate representation.

`apply_general_gate`(`gate: Union[tensorcircuit.gates.Gate, tensorcircuit.quantum.QuOperator]`, `*index: int`, `name: Optional[str] = None`, `split: Optional[Dict[str, Any]] = None`, `mpo: bool = False`, `ir_dict: Optional[Dict[str, Any]] = None`) → `None`

An implementation of this method should also append gate directionary to self._cir

`static apply_general_gate_delayed`(`gatef: Callable[[], tensorcircuit.gates.Gate]`, `name: Optional[str] = None`, `mpo: bool = False`) → `Callable[[], None]`

`static apply_general_variable_gate_delayed`(`gatef: Callable[[], tensorcircuit.gates.Gate]`, `name: Optional[str] = None`, `mpo: bool = False`) → `Callable[[], None]`

`barrier_instruction`(*`index: List[int]`) → `None`

add a barrier instruction flag, no effect on numerical simulation

Parameters `index` (`List[int]`) – the corresponding qubits

`circuit_param: Dict[str, Any]`

`cond_measure`(`index: int`) → `Any`

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a `int` Tensor and thus maintained a jittable pipeline.

Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
# two possible outputs: (1, 1) or (-1, -1)
```

Note: In terms of `DMCircuit`, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measurement results

Parameters `index (int)` – the qubit for the z-basis measurement

Returns 0 or 1 for z measurement on up and down freedom

Return type Tensor

`cond_measurement(index: int) → Any`

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a int Tensor and thus maintained a jittable pipeline.

Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
# two possible outputs: (1, 1) or (-1, -1)
```

Note: In terms of DMCircuit, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measuremet results

Parameters `index (int)` – the qubit for the z-basis measurement

Returns 0 or 1 for z measurement on up and down freedom

Return type Tensor

`conditional_gate(which: Any, kraus: Sequence[tensorcircuit.gates.Gate], *index: int) → None`

Apply `which`-th gate from `kraus` list, i.e. apply `kraus[which]`

Parameters

- `which (Tensor)` – Tensor of shape [] and dtype int
- `kraus (Sequence[Gate])` – A list of gate in the form of `tc.gate` or Tensor
- `index (int)` – the qubit lines the gate applied on

`draw(**kws: Any) → Any`

Visualise the circuit. This method recevies the keywords as same as `qiskit.circuit.QuantumCircuit.draw`. More details can be found here: <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.draw.html>.

Example

```
>>> c = tc.Circuit(3)
>>> c.H(1)
>>> c.X(2)
>>> c.CNOT(0, 1)
>>> c.draw(output='text')
q_0: _____
      |
q_1: H +---+
      |   |
      +---+
q_2: X +---+
```

```
expectation(*ops: Tuple[tensornetwork.network_components.Node, List[int]], reuse: bool = True,
noise_conf: Optional[Any] = None, nmc: int = 1000, status: Optional[Any] = None, **kws: Any) → Any
```

```
expectation_ps(x: Optional[Sequence[int]] = None, y: Optional[Sequence[int]] = None, z:
Optional[Sequence[int]] = None, reuse: bool = True, noise_conf: Optional[Any] = None,
nmc: int = 1000, status: Optional[Any] = None, **kws: Any) → Any
```

Shortcut for Pauli string expectation. x, y, z list are for X, Y, Z positions

Example

```
>>> c = tc.Circuit(2)
>>> c.X(0)
>>> c.H(1)
>>> c.expectation_ps(x=[1], z=[0])
array(-0.99999994+0.j, dtype=complex64)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> c.expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
(0.46274087-3.764033e-09j)
```

Parameters

- **x** (*Optional[Sequence[int]]*, *optional*) – sites to apply X gate, defaults to None
- **y** (*Optional[Sequence[int]]*, *optional*) – sites to apply Y gate, defaults to None
- **z** (*Optional[Sequence[int]]*, *optional*) – sites to apply Z gate, defaults to None
- **reuse** (*bool*, *optional*) – whether to cache and reuse the wavefunction, defaults to True
- **noise_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **nmc** (*int*, *optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **status** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

Returns Expectation value

Return type Tensor

```
classmethod from_json(jsonstr: str, circuit_params: Optional[Dict[str, Any]] = None) →
tensorcircuit.abstractcircuit.AbstractCircuit
```

load json str as a Circuit

Parameters

- **jsonstr** (*str*) – `_description_`
- **circuit_params** (*Optional[Dict[str, Any]]*, *optional*) – Extra circuit parameters in the format of `__init__`, defaults to None

Returns `_description_`

Return type `AbstractCircuit`

```
classmethod from_json_file(file: str, circuit_params: Optional[Dict[str, Any]] = None) →
    tensorcircuit.abstractcircuit.AbstractCircuit
```

load json file and convert it to a circuit

Parameters

- **file** (*str*) – filename
- **circuit_params** (*Optional[Dict[str, Any]]*, *optional*) – `_description_`, defaults to None

Returns `_description_`

Return type `AbstractCircuit`

```
classmethod from_openqasm(qasmstr: str, circuit_params: Optional[Dict[str, Any]] = None,
                           keep_measure_order: bool = False) →
    tensorcircuit.abstractcircuit.AbstractCircuit
```

```
classmethod from_openqasm_file(file: str, circuit_params: Optional[Dict[str, Any]] = None,
                               keep_measure_order: bool = False) →
    tensorcircuit.abstractcircuit.AbstractCircuit
```

```
classmethod from_qir(qir: List[Dict[str, Any]], circuit_params: Optional[Dict[str, Any]] = None) →
    tensorcircuit.abstractcircuit.AbstractCircuit
```

Restore the circuit from the quantum intermediate representation.

Example

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.rx(1, theta=tc.array_to_tensor(0.7))
>>> c.exp1(0, 1, unitary=tc.gates._zz_matrix, theta=tc.array_to_tensor(-0.2), ↴
    split=split)
>>> len(c)
7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)
>>> qirs = c.to_qir()
>>>
>>> c = tc.Circuit.from_qir(qirs, circuit_params={"nqubits": 3})
>>> len(c._nodes)
7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)
```

Parameters

- **qir** (*List[Dict[str, Any]]*) – The quantum intermediate representation of a circuit.
- **circuit_params** (*Optional[Dict[str, Any]]*) – Extra circuit parameters.

Returns The circuit have same gates in the qir.

Return type *Circuit*

```
classmethod from_qiskit(qc: Any, n: Optional[int] = None, inputs: Optional[List[float]] = None,
                        circuit_params: Optional[Dict[str, Any]] = None, binding_params:
                        Optional[Union[Sequence[float], Dict[Any, float]]] = None) →
                        tensorcircuit.abstractcircuit.AbstractCircuit
```

Import Qiskit QuantumCircuit object as a `tc.Circuit` object.

Example

```
>>> from qiskit import QuantumCircuit
>>> qisc = QuantumCircuit(3)
>>> qisc.h(2)
>>> qisc.cswap(1, 2, 0)
>>> qisc.swap(0, 1)
>>> c = tc.Circuit.from_qiskit(qisc)
```

Parameters

- **qc** (*QuantumCircuit in Qiskit*) – Qiskit Circuit object
- **n** (*int*) – The number of qubits for the circuit
- **inputs** (*Optional[List[float]]*, *optional*) – possible input wavefunction for `tc.Circuit`, defaults to None
- **circuit_params** (*Optional[Dict[str, Any]]*) – kwargs given in `Circuit.__init__` construction function, default to None.
- **binding_params** (*Optional[Union[Sequence[float], Dict[Any, float]]]*) – (variational) parameters for the circuit. Could be either a sequence or dictionary depending on the type of parameters in the Qiskit circuit. For `ParameterVectorElement` use sequence. For `Parameter` use dictionary

Returns The same circuit but as `tensorcircuit` object

Return type *Circuit*

```
classmethod from_qsim_file(file: str, circuit_params: Optional[Dict[str, Any]] = None) →
                        tensorcircuit.abstractcircuit.AbstractCircuit
```

```
gate_aliases = [['cnot', 'cx'], ['fredkin', 'cswap'], ['toffoli', 'ccnot'],
['toffoli', 'ccx'], ['any', 'unitary'], ['sd', 'sdg'], ['td', 'tdg']]
```

```
gate_count(gate_list: Optional[Sequence[str]] = None) → int
count the gate number of the circuit
```

Example

```
>>> c = tc.Circuit(3)
>>> c.h(0)
>>> c.multicontrol(0, 1, 2, ctrl=[0, 1], unitary=tc.gates._x_matrix)
>>> c.toffoli(1, 2, 0)
>>> c.gate_count()
3
>>> c.gate_count(["multicontrol", "toffoli"])
2
```

Parameters `gate_list` (*Optional[Sequence[str]]*, *optional*) – gate name list to be counted, defaults to None (counting all gates)

Returns the total number of all gates or gates in the `gate_list`

Return type int

`gate_summary()` → Dict[str, int]

return the summary dictionary on gate type - gate count pair

Returns the gate count dict by gate type

Return type Dict[str, int]

`get_positional_logical_mapping()` → Dict[int, int]

Get positional logical mapping dict based on measure instruction. This function is useful when we only measure part of the qubits in the circuit, to process the count result from partial measurement, we must be aware of the mapping, i.e. for each position in the count bitstring, what is the corresponding qubits (logical) defined on the circuit

Returns positional_logical_mapping

Return type Dict[int, int]

`initial_mapping(logical_physical_mapping: Dict[int, int], n: Optional[int] = None, circuit_params:`

`Optional[Dict[str, Any]] = None) → tensorcircuit.abstractcircuit.AbstractCircuit`

generate a new circuit with the qubit mapping given by `logical_physical_mapping`

Parameters

- `logical_physical_mapping` (`Dict[int, int]`) – how to map logical qubits to the physical qubits on the new circuit
- `n` (*Optional[int]*, *optional*) – number of qubit of the new circuit, can be different from the original one, defaults to None
- `circuit_params` (*Optional[Dict[str, Any]]*, *optional*) – `_description_`, defaults to None

Returns `_description_`

Return type `AbstractCircuit`

inputs: Any

`inverse(circuit_params: Optional[Dict[str, Any]] = None) → tensorcircuit.abstractcircuit.AbstractCircuit`

inverse the circuit, return a new inversed circuit

EXAMPLE

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rzz(1, 2, theta=0.8)
>>> c1 = c.inverse()
```

Parameters `circuit_params` (*Optional[Dict[str, Any]]*, *optional*) – keywords dict for initialization the new circuit, defaults to None

Returns the inversed circuit

Return type `Circuit`

is_mps: bool

measure_instruction(*index: int*) → None
add a measurement instruction flag, no effect on numerical simulation

Parameters **index** (*int*) – the corresponding qubit

mpogates = ['multicontrol', 'mpo']

prepend(*c: tensorcircuit.abstractcircuit.AbstractCircuit*) → *tensorcircuit.abstractcircuit.AbstractCircuit*
prepend circuit *c* before

Parameters **c** (*BaseCircuit*) – The other circuit to be prepended

Returns The composed circuit

Return type *BaseCircuit*

reset_instruction(*index: int*) → None
add a reset instruction flag, no effect on numerical simulation

Parameters **index** (*int*) – the corresponding qubit

select_gate(*which: Any, kraus: Sequence[tensorcircuit.gates.Gate], *index: int*) → None
Apply *which*-th gate from *kraus* list, i.e. apply *kraus[which]*

Parameters

- **which** (*Tensor*) – Tensor of shape [] and dtype int
- **kraus** (*Sequence[Gate]*) – A list of gate in the form of *tc.gate* or *Tensor*
- **index** (*int*) – the qubit lines the gate applied on

sgates = ['i', 'x', 'y', 'z', 'h', 't', 's', 'td', 'sd', 'wroot', 'cnot', 'cz', 'swap', 'cy', 'ox', 'oy', 'oz', 'toffoli', 'fredkin']

static standardize_gate(*name: str*) → *str*
standardize the gate name to tc common gate sets

Parameters **name** (*str*) – non-standard gate name

Returns the standard gate name

Return type *str*

tex(***kws: Any*) → *str*
Generate latex string based on quantikz latex package

Returns Latex string that can be directly compiled via, e.g. *latexit*

Return type *str*

to_cirq(*enable_instruction: bool = False*) → *Any*
Translate *tc.Circuit* to a cirq circuit object.

Parameters **enable_instruction** (*bool, defaults to False*) – whether also export measurement and reset instructions

Returns A cirq circuit of this circuit.

to_json(*file: Optional[str] = None, simplified: bool = False*) → *Any*
circuit dumps to json

Parameters

- **file** (*Optional[str], optional*) – file str to dump the json to, defaults to None, return the json str

- **simplified** (*bool*) – If False, keep all info for each gate, defaults to be False. If True, suitable for IO since less information is required

Returns None if dumps to file otherwise the json str

Return type Any

to_openqasm(***kws*: Any) → str

transform circuit to openqasm via qiskit circuit, see <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.qasm.html> for usage on possible options for *kws*

Returns circuit representation in openqasm format

Return type str

to_qir() → List[Dict[str, Any]]

Return the quantum intermediate representation of the circuit.

Example

```
>>> c = tc.Circuit(2)
>>> c.CNOT(0, 1)
>>> c.to_qir()
[{'gatef': cnot, 'gate': Gate(
    name: 'cnot',
    tensor:
        array([[[[1.+0.j, 0.+0.j],
               [0.+0.j, 0.+0.j]],
               [[0.+0.j, 1.+0.j],
               [0.+0.j, 0.+0.j]]],,
               [[[0.+0.j, 0.+0.j],
               [0.+0.j, 1.+0.j]],
               [[0.+0.j, 0.+0.j],
               [1.+0.j, 0.+0.j]]]], dtype=complex64),
    edges: [
        Edge(Dangling Edge)[0],
        Edge(Dangling Edge)[1],
        Edge('cnot'[2] -> 'qb-1'[0]),
        Edge('cnot'[3] -> 'qb-2'[0])
    ], 'index': (0, 1), 'name': 'cnot', 'split': None, 'mpo': False}]
```

Returns The quantum intermediate representation of the circuit.

Return type List[Dict[str, Any]]

to_qiskit(enable_instruction: bool = False) → Any

Translate `tc.Circuit` to a qiskit `QuantumCircuit` object.

Parameters `enable_instruction` (*bool*, defaults to False) – whether also export measurement and reset instructions

Returns A qiskit object of this circuit.

```
vgates = ['r', 'cr', 'u', 'cu', 'rx', 'ry', 'rz', 'phase', 'rxx', 'ryy', 'rzz',
'cphase', 'crx', 'cry', 'crz', 'orx', 'ory', 'orz', 'iswap', 'any', 'exp', 'exp1']
```

vis_tex(***kws*: Any) → str

Generate latex string based on quantikz latex package

Returns Latex string that can be directly compiled via, e.g. latexit**Return type** str

4.1.2 tensorcircuit.applications

tensorcircuit.applications.dqas

Modules for DQAS framework

```
tensorcircuit.applications.dqas.DQAS_search(kernel_func: Callable[[Any, Any, Sequence[int]], Tuple[Any, Any]], *, g: Optional[Iterator[Any]] = None, op_pool: Optional[Sequence[Any]] = None, p: Optional[int] = None, p_nnp: Optional[int] = None, p_stp: Optional[int] = None, batch: int = 300, prethermal: int = 0, epochs: int = 100, parallel_num: int = 0, verbose: bool = False, verbose_func: Optional[Callable[[], None]] = None, history_func: Optional[Callable[[], Any]] = None, prob_clip: Optional[float] = None, baseline_func: Optional[Callable[[Sequence[float]], float]] = None, perturbation_func: Optional[Callable[[], Any]] = None, nnp_initial_value: Optional[Any] = None, stp_initial_value: Optional[Any] = None, network_opt: Optional[Any] = None, structure_opt: Optional[Any] = None, prethermal_opt: Optional[Any] = None, prethermal_preset: Optional[Sequence[int]] = None, stp_regularization: Optional[Callable[[Any, Any], Any]] = None, nnp_regularization: Optional[Callable[[Any, Any], Any]] = None) → Tuple[Any, Any, Sequence[Any]]
```

DQAS framework entrypoint

Parameters

- **kernel_func** – function with input of data instance, circuit parameters theta and structural parameter k, return tuple of objective value and gradient with respect to theta
- **g** – data generator as dataset
- **op_pool** – list of operations as primitive operator pool
- **p** – the default layer number of the circuit ansatz
- **p_nnp** – shape of circuit parameter pool, in general $p_{stp} \times l$, where l is the max number of circuit parameters for op in the operator pool
- **p_stp** – the same as p in the most times
- **batch** – batch size of one epoch
- **prethermal** – prethermal update times
- **epochs** – training epochs
- **parallel_num** – parallel thread number, 0 to disable multiprocessing model by default
- **verbose** – set verbose log to print
- **verbose_func** – function to output verbose information

- **history_func** – function return intermediate result for final history list
- **prob_clip** – cutoff probability to avoid peak distribution
- **baseline_func** – function accepting list of objective values and return the baseline value used in the next round
- **perturbation_func** – return noise with the same shape as circuit parameter pool
- **nnp_initial_value** – initial values for circuit parameter pool
- **stp_initial_value** – initial values for probabilistic model parameters
- **network_opt** – optimizer for circuit parameters theta
- **structure_opt** – optimizer for model parameters alpha
- **prethermal_opt** – optimizer for circuit parameters in prethermal stage
- **prethermal_preset** – fixed structural parameters for prethermal training
- **stp_regularization** – regularization function for model parameters alpha
- **nnp_regularization** – regularization function for circuit parameters theta

Returns

```
tensorcircuit.applications.dqas.DQAS_search_pmb(kernel_func: Callable[[Any, Any, Sequence[int]], Tuple[Any, Any]], prob_model: Any, *, sample_func: Optional[Callable[[Any, int], Tuple[List[Any], List[List[Any]]]]] = None, g: Optional[Iterator[Any]] = None, op_pool: Optional[Sequence[Any]] = None, p: Optional[int] = None, batch: int = 300, prethermal: int = 0, epochs: int = 100, parallel_num: int = 0, verbose: bool = False, verbose_func: Optional[Callable[[], None]] = None, history_func: Optional[Callable[[], Any]] = None, baseline_func: Optional[Callable[[Sequence[float]], float]] = None, perturbation_func: Optional[Callable[[], Any]] = None, nnp_initial_value: Optional[Any] = None, stp_regularization: Optional[Callable[[Any, Any], Any]] = None, network_opt: Optional[Any] = None, structure_opt: Optional[Any] = None, prethermal_opt: Optional[Any] = None, loss_func: Optional[Callable[[Any], Any]] = None, loss_derivative_func: Optional[Callable[[Any], Any]] = None, validate_period: int = 0, validate_batch: int = 1, validate_func: Optional[Callable[[Any, Any, Sequence[int]], Tuple[Any, Any]]] = None, vg: Optional[Iterator[Any]] = None) → Tuple[Any, Any, Sequence[Any]]
```

The probabilistic model based DQAS, can use extensively for DQAS case for NMF probabilistic model.

Parameters

- **kernel_func** – vag func, return loss and nabla lnp
- **prob_model** – keras model
- **sample_func** – sample func of logic with keras model input

- **g** – input data pipeline generator
- **op_pool** – operation pool
- **p** – depth for DQAS
- **batch** –
- **prethermal** –
- **epochs** –
- **parallel_num** – parallel kernels
- **verbose** –
- **verbose_func** –
- **history_func** –
- **baseline_func** –
- **perturbation_func** –
- **nnp_initial_value** –
- **stp_regularization** –
- **network_opt** –
- **structure_opt** –
- **prethermal_opt** –
- **loss_func** – final loss function in terms of average of sub loss for each circuit
- **loss_derivative_func** – derivative function for `loss_func`

Returns

```
tensorcircuit.applications.dqas.evaluate_everyone(vag_func: Any, gdata: Iterator[Any], nnp: Any,  
presets: Sequence[Sequence[List[int]]], batch: int  
= 1) → Sequence[Tuple[Any, Any]]
```

```
tensorcircuit.applications.dqas.get_op_pool() → Sequence[Any]
```

```
tensorcircuit.applications.dqas.get_preset(stp: Any) → Any
```

```
tensorcircuit.applications.dqas.get_var(name: str) → Any
```

Call in customized functions and grab variables within DQAS framework function by var name str.

Parameters `name` (str) – The DQAS framework function

Returns Variables within the DQAS framework

Return type Any

```
tensorcircuit.applications.dqas.get_weights(nnp: Any, stp: Optional[Any] = None, preset:  
Optional[Sequence[int]] = None) → Any
```

This function works only when nnp has the same shape as stp, i.e. one parameter for each op.

Parameters

- **nnp** –
- **stp** –
- **preset** –

Returns

```
tensorcircuit.applications.dqas.get_weights_v2(nnp: Any, preset: Sequence[int]) → Any
tensorcircuit.applications.dqas.history_loss() → Any
tensorcircuit.applications.dqas.micro_sample(prob_model: Any, batch_size: int, repetitions:
                                              Optional[List[int]] = None) → Tuple[List[Any],
                                              List[List[Any]]]
tensorcircuit.applications.dqas.parallel_kernel(prob: Any, gdata: Any, nnp: Any, kernel_func:
                                                Callable[[Any, Any, Sequence[int]], Tuple[Any,
                                                Any]]) → Tuple[Any, Any, Any]
```

The kernel for multiprocess to run parallel in DQAS function/

Parameters

- **prob** –
- **gdata** –
- **nnp** –
- **kernel_func** –

Returns

```
tensorcircuit.applications.dqas.parallel_qaoa_train(preset: Sequence[int], g: Any, vag_func:
                                                    Optional[Any] = None, opt: Optional[Any] =
                                                    None, epochs: int = 60, tries: int = 16, batch:
                                                    int = 1, cores: int = 8, loc: float = 0.0, scale:
                                                    float = 1.0, nnp_shape: Optional[Sequence[int]] =
                                                    None, search_func: Optional[Callable[[...],
                                                    Any]] = None, kws: Optional[Dict[Any, Any]] =
                                                    None) → Sequence[Any]
```

parallel variational parameter training and search to avoid local minimum not limited to qaoa setup as the function name indicates, as long as you provided suitable *vag_func*

Parameters

- **preset** –
- **g** – data input generator for *vag_func*
- **vag_func** – *vag_kernel*
- **opt** –
- **epochs** –
- **tries** – number of tries
- **batch** – for optimization problem the input is in general fixed so batch is often 1
- **cores** – number of parallel jobs
- **loc** – mean value of normal distribution for *nnp*
- **scale** – std deviation of normal distribution for *nnp*

Returns

```
tensorcircuit.applications.dqas.preset_byprob(prob: Any) → Sequence[int]
```

```
tensorcircuit.applications.dqas.qaoa_simple_train(preset: Sequence[int], graph:  
    Union[Sequence[Any], Iterator[Any]], vag_func:  
    Optional[Callable[[Any, Any, Sequence[int]],  
    Tuple[Any, Any]]] = None, epochs: int = 60, batch:  
    int = 1, nnp_shape: Optional[Any] = None,  
    nnp_initial_value: Optional[Any] = None, opt:  
    Optional[Callable[..., Any]] = None, search_func:  
    Optional[Callable[[...], Any]] = None, kws:  
    Optional[Dict[Any, Any]] = None) → Tuple[Any,  
    float]  
  
tensorcircuit.applications.dqas.repr_op(element: Any) → str  
  
tensorcircuit.applications.dqas.set_op_pool(l: Sequence[Any]) → None  
  
tensorcircuit.applications.dqas.single_generator(g: Any) → Iterator[Any]  
  
tensorcircuit.applications.dqas.van_regularization(prob_model: Any, nnp: Optional[Any] = None,  
    lbd_w: float = 0.01, lbd_b: float = 0.01) → Any  
  
tensorcircuit.applications.dqas.van_sample(prob_model: Any, batch_size: int) → Tuple[List[Any],  
    List[List[Any]]]  
  
tensorcircuit.applications.dqas.verbose_output(max_prob: bool = True, weight: bool = True) → None  
    Doesn't support prob model DQAS search.
```

Parameters

- **max_prob** –
- **weight** –

Returns

```
tensorcircuit.applications.dqas.void_generator() → Iterator[Any]
```

tensorcircuit.applications.graphdata

Modules for graph instance data and more

```
tensorcircuit.applications.graphdata.Grid2D(m: int, n: int, pbc: bool = True) → Any  
tensorcircuit.applications.graphdata.Triangle2D(m: int, n: int) → Any  
tensorcircuit.applications.graphdata.all_nodes_covered(g: Any) → bool  
tensorcircuit.applications.graphdata.dict2graph(d: Dict[Any, Any]) → Any  
`python d = nx.to_dict_of_dicts(g)`
```

Parameters d –**Returns**

```
tensorcircuit.applications.graphdata.dress_graph_with_cirq_qubit(g: Any) → Any  
tensorcircuit.applications.graphdata.ensemble_maxcut_solution(g: Any, samples: int = 100) →  
    Tuple[float, float]  
tensorcircuit.applications.graphdata.erdos_graph_generator(*args, **kwargs) → Iterator[Any]  
tensorcircuit.applications.graphdata.even1D(n: int, s: int = 0) → Any  
tensorcircuit.applications.graphdata.get_graph(c: str) → Any
```

`tensorcircuit.applications.graphdata.graph1D(n: int, pbc: bool = True) → Any`
 1D PBC chain with n sites.

Parameters `n (int)` – The number of nodes

Returns The resulted graph g

Return type Graph

`tensorcircuit.applications.graphdata.maxcut_solution_bruteforce(g: Any) → Tuple[float, Sequence[int]]`

`tensorcircuit.applications.graphdata.odd1D(n: int, *, s: int = 1) → Any`

`tensorcircuit.applications.graphdata.reduce_edges(g: Any, m: int = 1) → Sequence[Any]`

Parameters

- `g` –
- `m` –

Returns all graphs with m edge out from g

`tensorcircuit.applications.graphdata.reduced_ansatz(g: Any, ratio: Optional[int] = None) → Any`
 Generate a reduced graph with given ratio of edges compared to the original graph g.

Parameters

- `g (Graph)` – The base graph
- `ratio` – number of edges kept, default half of the edges

Returns The resulted reduced graph

Return type Graph

`tensorcircuit.applications.graphdata.regular_graph_generator(d: int, n: int, weights: bool = False) → Iterator[Any]`

`tensorcircuit.applications.graphdata.split_ansatz(g: Any, split: int = 2) → Sequence[Any]`

Split the graph in exactly split piece evenly.

Parameters

- `g (Graph)` – The mother graph
- `split (int, optional)` – The number of the graph we want to divide into, defaults to 2

Returns List of graph instance of size split

Return type Sequence[Graph]

tensorcircuit.applications.layers

Module for functions adding layers of circuits

`tensorcircuit.applications.layers.Hlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Optional[Union[Any, float]] = None, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.Ilayer(circuit: tensorcircuit.circuit.Circuit, symbol: Optional[Union[Any, float]] = None, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

```
tensorcircuit.applications.layers.anyHlayer(circuit: tensorcircuit.circuit.Circuit, symbol:  
Optional[Union[Any, float]] = None, g: Optional[Any] =  
None) → tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyIlayer(circuit: tensorcircuit.circuit.Circuit, symbol:  
Optional[Union[Any, float]] = None, g: Optional[Any] =  
None) → tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyrxlayer(circuit: tensorcircuit.circuit.Circuit, symbol:  
Optional[Union[Any, float]] = None, g: Optional[Any] =  
None) → tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyrylayer(circuit: tensorcircuit.circuit.Circuit, symbol:  
Optional[Union[Any, float]] = None, g: Optional[Any] =  
None) → tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyrzlayer(circuit: tensorcircuit.circuit.Circuit, symbol:  
Optional[Union[Any, float]] = None, g: Optional[Any] =  
None) → tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyswaplayer(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g:  
Any) → tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyswaplayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit,  
symbol: Any, g: Any, px: float, py: float,  
pz: float) → tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyxxlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any,  
float], g: Optional[Any] = None) →  
tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyxxlayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit,  
symbol: Union[Any, float], g:  
Optional[Any] = None, *params: float) →  
tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyxylayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any,  
float], g: Optional[Any] = None) →  
tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyxylayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit,  
symbol: Union[Any, float], g:  
Optional[Any] = None, *params: float) →  
tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyxzlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any,  
float], g: Optional[Any] = None) →  
tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyxzlayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit,  
symbol: Union[Any, float], g:  
Optional[Any] = None, *params: float) →  
tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.layers.anyyxlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any,  
float], g: Optional[Any] = None) →  
tensorcircuit.circuit.Circuit
```

`tensorcircuit.applications.layers.anyxlayer_bitflip_mc`(*circuit*: `tensorcircuit.circuit.Circuit`,
symbol: `Union[Any, float]`, *g*:
`Optional[Any] = None`, `*params: float`) →
`tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.anyyylayer`(*circuit*: `tensorcircuit.circuit.Circuit`, *symbol*: `Union[Any, float]`,
g: `Optional[Any] = None`) →
`tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.anyyylayer_bitflip_mc`(*circuit*: `tensorcircuit.circuit.Circuit`,
symbol: `Union[Any, float]`, *g*:
`Optional[Any] = None`, `*params: float`) →
`tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.anyyzlayer`(*circuit*: `tensorcircuit.circuit.Circuit`, *symbol*: `Union[Any, float]`,
g: `Optional[Any] = None`) →
`tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.anyyzlayer_bitflip_mc`(*circuit*: `tensorcircuit.circuit.Circuit`,
symbol: `Union[Any, float]`, *g*:
`Optional[Any] = None`, `*params: float`) →
`tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.anyzxlayer`(*circuit*: `tensorcircuit.circuit.Circuit`, *symbol*: `Union[Any, float]`,
g: `Optional[Any] = None`) →
`tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.anyzxlayer_bitflip_mc`(*circuit*: `tensorcircuit.circuit.Circuit`,
symbol: `Union[Any, float]`, *g*:
`Optional[Any] = None`, `*params: float`) →
`tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.anyzylayer`(*circuit*: `tensorcircuit.circuit.Circuit`, *symbol*: `Union[Any, float]`,
g: `Optional[Any] = None`) →
`tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.anyzylayer_bitflip_mc`(*circuit*: `tensorcircuit.circuit.Circuit`,
symbol: `Union[Any, float]`, *g*:
`Optional[Any] = None`, `*params: float`) →
`tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.anyzzlayer`(*circuit*: `tensorcircuit.circuit.Circuit`, *symbol*: `Union[Any, float]`,
g: `Optional[Any] = None`) →
`tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.anyzzlayer_bitflip_mc`(*circuit*: `tensorcircuit.circuit.Circuit`,
symbol: `Union[Any, float]`, *g*:
`Optional[Any] = None`, `*params: float`) →
`tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.bitfliplayer`(*ci*: `tensorcircuit.densitymatrix.DMCircuit`, *g*: `Any`, *px*:
`float`, *py*: `float`, *pz*: `float`) → `None`

`tensorcircuit.applications.layers.bitfliplayer_mc`(*ci*: `tensorcircuit.circuit.Circuit`, *g*: `Any`, *px*: `float`,
py: `float`, *pz*: `float`) → `None`

`tensorcircuit.applications.layers.cirqHlayer`(*circuit*: `cirq.circuits.circuit.Circuit`, *g*: `Any`, *symbol*: `Any`,
qubits: `Optional[Sequence[Any]] = None`) →
`cirq.circuits.circuit.Circuit`

Hlayer

`tensorcircuit.applications.layers.cirqanyrxlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyrxlayer

`tensorcircuit.applications.layers.cirqanyrylayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyrylayer

`tensorcircuit.applications.layers.cirqanyrzlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyrzlayer

`tensorcircuit.applications.layers.cirqanyswaplayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyswaplayer

`tensorcircuit.applications.layers.cirqanyxxlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyxxlayer

`tensorcircuit.applications.layers.cirqanyxylayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyxylayer

`tensorcircuit.applications.layers.cirqanyxzlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyxzlayer

`tensorcircuit.applications.layers.cirqanyyxlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyyxlayer

`tensorcircuit.applications.layers.cirqanyyylayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyyylayer

`tensorcircuit.applications.layers.cirqanyyzlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyyzlayer

`tensorcircuit.applications.layers.cirqanyzxlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyzxlayer

`tensorcircuit.applications.layers.cirqanyzylayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyzylayer

`tensorcircuit.applications.layers.cirqanyzzlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

anyzzlayer

`tensorcircuit.applications.layers.cirqcnotgate`(*circuit*: *cirq.circuits.circuit.Circuit*, *qubit1*: *cirq.devices.grid_qubit.GridQubit*, *qubit2*: *cirq.devices.grid_qubit.GridQubit*, *symbol*: *Any*) → *cirq.circuits.circuit.Circuit*

`tensorcircuit.applications.layers.cirqcnotlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

cnotlayer

`tensorcircuit.applications.layers.cirqrxlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

rxlayer

`tensorcircuit.applications.layers.cirqrylayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

rylayer

`tensorcircuit.applications.layers.cirqrzlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

rzlayer

`tensorcircuit.applications.layers.cirqswapgate`(*circuit*: *cirq.circuits.circuit.Circuit*, *qubit1*: *cirq.devices.grid_qubit.GridQubit*, *qubit2*: *cirq.devices.grid_qubit.GridQubit*, *symbol*: *Any*) → *cirq.circuits.circuit.Circuit*

swaplayer

`tensorcircuit.applications.layers.cirqxxgate`(*circuit*: *cirq.circuits.circuit.Circuit*, *qubit1*: *cirq.devices.grid_qubit.GridQubit*, *qubit2*: *cirq.devices.grid_qubit.GridQubit*, *symbol*: *Any*) → *cirq.circuits.circuit.Circuit*

xxgate

`tensorcircuit.applications.layers.cirqxxlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

xxlayer

`tensorcircuit.applications.layers.cirqxygate`(*circuit*: *cirq.circuits.circuit.Circuit*, *qubit1*: *cirq.devices.grid_qubit.GridQubit*, *qubit2*: *cirq.devices.grid_qubit.GridQubit*, *symbol*: *Any*) → *cirq.circuits.circuit.Circuit*

xygate

`tensorcircuit.applications.layers.cirqxylayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

xylayer

`tensorcircuit.applications.layers.cirqxzgate`(*circuit*: *cirq.circuits.circuit.Circuit*, *qubit1*: *cirq.devices.grid_qubit.GridQubit*, *qubit2*: *cirq.devices.grid_qubit.GridQubit*, *symbol*: *Any*) → *cirq.circuits.circuit.Circuit*

xzgate

`tensorcircuit.applications.layers.cirqxzlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

xzlayer

`tensorcircuit.applications.layers.cirqyxgate`(*circuit*: *cirq.circuits.circuit.Circuit*, *qubit1*: *cirq.devices.grid_qubit.GridQubit*, *qubit2*: *cirq.devices.grid_qubit.GridQubit*, *symbol*: *Any*) → *cirq.circuits.circuit.Circuit*

yxgate

`tensorcircuit.applications.layers.cirqyxlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

yxlayer

`tensorcircuit.applications.layers.cirqyygate`(*circuit*: *cirq.circuits.circuit.Circuit*, *qubit1*: *cirq.devices.grid_qubit.GridQubit*, *qubit2*: *cirq.devices.grid_qubit.GridQubit*, *symbol*: *Any*) → *cirq.circuits.circuit.Circuit*

yygate

`tensorcircuit.applications.layers.cirqyylayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

yylayer

`tensorcircuit.applications.layers.cirqyzgate`(*circuit*: *cirq.circuits.circuit.Circuit*, *qubit1*: *cirq.devices.grid_qubit.GridQubit*, *qubit2*: *cirq.devices.grid_qubit.GridQubit*, *symbol*: *Any*) → *cirq.circuits.circuit.Circuit*

yzgate

`tensorcircuit.applications.layers.cirqyzlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

yzlayer

`tensorcircuit.applications.layers.cirqzxgate`(*circuit*: *cirq.circuits.circuit.Circuit*, *qubit1*: *cirq.devices.grid_qubit.GridQubit*, *qubit2*: *cirq.devices.grid_qubit.GridQubit*, *symbol*: *Any*) → *cirq.circuits.circuit.Circuit*

zxgate

`tensorcircuit.applications.layers.cirqzxlayer`(*circuit*: *cirq.circuits.circuit.Circuit*, *g*: *Any*, *symbol*: *Any*, *qubits*: *Optional[Sequence[Any]]* = *None*) → *cirq.circuits.circuit.Circuit*

zxlayer

`tensorcircuit.applications.layers.cirqzygate(circuit: cirq.circuits.circuit.Circuit, qubit1: cirq.devices.grid_qubit.GridQubit, qubit2: cirq.devices.grid_qubit.GridQubit, symbol: Any) → cirq.circuits.circuit.Circuit`

zygate

`tensorcircuit.applications.layers.cirqzylayer(circuit: cirq.circuits.circuit.Circuit, g: Any, symbol: Any, qubits: Optional[Sequence[Any]] = None) → cirq.circuits.circuit.Circuit`

zylayer

`tensorcircuit.applications.layers.cirqzzgate(circuit: cirq.circuits.circuit.Circuit, qubit1: cirq.devices.grid_qubit.GridQubit, qubit2: cirq.devices.grid_qubit.GridQubit, symbol: Any) → cirq.circuits.circuit.Circuit`

zzgate

`tensorcircuit.applications.layers.cirqzzlayer(circuit: cirq.circuits.circuit.Circuit, g: Any, symbol: Any, qubits: Optional[Sequence[Any]] = None) → cirq.circuits.circuit.Circuit`

zzlayer

`tensorcircuit.applications.layers.generate_any_double_gate_layer(gates: str) → None`

`tensorcircuit.applications.layers.generate_any_double_gate_layer_bitflip_mc(gates: str) → None`

`tensorcircuit.applications.layers.generate_any_gate_layer(gate: str) → None`

$\$\$e^{\{-i\theta\sigma_i\}}\$$

Parameters `gate` (str) –

Returns

`tensorcircuit.applications.layers.generate_cirq_any_double_gate_layer(gates: str) → None`

The following function should be used to generate layers with special case. As its soundness depends on the nature of the task or problem, it doesn't always make sense.

Parameters `gates` (str) –

Returns

`tensorcircuit.applications.layers.generate_cirq_any_gate_layer(gate: str) → None`

$\$\$e^{\{-i\theta\sigma_i\}}\$$

Parameters `gate` (str) –

Returns

`tensorcircuit.applications.layers.generate_cirq_double_gate(gates: str) → None`

`tensorcircuit.applications.layers.generate_cirq_double_gate_layer(gates: str) → None`

`tensorcircuit.applications.layers.generate_cirq_gate_layer(gate: str) → None`

$\$\$e^{\{-i\theta\sigma_i\}}\$$

Parameters `gate` (str) –

Returns

`tensorcircuit.applications.layers.generate_double_gate(gates: str) → None`

`tensorcircuit.applications.layers.generate_double_gate_layer(gates: str) → None`

`tensorcircuit.applications.layers.generate_double_gate_layer_bitflip(gates: str) → None`

```
tensorcircuit.applications.layers.generate_double_gate_layer_bitflip_mc(gates: str) → None
tensorcircuit.applications.layers.generate_double_layer_block(gates: Tuple[str]) → None
tensorcircuit.applications.layers.generate_gate_layer(gate: str) → None
    $$e^{-i\theta} \sigma_z$$
```

Parameters `gate` (`str`) –

Returns

```
tensorcircuit.applications.layers.generate_qubits(g: Any) → List[Any]
tensorcircuit.applications.layers.rx_rx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.rx_ry_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.rx_rz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.rx_xx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.rx_yy_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.rx_zz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.rxlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Optional[Union[Any, float]] = None, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.ry_rx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.ry_ry_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.ry_rz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.ry_xx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.ry_yy_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.ry_zz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.rylayer(circuit: tensorcircuit.circuit.Circuit, symbol: Optional[Union[Any, float]] = None, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.rz_rx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.rz_ry_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
tensorcircuit.applications.layers.rz_rz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit
```

`tensorcircuit.applications.layers.rz_xx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.rz_yy_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.rz_zz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.rzlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Optional[Union[Any, float]] = None, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xx_rx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xx_ry_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xx_rz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xx_xx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xx_yy_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xx_zz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xxgate(circuit: tensorcircuit.circuit.Circuit, qubit1: int, qubit2: int, symbol: Union[Any, float]) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xxlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xxlayer_bitflip(circuit: tensorcircuit.densitymatrix.DMCircuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.densitymatrix.DMCircuit`

`tensorcircuit.applications.layers.xxlayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xygate(circuit: tensorcircuit.circuit.Circuit, qubit1: int, qubit2: int, symbol: Union[Any, float]) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xylayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xylayer_bitflip(circuit: tensorcircuit.densitymatrix.DMCircuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.densitymatrix.DMCircuit`

`tensorcircuit.applications.layers.xylayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.xzgate(circuit: tensorcircuit.circuit.Circuit, qubit1: int, qubit2: int, symbol: Union[Any, float]) → tensorcircuit.circuit.Circuit`

```
tensorcircuit.applications.layers.xzlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any,  
float], g: Optional[Any] = None) →  
    tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.xzlayer_bitflip(circuit: tensorcircuit.densitymatrix.DMCircuit,  
    symbol: Union[Any, float], g: Any, *params: float)  
    → tensorcircuit.densitymatrix.DMCircuit  
  
tensorcircuit.applications.layers.xzlayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit, symbol:  
    Union[Any, float], g: Any, *params: float) →  
    tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yxgate(circuit: tensorcircuit.circuit.Circuit, qubit1: int, qubit2: int,  
    symbol: Union[Any, float]) → tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yxlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any,  
float], g: Optional[Any] = None) →  
    tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yxlayer_bitflip(circuit: tensorcircuit.densitymatrix.DMCircuit,  
    symbol: Union[Any, float], g: Any, *params: float)  
    → tensorcircuit.densitymatrix.DMCircuit  
  
tensorcircuit.applications.layers.yxlayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit, symbol:  
    Union[Any, float], g: Any, *params: float) →  
    tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yy_rx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g:  
    Optional[Any] = None) → tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yy_ry_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g:  
    Optional[Any] = None) → tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yy_rz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g:  
    Optional[Any] = None) → tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yy_xx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g:  
    Optional[Any] = None) → tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yy_yy_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g:  
    Optional[Any] = None) → tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yy_zz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g:  
    Optional[Any] = None) → tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yygate(circuit: tensorcircuit.circuit.Circuit, qubit1: int, qubit2: int,  
    symbol: Union[Any, float]) → tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yylayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any,  
float], g: Optional[Any] = None) →  
    tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yylayer_bitflip(circuit: tensorcircuit.densitymatrix.DMCircuit,  
    symbol: Union[Any, float], g: Any, *params: float)  
    → tensorcircuit.densitymatrix.DMCircuit  
  
tensorcircuit.applications.layers.yylayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit, symbol:  
    Union[Any, float], g: Any, *params: float) →  
    tensorcircuit.circuit.Circuit  
  
tensorcircuit.applications.layers.yzgate(circuit: tensorcircuit.circuit.Circuit, qubit1: int, qubit2: int,  
    symbol: Union[Any, float]) → tensorcircuit.circuit.Circuit
```

`tensorcircuit.applications.layers.yzlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.yzlayer_bitflip(circuit: tensorcircuit.densitymatrix.DMCircuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.densitymatrix.DMCircuit`

`tensorcircuit.applications.layers.yzlayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zxgate(circuit: tensorcircuit.circuit.Circuit, qubit1: int, qubit2: int, symbol: Union[Any, float]) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zxlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zxlayer_bitflip(circuit: tensorcircuit.densitymatrix.DMCircuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.densitymatrix.DMCircuit`

`tensorcircuit.applications.layers.zxlayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zygate(circuit: tensorcircuit.circuit.Circuit, qubit1: int, qubit2: int, symbol: Union[Any, float]) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zylayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zylayer_bitflip(circuit: tensorcircuit.densitymatrix.DMCircuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.densitymatrix.DMCircuit`

`tensorcircuit.applications.layers.zylayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zz_rx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zz_ry_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zz_rz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zz_xx_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zz_yy_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zz_zz_block(circuit: tensorcircuit.circuit.Circuit, symbol: Any, g: Optional[Any] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.layers.zzgate(circuit: tensorcircuit.circuit.Circuit, qubit1: int, qubit2: int, symbol: Union[Any, float]) → tensorcircuit.circuit.Circuit`

```
tensorcircuit.applications.layers.zzlayer(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Optional[Any] = None) → tensorcircuit.circuit.Circuit

tensorcircuit.applications.layers.zzlayer_bitflip(circuit: tensorcircuit.densitymatrix.DMCircuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.densitymatrix.DMCircuit

tensorcircuit.applications.layers.zzlayer_bitflip_mc(circuit: tensorcircuit.circuit.Circuit, symbol: Union[Any, float], g: Any, *params: float) → tensorcircuit.circuit.Circuit
```

tensorcircuit.applications.utils

A collection of useful function snippets that irrelevant with the main modules or await for further refactor

```
class tensorcircuit.applications.utils.FakeModule
    Bases: object
```

```
tensorcircuit.applications.utils.Heisenberg1Denergy(L: int, Pauli: bool = True, maxiters: int = 1000) → float
```

```
tensorcircuit.applications.utils.TFIM1Denergy(L: int, Jzz: float = 1.0, Jx: float = 1.0, Pauli: bool = True) → float
```

```
tensorcircuit.applications.utils.amplitude_encoding(fig: Any, qubits: int, index: Optional[Sequence[int]] = None, index_func: Optional[Callable[[int, int], Sequence[int]]] = None) → Any
```

```
tensorcircuit.applications.utils.color_svg(circuit: cirq.circuits.circuit.Circuit, *coords: Tuple[int, int]) → Any
```

color cirq circuit SVG for given gates, a small tool to hack the cirq SVG

Parameters

- **circuit** –
- **coords** – integer coordinate which gate is colored

Returns

```
tensorcircuit.applications.utils.generate_random_circuit(inputs: Any, nqubits: int = 10, epochs: int = 3, layouts: Optional[Any] = None) → tensorcircuit.circuit.Circuit
```

```
tensorcircuit.applications.utils.mnist_amplitude_data(a: int, b: int, binarize: bool = False, index: Optional[Sequence[int]] = None, index_func: Optional[Callable[[int, int], Sequence[int]]] = None, loader: Optional[Any] = None, threshold: float = 0.4) → Any
```

```
tensorcircuit.applications.utils.mnist_generator(x_train: Any, y_train: Any, batch: int = 1, random: bool = True) → Iterator[Any]
```

```
tensorcircuit.applications.utils.naive_qml_vag(gdata: Any, nnp: Any, preset: Sequence[int], nqubits: int = 10, epochs: int = 3, target: int = 0) → Tuple[Any, Any]
```

```
tensorcircuit.applications.utils.recursive_index(x: int, y: int) → Sequence[int]
```

`tensorcircuit.applications.utils.repr2array(inputs: str) → Any`
 transform repr form of an array to real numpy array

Parameters `inputs` –

Returns

`tensorcircuit.applications.utils.train_qml_vag(gdata: Any, nnp: Any, preset: Optional[Sequence[int]] = None, nqubits: int = 10, epochs: int = 3, batch: int = 64, validation: bool = False) → Any`

`tensorcircuit.applications.utils.validate_qml_vag(gdata: Any, nnp: Any, preset: Optional[Sequence[int]] = None, nqubits: int = 10, epochs: int = 3, batch: int = 64) → Any`

tensorcircuit.applications.vags

DQAS application kernels as vag functions

`tensorcircuit.applications.vags.GHZ_vag(gdata: Any, nnp: Any, preset: Sequence[int], verbose: bool = False, n: int = 3) → Tuple[Any, Any]`

`tensorcircuit.applications.vags.GHZ_vag_tfq(gdata: Any, nnp: Any, preset: Sequence[int], verbose: bool = False, index: Tuple[int, int, int, int, int, int, int] = (1, 1, 0, 0, 1, 0, 0)) → Tuple[Any, Any]`

`tensorcircuit.applications.vags.ave_func(state: Any, g: Any, *fs: Union[Tuple[Callable[[float], float], Callable[[Any, Any]], Tuple[Callable[[float], float], Callable[[Any], Any], Callable[[Any, Any], Any]]]) → Sequence[Any]`

Parameters

- `state` – 1D array for full wavefunction, the basis is in lexical order
- `g` – nx.Graph
- `fs` – transformation functions before averaged

Returns

`tensorcircuit.applications.vags.compose_tc_circuit_with_multiple_pools(theta: Any, preset: Sequence[int], g: Any, pool_choice: Sequence[int], cset: Optional[Sequence[Any]] = None, measure_func: Optional[Callable[[tensorcircuit.densitymatrix.Any], Any]] = None) → tensorcircuit.circuit.Circuit`

`tensorcircuit.applications.vags.correlation(m: Any, rho: Any) → Any`

`tensorcircuit.applications.vags.cvar(r: List[float], p: Any, percent: float = 0.2) → Sequence[float]`
 as f3

Parameters

- **r** –
- **p** –
- **percent** –

Returns

`tensorcircuit.applications.vags.double_qubits_initial()` → `Iterator[Sequence[Any]]`

`tensorcircuit.applications.vags.double_state(h: Any, beta: float = 1)` → `Any`

`tensorcircuit.applications.vags.energy(i: int, n: int, g: Any)` → `float`

maxcut energy for n qubit wavefunction i-th basis

Parameters

- **i** – ranged from 0 to 2^{**n-1}
- **n** – number of qubits
- **g** – `nx.Graph`

Returns

`tensorcircuit.applications.vags.entanglement_entropy(state: Any)` → `Any`

deprecated as non tf and non flexible, use the combination of `reduced_density_matrix` and `entropy` instead.

`tensorcircuit.applications.vags.entropy(rho: Any, eps: float = 1e-12)` → `Any`

deprecated, current version in `tc.quantum`

`tensorcircuit.applications.vags.evaluate_vag(params: Any, preset: Sequence[int], g: Any, lbd: float = 0.0, overlap_threshold: float = 0.0)` → `Tuple[Any, Any, Any]`

value and gradient, currently only tensorflow backend is supported jax and numpy seems to be slow in circuit simulation anyhow. *deprecated*

Parameters

- **params** –
- **preset** –
- **g** –
- **lbd** – if `lbd=0`, take energy as objective
- **overlap_threshold** – if as default 0, overlap will not compute in the process

Returns

`tensorcircuit.applications.vags.exp_forward(theta: Any, preset: Sequence[int], g: Any, *fs: Tuple[Callable[[float], float], Callable[[Any], Any]])` → `Sequence[Any]`

`tensorcircuit.applications.vags.fidelity(rho: Any, rho0: Any)` → `Any`

`tensorcircuit.applications.vags.free_energy(rho: Any, h: Any, beta: float = 1, eps: float = 1e-12)` → `Any`

`tensorcircuit.applications.vags.gapfilling(circuit: cirq.circuits.circuit.Circuit, placeholder: Sequence[Any])` → `cirq.circuits.circuit.Circuit`

Fill single qubit gates according to placeholder on circuit

Parameters

- **circuit** –

- **placeholder** –

Returns

```
tensorcircuit.applications.vags.gatewise_vqe_vag(gdata: Any, nnp: Any, preset: Sequence[int],
                                                pool_choice: Sequence[int], measure_func: Optional[Callable[[Union[tensorcircuit.circuit.Circuit,
                                                tensorcircuit.densitymatrix.DMCircuit], Any], Any]] = None) → Tuple[Any, Any]
```

```
tensorcircuit.applications.vags.gibbs_state(h: Any, beta: float = 1) → Any
```

```
tensorcircuit.applications.vags.heisenberg_measurements(g: Any, hxx: float = 1.0, hy: float = 1.0,
                                                       hzz: float = 1.0, one: bool = True) → Any
```

Hamiltonian measurements for Heisenberg model on graph lattice g

Parameters

- **g** –
- **hxx** –
- **hy** –
- **hzz** –
- **one** –

Returns

```
tensorcircuit.applications.vags.heisenberg_measurements_tc(c: Union[tensorcircuit.circuit.Circuit,
                                                               tensorcircuit.densitymatrix.DMCircuit], g: Any,
                                                               hzz: float = 1.0, hxx: float = 1.0, hy: float = 1.0,
                                                               hz: float = 0.0, hx: float = 0.0, hy: float = 0.0, reuse: bool = True) → Any
```

```
tensorcircuit.applications.vags.maxcut_measurements_tc(c: Union[tensorcircuit.circuit.Circuit,
                                                               tensorcircuit.densitymatrix.DMCircuit], g: Any) → Any
```

```
tensorcircuit.applications.vags.noise_forward(theta: Any, preset: Sequence[int], g: Any,
                                               measure_func: Callable[[tensorcircuit.densitymatrix.DMCircuit, Any], Any], is_mc: bool = False) → Any
```

```
tensorcircuit.applications.vags.noisyfy(circuit: cirq.circuits.circuit.Circuit, error_model: str = 'bit_flip',
                                         p_idle: float = 0.2, p_sep: float = 0.02) → cirq.circuits.circuit.Circuit
```

```
tensorcircuit.applications.vags.q(i: int) → cirq.devices.line_qubit.LineQubit
short cut for cirq.LineQubit(i)
```

Parameters i –

Returns

```
tensorcircuit.applications.vags.qaoa_block_vag(gdata: Any, nnp: Any, preset: Sequence[int], f: Tuple[Callable[[float], float], Callable[[Any], Any]]) → Tuple[Any, Any]
```

QAOA block encoding kernel, support 2 params in one op

Parameters

- **gdata** –
- **nnp** –
- **preset** –
- **f** –

Returns

```
tensorcircuit.applications.vags.qaoa_block_vag_energy(gdata: Any, nnp: Any, preset: Sequence[int],  
*, f: Tuple[Callable[[float], float],  
Callable[[Any], Any]] = (<function  
_identity>, <function _neg>)) → Tuple[Any,  
Any]
```

QAOA block encoding kernel, support 2 params in one op

Parameters

- **gdata** –
- **nnp** –
- **preset** –
- **f** –

Returns

```
tensorcircuit.applications.vags.qaoa_noise_vag(gdata: Any, nnp: Any, preset: Sequence[int],  
measure_func: Op-  
tional[Callable[[tensorcircuit.densitymatrix.DMCircuit,  
Any], Any]] = None, forward_func:  
Optional[Callable[[Any, Sequence[int]], Any,  
Callable[[tensorcircuit.densitymatrix.DMCircuit,  
Any], Any]], Any]] = None, **kws: Any) →  
Tuple[Any, Any]
```

```
tensorcircuit.applications.vags.qaoa_train(preset: Sequence[int], g: Union[Any, Iterator[Any]], *,  
epochs: int = 100, batch: int = 1, initial_param:  
Optional[Any] = None, opt: Optional[Any] = None, lbd:  
float = 0.0, overlap_threshold: float = 0.0, verbose: bool =  
True) → Tuple[Any, Sequence[float], Sequence[float],  
Sequence[float]]
```

training QAOA with only optimizing circuit parameters, can be well replaced with more general function *DQAS_search*

Parameters

- **preset** –
- **g** –
- **epochs** –
- **batch** –
- **initial_param** –
- **opt** –
- **lbd** –
- **overlap_threshold** –

- **verbose** –

Returns

```
tensorcircuit.applications.vags.qaoa_vag(gdata: Any, nnp: Any, preset: Sequence[int], f:
    Optional[Tuple[Callable[[float], float], Callable[[Any], Any]]]
    = None, forward_func: Optional[Callable[[Any,
        Sequence[int], Any, Tuple[Callable[[float], float],
        Callable[[Any], Any]]], Tuple[Any, Any]]] = None,
    verbose_fs: Optional[Sequence[Tuple[Callable[[float], float],
        Callable[[Any], Any]]]] = None) → Tuple[Any, Any]
```

```
tensorcircuit.applications.vags.qaoa_vag_energy(gdata: Any, nnp: Any, preset: Sequence[int], *, f:
    Optional[Tuple[Callable[[float], float],
    Callable[[Any], Any]]] = (<function _identity>,
    <function _neg>), forward_func:
    Optional[Callable[[Any, Sequence[int], Any,
        Tuple[Callable[[float], float], Callable[[Any], Any]]], Tuple[Any, Any]]] = None, verbose_fs:
    Optional[Sequence[Tuple[Callable[[float], float],
        Callable[[Any], Any]]]] = None) → Tuple[Any, Any]
```

`tensorcircuit.applications.vags.qft_circuit(n: int)` → `cirq.circuits.circuit.Circuit`

```
tensorcircuit.applications.vags.qft_qem_vag(gdata: Any, nnp: Any, preset: Sequence[int], n: int = 3,
    p_idle: float = 0.2, p_sep: float = 0.02) → Tuple[Any, Any]
```

```
tensorcircuit.applications.vags.quantum_mp_qaoa_vag(gdata: Any, nnp: Any, preset: Sequence[int],
    measurements_func: Optional[Callable[..., Any]] = None, **kws: Any) → Tuple[Any, Any]
```

multi parameter for one layer

Parameters

- **gdata** –
- **nnp** –
- **preset** –
- **measurements_func** –
- **kws** – kw arguments for `measurements_func`

Returns loss function, gradient of `nnp`

```
tensorcircuit.applications.vags.quantum_qaoa_vag(gdata: Any, nnp: Any, preset: Sequence[int],
    measurements_func: Optional[Callable[..., Any]] = None, **kws: Any) → Tuple[Any, Any]
```

tensorflow quantum backend compare to `qaoa_vag` which is tensorcircuit backend

Parameters

- **gdata** –
- **nnp** –
- **preset** –
- **measurements_func** –
- **kws** – kw arguments for `measurements_func`

Returns

```
tensorcircuit.applications.vags.reduced_density_matrix(state: Any, freedom: int, cut: Union[int, List[int]], p: Optional[Any] = None) → Any
```

deprecated, current version in tc.quantum

```
tensorcircuit.applications.vags.renyi_entropy(rho: Any, k: int = 2, eps: float = 1e-12) → Any
```

```
tensorcircuit.applications.vags.renyi_free_energy(rho: Any, h: Any, beta: float = 1) → Any
```

```
tensorcircuit.applications.vags.taylorlnm(x: Any, k: int) → Any
```

```
tensorcircuit.applications.vags.tfim_measurements(g: Any, hzz: float = 1, hx: float = 0, hz: float = 0, one: bool = True) → Any
```

Hamiltonian for tfim on lattice defined by graph g

Parameters

- **g** –
- **hzz** –
- **hx** –
- **hz** –
- **one** –

Returns cirq.PauliSum as operators for tfq expectation layer

```
tensorcircuit.applications.vags.tfim_measurements_tc(c: Union[tensorcircuit.circuit.Circuit, tensorcircuit.densitymatrix.DMCircuit], g: Any, hzz: float = 1.0, hx: float = 0.0, hz: float = 0.0) → Any
```

```
tensorcircuit.applications.vags.trace_distance(rho: Any, rho0: Any, eps: float = 1e-12) → Any
```

```
tensorcircuit.applications.vags.truncated_free_energy(rho: Any, h: Any, beta: float = 1, k: int = 2, eps: float = 1e-12) → Any
```

```
tensorcircuit.applications.vags.unitary_design(n: int, l: int = 3) → cirq.circuits.circuit.Circuit  
generate random wavefunction from approximately Haar measure, reference: https://doi.org/10.1063/1.4983266
```

Parameters

- **n** – number of qubits
- **l** – repetition of the blocks

Returns

```
tensorcircuit.applications.vags.unitary_design_block(circuit: cirq.circuits.circuit.Circuit, n: int) → cirq.circuits.circuit.Circuit
```

random Haar measure approximation

Parameters

- **circuit** – cirq.Circuit, empty circuit
- **n** – # of qubit

Returns

tensorcircuit.applications.van

One-hot variational autoregressive models for multiple categorical choices beyond binary

```
class tensorcircuit.applications.van.MADE(*args, **kwargs)
    Bases: keras.engine.training.Model

    __init__(input_space: int, output_space: int, hidden_space: int, spin_channel: int, depth: int, evenly: bool
         = True, dtype: Optional[tensorflow.python.framework.dtypes.DType] = None, activation:
         Optional[Any] = None, nonmerge: bool = True, probamp:
         Optional[tensorflow.python.framework.ops.Tensor] = None)
```

property activity_regularizer

Optional regularizer function for the output of this layer.

add_loss(losses, **kwargs)

Add loss tensor(s), potentially dependent on layer inputs.

Some losses (for instance, activity regularization losses) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs a and b , some entries in `layer.losses` may be dependent on a and some on b . This method automatically keeps track of dependencies.

This method can be used inside a subclassed layer or model's `call` function, in which case `losses` should be a Tensor or list of Tensors.

Example:

```
'''python class MyLayer(tf.keras.layers.Layer):
    def call(self, inputs): self.add_loss(tf.abs(tf.reduce_mean(inputs))) return inputs
'''
```

This method can also be called directly on a Functional Model during construction. In this case, any loss Tensors passed to this Model must be symbolic and be able to be traced back to the model's `Input`'s. *These losses become part of the model's topology and are tracked in 'get_config'.*

Example:

```
python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.
Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.
Model(inputs, outputs) # Activity regularization. model.add_loss(tf.abs(tf.
reduce_mean(x)))`
```

If this is not the case for your loss (if, for example, your loss references a *Variable* of one of the model's layers), you can wrap your loss in a zero-argument lambda. These losses are not tracked as part of the model's topology since they can't be serialized.

Example:

```
python inputs = tf.keras.Input(shape=(10,)) d = tf.keras.layers.Dense(10) x =
d(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs,
outputs) # Weight regularization. model.add_loss(lambda: tf.reduce_mean(d.
kernel))`
```

Parameters

- **losses** – Loss tensor, or list/tuple of tensors. Rather than tensors, losses may also be zero-argument callables which create a loss tensor.
- ****kwargs** – Used for backwards compatibility only.

add_metric(*value*, *name*=None, ***kwargs*)

Adds metric tensor to the layer.

This method can be used inside the *call()* method of a subclassed layer or model.

```
```python class MyMetricLayer(tf.keras.layers.Layer):
```

```
 def __init__(self): super(MyMetricLayer, self).__init__(name='my_metric_layer') self.mean = tf.keras.metrics.Mean(name='metric_1')

 def call(self, inputs): self.add_metric(self.mean(inputs)) self.add_metric(tf.reduce_sum(inputs), name='metric_2') return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any tensor passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These metrics become part of the model's topology and are tracked when you save the model via 'save()'*.

```
```python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) model.add_metric(tf.reduce_sum(x), name='metric_1')
```

Note: Calling *add_metric()* with the result of a metric object on a Functional Model, as shown in the example below, is not supported. This is because we cannot trace the metric result tensor back to the model's inputs.

```
```python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) model.add_metric(tf.keras.metrics.Mean()(x), name='metric_1')``
```

**Parameters**

- **value** – Metric tensor.
- **name** – String metric name.
- **\*\*kwargs** – Additional keyword arguments for backward compatibility. Accepted values:  
*aggregation* - When the *value* tensor provided is not the result of calling a *keras.Metric* instance, it will be aggregated by default using a *keras.Metric.Mean*.

**add\_update**(*updates*)

Add update op(s), potentially dependent on layer inputs.

Weight updates (for instance, the updates of the moving mean and variance in a BatchNormalization layer) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs *a* and *b*, some entries in *layer.updates* may be dependent on *a* and some on *b*. This method automatically keeps track of dependencies.

This call is ignored when eager execution is enabled (in that case, variable updates are run on the fly and thus do not need to be tracked for later execution).

**Parameters** **updates** – Update op, or list/tuple of update ops, or zero-arg callable that returns an update op. A zero-arg callable should be passed in order to disable running the updates by setting *trainable=False* on this Layer, when executing in Eager mode.

**add\_variable**(\**args*, \*\**kwargs*)

Deprecated, do NOT use! Alias for *add\_weight*.

---

**add\_weight**(*name=None*, *shape=None*, *dtype=None*, *initializer=None*, *regularizer=None*, *trainable=None*, *constraint=None*, *use\_resource=None*, *synchronization=VariableSynchronization.AUTO*, *aggregation=VariableAggregationV2.NONE*, *\*\*kwargs*)

Adds a new variable to the layer.

#### Parameters

- **name** – Variable name.
- **shape** – Variable shape. Defaults to scalar if unspecified.
- **dtype** – The type of the variable. Defaults to *self.dtype*.
- **initializer** – Initializer instance (callable).
- **regularizer** – Regularizer instance (callable).
- **trainable** – Boolean, whether the variable should be part of the layer’s “trainable\_variables” (e.g. variables, biases) or “non\_trainable\_variables” (e.g. BatchNorm mean and variance). Note that *trainable* cannot be *True* if *synchronization* is set to *ON\_READ*.
- **constraint** – Constraint instance (callable).
- **use\_resource** – Whether to use a *ResourceVariable* or not. See [this guide]([https://www.tensorflow.org/guide/migrate/tf1\\_vs\\_tf2#resourcevariables\\_instead\\_of\\_referencevariables](https://www.tensorflow.org/guide/migrate/tf1_vs_tf2#resourcevariables_instead_of_referencevariables)) for more information.
- **synchronization** – Indicates when a distributed a variable will be aggregated. Accepted values are constants defined in the class *tf.VariableSynchronization*. By default the synchronization is set to *AUTO* and the current *DistributionStrategy* chooses when to synchronize. If *synchronization* is set to *ON\_READ*, *trainable* must not be set to *True*.
- **aggregation** – Indicates how a distributed variable will be aggregated. Accepted values are constants defined in the class *tf.VariableAggregation*.
- **\*\*kwargs** – Additional keyword arguments. Accepted values are *getter*, *collections*, *experimental\_autocast* and *caching\_device*.

**Returns** The variable created.

**Raises** **ValueError** – When giving unsupported *dtype* and no initializer or when *trainable* has been set to *True* with *synchronization* set as *ON\_READ*.

**build**(*input\_shape*)

Builds the model based on input shapes received.

This is to be used for subclassed models, which do not know at instantiation time what their inputs look like.

This method only exists for users who want to call *model.build()* in a standalone way (as a substitute for calling the model on real data to build it). It will never be called by the framework (and thus it will never throw unexpected errors in an unrelated workflow).

**Parameters** **input\_shape** – Single tuple, *TensorShape* instance, or list/dict of shapes, where shapes are tuples, integers, or *TensorShape* instances.

**Raises**

- **ValueError** –

1. In case of invalid user-provided data (not of type tuple, list, *TensorShape*, or dict). 2. If the model requires call arguments that are agnostic to the input shapes (positional or

keyword arg in call signature). 3. If not all layers were properly built. 4. If float type inputs are not supported within the layers.

- In each of these cases, the user should build their model by calling  
–  
• it on real tensor data. –

`call(inputs: tensorflow.python.framework.ops.Tensor) → tensorflow.python.framework.ops.Tensor`

Calls the model on new inputs and returns the outputs as tensors.

In this case `call()` just reapplyes all ops in the graph to the new inputs (e.g. build a new computational graph from the provided inputs).

Note: This method should not be called directly. It is only meant to be overridden when subclassing `tf.keras.Model`. To call a model on an input, always use the `__call__()` method, i.e. `model(inputs)`, which relies on the underlying `call()` method.

#### Parameters

- **inputs** – Input tensor, or dict/list/tuple of input tensors.
- **training** – Boolean or boolean scalar tensor, indicating whether to run the *Network* in training mode or inference mode.
- **mask** – A mask or list of masks. A mask can be either a boolean tensor or None (no mask). For more details, check the guide [here]([https://www.tensorflow.org/guide/keras/masking\\_and\\_padding](https://www.tensorflow.org/guide/keras/masking_and_padding)).

**Returns** A tensor if there is a single output, or a list of tensors if there are more than one outputs.

`compile(optimizer='rmsprop', loss=None, metrics=None, loss_weights=None, weighted_metrics=None, run_eagerly=None, steps_per_execution=None, jit_compile=None, **kwargs)`

Configures the model for training.

Example:

```
```python
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
               loss=tf.keras.losses.BinaryCrossentropy(),
               metrics=[tf.keras.metrics.BinaryAccuracy(),
                        tf.keras.metrics.FalseNegatives()])
````
```

#### Parameters

- **optimizer** – String (name of optimizer) or optimizer instance. See `tf.keras.optimizers`.
- **loss** – Loss function. May be a string (name of loss function), or a `tf.keras.losses.Loss` instance. See `tf.keras.losses`. A loss function is any callable with the signature `loss = fn(y_true, y_pred)`, where `y_true` are the ground truth values, and `y_pred` are the model's predictions. `y_true` should have shape `(batch_size, d0, .. dN)` (except in the case of sparse loss functions such as sparse categorical crossentropy which expects integer arrays of shape `(batch_size, d0, .. dN-1)`). `y_pred` should have shape `(batch_size, d0, .. dN)`. The loss function should return a float tensor. If a custom `Loss` instance is used and reduction is set to `None`, return value has shape `(batch_size, d0, .. dN-1)` i.e. per-sample or per-timestep loss values; otherwise, it is a scalar. If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses, unless `loss_weights` is specified.
- **metrics** – List of metrics to be evaluated by the model during training and testing. Each of this can be a string (name of a built-in function), function or a `tf.keras.metrics.Metric`

instance. See `tf.keras.metrics`. Typically you will use `metrics=['accuracy']`. A function is any callable with the signature `result = fn(y_true, y_pred)`. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as `metrics={'output_a': 'accuracy', 'output_b': ['accuracy', 'mse']}`. You can also pass a list to specify a metric or a list of metrics for each output, such as `metrics=[[['accuracy'], ['accuracy', 'mse']]]` or `metrics=['accuracy', ['accuracy', 'mse']]`. When you pass the strings ‘accuracy’ or ‘acc’, we convert this to one of `tf.keras.metrics.BinaryAccuracy`, `tf.keras.metrics.CategoricalAccuracy`, `tf.keras.metrics.SparseCategoricalAccuracy` based on the shapes of the targets and of the model output. We do a similar conversion for the strings ‘crossentropy’ and ‘ce’ as well. The metrics passed here are evaluated without sample weighting; if you would like sample weighting to apply, you can specify your metrics via the `weighted_metrics` argument instead.

- **loss\_weights** – Optional list or dictionary specifying scalar coefficients (Python floats) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the *weighted sum* of all individual losses, weighted by the `loss_weights` coefficients. If a list, it is expected to have a 1:1 mapping to the model’s outputs. If a dict, it is expected to map output names (strings) to scalar coefficients.
- **weighted\_metrics** – List of metrics to be evaluated and weighted by `sample_weight` or `class_weight` during training and testing.
- **run\_eagerly** – Bool. Defaults to `False`. If `True`, this *Model*’s logic will not be wrapped in a `tf.function`. Recommended to leave this as `None` unless your *Model* cannot be run inside a `tf.function`. `run_eagerly=True` is not supported when using `tf.distribute.experimental.ParameterServerStrategy`.
- **steps\_per\_execution** – Int. Defaults to 1. The number of batches to run during each `tf.function` call. Running multiple batches inside a single `tf.function` call can greatly improve performance on TPUs or small models with a large Python overhead. At most, one full epoch will be run each execution. If a number larger than the size of the epoch is passed, the execution will be truncated to the size of the epoch. Note that if `steps_per_execution` is set to  $N$ , `Callback.on_batch_begin` and `Callback.on_batch_end` methods will only be called every  $N$  batches (i.e. before/after each `tf.function` execution).
- **jit\_compile** – If `True`, compile the model training step with XLA. [XLA](<https://www.tensorflow.org/xla>) is an optimizing compiler for machine learning. `jit_compile` is not enabled for by default. This option cannot be enabled with `run_eagerly=True`. Note that `jit_compile=True` may not necessarily work for all models. For more information on supported operations please refer to the [XLA documentation](<https://www.tensorflow.org/xla>). Also refer to [known XLA issues]([https://www.tensorflow.org/xla/known\\_issues](https://www.tensorflow.org/xla/known_issues)) for more details.
- **\*\*kwargs** – Arguments supported for backwards compatibility only.

#### **property compute\_dtype**

The dtype of the layer’s computations.

This is equivalent to `Layer.dtype_policy.compute_dtype`. Unless mixed precision is used, this is the same as `Layer.dtype`, the dtype of the weights.

Layers automatically cast their inputs to the compute dtype, which causes computations and the output to be in the compute dtype as well. This is done by the base Layer class in `Layer.__call__`, so you do not have to insert these casts if implementing your own layer.

Layers often perform certain internal computations in higher precision when `compute_dtype` is `float16` or `bfloat16` for numeric stability. The output will still typically be `float16` or `bfloat16` in such cases.

**Returns** The layer’s compute dtype.

**compute\_loss**(*x=None*, *y=None*, *y\_pred=None*, *sample\_weight=None*)

Compute the total loss, validate it, and return it.

Subclasses can optionally override this method to provide custom loss computation logic.

Example: ```` python class MyModel(tf.keras.Model):`

```
def __init__(self, *args, **kwargs): super(MyModel, self).__init__(*args, **kwargs)
 self.loss_tracker = tf.keras.metrics.Mean(name='loss')

def compute_loss(self, x, y, y_pred, sample_weight): loss = tf.reduce_mean(tf.math.squared_difference(y_pred,
 y)) loss += tf.add_n(self.losses) self.loss_tracker.update_state(loss) return loss

def reset_metrics(self): self.loss_tracker.reset_states()

@property def metrics(self):
 return [self.loss_tracker]

tensors = tf.random.uniform((10, 10)), tf.random.uniform((10,)) dataset =
tf.data.Dataset.from_tensor_slices(tensors).repeat().batch(1)

inputs = tf.keras.layers.Input(shape=(10,), name='my_input') outputs = tf.keras.layers.Dense(10)(inputs)
model = MyModel(inputs, outputs) model.add_loss(tf.reduce_sum(outputs))

optimizer = tf.keras.optimizers.SGD() model.compile(optimizer, loss='mse', steps_per_execution=10)
model.fit(dataset, epochs=2, steps_per_epoch=10) print('My custom loss: ', model.loss_tracker.result().numpy())````
```

**Parameters**

- **x** – Input data.
- **y** – Target data.
- **y\_pred** – Predictions returned by the model (output of *model(x)*)
- **sample\_weight** – Sample weights for weighting the loss function.

**Returns** The total loss as a *tf.Tensor*, or *None* if no loss results (which is the case when called by *Model.test\_step*).

**compute\_mask**(*inputs*, *mask=None*)

Computes an output mask tensor.

**Parameters**

- **inputs** – Tensor or list of tensors.
- **mask** – Tensor or list of tensors.

**Returns**

**None or a tensor (or list of tensors**, one per output tensor of the layer).

**compute\_metrics**(*x*, *y*, *y\_pred*, *sample\_weight*)

Update metric states and collect all metrics to be returned.

Subclasses can optionally override this method to provide custom metric updating and collection logic.

Example: ```` python class MyModel(tf.keras.Sequential):`

```
def compute_metrics(self, x, y, y_pred, sample_weight):
 # This super call updates self.compiled_metrics and returns # results for all metrics listed
 # in self.metrics. metric_results = super(MyModel, self).compute_metrics(
 x, y, y_pred, sample_weight)
```

```

Note that self.custom_metric is not listed in self.metrics.
self.custom_metric.update_state(x, y, y_pred, sample_weight)
metric_results['custom_metric_name'] = self.custom_metric.result()
return metric_results
```

```

Parameters

- **x** – Input data.
- **y** – Target data.
- **y_pred** – Predictions returned by the model (output of *model.call(x)*)
- **sample_weight** – Sample weights for weighting the loss function.

Returns A *dict* containing values that will be passed to *tf.keras.callbacks.CallbackList.on_train_batch_end()*. Typically, the values of the metrics listed in *self.metrics* are returned. Example: {'loss': 0.2, 'accuracy': 0.7}.

compute_output_shape(*input_shape*)

Computes the output shape of the layer.

This method will cause the layer’s state to be built, if that has not happened before. This requires that the layer will later be used with inputs that match the input shape provided here.

Parameters **input_shape** – Shape tuple (tuple of integers) or *tf.TensorShape*, or structure of shape tuples / *tf.TensorShape* instances (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns A *tf.TensorShape* instance or structure of *tf.TensorShape* instances.

compute_output_signature(*input_signature*)

Compute the output tensor signature of the layer based on the inputs.

Unlike a *TensorShape* object, a *TensorSpec* object contains both shape and dtype information for a tensor. This method allows layers to provide output dtype information if it is different from the input dtype. For any layer that doesn’t implement this function, the framework will fall back to use *compute_output_shape*, and will assume that the output dtype matches the input dtype.

Parameters **input_signature** – Single *TensorSpec* or nested structure of *TensorSpec* objects, describing a candidate input for the layer.

Returns

Single *TensorSpec* or nested structure of *TensorSpec* objects, describing how the layer would transform the provided input.

Raises **TypeError** – If *input_signature* contains a non-*TensorSpec* object.

count_params()

Count the total number of scalars composing the weights.

Returns An integer count.

Raises **ValueError** – if the layer isn’t yet built (in which case its weights aren’t yet defined).

property distribute_reduction_method

The method employed to reduce per-replica values during training.

Unless specified, the value “auto” will be assumed, indicating that the reduction strategy should be chosen based on the current running environment. See *reduce_per_replica* function for more details.

property distribute_strategy

The *tf.distribute.Strategy* this model was created under.

property dtype

The dtype of the layer weights.

This is equivalent to `Layer.dtype_policy.variable_dtype`. Unless mixed precision is used, this is the same as `Layer.compute_dtype`, the dtype of the layer's computations.

property dtype_policy

The dtype policy associated with this layer.

This is an instance of a `tf.keras.mixed_precision.Policy`.

property dynamic

Whether the layer is dynamic (eager-only); set in the constructor.

evaluate(*x=None*, *y=None*, *batch_size=None*, *verbose='auto'*, *sample_weight=None*, *steps=None*,
callbacks=None, *max_queue_size=10*, *workers=1*, *use_multiprocessing=False*, *return_dict=False*,
***kwargs*)

Returns the loss value & metrics values for the model in test mode.

Computation is done in batches (see the `batch_size` arg.)

Parameters

- **x** – Input data. It could be:
 - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
 - A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
 - A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
 - A `tf.data` dataset. Should return a tuple of either (`inputs`, `targets`) or (`inputs`, `targets`, `sample_weights`).
 - A generator or `keras.utils.Sequence` returning (`inputs`, `targets`) or (`inputs`, `targets`, `sample_weights`).

A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given in the *Unpacking behavior for iterator-like inputs* section of *Model.fit*.

- **y** – Target data. Like the input data *x*, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with *x* (you cannot have Numpy inputs and tensor targets, or inversely). If *x* is a dataset, generator or `keras.utils.Sequence` instance, *y* should not be specified (since targets will be obtained from the iterator/dataset).
- **batch_size** – Integer or *None*. Number of samples per batch of computation. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` if your data is in the form of a dataset, generators, or `keras.utils.Sequence` instances (since they generate batches).
- **verbose** – “auto”, 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line. “auto” defaults to 1 for most cases, and to 2 when used with *ParameterServerStrategy*. Note that the progress bar is not particularly useful when logged to a file, so `verbose=2` is recommended when not running interactively (e.g. in a production environment).
- **sample_weight** – Optional Numpy array of weights for the test samples, used for weighting the loss function. You can either pass a flat (1D) Numpy array with the same length as the input samples

(1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape (`samples`, `sequence_length`), to apply a different

weight to every timestep of every sample. This argument is not supported when x is a dataset, instead pass sample weights as the third element of x .

- **steps** – Integer or *None*. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of *None*. If x is a *tf.data* dataset and *steps* is *None*, ‘evaluate’ will run until the dataset is exhausted. This argument is not supported with array inputs.
- **callbacks** – List of *keras.callbacks.Callback* instances. List of callbacks to apply during evaluation. See [callbacks](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks).
- **max_queue_size** – Integer. Used for generator or *keras.utils.Sequence* input only. Maximum size for the generator queue. If unspecified, *max_queue_size* will default to 10.
- **workers** – Integer. Used for generator or *keras.utils.Sequence* input only. Maximum number of processes to spin up when using process-based threading. If unspecified, *workers* will default to 1.
- **use_multiprocessing** – Boolean. Used for generator or *keras.utils.Sequence* input only. If *True*, use process-based threading. If unspecified, *use_multiprocessing* will default to *False*. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can’t be passed easily to children processes.
- **return_dict** – If *True*, loss and metric results are returned as a dict, with each key being the name of the metric. If *False*, they are returned as a list.
- ****kwargs** – Unused at this time.

See the discussion of *Unpacking behavior for iterator-like inputs* for *Model.fit*.

Returns Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute *model.metrics_names* will give you the display labels for the scalar outputs.

Raises RuntimeError – If *model.evaluate* is wrapped in a *tf.function*.

evaluate_generator(*generator*, *steps=None*, *callbacks=None*, *max_queue_size=10*, *workers=1*,
 use_multiprocessing=False, *verbose=0*)

Evaluates the model on a data generator.

DEPRECATED: *Model.evaluate* now supports generators, so there is no longer any need to use this endpoint.

finalize_state()

Finalizes the layers state after updating layer weights.

This function can be subclassed in a layer and will be called after updating a layer weights. It can be overridden to finalize any additional layer state after a weight update.

This function will be called after weights of a layer have been restored from a loaded model.

fit(*x=None*, *y=None*, *batch_size=None*, *epochs=1*, *verbose='auto'*, *callbacks=None*, *validation_split=0.0*,
 validation_data=None, *shuffle=True*, *class_weight=None*, *sample_weight=None*, *initial_epoch=0*,
 steps_per_epoch=None, *validation_steps=None*, *validation_batch_size=None*, *validation_freq=1*,
 max_queue_size=10, *workers=1*, *use_multiprocessing=False*)

Trains the model for a fixed number of epochs (iterations on a dataset).

Parameters

- **x** – Input data. It could be: - A Numpy array (or array-like), or a list of arrays

(in case the model has multiple inputs).

- A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
- A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
- A `tf.data` dataset. Should return a tuple of either `(inputs, targets)` or `(inputs, targets, sample_weights)`.
- A generator or `keras.utils.Sequence` returning `(inputs, targets)` or `(inputs, targets, sample_weights)`.
- A `tf.keras.utils.experimental.DatasetCreator`, which wraps a callable that takes a single argument of type `tf.distribute.InputContext`, and returns a `tf.data.Dataset`. `DatasetCreator` should be used when users prefer to specify the per-replica batching and sharding logic for the `Dataset`. See `tf.keras.utils.experimental.DatasetCreator` doc for more information.

A more detailed description of unpacking behavior for iterator types (`Dataset`, `generator`, `Sequence`) is given below. If these include `sample_weights` as a third component, note that sample weighting applies to the `weighted_metrics` argument but not the `metrics` argument in `compile()`. If using `tf.distribute.experimental.ParameterServerStrategy`, only `DatasetCreator` type is supported for `x`.

- **y** – Target data. Like the input data `x`, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with `x` (you cannot have Numpy inputs and tensor targets, or inversely). If `x` is a dataset, generator, or `keras.utils.Sequence` instance, `y` should not be specified (since targets will be obtained from `x`).
- **batch_size** – Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` if your data is in the form of datasets, generators, or `keras.utils.Sequence` instances (since they generate batches).
- **epochs** – Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided (unless the `steps_per_epoch` flag is set to something other than `None`). Note that in conjunction with `initial_epoch`, `epochs` is to be understood as “final epoch”. The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.
- **verbose** – ‘auto’, 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch. ‘auto’ defaults to 1 for most cases, but 2 when used with `ParameterServerStrategy`. Note that the progress bar is not particularly useful when logged to a file, so `verbose=2` is recommended when not running interactively (eg, in a production environment).
- **callbacks** – List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See `tf.keras.callbacks`. Note `tf.keras.callbacks.ProgbarLogger` and `tf.keras.callbacks.History` callbacks are created automatically and need not be passed into `model.fit`. `tf.keras.callbacks.ProgbarLogger` is created or not based on `verbose` argument to `model.fit`. Callbacks with batch-level calls are currently unsupported with `tf.distribute.experimental.ParameterServerStrategy`, and users are advised to implement epoch-level calls instead with an appropriate `steps_per_epoch` value.
- **validation_split** – Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of

each epoch. The validation data is selected from the last samples in the x and y data provided, before shuffling. This argument is not supported when x is a dataset, generator or `keras.utils.Sequence` instance. If both `validation_data` and `validation_split` are provided, `validation_data` will override `validation_split`. `validation_split` is not yet supported with `tf.distribute.experimental.ParameterServerStrategy`.

- **validation_data** – Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. Thus, note the fact that the validation loss of data provided using `validation_split` or `validation_data` is not affected by regularization layers like noise and dropout. `validation_data` will override `validation_split`. `validation_data` could be:

- A tuple $(x_{\text{val}}, y_{\text{val}})$ of Numpy arrays or tensors.
- A tuple $(x_{\text{val}}, y_{\text{val}}, \text{val_sample_weights})$ of NumPy arrays.
- A `tf.data.Dataset`.
- A Python generator or `keras.utils.Sequence` returning $(\text{inputs}, \text{targets})$ or $(\text{inputs}, \text{targets}, \text{sample_weights})$.

`validation_data` is not yet supported with `tf.distribute.experimental.ParameterServerStrategy`.

- **shuffle** – Boolean (whether to shuffle the training data before each epoch) or str (for ‘batch’). This argument is ignored when x is a generator or an object of `tf.data.Dataset`. ‘batch’ is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not `None`.
- **class_weight** – Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to “pay more attention” to samples from an under-represented class.
- **sample_weight** – Optional Numpy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape $(\text{samples}, \text{sequence_length})$, to apply a different weight to every timestep of every sample. This argument is not supported when x is a dataset, generator, or `keras.utils.Sequence` instance, instead provide the `sample_weights` as the third element of x . Note that sample weighting does not apply to metrics specified via the `metrics` argument in `compile()`. To apply sample weighting to your metrics, you can specify them via the `weighted_metrics` in `compile()` instead.
- **initial_epoch** – Integer. Epoch at which to start training (useful for resuming a previous training run).
- **steps_per_epoch** – Integer or `None`. Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as TensorFlow data tensors, the default `None` is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined. If x is a `tf.data` dataset, and ‘`steps_per_epoch`’ is `None`, the epoch will run until the input dataset is exhausted. When passing an infinitely repeating dataset, you must specify the `steps_per_epoch` argument. If `steps_per_epoch=-1` the training will run indefinitely with an infinitely repeating dataset. This argument is not supported with array inputs. When using `tf.distribute.experimental.ParameterServerStrategy`:
 - `steps_per_epoch=None` is not supported.

- **validation_steps** – Only relevant if *validation_data* is provided and is a *tf.data* dataset. Total number of steps (batches of samples) to draw before stopping when performing validation at the end of every epoch. If ‘*validation_steps*’ is None, validation will run until the *validation_data* dataset is exhausted. In the case of an infinitely repeated dataset, it will run into an infinite loop. If ‘*validation_steps*’ is specified and only part of the dataset will be consumed, the evaluation will start from the beginning of the dataset at each epoch. This ensures that the same validation samples are used every time.
- **validation_batch_size** – Integer or *None*. Number of samples per validation batch. If unspecified, will default to *batch_size*. Do not specify the *validation_batch_size* if your data is in the form of datasets, generators, or *keras.utils.Sequence* instances (since they generate batches).
- **validation_freq** – Only relevant if validation data is provided. Integer or *collections.abc.Container* instance (e.g. list, tuple, etc.). If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. *validation_freq*=2 runs validation every 2 epochs. If a Container, specifies the epochs on which to run validation, e.g. *validation_freq*=[1, 2, 10] runs validation at the end of the 1st, 2nd, and 10th epochs.
- **max_queue_size** – Integer. Used for generator or *keras.utils.Sequence* input only. Maximum size for the generator queue. If unspecified, *max_queue_size* will default to 10.
- **workers** – Integer. Used for generator or *keras.utils.Sequence* input only. Maximum number of processes to spin up when using process-based threading. If unspecified, *workers* will default to 1.
- **use_multiprocessing** – Boolean. Used for generator or *keras.utils.Sequence* input only. If *True*, use process-based threading. If unspecified, *use_multiprocessing* will default to *False*. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.

Unpacking behavior for iterator-like inputs:

A common pattern is to pass a *tf.data.Dataset*, generator, or

tf.keras.utils.Sequence to the *x* argument of *fit*, which will in fact yield not only features (*x*) but optionally targets (*y*) and sample weights. Keras requires that the output of such iterator-likes be unambiguous. The iterator should return a tuple of length 1, 2, or 3, where the optional second and third elements will be used for *y* and *sample_weight* respectively. Any other type provided will be wrapped in a length one tuple, effectively treating everything as ‘*x*’. When yielding dicts, they should still adhere to the top-level tuple structure. e.g. ({“*x0*”: *x0*, “*x1*”: *x1*}, *y*). Keras will not attempt to separate features, targets, and weights from the keys of a single dict.

A notable unsupported data type is the namedtuple. The reason is

that it behaves like both an ordered datatype (tuple) and a mapping datatype (dict). So given a namedtuple of the form:

```
namedtuple("example_tuple", ["y", "x"])
```

it is ambiguous whether to reverse the order of the elements when interpreting the value. Even worse is a tuple of the form:

```
namedtuple("other_tuple", ["x", "y", "z"])
```

where it is unclear if the tuple was intended to be unpacked into `x`, `y`, and `sample_weight` or passed through as a single element to `x`. As a result the data processing code will simply raise a `ValueError` if it encounters a namedtuple. (Along with instructions to remedy the issue.)

Returns A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises

- **RuntimeError** –
 1. If the model was never compiled or,
 2. If `model.fit` is wrapped in `tf.function`.
- **ValueError** – In case of mismatch between the provided input data and what the model expects or when the input data is empty.

```
fit_generator(generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None,
              validation_data=None, validation_steps=None, validation_freq=1, class_weight=None,
              max_queue_size=10, workers=1, use_multiprocessing=False, shuffle=True,
              initial_epoch=0)
```

Fits the model on data yielded batch-by-batch by a Python generator.

DEPRECATED: `Model.fit` now supports generators, so there is no longer any need to use this endpoint.

```
classmethod from_config(config, custom_objects=None)
```

Creates a layer from its config.

This method is the reverse of `get_config`, capable of instantiating the same layer from the config dictionary. It does not handle layer connectivity (handled by Network), nor weights (handled by `set_weights`).

Parameters `config` – A Python dictionary, typically the output of `get_config`.

Returns A layer instance.

```
get_config()
```

Returns the config of the `Model`.

Config is a Python dictionary (serializable) containing the configuration of an object, which in this case is a `Model`. This allows the `Model` to be reinstated later (without its trained weights) from this configuration.

Note that `get_config()` does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Developers of subclassed `Model` are advised to override this method, and continue to update the dict from `super(MyModel, self).get_config()` to provide the proper configuration of this `Model`. The default config is an empty dict. Optionally, raise `NotImplementedError` to allow Keras to attempt a default serialization.

Returns Python dictionary containing the configuration of this `Model`.

```
get_input_at(node_index)
```

Retrieves the input tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first input node of the layer.

Returns A tensor (or list of tensors if the layer has multiple inputs).

Raises **RuntimeError** – If called in Eager mode.

get_input_mask_at(node_index)

Retrieves the input mask tensor(s) of a layer at a given node.

Parameters **node_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A mask tensor (or list of tensors if the layer has multiple inputs).

get_input_shape_at(node_index)

Retrieves the input shape(s) of a layer at a given node.

Parameters **node_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A shape tuple (or list of shape tuples if the layer has multiple inputs).

Raises `RuntimeError` – If called in Eager mode.

get_layer(name=None, index=None)

Retrieves a layer based on either its name (unique) or index.

If `name` and `index` are both provided, `index` will take precedence. Indices are based on order of horizontal graph traversal (bottom-up).

Parameters

- **name** – String, name of layer.
- **index** – Integer, index of layer.

Returns A layer instance.

get_metrics_result()

Returns the model's metrics values as a dict.

If any of the metric result is a dict (containing multiple metrics), each of them gets added to the top level returned dict of this method.

Returns A `dict` containing values of the metrics listed in `self.metrics`. Example: `{'loss': 0.2, 'accuracy': 0.7}`.

get_output_at(node_index)

Retrieves the output tensor(s) of a layer at a given node.

Parameters **node_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first output node of the layer.

Returns A tensor (or list of tensors if the layer has multiple outputs).

Raises `RuntimeError` – If called in Eager mode.

get_output_mask_at(node_index)

Retrieves the output mask tensor(s) of a layer at a given node.

Parameters **node_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A mask tensor (or list of tensors if the layer has multiple outputs).

get_output_shape_at(node_index)

Retrieves the output shape(s) of a layer at a given node.

Parameters **node_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A shape tuple (or list of shape tuples if the layer has multiple outputs).

Raises `RuntimeError` – If called in Eager mode.

`get_weight_paths()`

Retrieve all the variables and their paths for the model.

The variable path (string) is a stable key to indentify a `tf.Variable` instance owned by the model. It can be used to specify variable-specific configurations (e.g. DTensor, quantization) from a global view.

This method returns a dict with weight object paths as keys and the corresponding `tf.Variable` instances as values.

Note that if the model is a subclassed model and the weights haven't been initialized, an empty dict will be returned.

Returns

A dict where keys are variable paths and values are `tf.Variable` instances.

Example:

```
```python
class SubclassModel(tf.keras.Model):
 def __init__(self, name=None):
 super().__init__(name=name)
 self.d1 = tf.keras.layers.Dense(10)
 self.d2 = tf.keras.layers.Dense(20)

 def call(self, inputs):
 x = self.d1(inputs)
 return self.d2(x)

model = SubclassModel()
model(tf.zeros((10, 10)))
weight_paths = model.get_weight_paths() # weight_paths: { # 'd1.kernel': model.d1.kernel, # 'd1.bias': model.d1.bias, # 'd2.kernel': model.d2.kernel, # 'd2.bias': model.d2.bias, # }

Functional model
inputs = tf.keras.Input((10,), batch_size=10)
x = tf.keras.layers.Dense(20, name='d1')(inputs)
output = tf.keras.layers.Dense(30, name='d2')(x)
model = tf.keras.Model(inputs, output)
d1 = model.layers[1]
d2 = model.layers[2]
weight_paths = model.get_weight_paths() # weight_paths: { # 'd1.kernel': d1.kernel, # 'd1.bias': d1.bias, # 'd2.kernel': d2.kernel, # 'd2.bias': d2.bias, # }
```

```

`get_weights()`

Retrieves the weights of the model.

Returns A flat list of Numpy arrays.

`property inbound_nodes`

Return Functional API nodes upstream of this layer.

`property input`

Retrieves the input tensor(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer.

Returns Input tensor or list of input tensors.

Raises

- `RuntimeError` – If called in Eager mode.
- `AttributeError` – If no inbound nodes are found.

`property input_mask`

Retrieves the input mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Input mask tensor (potentially None) or list of input mask tensors.

Raises

- `AttributeError` – if the layer is connected to

- more than one incoming layers. –

property input_shape

Retrieves the input shape(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer, or if all inputs have the same shape.

Returns Input shape, as an integer shape tuple (or list of shape tuples, one tuple per input tensor).

Raises

- **AttributeError** – if the layer has no defined input_shape.

- **RuntimeError** – if called in Eager mode.

property input_spec

InputSpec instance(s) describing the input format for this layer.

When you create a layer subclass, you can set *self.input_spec* to enable the layer to run input compatibility checks when it is called. Consider a *Conv2D* layer: it can only be called on a single input tensor of rank 4. As such, you can set, in *__init__()*:

```
`python self.input_spec = tf.keras.layers.InputSpec(ndim=4)`
```

Now, if you try to call the layer on an input that isn't rank 4 (for instance, an input of shape (2,), it will raise a nicely-formatted error:

```
`ValueError: Input 0 of layer conv2d is incompatible with the layer: expected ndim=4, found ndim=1. Full shape received: [2]`
```

Input checks that can be specified via *input_spec* include: - Structure (e.g. a single input, a list of 2 inputs, etc) - Shape - Rank (ndim) - Dtype

For more information, see *tf.keras.layers.InputSpec*.

Returns A *tf.keras.layers.InputSpec* instance, or nested structure thereof.

property layers**load_weights(filepath, by_name=False, skip_mismatch=False, options=None)**

Loads all layer weights, either from a TensorFlow or an HDF5 weight file.

If *by_name* is False weights are loaded based on the network's topology. This means the architecture should be the same as when the weights were saved. Note that layers that don't have weights are not taken into account in the topological ordering, so adding or removing layers is fine as long as they don't have weights.

If *by_name* is True, weights are loaded into layers only if they share the same name. This is useful for fine-tuning or transfer-learning models where some of the layers have changed.

Only topological loading (*by_name=False*) is supported when loading weights from the TensorFlow format. Note that topological loading differs slightly between TensorFlow and HDF5 formats for user-defined classes inheriting from *tf.keras.Model*: HDF5 loads based on a flattened list of weights, while the TensorFlow format loads based on the object-local names of attributes to which layers are assigned in the *Model*'s constructor.

Parameters

- **filepath** – String, path to the weights file to load. For weight files in TensorFlow format, this is the file prefix (the same as was passed to *save_weights*). This can also be a path to a SavedModel saved from *model.save*.

- **by_name** – Boolean, whether to load weights by name or by topological order. Only topological loading is supported for weight files in TensorFlow format.
- **skip_mismatch** – Boolean, whether to skip loading of layers where there is a mismatch in the number of weights, or a mismatch in the shape of the weight (only valid when *by_name=True*).
- **options** – Optional *tf.train.CheckpointOptions* object that specifies options for loading weights.

Returns

When loading a weight file in TensorFlow format, returns the same status object as *tf.train.Checkpoint.restore*. When graph building, restore ops are run automatically as soon as the network is built (on first call for user-defined classes inheriting from *Model*, immediately if it is already built).

When loading weights in HDF5 format, returns *None*.

Raises

- **ImportError** – If *h5py* is not available and the weight file is in HDF5 format.
- **ValueError** – If *skip_mismatch* is set to *True* when *by_name* is *False*.

log_prob(*sample*: tensorflow.python.framework.ops.Tensor) → tensorflow.python.framework.ops.Tensor

property losses

List of losses added using the *add_loss()* API.

Variable regularization tensors are created when this property is accessed, so it is eager safe: accessing *losses* under a *tf.GradientTape* will propagate gradients back to the corresponding variables.

Examples:

```
>>> class MyLayer(tf.keras.layers.Layer):
...     def call(self, inputs):
...         self.add_loss(tf.abs(tf.reduce_mean(inputs)))
...         return inputs
>>> l = MyLayer()
>>> l(np.ones((10, 1)))
>>> l.losses
[1.0]
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> x = tf.keras.layers.Dense(10)(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Activity regularization.
>>> len(model.losses)
0
>>> model.add_loss(tf.abs(tf.reduce_mean(x)))
>>> len(model.losses)
1
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> d = tf.keras.layers.Dense(10, kernel_initializer='ones')
>>> x = d(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
```

(continues on next page)

(continued from previous page)

```
>>> model = tf.keras.Model(inputs, outputs)
>>> # Weight regularization.
>>> model.add_loss(lambda: tf.reduce_mean(d.kernel))
>>> model.losses
[<tf.Tensor: shape=(), dtype=float32, numpy=1.0>]
```

Returns A list of tensors.**make_predict_function(force=False)**

Creates a function that executes one step of inference.

This method can be overridden to support custom inference logic. This method is called by *Model.predict* and *Model.predict_on_batch*.

Typically, this method directly controls *tf.function* and *tf.distribute.Strategy* settings, and delegates the actual evaluation logic to *Model.predict_step*.

This function is cached the first time *Model.predict* or *Model.predict_on_batch* is called. The cache is cleared whenever *Model.compile* is called. You can skip the cache and generate again the function with *force=True*.

Parameters force – Whether to regenerate the predict function and skip the cached function if available.

Returns Function. The function created by this method should accept a *tf.data.Iterator*, and return the outputs of the *Model*.

make_test_function(force=False)

Creates a function that executes one step of evaluation.

This method can be overridden to support custom evaluation logic. This method is called by *Model.evaluate* and *Model.test_on_batch*.

Typically, this method directly controls *tf.function* and *tf.distribute.Strategy* settings, and delegates the actual evaluation logic to *Model.test_step*.

This function is cached the first time *Model.evaluate* or *Model.test_on_batch* is called. The cache is cleared whenever *Model.compile* is called. You can skip the cache and generate again the function with *force=True*.

Parameters force – Whether to regenerate the test function and skip the cached function if available.

Returns Function. The function created by this method should accept a *tf.data.Iterator*, and return a *dict* containing values that will be passed to *tf.keras.Callbacks.on_test_batch_end*.

make_train_function(force=False)

Creates a function that executes one step of training.

This method can be overridden to support custom training logic. This method is called by *Model.fit* and *Model.train_on_batch*.

Typically, this method directly controls *tf.function* and *tf.distribute.Strategy* settings, and delegates the actual training logic to *Model.train_step*.

This function is cached the first time *Model.fit* or *Model.train_on_batch* is called. The cache is cleared whenever *Model.compile* is called. You can skip the cache and generate again the function with *force=True*.

Parameters force – Whether to regenerate the train function and skip the cached function if available.

Returns Function. The function created by this method should accept a `tf.data.Iterator`, and return a `dict` containing values that will be passed to `tf.keras.Callbacks.on_train_batch_end`, such as `{'loss': 0.2, 'accuracy': 0.7}`.

property metrics

Returns the model's metrics added using `compile()`, `add_metric()` APIs.

Note: Metrics passed to `compile()` are available only after a `keras.Model` has been trained/evaluated on actual data.

Examples:

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> outputs = tf.keras.layers.Dense(2)(inputs)
>>> model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
>>> [m.name for m in model.metrics]
[]
```

```
>>> x = np.random.random((2, 3))
>>> y = np.random.randint(0, 2, (2, 2))
>>> model.fit(x, y)
>>> [m.name for m in model.metrics]
['loss', 'mae']
```

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2, name='out')
>>> output_1 = d(inputs)
>>> output_2 = d(inputs)
>>> model = tf.keras.models.Model(
...     inputs=inputs, outputs=[output_1, output_2])
>>> model.add_metric(
...     tf.reduce_sum(output_2), name='mean', aggregation='mean')
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae", "acc"])
>>> model.fit(x, (y, y))
>>> [m.name for m in model.metrics]
['loss', 'out_loss', 'out_1_loss', 'out_mae', 'out_acc', 'out_1_mae',
 'out_1_acc', 'mean']
```

property metrics_names

Returns the model's display labels for all outputs.

Note: `metrics_names` are available only after a `keras.Model` has been trained/evaluated on actual data.

Examples:

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> outputs = tf.keras.layers.Dense(2)(inputs)
>>> model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
>>> model.metrics_names
[]
```

```
>>> x = np.random.random((2, 3))
>>> y = np.random.randint(0, 2, (2, 2))
```

(continues on next page)

(continued from previous page)

```
>>> model.fit(x, y)
>>> model.metrics_names
['loss', 'mae']

>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2, name='out')
>>> output_1 = d(inputs)
>>> output_2 = d(inputs)
>>> model = tf.keras.models.Model(
...     inputs=inputs, outputs=[output_1, output_2])
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae", "acc"])
>>> model.fit(x, (y, y))
>>> model.metrics_names
['loss', 'out_loss', 'out_1_loss', 'out_mae', 'out_acc', 'out_1_mae',
'out_1_acc']
```

model() → Any**property name**

Name of the layer (string), set in the constructor.

property name_scope

Returns a `tf.name_scope` instance for this class.

property non_trainable_variables

Sequence of non-trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don't expect the return value to change.

Returns A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

property non_trainable_weights

List of all non-trainable weights tracked by this layer.

Non-trainable weights are *not* updated during training. They are expected to be updated manually in `call()`.

Returns A list of non-trainable variables.

property outbound_nodes

Return Functional API nodes downstream of this layer.

property output

Retrieves the output tensor(s) of a layer.

Only applicable if the layer has exactly one output, i.e. if it is connected to one incoming layer.

Returns Output tensor or list of output tensors.

Raises

- **AttributeError** – if the layer is connected to more than one incoming layers.
- **RuntimeError** – if called in Eager mode.

property output_mask

Retrieves the output mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Output mask tensor (potentially `None`) or list of output mask tensors.

Raises

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

property `output_shape`

Retrieves the output shape(s) of a layer.

Only applicable if the layer has one output, or if all outputs have the same shape.

Returns Output shape, as an integer shape tuple (or list of shape tuples, one tuple per output tensor).

Raises

- **AttributeError** – if the layer has no defined output shape.
- **RuntimeError** – if called in Eager mode.

`predict(x, batch_size=None, verbose='auto', steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiprocessing=False)`

Generates output predictions for the input samples.

Computation is done in batches. This method is designed for batch processing of large numbers of inputs. It is not intended for use inside of loops that iterate over your data and process small numbers of inputs at a time.

For small numbers of inputs that fit in one batch, directly use `__call__()` for faster execution, e.g., `model(x)`, or `model(x, training=False)` if you have layers such as `tf.keras.layers.BatchNormalization` that behave differently during inference. You may pair the individual model call with a `tf.function` for additional performance inside your inner loop. If you need access to numpy array values instead of tensors after your model call, you can use `tensor.numpy()` to get the numpy array value of an eager tensor.

Also, note the fact that test loss is not affected by regularization layers like noise and dropout.

Note: See [this FAQ entry](https://keras.io/getting_started/faq/#whats-the-difference-between-model-methods-predict-and-call) for more details about the difference between `Model` methods `predict()` and `__call__()`.

Parameters

- **x** – Input samples. It could be:
 - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
 - A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
 - A `tf.data` dataset.
 - A generator or `keras.utils.Sequence` instance.

A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given in the *Unpacking behavior for iterator-like inputs* section of `Model.fit`.

- **batch_size** – Integer or `None`. Number of samples per batch. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` if your data is in the form of dataset, generators, or `keras.utils.Sequence` instances (since they generate batches).
- **verbose** – “`auto`”, 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line. “`auto`” defaults to 1 for most cases, and to 2 when used with `ParameterServerStrategy`. Note that the progress bar is not particularly useful when logged to a file,

so `verbose=2` is recommended when not running interactively (e.g. in a production environment).

- **steps** – Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of `None`. If `x` is a `tf.data` dataset and `steps` is `None`, `predict()` will run until the input dataset is exhausted.
- **callbacks** – List of `keras.callbacks.Callback` instances. List of callbacks to apply during prediction. See [callbacks](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks).
- **max_queue_size** – Integer. Used for generator or `keras.utils.Sequence` input only. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers** – Integer. Used for generator or `keras.utils.Sequence` input only. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1.
- **use_multiprocessing** – Boolean. Used for generator or `keras.utils.Sequence` input only. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-pickleable arguments to the generator as they can't be passed easily to children processes.

See the discussion of *Unpacking behavior for iterator-like inputs* for `Model.fit`. Note that `Model.predict` uses the same interpretation rules as `Model.fit` and `Model.evaluate`, so inputs must be unambiguous for all three methods.

Returns Numpy array(s) of predictions.

Raises

- **RuntimeError** – If `model.predict` is wrapped in a `tf.function`.
- **ValueError** – In case of mismatch between the provided input data and the model's expectations, or in case a stateful model receives a number of samples that is not a multiple of the batch size.

`predict_generator(generator, steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiprocessing=False, verbose=0)`

Generates predictions for the input samples from a data generator.

DEPRECATED: `Model.predict` now supports generators, so there is no longer any need to use this endpoint.

`predict_on_batch(x)`

Returns predictions for a single batch of samples.

Parameters `x` – Input data. It could be: - A Numpy array (or array-like), or a list of arrays (in case the

model has multiple inputs).

- **A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).**

Returns Numpy array(s) of predictions.

Raises **RuntimeError** – If `model.predict_on_batch` is wrapped in a `tf.function`.

`predict_step(data)`

The logic for one inference step.

This method can be overridden to support custom inference logic. This method is called by `Model.make_predict_function`.

This method should contain the mathematical logic for one step of inference. This typically includes the forward pass.

Configuration details for *how* this logic is run (e.g. `tf.function` and `tf.distribute.Strategy` settings), should be left to `Model.make_predict_function`, which can also be overridden.

Parameters `data` – A nested structure of `Tensor`'s.

Returns The result of one inference step, typically the output of calling the `Model` on data.

`regularization(lbd_w: float = 1.0, lbd_b: float = 1.0) → tensorflow.python.framework.ops.Tensor`

`reset_metrics()`

Resets the state of all the metrics in the model.

Examples:

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> outputs = tf.keras.layers.Dense(2)(inputs)
>>> model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
```

```
>>> x = np.random.random((2, 3))
>>> y = np.random.randint(0, 2, (2, 2))
>>> _ = model.fit(x, y, verbose=0)
>>> assert all(float(m.result()) for m in model.metrics)
```

```
>>> model.reset_metrics()
>>> assert all(float(m.result()) == 0 for m in model.metrics)
```

`reset_states()`

property `run_eagerly`

Settable attribute indicating whether the model should run eagerly.

Running eagerly means that your model will be run step by step, like Python code. Your model might run slower, but it should become easier for you to debug it by stepping into individual layer calls.

By default, we will attempt to compile your model to a static graph to deliver the best execution performance.

Returns Boolean, whether the model should run eagerly.

`sample(batch_size: int) → Tuple[numpy.array, tensorflow.python.framework.ops.Tensor]`

`save(filepath, overwrite=True, include_optimizer=True, save_format=None, signatures=None, options=None, save_traces=True)`

Saves the model to Tensorflow SavedModel or a single HDF5 file.

Please see `tf.keras.models.save_model` or the [Serialization and Saving guide](https://keras.io/guides/serialization_and_saving/) for details.

Parameters

- **filepath** – String, PathLike, path to SavedModel or H5 file to save the model.
- **overwrite** – Whether to silently overwrite any existing file at the target location, or provide the user with a manual prompt.
- **include_optimizer** – If True, save optimizer's state together.

- **save_format** – Either ‘tf’ or ‘h5’, indicating whether to save the model to Tensorflow SavedModel or HDF5. Defaults to ‘tf’ in TF 2.X, and ‘h5’ in TF 1.X.
- **signatures** – Signatures to save with the SavedModel. Applicable to the ‘tf’ format only. Please see the *signatures* argument in *tf.saved_model.save* for details.
- **options** – (only applies to SavedModel format) *tf.saved_model.SaveOptions* object that specifies options for saving to SavedModel.
- **save_traces** – (only applies to SavedModel format) When enabled, the SavedModel will store the function traces for each layer. This can be disabled, so that only the configs of each layer are stored. Defaults to *True*. Disabling this will decrease serialization time and reduce file size, but it requires that all custom layers/models implement a *get_config()* method.

Example:

```
```python
from keras.models import load_model
model.save('my_model.h5') # creates a HDF5 file 'my_model.h5' del model # deletes the existing model
returns a compiled model # identical to the previous one
model = load_model('my_model.h5')
````
```

save_spec(dynamic_batch=True)

Returns the *tf.TensorSpec* of call inputs as a tuple (*args*, *kwargs*).

This value is automatically defined after calling the model for the first time. Afterwards, you can use it when exporting the model for serving:

```
```python
model = tf.keras.Model(...)
@tf.function def serve(*args, **kwargs):
 outputs = model(*args, **kwargs) # Apply postprocessing steps, or add additional outputs. ...
 return outputs
arg_specs is [tf.TensorSpec(...), ...]. kwarg_specs, in this # example, is an empty dict since functional
models do not use keyword # arguments. arg_specs, kwarg_specs = model.save_spec()
model.save(path, signatures={
```

```
 'serving_default': serve.get_concrete_function(*arg_specs, **kwarg_specs)
}
```

**Parameters dynamic\_batch** – Whether to set the batch sizes of all the returned *tf.TensorSpec* to *None*. (Note that when defining functional or Sequential models with *tf.keras.Input(..., batch\_size=X)*, the batch size will always be preserved). Defaults to *True*.

**Returns** If the model inputs are defined, returns a tuple (*args*, *kwargs*). All elements in *args* and *kwargs* are *tf.TensorSpec*. If the model inputs are not defined, returns *None*. The model inputs are automatically set when calling the model, *model.fit*, *model.evaluate* or *model.predict*.

#### **save\_weights(filepath, overwrite=True, save\_format=None, options=None)**

Saves all layer weights.

Either saves in HDF5 or in TensorFlow format based on the *save\_format* argument.

**When saving in HDF5 format, the weight file has:**

- **layer\_names (attribute), a list of strings** (ordered names of model layers).
- **For every layer, a group named layer.name**

- **For every such layer group, a group attribute `weight_names`,** a list of strings (ordered names of weights tensor of the layer).
- **For every weight in the layer, a dataset** storing the weight value, named after the weight tensor.

When saving in TensorFlow format, all objects referenced by the network are saved in the same format as `tf.train.Checkpoint`, including any `Layer` instances or `Optimizer` instances assigned to object attributes. For networks constructed from inputs and outputs using `tf.keras.Model(inputs, outputs)`, `Layer` instances used by the network are tracked/saved automatically. For user-defined classes which inherit from `tf.keras.Model`, `Layer` instances must be assigned to object attributes, typically in the constructor. See the documentation of `tf.train.Checkpoint` and `tf.keras.Model` for details.

While the formats are the same, do not mix `save_weights` and `tf.train.Checkpoint`. Checkpoints saved by `Model.save_weights` should be loaded using `Model.load_weights`. Checkpoints saved using `tf.train.Checkpoint.save` should be restored using the corresponding `tf.train.Checkpoint.restore`. Prefer `tf.train.Checkpoint` over `save_weights` for training checkpoints.

The TensorFlow format matches objects and variables by starting at a root object, `self` for `save_weights`, and greedily matching attribute names. For `Model.save` this is the `Model`, and for `Checkpoint.save` this is the `Checkpoint` even if the `Checkpoint` has a model attached. This means saving a `tf.keras.Model` using `save_weights` and loading into a `tf.train.Checkpoint` with a `Model` attached (or vice versa) will not match the `Model`'s variables. See the [guide to training checkpoints](<https://www.tensorflow.org/guide/checkpoint>) for details on the TensorFlow format.

### Parameters

- **`filepath`** – String or PathLike, path to the file to save the weights to. When saving in TensorFlow format, this is the prefix used for checkpoint files (multiple files are generated). Note that the ‘.h5’ suffix causes weights to be saved in HDF5 format.
- **`overwrite`** – Whether to silently overwrite any existing file at the target location, or provide the user with a manual prompt.
- **`save_format`** – Either ‘tf’ or ‘h5’. A `filepath` ending in ‘.h5’ or ‘.keras’ will default to HDF5 if `save_format` is `None`. Otherwise `None` defaults to ‘tf’.
- **`options`** – Optional `tf.train.CheckpointOptions` object that specifies options for saving weights.

**Raises `ImportError`** – If `h5py` is not available when attempting to save in HDF5 format.

### `set_weights(weights)`

Sets the weights of the layer, from NumPy arrays.

The weights of a layer represent the state of the layer. This function sets the weight values from numpy arrays. The weight values should be passed in the order they are created by the layer. Note that the layer's weights must be instantiated before calling this function, by calling the layer.

For example, a `Dense` layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another `Dense` layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
... kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
 [1.],
 [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
```

(continues on next page)

(continued from previous page)

```

... kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
 [2.],
 [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
 [1.],
 [1.]], dtype=float32), array([0.], dtype=float32)]

```

**Parameters** **weights** – a list of NumPy arrays. The number of arrays and their shape must match number of the dimensions of the weights of the layer (i.e. it should match the output of *get\_weights*).

**Raises** **ValueError** – If the provided weights list does not match the layer’s specifications.

#### property state\_updates

Deprecated, do NOT use!

Returns the *updates* from all layers that are stateful.

This is useful for separating training updates and state updates, e.g. when we need to update a layer’s internal state during prediction.

**Returns** A list of update ops.

#### property stateful

#### property submodules

Sequence of all sub-modules.

Submodules are modules which are properties of this module, or found as properties of modules which are properties of this module (and so on).

```

>>> a = tf.Module()
>>> b = tf.Module()
>>> c = tf.Module()
>>> a.b = b
>>> b.c = c
>>> list(a.submodules) == [b, c]
True
>>> list(b.submodules) == [c]
True
>>> list(c.submodules) == []
True

```

**Returns** A sequence of all submodules.

**summary**(*line\_length=None*, *positions=None*, *print\_fn=None*, *expand\_nested=False*, *show\_trainable=False*, *layer\_range=None*)

Prints a string summary of the network.

#### Parameters

- **line\_length** – Total length of printed lines (e.g. set this to adapt the display to different terminal window sizes).
- **positions** – Relative or absolute positions of log elements in each line. If not provided, defaults to `[.33, .55, .67, 1.]`.
- **print\_fn** – Print function to use. Defaults to `print`. It will be called on each line of the summary. You can set it to a custom function in order to capture the string summary.
- **expand\_nested** – Whether to expand the nested models. If not provided, defaults to `False`.
- **show\_trainable** – Whether to show if a layer is trainable. If not provided, defaults to `False`.
- **layer\_range** – a list or tuple of 2 strings, which is the starting layer name and ending layer name (both inclusive) indicating the range of layers to be printed in summary. It also accepts regex patterns instead of exact name. In such case, start predicate will be the first element it matches to `layer_range[0]` and the end predicate will be the last element it matches to `layer_range[1]`. By default `None` which considers all layers of model.

**Raises** `ValueError` – if `summary()` is called before the model is built.

#### **property supports\_masking**

Whether this layer supports computing a mask using `compute_mask`.

#### **test\_on\_batch(*x*, *y*=*None*, *sample\_weight*=*None*, *reset\_metrics*=*True*, *return\_dict*=*False*)**

Test the model on a single batch of samples.

##### **Parameters**

- **x** – Input data. It could be:
  - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
  - **A TensorFlow tensor, or a list of tensors (in case the model has** multiple inputs).
  - **A dict mapping input names to the corresponding array/tensors**, if the model has named inputs.
- **y** – Target data. Like the input data *x*, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with *x* (you cannot have Numpy inputs and tensor targets, or inversely).
- **sample\_weight** – Optional array of the same length as *x*, containing weights to apply to the model’s loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence\_length), to apply a different weight to every timestep of every sample.
- **reset\_metrics** – If *True*, the metrics returned will be only for this batch. If *False*, the metrics will be statefully accumulated across batches.
- **return\_dict** – If *True*, loss and metric results are returned as a dict, with each key being the name of the metric. If *False*, they are returned as a list.

**Returns** Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

**Raises `RuntimeError`** – If `model.test_on_batch` is wrapped in a `tf.function`.

**test\_step(`data`)**

The logic for one evaluation step.

This method can be overridden to support custom evaluation logic. This method is called by `Model.make_test_function`.

This function should contain the mathematical logic for one step of evaluation. This typically includes the forward pass, loss calculation, and metrics updates.

Configuration details for *how* this logic is run (e.g. `tf.function` and `tf.distribute.Strategy` settings), should be left to `Model.make_test_function`, which can also be overridden.

**Parameters `data`** – A nested structure of `Tensor`s.

**Returns** A `dict` containing values that will be passed to `tf.keras.callbacks.CallbackList.on_train_batch_end`. Typically, the values of the `Model`'s metrics are returned.

**to\_json(\*\*kwargs)**

Returns a JSON string containing the network configuration.

To load a network from a JSON save file, use `keras.models.model_from_json(json_string, custom_objects={})`.

**Parameters `**kwargs`** – Additional keyword arguments to be passed to `*json.dumps()`.

**Returns** A JSON string.

**to\_yaml(\*\*kwargs)**

Returns a yaml string containing the network configuration.

Note: Since TF 2.6, this method is no longer supported and will raise a `RuntimeError`.

To load a network from a yaml save file, use `keras.models.model_from_yaml(yaml_string, custom_objects={})`.

`custom_objects` should be a dictionary mapping the names of custom losses / layers / etc to the corresponding functions / classes.

**Parameters `**kwargs`** – Additional keyword arguments to be passed to `yaml.dump()`.

**Returns** A YAML string.

**Raises `RuntimeError`** – announces that the method poses a security risk

**train\_on\_batch(`x, y=None, sample_weight=None, class_weight=None, reset_metrics=True, return_dict=False`)**

Runs a single gradient update on a single batch of data.

**Parameters**

- **x** – Input data. It could be: - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
  - **A TensorFlow tensor, or a list of tensors** (in case the model has multiple inputs).
  - **A dict mapping input names to the corresponding array/tensors**, if the model has named inputs.
- **y** – Target data. Like the input data `x`, it could be either Numpy array(s) or TensorFlow tensor(s).

- **sample\_weight** – Optional array of the same length as  $x$ , containing weights to apply to the model’s loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence\_length), to apply a different weight to every timestep of every sample.
- **class\_weight** – Optional dictionary mapping class indices (integers) to a weight (float) to apply to the model’s loss for the samples from this class during training. This can be useful to tell the model to “pay more attention” to samples from an under-represented class.
- **reset\_metrics** – If *True*, the metrics returned will be only for this batch. If *False*, the metrics will be statefully accumulated across batches.
- **return\_dict** – If *True*, loss and metric results are returned as a dict, with each key being the name of the metric. If *False*, they are returned as a list.

**Returns** Scalar training loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute *model.metrics\_names* will give you the display labels for the scalar outputs.

**Raises** `RuntimeError` – If *model.train\_on\_batch* is wrapped in a *tf.function*.

### **train\_step**(*data*)

The logic for one training step.

This method can be overridden to support custom training logic. For concrete examples of how to override this method see [Customizing what happens in fit]([https://www.tensorflow.org/guide/keras/customizing\\_what\\_happens\\_in\\_fit](https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit)). This method is called by *Model.make\_train\_function*.

This method should contain the mathematical logic for one step of training. This typically includes the forward pass, loss calculation, backpropagation, and metric updates.

Configuration details for *how* this logic is run (e.g. *tf.function* and *tf.distribute.Strategy* settings), should be left to *Model.make\_train\_function*, which can also be overridden.

**Parameters** *data* – A nested structure of `Tensor`’s.

**Returns** A *dict* containing values that will be passed to *tf.keras.callbacks.CallbackList.on\_train\_batch\_end*. Typically, the values of the *Model*’s metrics are returned. Example: {‘loss’: 0.2, ‘accuracy’: 0.7}.

### **property trainable**

### **property trainable\_variables**

Sequence of trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don’t expect the return value to change.

**Returns** A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

### **property trainable\_weights**

List of all trainable weights tracked by this layer.

Trainable weights are updated via gradient descent during training.

**Returns** A list of trainable variables.

### **property updates**

### **property variable\_dtype**

Alias of *Layer.dtype*, the dtype of the weights.

**property variables**

Returns the list of all layer variables/weights.

Alias of `self.weights`.

Note: This will not track the weights of nested `tf.Modules` that are not themselves Keras layers.

**Returns** A list of variables.

**property weights**

Returns the list of all layer variables/weights.

Note: This will not track the weights of nested `tf.Modules` that are not themselves Keras layers.

**Returns** A list of variables.

**classmethod with\_name\_scope(method)**

Decorator to automatically enter the module name scope.

```
>>> class MyModule(tf.Module):
... @tf.Module.with_name_scope
... def __call__(self, x):
... if not hasattr(self, 'w'):
... self.w = tf.Variable(tf.random.normal([x.shape[1], 3]))
... return tf.matmul(x, self.w)
```

Using the above module would produce `tf.Variable`'s and `tf.Tensor`'s whose names included the module name:

```
>>> mod = MyModule()
>>> mod(tf.ones([1, 2]))
<tf.Tensor: shape=(1, 3), dtype=float32, numpy=..., dtype=float32>
>>> mod.w
<tf.Variable 'my_module/Variable:0' shape=(2, 3) dtype=float32,
numpy=..., dtype=float32>
```

**Parameters** `method` – The method to wrap.

**Returns** The original method wrapped such that it enters the module's name scope.

**class tensorcircuit.applications.van.MaskedConv2D(\*args, \*\*kwargs)**

Bases: `keras.engine.base_layer.Layer`

**\_\_init\_\_(mask\_type: str, \*\*kwargs: Any)****property activity\_regularizer**

Optional regularizer function for the output of this layer.

**add\_loss(losses, \*\*kwargs)**

Add loss tensor(s), potentially dependent on layer inputs.

Some losses (for instance, activity regularization losses) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs `a` and `b`, some entries in `layer.losses` may be dependent on `a` and some on `b`. This method automatically keeps track of dependencies.

This method can be used inside a subclassed layer or model's `call` function, in which case `losses` should be a Tensor or list of Tensors.

Example:

```
```python
class MyLayer(tf.keras.layers.Layer):
```

```
def call(self, inputs): self.add_loss(tf.abs(tf.reduce_mean(inputs))) return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any loss Tensors passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These losses become part of the model's topology and are tracked in `get_config`.*

Example:

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.  
Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.  
Model(inputs, outputs) # Activity regularization. model.add_loss(tf.abs(tf.  
reduce_mean(x)))`
```

If this is not the case for your loss (if, for example, your loss references a *Variable* of one of the model's layers), you can wrap your loss in a zero-argument lambda. These losses are not tracked as part of the model's topology since they can't be serialized.

Example:

```
`python inputs = tf.keras.Input(shape=(10,)) d = tf.keras.layers.Dense(10) x =  
d(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs,  
outputs) # Weight regularization. model.add_loss(lambda: tf.reduce_mean(d.  
kernel))`
```

Parameters

- **losses** – Loss tensor, or list/tuple of tensors. Rather than tensors, losses may also be zero-argument callables which create a loss tensor.
- ****kwargs** – Used for backwards compatibility only.

add_metric(value, name=None, **kwargs)

Adds metric tensor to the layer.

This method can be used inside the *call()* method of a subclassed layer or model.

```
```python class MyMetricLayer(tf.keras.layers.Layer):  

 def __init__(self): super(MyMetricLayer, self).__init__(name='my_metric_layer') self.mean
 = tf.keras.metrics.Mean(name='metric_1')

 def call(self, inputs): self.add_metric(self.mean(inputs)) self.add_metric(tf.reduce_sum(inputs),
 name='metric_2') return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any tensor passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These metrics become part of the model's topology and are tracked when you save the model via `save()`.*

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.
Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.
keras.Model(inputs, outputs) model.add_metric(math_ops.reduce_sum(x),
name='metric_1')`
```

Note: Calling *add\_metric()* with the result of a metric object on a Functional Model, as shown in the example below, is not supported. This is because we cannot trace the metric result tensor back to the model's inputs.

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.
Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.
```

```
Model(inputs, outputs) model.add_metric(tf.keras.metrics.Mean()(x),
name='metric_1')
```

#### Parameters

- **value** – Metric tensor.
- **name** – String metric name.
- **\*\*kwargs** – Additional keyword arguments for backward compatibility. Accepted values: *aggregation* - When the *value* tensor provided is not the result of calling a *keras.Metric* instance, it will be aggregated by default using a *keras.Metric.Mean*.

#### add\_update(updates)

Add update op(s), potentially dependent on layer inputs.

Weight updates (for instance, the updates of the moving mean and variance in a BatchNormalization layer) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs *a* and *b*, some entries in *layer.updates* may be dependent on *a* and some on *b*. This method automatically keeps track of dependencies.

This call is ignored when eager execution is enabled (in that case, variable updates are run on the fly and thus do not need to be tracked for later execution).

**Parameters** **updates** – Update op, or list/tuple of update ops, or zero-arg callable that returns an update op. A zero-arg callable should be passed in order to disable running the updates by setting *trainable=False* on this Layer, when executing in Eager mode.

#### add\_variable(\*args, \*\*kwargs)

Deprecated, do NOT use! Alias for *add\_weight*.

#### add\_weight(name=None, shape=None, dtype=None, initializer=None, regularizer=None, trainable=None, constraint=None, use\_resource=None, synchronization=VariableSynchronization.AUTO, aggregation=VariableAggregationV2.NONE, \*\*kwargs)

Adds a new variable to the layer.

#### Parameters

- **name** – Variable name.
- **shape** – Variable shape. Defaults to scalar if unspecified.
- **dtype** – The type of the variable. Defaults to *self.dtype*.
- **initializer** – Initializer instance (callable).
- **regularizer** – Regularizer instance (callable).
- **trainable** – Boolean, whether the variable should be part of the layer’s “trainable\_variables” (e.g. variables, biases) or “non\_trainable\_variables” (e.g. BatchNorm mean and variance). Note that *trainable* cannot be *True* if *synchronization* is set to *ON\_READ*.
- **constraint** – Constraint instance (callable).
- **use\_resource** – Whether to use a *ResourceVariable* or not. See [this guide]([https://www.tensorflow.org/guide/migrate/tf1\\_vs\\_tf2#resourcevariables\\_instead\\_of\\_referencevariables](https://www.tensorflow.org/guide/migrate/tf1_vs_tf2#resourcevariables_instead_of_referencevariables)) for more information.
- **synchronization** – Indicates when a distributed a variable will be aggregated. Accepted values are constants defined in the class *tf.VariableSynchronization*. By default the synchronization is set to *AUTO* and the current *DistributionStrategy* chooses when

to synchronize. If *synchronization* is set to *ON\_READ*, *trainable* must not be set to *True*.

- **aggregation** – Indicates how a distributed variable will be aggregated. Accepted values are constants defined in the class *tf.VariableAggregation*.
- **\*\*kwargs** – Additional keyword arguments. Accepted values are *getter*, *collections*, *experimental\_autocast* and *caching\_device*.

**Returns** The variable created.

**Raises ValueError** – When giving unsupported dtype and no initializer or when trainable has been set to True with synchronization set as *ON\_READ*.

**build**(*input\_shape*: Union[List[*tensorflow.python.framework.tensor\_shape.TensorShape*], *tensorflow.python.framework.tensor\_shape.TensorShape*]) → None

Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

**Parameters** **input\_shape** – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

**call**(*inputs*: *tensorflow.python.framework.ops.Tensor*) → *tensorflow.python.framework.ops.Tensor*

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init\_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *\_\_init\_\_()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

### Parameters

- **inputs** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero arguments, and *inputs* cannot be provided via the default value of a keyword argument.
  - NumPy array or Python scalar values in *inputs* get cast as tensors.
  - Keras mask metadata is only collected from *inputs*.
  - Layers are built (*build(input\_shape)* method) using shape info from *inputs* only.
  - *input\_spec* compatibility is only checked against *inputs*.
  - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in *\*args* or *\*\*kwargs*, their casting behavior in mixed precision should be handled manually.
  - The SavedModel input specification is generated using *inputs* only.
  - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- **\*args** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- **\*\*kwargs** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved:
  - *training*: Boolean scalar tensor of Python boolean indicating whether the *call* is meant for training or inference.
  - *mask*: Boolean input mask. If the layer’s *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

**Returns** A tensor or list/tuple of tensors.

#### **property compute\_dtype**

The dtype of the layer’s computations.

This is equivalent to *Layer.dtype\_policy.compute\_dtype*. Unless mixed precision is used, this is the same as *Layer.dtype*, the dtype of the weights.

Layers automatically cast their inputs to the compute dtype, which causes computations and the output to be in the compute dtype as well. This is done by the base Layer class in *Layer.\_\_call\_\_*, so you do not have to insert these casts if implementing your own layer.

Layers often perform certain internal computations in higher precision when *compute\_dtype* is float16 or bfloat16 for numeric stability. The output will still typically be float16 or bfloat16 in such cases.

**Returns** The layer’s compute dtype.

#### **compute\_mask(inputs, mask=None)**

Computes an output mask tensor.

##### **Parameters**

- **inputs** – Tensor or list of tensors.
- **mask** – Tensor or list of tensors.

##### **Returns**

**None or a tensor (or list of tensors**, one per output tensor of the layer).

#### **compute\_output\_shape(input\_shape)**

Computes the output shape of the layer.

This method will cause the layer’s state to be built, if that has not happened before. This requires that the layer will later be used with inputs that match the input shape provided here.

**Parameters** **input\_shape** – Shape tuple (tuple of integers) or *tf.TensorShape*, or structure of shape tuples / *tf.TensorShape* instances (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

**Returns** A *tf.TensorShape* instance or structure of *tf.TensorShape* instances.

#### **compute\_output\_signature(input\_signature)**

Compute the output tensor signature of the layer based on the inputs.

Unlike a *TensorShape* object, a *TensorSpec* object contains both shape and dtype information for a tensor. This method allows layers to provide output dtype information if it is different from the input dtype. For any layer that doesn’t implement this function, the framework will fall back to use *compute\_output\_shape*, and will assume that the output dtype matches the input dtype.

**Parameters** `input_signature` – Single TensorSpec or nested structure of TensorSpec objects, describing a candidate input for the layer.

**Returns**

**Single TensorSpec or nested structure of TensorSpec objects**, describing how the layer would transform the provided input.

**Raises** `TypeError` – If `input_signature` contains a non-TensorSpec object.

**count\_params()**

Count the total number of scalars composing the weights.

**Returns** An integer count.

**Raises** `ValueError` – if the layer isn't yet built (in which case its weights aren't yet defined).

**property** `dtype`

The dtype of the layer weights.

This is equivalent to `Layer.dtype_policy.variable_dtype`. Unless mixed precision is used, this is the same as `Layer.compute_dtype`, the dtype of the layer's computations.

**property** `dtype_policy`

The dtype policy associated with this layer.

This is an instance of a `tf.keras.mixed_precision.Policy`.

**property** `dynamic`

Whether the layer is dynamic (eager-only); set in the constructor.

**finalize\_state()**

Finalizes the layers state after updating layer weights.

This function can be subclassed in a layer and will be called after updating a layer weights. It can be overridden to finalize any additional layer state after a weight update.

This function will be called after weights of a layer have been restored from a loaded model.

**classmethod** `from_config(config)`

Creates a layer from its config.

This method is the reverse of `get_config`, capable of instantiating the same layer from the config dictionary. It does not handle layer connectivity (handled by Network), nor weights (handled by `set_weights`).

**Parameters** `config` – A Python dictionary, typically the output of `get_config`.

**Returns** A layer instance.

**get\_config()**

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by `Network` (one layer of abstraction above).

Note that `get_config()` does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

**Returns** Python dictionary.

**get\_input\_at(node\_index)**

Retrieves the input tensor(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first input node of the layer.

**Returns** A tensor (or list of tensors if the layer has multiple inputs).

**Raises** `RuntimeError` – If called in Eager mode.

**get\_input\_mask\_at(node\_index)**

Retrieves the input mask tensor(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

**Returns** A mask tensor (or list of tensors if the layer has multiple inputs).

**get\_input\_shape\_at(node\_index)**

Retrieves the input shape(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

**Returns** A shape tuple (or list of shape tuples if the layer has multiple inputs).

**Raises** `RuntimeError` – If called in Eager mode.

**get\_output\_at(node\_index)**

Retrieves the output tensor(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first output node of the layer.

**Returns** A tensor (or list of tensors if the layer has multiple outputs).

**Raises** `RuntimeError` – If called in Eager mode.

**get\_output\_mask\_at(node\_index)**

Retrieves the output mask tensor(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

**Returns** A mask tensor (or list of tensors if the layer has multiple outputs).

**get\_output\_shape\_at(node\_index)**

Retrieves the output shape(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

**Returns** A shape tuple (or list of shape tuples if the layer has multiple outputs).

**Raises** `RuntimeError` – If called in Eager mode.

**get\_weights()**

Returns the current weights of the layer, as NumPy arrays.

The weights of a layer represent the state of the layer. This function returns both trainable and non-trainable weight values associated with this layer as a list of NumPy arrays, which can in turn be used to load state into similarly parameterized layers.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```

>>> layer_a = tf.keras.layers.Dense(1,
... kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
 [1.],
 [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
... kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
 [2.],
 [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
 [1.],
 [1.]], dtype=float32), array([0.], dtype=float32)]

```

**Returns** Weights values as a list of NumPy arrays.

#### property inbound\_nodes

Return Functional API nodes upstream of this layer.

#### property input

Retrieves the input tensor(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer.

**Returns** Input tensor or list of input tensors.

#### Raises

- **RuntimeError** – If called in Eager mode.
- **AttributeError** – If no inbound nodes are found.

#### property input\_mask

Retrieves the input mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

**Returns** Input mask tensor (potentially None) or list of input mask tensors.

#### Raises

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

#### property input\_shape

Retrieves the input shape(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer, or if all inputs have the same shape.

**Returns** Input shape, as an integer shape tuple (or list of shape tuples, one tuple per input tensor).

#### Raises

- **AttributeError** – if the layer has no defined input\_shape.
- **RuntimeError** – if called in Eager mode.

**property input\_spec**

*InputSpec* instance(s) describing the input format for this layer.

When you create a layer subclass, you can set *self.input\_spec* to enable the layer to run input compatibility checks when it is called. Consider a *Conv2D* layer: it can only be called on a single input tensor of rank 4. As such, you can set, in *\_\_init\_\_()*:

```
`python self.input_spec = tf.keras.layers.InputSpec(ndim=4)`
```

Now, if you try to call the layer on an input that isn't rank 4 (for instance, an input of shape (2,), it will raise a nicely-formatted error:

```
`ValueError: Input 0 of layer conv2d is incompatible with the layer: expected ndim=4, found ndim=1. Full shape received: [2]`
```

Input checks that can be specified via *input\_spec* include:

- Structure (e.g. a single input, a list of 2 inputs, etc)
- Shape
- Rank (ndim)
- Dtype

For more information, see *tf.keras.layers.InputSpec*.

**Returns** A *tf.keras.layers.InputSpec* instance, or nested structure thereof.

**property losses**

List of losses added using the *add\_loss()* API.

Variable regularization tensors are created when this property is accessed, so it is eager safe: accessing *losses* under a *tf.GradientTape* will propagate gradients back to the corresponding variables.

Examples:

```
>>> class MyLayer(tf.keras.layers.Layer):
... def call(self, inputs):
... self.add_loss(tf.abs(tf.reduce_mean(inputs)))
... return inputs
>>> l = MyLayer()
>>> l(np.ones((10, 1)))
>>> l.losses
[1.0]
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> x = tf.keras.layers.Dense(10)(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Activity regularization.
>>> len(model.losses)
0
>>> model.add_loss(tf.abs(tf.reduce_mean(x)))
>>> len(model.losses)
1
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> d = tf.keras.layers.Dense(10, kernel_initializer='ones')
>>> x = d(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
```

(continues on next page)

(continued from previous page)

```
>>> # Weight regularization.
>>> model.add_loss(lambda: tf.reduce_mean(d.kernel))
>>> model.losses
[<tf.Tensor: shape=(), dtype=float32, numpy=1.0>]
```

**Returns** A list of tensors.**property metrics**List of metrics added using the `add_metric()` API.

Example:

```
>>> input = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2)
>>> output = d(input)
>>> d.add_metric(tf.reduce_max(output), name='max')
>>> d.add_metric(tf.reduce_min(output), name='min')
>>> [m.name for m in d.metrics]
['max', 'min']
```

**Returns** A list of *Metric* objects.**property name**

Name of the layer (string), set in the constructor.

**property name\_scope**Returns a `tf.name_scope` instance for this class.**property non\_trainable\_variables**

Sequence of non-trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don't expect the return value to change.

**Returns** A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

**property non\_trainable\_weights**

List of all non-trainable weights tracked by this layer.

Non-trainable weights are *not* updated during training. They are expected to be updated manually in `call()`.**Returns** A list of non-trainable variables.**property outbound\_nodes**

Return Functional API nodes downstream of this layer.

**property output**

Retrieves the output tensor(s) of a layer.

Only applicable if the layer has exactly one output, i.e. if it is connected to one incoming layer.

**Returns** Output tensor or list of output tensors.**Raises**

- **AttributeError** – if the layer is connected to more than one incoming layers.

- **RuntimeError** – if called in Eager mode.

**property output\_mask**

Retrieves the output mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

**Returns** Output mask tensor (potentially None) or list of output mask tensors.

**Raises**

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

**property output\_shape**

Retrieves the output shape(s) of a layer.

Only applicable if the layer has one output, or if all outputs have the same shape.

**Returns** Output shape, as an integer shape tuple (or list of shape tuples, one tuple per output tensor).

**Raises**

- **AttributeError** – if the layer has no defined output shape.
- **RuntimeError** – if called in Eager mode.

**set\_weights(weights)**

Sets the weights of the layer, from NumPy arrays.

The weights of a layer represent the state of the layer. This function sets the weight values from numpy arrays. The weight values should be passed in the order they are created by the layer. Note that the layer's weights must be instantiated before calling this function, by calling the layer.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
... kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
 [1.],
 [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
... kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
 [2.],
 [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
 [1.],
 [1.]], dtype=float32), array([0.], dtype=float32)]
```

**Parameters** **weights** – a list of NumPy arrays. The number of arrays and their shape must match number of the dimensions of the weights of the layer (i.e. it should match the output

of `get_weights`).

**Raises ValueError** – If the provided weights list does not match the layer’s specifications.

#### property `stateful`

#### property `submodules`

Sequence of all sub-modules.

Submodules are modules which are properties of this module, or found as properties of modules which are properties of this module (and so on).

```
>>> a = tf.Module()
>>> b = tf.Module()
>>> c = tf.Module()
>>> a.b = b
>>> b.c = c
>>> list(a.submodules) == [b, c]
True
>>> list(b.submodules) == [c]
True
>>> list(c.submodules) == []
True
```

**Returns** A sequence of all submodules.

#### property `supports_masking`

Whether this layer supports computing a mask using `compute_mask`.

#### property `trainable`

#### property `trainable_variables`

Sequence of trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don’t expect the return value to change.

**Returns** A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

#### property `trainable_weights`

List of all trainable weights tracked by this layer.

Trainable weights are updated via gradient descent during training.

**Returns** A list of trainable variables.

#### property `updates`

#### property `variable_dtype`

Alias of `Layer.dtype`, the dtype of the weights.

#### property `variables`

Returns the list of all layer variables/weights.

Alias of `self.weights`.

Note: This will not track the weights of nested `tf.Modules` that are not themselves Keras layers.

**Returns** A list of variables.

**property weights**

Returns the list of all layer variables/weights.

**Returns** A list of variables.

**classmethod with\_name\_scope(method)**

Decorator to automatically enter the module name scope.

```
>>> class MyModule(tf.Module):
... @tf.Module.with_name_scope
... def __call__(self, x):
... if not hasattr(self, 'w'):
... self.w = tf.Variable(tf.random.normal([x.shape[1], 3]))
... return tf.matmul(x, self.w)
```

Using the above module would produce `tf.Variable`s and `tf.Tensor`s whose names included the module name:

```
>>> mod = MyModule()
>>> mod(tf.ones([1, 2]))
<tf.Tensor: shape=(1, 3), dtype=float32, numpy=..., dtype=float32>
>>> mod.w
<tf.Variable 'my_module/Variable:0' shape=(2, 3) dtype=float32,
numpy=..., dtype=float32>
```

**Parameters** **method** – The method to wrap.

**Returns** The original method wrapped such that it enters the module's name scope.

**class tensorcircuit.applications.van.MaskedLinear(\*args, \*\*kwargs)**

Bases: keras.engine.base\_layer.Layer

```
__init__(input_space: int, output_space: int, spin_channel: int, mask:
 Optional[tensorflow.python.framework.ops.Tensor] = None, dtype:
 Optional[tensorflow.python.framework.dtypes.DType] = None)
```

**property activity\_regularizer**

Optional regularizer function for the output of this layer.

**add\_loss(losses, \*\*kwargs)**

Add loss tensor(s), potentially dependent on layer inputs.

Some losses (for instance, activity regularization losses) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs  $a$  and  $b$ , some entries in  $layer.losses$  may be dependent on  $a$  and some on  $b$ . This method automatically keeps track of dependencies.

This method can be used inside a subclassed layer or model's *call* function, in which case *losses* should be a Tensor or list of Tensors.

Example:

```
```python
class MyLayer(tf.keras.layers.Layer):

    def call(self, inputs):
        self.add_loss(tf.abs(tf.reduce_mean(inputs)))
        return inputs
```

```

This method can also be called directly on a Functional Model during construction. In this case, any loss Tensors passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These losses become part of the model's topology and are tracked in 'get\_config'.*

Example:

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.
Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.
Model(inputs, outputs) # Activity regularization. model.add_loss(tf.abs(tf.
reduce_mean(x)))`
```

If this is not the case for your loss (if, for example, your loss references a *Variable* of one of the model's layers), you can wrap your loss in a zero-argument lambda. These losses are not tracked as part of the model's topology since they can't be serialized.

Example:

```
`python inputs = tf.keras.Input(shape=(10,)) d = tf.keras.layers.Dense(10) x =
d(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs,
outputs) # Weight regularization. model.add_loss(lambda: tf.reduce_mean(d.
kernel))`
```

#### Parameters

- **losses** – Loss tensor, or list/tuple of tensors. Rather than tensors, losses may also be zero-argument callables which create a loss tensor.
- **\*\*kwargs** – Used for backwards compatibility only.

**add\_metric**(value, name=None, \*\*kwargs)

Adds metric tensor to the layer.

This method can be used inside the *call()* method of a subclassed layer or model.

```
```python class MyMetricLayer(tf.keras.layers.Layer):
```

```
    def __init__(self): super(MyMetricLayer, self).__init__(name='my_metric_layer') self.mean  
        = tf.keras.metrics.Mean(name='metric_1')  
  
    def call(self, inputs): self.add_metric(self.mean(inputs)) self.add_metric(tf.reduce_sum(inputs),  
        name='metric_2') return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any tensor passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These metrics become part of the model's topology and are tracked when you save the model via 'save()'*.

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.  
Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.  
keras.Model(inputs, outputs) model.add_metric(math_ops.reduce_sum(x),  
name='metric_1')`
```

Note: Calling *add_metric()* with the result of a metric object on a Functional Model, as shown in the example below, is not supported. This is because we cannot trace the metric result tensor back to the model's inputs.

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.  
Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.  
Model(inputs, outputs) model.add_metric(tf.keras.metrics.Mean()(x),  
name='metric_1')`
```

Parameters

- **value** – Metric tensor.
- **name** – String metric name.

- ****kwargs** – Additional keyword arguments for backward compatibility. Accepted values: *aggregation* - When the *value* tensor provided is not the result of calling a *keras.Metric* instance, it will be aggregated by default using a *keras.Metric.Mean*.

add_update(updates)

Add update op(s), potentially dependent on layer inputs.

Weight updates (for instance, the updates of the moving mean and variance in a BatchNormalization layer) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs *a* and *b*, some entries in *layer.updates* may be dependent on *a* and some on *b*. This method automatically keeps track of dependencies.

This call is ignored when eager execution is enabled (in that case, variable updates are run on the fly and thus do not need to be tracked for later execution).

Parameters **updates** – Update op, or list/tuple of update ops, or zero-arg callable that returns an update op. A zero-arg callable should be passed in order to disable running the updates by setting *trainable=False* on this Layer, when executing in Eager mode.

add_variable(*args, **kwargs)

Deprecated, do NOT use! Alias for *add_weight*.

add_weight(name=None, shape=None, dtype=None, initializer=None, regularizer=None, trainable=None, constraint=None, use_resource=None, synchronization=VariableSynchronization.AUTO, aggregation=VariableAggregationV2.NONE, **kwargs)

Adds a new variable to the layer.

Parameters

- **name** – Variable name.
- **shape** – Variable shape. Defaults to scalar if unspecified.
- **dtype** – The type of the variable. Defaults to *self.dtype*.
- **initializer** – Initializer instance (callable).
- **regularizer** – Regularizer instance (callable).
- **trainable** – Boolean, whether the variable should be part of the layer’s “trainable_variables” (e.g. variables, biases) or “non_trainable_variables” (e.g. BatchNorm mean and variance). Note that *trainable* cannot be *True* if *synchronization* is set to *ON_READ*.
- **constraint** – Constraint instance (callable).
- **use_resource** – Whether to use a *ResourceVariable* or not. See [this guide](https://www.tensorflow.org/guide/migrate/tf1_vs_tf2#resourcevariables_instead_of_referencevariables) for more information.
- **synchronization** – Indicates when a distributed a variable will be aggregated. Accepted values are constants defined in the class *tf.VariableSynchronization*. By default the synchronization is set to *AUTO* and the current *DistributionStrategy* chooses when to synchronize. If *synchronization* is set to *ON_READ*, *trainable* must not be set to *True*.
- **aggregation** – Indicates how a distributed variable will be aggregated. Accepted values are constants defined in the class *tf.VariableAggregation*.
- ****kwargs** – Additional keyword arguments. Accepted values are *getter*, *collections*, *experimental_autocast* and *caching_device*.

Returns The variable created.

Raises ValueError – When giving unsupported dtype and no initializer or when trainable has been set to True with synchronization set as *ON_READ*.

build(*input_shape*)

Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters *input_shape* – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*inputs*: tensorflow.python.framework.ops.Tensor) → tensorflow.python.framework.ops.Tensor

This is where the layer’s logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- **inputs** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero arguments, and *inputs* cannot be provided via the default value of a keyword argument.
 - NumPy array or Python scalar values in *inputs* get cast as tensors.
 - Keras mask metadata is only collected from *inputs*.
 - Layers are built (*build(input_shape)* method) using shape info from *inputs* only.
 - *input_spec* compatibility is only checked against *inputs*.
 - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.
 - The SavedModel input specification is generated using *inputs* only.
 - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.
- ***args** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- ****kwargs** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor or Python boolean indicating whether the *call* is meant for training or inference.

- *mask*: Boolean input mask. If the layer’s *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns A tensor or list/tuple of tensors.

property `compute_dtype`

The dtype of the layer’s computations.

This is equivalent to *Layer.dtype_policy.compute_dtype*. Unless mixed precision is used, this is the same as *Layer.dtype*, the dtype of the weights.

Layers automatically cast their inputs to the compute dtype, which causes computations and the output to be in the compute dtype as well. This is done by the base Layer class in *Layer.__call__*, so you do not have to insert these casts if implementing your own layer.

Layers often perform certain internal computations in higher precision when *compute_dtype* is float16 or bfloat16 for numeric stability. The output will still typically be float16 or bfloat16 in such cases.

Returns The layer’s compute dtype.

`compute_mask(inputs, mask=None)`

Computes an output mask tensor.

Parameters

- **inputs** – Tensor or list of tensors.
- **mask** – Tensor or list of tensors.

Returns

None or a tensor (or list of tensors, one per output tensor of the layer).

`compute_output_shape(input_shape)`

Computes the output shape of the layer.

This method will cause the layer’s state to be built, if that has not happened before. This requires that the layer will later be used with inputs that match the input shape provided here.

Parameters **input_shape** – Shape tuple (tuple of integers) or *tf.TensorShape*, or structure of shape tuples / *tf.TensorShape* instances (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns A *tf.TensorShape* instance or structure of *tf.TensorShape* instances.

`compute_output_signature(input_signature)`

Compute the output tensor signature of the layer based on the inputs.

Unlike a *TensorShape* object, a *TensorSpec* object contains both shape and dtype information for a tensor. This method allows layers to provide output dtype information if it is different from the input dtype. For any layer that doesn’t implement this function, the framework will fall back to use *compute_output_shape*, and will assume that the output dtype matches the input dtype.

Parameters **input_signature** – Single *TensorSpec* or nested structure of *TensorSpec* objects, describing a candidate input for the layer.

Returns

Single *TensorSpec* or nested structure of *TensorSpec* objects, describing how the layer would transform the provided input.

Raises **TypeError** – If *input_signature* contains a non-*TensorSpec* object.

count_params()

Count the total number of scalars composing the weights.

Returns An integer count.

Raises ValueError – if the layer isn’t yet built (in which case its weights aren’t yet defined).

property dtype

The dtype of the layer weights.

This is equivalent to *Layer.dtype_policy.variable_dtype*. Unless mixed precision is used, this is the same as *Layer.compute_dtype*, the dtype of the layer’s computations.

property dtype_policy

The dtype policy associated with this layer.

This is an instance of a *tf.keras.mixed_precision.Policy*.

property dynamic

Whether the layer is dynamic (eager-only); set in the constructor.

finalize_state()

Finalizes the layers state after updating layer weights.

This function can be subclassed in a layer and will be called after updating a layer weights. It can be overridden to finalize any additional layer state after a weight update.

This function will be called after weights of a layer have been restored from a loaded model.

classmethod from_config(config)

Creates a layer from its config.

This method is the reverse of *get_config*, capable of instantiating the same layer from the config dictionary. It does not handle layer connectivity (handled by Network), nor weights (handled by *set_weights*).

Parameters config – A Python dictionary, typically the output of *get_config*.

Returns A layer instance.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns Python dictionary.

get_input_at(node_index)

Retrieves the input tensor(s) of a layer at a given node.

Parameters node_index – Integer, index of the node from which to retrieve the attribute. E.g. *node_index=0* will correspond to the first input node of the layer.

Returns A tensor (or list of tensors if the layer has multiple inputs).

Raises RuntimeError – If called in Eager mode.

get_input_mask_at(node_index)

Retrieves the input mask tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A mask tensor (or list of tensors if the layer has multiple inputs).

`get_input_shape_at(node_index)`

Retrieves the input shape(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A shape tuple (or list of shape tuples if the layer has multiple inputs).

Raises `RuntimeError` – If called in Eager mode.

`get_output_at(node_index)`

Retrieves the output tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first output node of the layer.

Returns A tensor (or list of tensors if the layer has multiple outputs).

Raises `RuntimeError` – If called in Eager mode.

`get_output_mask_at(node_index)`

Retrieves the output mask tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A mask tensor (or list of tensors if the layer has multiple outputs).

`get_output_shape_at(node_index)`

Retrieves the output shape(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A shape tuple (or list of shape tuples if the layer has multiple outputs).

Raises `RuntimeError` – If called in Eager mode.

`get_weights()`

Returns the current weights of the layer, as NumPy arrays.

The weights of a layer represent the state of the layer. This function returns both trainable and non-trainable weight values associated with this layer as a list of NumPy arrays, which can in turn be used to load state into similarly parameterized layers.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
```

(continues on next page)

(continued from previous page)

```
>>> layer_b.get_weights()
[array([[2.],
       [2.],
       [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
```

Returns Weights values as a list of NumPy arrays.

property inbound_nodes

Return Functional API nodes upstream of this layer.

property input

Retrieves the input tensor(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer.

Returns Input tensor or list of input tensors.

Raises

- **RuntimeError** – If called in Eager mode.
- **AttributeError** – If no inbound nodes are found.

property input_mask

Retrieves the input mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Input mask tensor (potentially None) or list of input mask tensors.

Raises

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

property input_shape

Retrieves the input shape(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer, or if all inputs have the same shape.

Returns Input shape, as an integer shape tuple (or list of shape tuples, one tuple per input tensor).

Raises

- **AttributeError** – if the layer has no defined input_shape.
- **RuntimeError** – if called in Eager mode.

property input_spec

InputSpec instance(s) describing the input format for this layer.

When you create a layer subclass, you can set *self.input_spec* to enable the layer to run input compatibility checks when it is called. Consider a *Conv2D* layer: it can only be called on a single input tensor of rank 4. As such, you can set, in *__init__()*:

```
`python self.input_spec = tf.keras.layers.InputSpec(ndim=4)`
```

Now, if you try to call the layer on an input that isn't rank 4 (for instance, an input of shape (2,), it will raise a nicely-formatted error:

```
`ValueError: Input 0 of layer conv2d is incompatible with the layer: expected ndim=4, found ndim=1. Full shape received: [2]`
```

Input checks that can be specified via *input_spec* include:

- Structure (e.g. a single input, a list of 2 inputs, etc)
- Shape
- Rank (ndim)
- Dtype

For more information, see *tf.keras.layers.InputSpec*.

Returns A *tf.keras.layers.InputSpec* instance, or nested structure thereof.

property losses

List of losses added using the *add_loss()* API.

Variable regularization tensors are created when this property is accessed, so it is eager safe: accessing *losses* under a *tf.GradientTape* will propagate gradients back to the corresponding variables.

Examples:

```
>>> class MyLayer(tf.keras.layers.Layer):
...     def call(self, inputs):
...         self.add_loss(tf.abs(tf.reduce_mean(inputs)))
...     return inputs
>>> l = MyLayer()
>>> l(np.ones((10, 1)))
>>> l.losses
[1.0]
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> x = tf.keras.layers.Dense(10)(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Activity regularization.
>>> len(model.losses)
0
>>> model.add_loss(tf.abs(tf.reduce_mean(x)))
>>> len(model.losses)
1
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> d = tf.keras.layers.Dense(10, kernel_initializer='ones')
>>> x = d(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Weight regularization.
>>> model.add_loss(lambda: tf.reduce_mean(d.kernel))
>>> model.losses
[<tf.Tensor: shape=(), dtype=float32, numpy=1.0>]
```

Returns A list of tensors.

property metrics

List of metrics added using the *add_metric()* API.

Example:

```
>>> input = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2)
>>> output = d(input)
>>> d.add_metric(tf.reduce_max(output), name='max')
>>> d.add_metric(tf.reduce_min(output), name='min')
>>> [m.name for m in d.metrics]
['max', 'min']
```

Returns A list of *Metric* objects.

property name

Name of the layer (string), set in the constructor.

property name_scope

Returns a *tf.name_scope* instance for this class.

property non_trainable_variables

Sequence of non-trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don't expect the return value to change.

Returns A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

property non_trainable_weights

List of all non-trainable weights tracked by this layer.

Non-trainable weights are *not* updated during training. They are expected to be updated manually in *call()*.

Returns A list of non-trainable variables.

property outbound_nodes

Return Functional API nodes downstream of this layer.

property output

Retrieves the output tensor(s) of a layer.

Only applicable if the layer has exactly one output, i.e. if it is connected to one incoming layer.

Returns Output tensor or list of output tensors.

Raises

- **AttributeError** – if the layer is connected to more than one incoming layers.
- **RuntimeError** – if called in Eager mode.

property output_mask

Retrieves the output mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Output mask tensor (potentially None) or list of output mask tensors.

Raises

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

property output_shape

Retrieves the output shape(s) of a layer.

Only applicable if the layer has one output, or if all outputs have the same shape.

Returns Output shape, as an integer shape tuple (or list of shape tuples, one tuple per output tensor).

Raises

- **AttributeError** – if the layer has no defined output shape.
- **RuntimeError** – if called in Eager mode.

regularization(*lbd_w*: float = 1.0, *lbd_b*: float = 1.0) → tensorflow.python.framework.ops.Tensor

set_weights(*weights*)

Sets the weights of the layer, from NumPy arrays.

The weights of a layer represent the state of the layer. This function sets the weight values from numpy arrays. The weight values should be passed in the order they are created by the layer. Note that the layer's weights must be instantiated before calling this function, by calling the layer.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
       [2.],
       [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
```

Parameters **weights** – a list of NumPy arrays. The number of arrays and their shape must match number of the dimensions of the weights of the layer (i.e. it should match the output of *get_weights*).

Raises **ValueError** – If the provided weights list does not match the layer's specifications.

property stateful**property submodules**

Sequence of all sub-modules.

Submodules are modules which are properties of this module, or found as properties of modules which are properties of this module (and so on).

```
>>> a = tf.Module()
>>> b = tf.Module()
>>> c = tf.Module()
>>> a.b = b
>>> b.c = c
>>> list(a.submodules) == [b, c]
True
>>> list(b.submodules) == [c]
True
>>> list(c.submodules) == []
True
```

Returns A sequence of all submodules.

property supports_masking

Whether this layer supports computing a mask using *compute_mask*.

property trainable

property trainable_variables

Sequence of trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don't expect the return value to change.

Returns A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

property trainable_weights

List of all trainable weights tracked by this layer.

Trainable weights are updated via gradient descent during training.

Returns A list of trainable variables.

property updates

property variable_dtype

Alias of *Layer.dtype*, the dtype of the weights.

property variables

Returns the list of all layer variables/weights.

Alias of *self.weights*.

Note: This will not track the weights of nested *tf.Modules* that are not themselves Keras layers.

Returns A list of variables.

property weights

Returns the list of all layer variables/weights.

Returns A list of variables.

classmethod with_name_scope(method)

Decorator to automatically enter the module name scope.

```
>>> class MyModule(tf.Module):
...     @tf.Module.with_name_scope
```

(continues on next page)

(continued from previous page)

```

...     def __call__(self, x):
...         if not hasattr(self, 'w'):
...             self.w = tf.Variable(tf.random.normal([x.shape[1], 3]))
...         return tf.matmul(x, self.w)

```

Using the above module would produce `tf.Variable`'s and `tf.Tensor`'s whose names included the module name:

```

>>> mod = MyModule()
>>> mod(tf.ones([1, 2]))
<tf.Tensor: shape=(1, 3), dtype=float32, numpy=..., dtype=float32>
>>> mod.w
<tf.Variable 'my_module/Variable:0' shape=(2, 3) dtype=float32,
numpy=..., dtype=float32>

```

Parameters `method` – The method to wrap.

Returns The original method wrapped such that it enters the module's name scope.

```

class tensorcircuit.applications.van.NMF(*args, **kwargs)
Bases: keras.engine.training.Model

_init_(spin_channel: int, *dimensions: int, _dtype: tensorflow.python.framework.dtypes.DType =
          tf.float32, probamp: Optional[tensorflow.python.framework.ops.Tensor] = None)

property activity_regularizer
    Optional regularizer function for the output of this layer.

add_loss(losses, **kwargs)
    Add loss tensor(s), potentially dependent on layer inputs.

Some losses (for instance, activity regularization losses) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs a and b, some entries in layer.losses may be dependent on a and some on b. This method automatically keeps track of dependencies.

This method can be used inside a subclassed layer or model's call function, in which case losses should be a Tensor or list of Tensors.

Example:
```
python class MyLayer(tf.keras.layers.Layer):

 def call(self, inputs): self.add_loss(tf.abs(tf.reduce_mean(inputs))) return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any loss Tensors passed to this Model must be symbolic and be able to be traced back to the model's Input's. These losses become part of the model's topology and are tracked in get_config.
```

Example:

```

`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.
Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.
Model(inputs, outputs) # Activity regularization. model.add_loss(tf.abs(tf.
reduce_mean(x)))`
```

If this is not the case for your loss (if, for example, your loss references a *Variable* of one of the model's layers), you can wrap your loss in a zero-argument lambda. These losses are not tracked as part of the model's topology since they can't be serialized.

Example:

```
`python inputs = tf.keras.Input(shape=(10,)) d = tf.keras.layers.Dense(10) x = d(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) # Weight regularization. model.add_loss(lambda: tf.reduce_mean(d.kernel))`
```

Parameters

- **losses** – Loss tensor, or list/tuple of tensors. Rather than tensors, losses may also be zero-argument callables which create a loss tensor.
- ****kwargs** – Used for backwards compatibility only.

add_metric(*value*, *name*=None, ***kwargs*)

Adds metric tensor to the layer.

This method can be used inside the *call()* method of a subclassed layer or model.

```
```python class MyMetricLayer(tf.keras.layers.Layer):
```

```
 def __init__(self): super(MyMetricLayer, self).__init__(name='my_metric_layer') self.mean = tf.keras.metrics.Mean(name='metric_1')

 def call(self, inputs): self.add_metric(self.mean(inputs)) self.add_metric(tf.reduce_sum(inputs), name='metric_2') return inputs
```

```
```
```

This method can also be called directly on a Functional Model during construction. In this case, any tensor passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These metrics become part of the model's topology and are tracked when you save the model via 'save()'*.

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) model.add_metric(math_ops.reduce_sum(x), name='metric_1')`
```

Note: Calling *add_metric()* with the result of a metric object on a Functional Model, as shown in the example below, is not supported. This is because we cannot trace the metric result tensor back to the model's inputs.

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) model.add_metric(tf.keras.metrics.Mean()(x), name='metric_1')`
```

Parameters

- **value** – Metric tensor.
- **name** – String metric name.
- ****kwargs** – Additional keyword arguments for backward compatibility. Accepted values: *aggregation* - When the *value* tensor provided is not the result of calling a *keras.Metric* instance, it will be aggregated by default using a *keras.Metric.Mean*.

add_update(*updates*)

Add update op(s), potentially dependent on layer inputs.

Weight updates (for instance, the updates of the moving mean and variance in a BatchNormalization layer) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs a and b , some entries in `layer.updates` may be dependent on a and some on b . This method automatically keeps track of dependencies.

This call is ignored when eager execution is enabled (in that case, variable updates are run on the fly and thus do not need to be tracked for later execution).

Parameters `updates` – Update op, or list/tuple of update ops, or zero-arg callable that returns an update op. A zero-arg callable should be passed in order to disable running the updates by setting `trainable=False` on this Layer, when executing in Eager mode.

`add_variable(*args, **kwargs)`

Deprecated, do NOT use! Alias for `add_weight`.

`add_weight(name=None, shape=None, dtype=None, initializer=None, regularizer=None, trainable=None, constraint=None, use_resource=None, synchronization=VariableSynchronization.AUTO, aggregation=VariableAggregationV2.NONE, **kwargs)`

Adds a new variable to the layer.

Parameters

- **name** – Variable name.
- **shape** – Variable shape. Defaults to scalar if unspecified.
- **dtype** – The type of the variable. Defaults to `self.dtype`.
- **initializer** – Initializer instance (callable).
- **regularizer** – Regularizer instance (callable).
- **trainable** – Boolean, whether the variable should be part of the layer’s “trainable_variables” (e.g. variables, biases) or “non_trainable_variables” (e.g. BatchNorm mean and variance). Note that `trainable` cannot be `True` if `synchronization` is set to `ON_READ`.
- **constraint** – Constraint instance (callable).
- **use_resource** – Whether to use a `ResourceVariable` or not. See [this guide](https://www.tensorflow.org/guide/migrate/tf1_vs_tf2#resourcevariables_instead_of_referencevariables)
for more information.
- **synchronization** – Indicates when a distributed a variable will be aggregated. Accepted values are constants defined in the class `tf.VariableSynchronization`. By default the synchronization is set to `AUTO` and the current `DistributionStrategy` chooses when to synchronize. If `synchronization` is set to `ON_READ`, `trainable` must not be set to `True`.
- **aggregation** – Indicates how a distributed variable will be aggregated. Accepted values are constants defined in the class `tf.VariableAggregation`.
- ****kwargs** – Additional keyword arguments. Accepted values are `getter`, `collections`, `experimental_autocast` and `caching_device`.

Returns The variable created.

Raises `ValueError` – When giving unsupported dtype and no initializer or when trainable has been set to True with synchronization set as `ON_READ`.

`build(input_shape)`

Builds the model based on input shapes received.

This is to be used for subclassed models, which do not know at instantiation time what their inputs look like.

This method only exists for users who want to call `model.build()` in a standalone way (as a substitute for calling the model on real data to build it). It will never be called by the framework (and thus it will never throw unexpected errors in an unrelated workflow).

Parameters `input_shape` – Single tuple, `TensorShape` instance, or list/dict of shapes, where shapes are tuples, integers, or `TensorShape` instances.

Raises

- `ValueError` –

1. In case of invalid user-provided data (not of type tuple, list, `TensorShape`, or dict). 2. If the model requires call arguments that are agnostic to the input shapes (positional or keyword arg in call signature). 3. If not all layers were properly built. 4. If float type inputs are not supported within the layers.

- **In each of these cases, the user should build their model by calling –**

- **it on real tensor data.** –

`call(inputs: Optional[tensorflow.python.framework.ops.Tensor] = None) → tensorflow.python.framework.ops.Tensor`

Calls the model on new inputs and returns the outputs as tensors.

In this case `call()` just reapplies all ops in the graph to the new inputs (e.g. build a new computational graph from the provided inputs).

Note: This method should not be called directly. It is only meant to be overridden when subclassing `tf.keras.Model`. To call a model on an input, always use the `__call__()` method, i.e. `model(inputs)`, which relies on the underlying `call()` method.

Parameters

- `inputs` – Input tensor, or dict/list/tuple of input tensors.
- `training` – Boolean or boolean scalar tensor, indicating whether to run the *Network* in training mode or inference mode.
- `mask` – A mask or list of masks. A mask can be either a boolean tensor or None (no mask). For more details, check the guide [here](https://www.tensorflow.org/guide/keras/masking_and_padding).

Returns A tensor if there is a single output, or a list of tensors if there are more than one outputs.

`compile(optimizer='rmsprop', loss=None, metrics=None, loss_weights=None, weighted_metrics=None, run_eagerly=None, steps_per_execution=None, jit_compile=None, **kwargs)`

Configures the model for training.

Example:

```
```python
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),

```

```
loss=tf.keras.losses.BinaryCrossentropy(),
metrics=[tf.keras.metrics.BinaryAccuracy(),
tf.keras.metrics.FalseNegatives()])
```

```

Parameters

- **optimizer** – String (name of optimizer) or optimizer instance. See `tf.keras.optimizers`.
- **loss** – Loss function. May be a string (name of loss function), or a `tf.keras.losses.Loss` instance. See `tf.keras.losses`. A loss function is any callable with the signature `loss = fn(y_true, y_pred)`, where `y_true` are the ground truth values, and `y_pred` are the model’s predictions. `y_true` should have shape `(batch_size, d0, .. dN)` (except in the case of sparse loss functions such as sparse categorical crossentropy which expects integer arrays of shape `(batch_size, d0, .. dN-1)`). `y_pred` should have shape `(batch_size, d0, .. dN)`. The loss function should return a float tensor. If a custom `Loss` instance is used and reduction is set to `None`, return value has shape `(batch_size, d0, .. dN-1)` i.e. per-sample or per-timestep loss values; otherwise, it is a scalar. If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses, unless `loss_weights` is specified.
- **metrics** – List of metrics to be evaluated by the model during training and testing. Each of this can be a string (name of a built-in function), function or a `tf.keras.metrics.Metric` instance. See `tf.keras.metrics`. Typically you will use `metrics=['accuracy']`. A function is any callable with the signature `result = fn(y_true, y_pred)`. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as `metrics={'output_a':'accuracy', 'output_b':['accuracy', 'mse']}`. You can also pass a list to specify a metric or a list of metrics for each output, such as `metrics=[[['accuracy'], ['accuracy', 'mse']]` or `metrics=[{'accuracy', 'accuracy', 'mse'}]`. When you pass the strings ‘accuracy’ or ‘acc’, we convert this to one of `tf.keras.metrics.BinaryAccuracy`, `tf.keras.metrics.CategoricalAccuracy`, `tf.keras.metrics.SparseCategoricalAccuracy` based on the shapes of the targets and of the model output. We do a similar conversion for the strings ‘crossentropy’ and ‘ce’ as well. The metrics passed here are evaluated without sample weighting; if you would like sample weighting to apply, you can specify your metrics via the `weighted_metrics` argument instead.
- **loss_weights** – Optional list or dictionary specifying scalar coefficients (Python floats) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the *weighted sum* of all individual losses, weighted by the `loss_weights` coefficients. If a list, it is expected to have a 1:1 mapping to the model’s outputs. If a dict, it is expected to map output names (strings) to scalar coefficients.
- **weighted_metrics** – List of metrics to be evaluated and weighted by `sample_weight` or `class_weight` during training and testing.
- **run_eagerly** – Bool. Defaults to `False`. If `True`, this `Model`’s logic will not be wrapped in a `tf.function`. Recommended to leave this as `None` unless your `Model` cannot be run inside a `tf.function`. `run_eagerly=True` is not supported when using `tf.distribute.experimental.ParameterServerStrategy`.
- **steps_per_execution** – Int. Defaults to 1. The number of batches to run during each `tf.function` call. Running multiple batches inside a single `tf.function` call can greatly improve performance on TPUs or small models with a large Python overhead. At most, one full epoch will be run each execution. If a number larger than the size of the epoch is passed, the execution will be truncated to the size of the epoch. Note that if `steps_per_execution` is set to `N`, `Callback.on_batch_begin` and `Callback.on_batch_end` methods will only be called every `N` batches (i.e. before/after each `tf.function` execution).

- **`jit_compile`** – If *True*, compile the model training step with XLA. [XLA](<https://www.tensorflow.org/xla>) is an optimizing compiler for machine learning. *jit_compile* is not enabled for by default. This option cannot be enabled with *run_eagerly=True*. Note that *jit_compile=True* may not necessarily work for all models. For more information on supported operations please refer to the [XLA documentation](<https://www.tensorflow.org/xla>). Also refer to [known XLA issues](https://www.tensorflow.org/xla/known_issues) for more details.
- **`**kwargs`** – Arguments supported for backwards compatibility only.

property `compute_dtype`

The dtype of the layer's computations.

This is equivalent to *Layer.dtype_policy.compute_dtype*. Unless mixed precision is used, this is the same as *Layer.dtype*, the dtype of the weights.

Layers automatically cast their inputs to the compute dtype, which causes computations and the output to be in the compute dtype as well. This is done by the base Layer class in *Layer.__call__*, so you do not have to insert these casts if implementing your own layer.

Layers often perform certain internal computations in higher precision when *compute_dtype* is float16 or bfloat16 for numeric stability. The output will still typically be float16 or bfloat16 in such cases.

Returns The layer's compute dtype.

`compute_loss(x=None, y=None, y_pred=None, sample_weight=None)`

Compute the total loss, validate it, and return it.

Subclasses can optionally override this method to provide custom loss computation logic.

Example: ````python class MyModel(tf.keras.Model):

```
def __init__(self, *args, **kwargs): super(MyModel, self).__init__(*args, **kwargs)
    self.loss_tracker = tf.keras.metrics.Mean(name='loss')

def compute_loss(self, x, y, y_pred, sample_weight): loss =
    tf.reduce_mean(tf.math.squared_difference(y_pred, y)) loss += tf.add_n(self.losses)
    self.loss_tracker.update_state(loss) return loss

def reset_metrics(self): self.loss_tracker.reset_states()

@property def metrics(self):
    return [self.loss_tracker]

tensors = tf.random.uniform((10, 10), tf.random.uniform((10,)) dataset =
tf.data.Dataset.from_tensor_slices(tensors).repeat().batch(1)

inputs = tf.keras.layers.Input(shape=(10,), name='my_input') outputs = tf.keras.layers.Dense(10)(inputs)
model = MyModel(inputs, outputs) model.add_loss(tf.reduce_sum(outputs))

optimizer = tf.keras.optimizers.SGD() model.compile(optimizer, loss='mse', steps_per_execution=10)
model.fit(dataset, epochs=2, steps_per_epoch=10) print('My custom loss:', model.loss_tracker.result().numpy()) ````
```

Parameters

- **`x`** – Input data.
- **`y`** – Target data.
- **`y_pred`** – Predictions returned by the model (output of *model(x)*)
- **`sample_weight`** – Sample weights for weighting the loss function.

Returns The total loss as a *tf.Tensor*, or *None* if no loss results (which is the case when called by *Model.test_step*).

compute_mask(*inputs*, *mask*=*None*)

Computes an output mask tensor.

Parameters

- **inputs** – Tensor or list of tensors.
- **mask** – Tensor or list of tensors.

Returns

None or a tensor (or list of tensors), one per output tensor of the layer).

compute_metrics(*x*, *y*, *y_pred*, *sample_weight*)

Update metric states and collect all metrics to be returned.

Subclasses can optionally override this method to provide custom metric updating and collection logic.

Example: ````python class MyModel(tf.keras.Sequential):`

```
def compute_metrics(self, x, y, y_pred, sample_weight):
    # This super call updates self.compiled_metrics and returns # results for all metrics
    # listed in self.metrics. metric_results = super(MyModel, self).compute_metrics(
        x, y, y_pred, sample_weight)

    # Note that self.custom_metric is not listed in self.metrics.
    self.custom_metric.update_state(x, y, y_pred, sample_weight) metric_results['custom_metric_name'] = self.custom_metric.result() return metric_results
```

`````

**Parameters**

- **x** – Input data.
- **y** – Target data.
- **y\_pred** – Predictions returned by the model (output of *model.call(x)*)
- **sample\_weight** – Sample weights for weighting the loss function.

**Returns** A *dict* containing values that will be passed to *tf.keras.callbacks.CallbackList.on\_train\_batch\_end()*. Typically, the values of the metrics listed in *self.metrics* are returned. Example: `{'loss': 0.2, 'accuracy': 0.7}`.

**compute\_output\_shape(*input\_shape*)**

Computes the output shape of the layer.

This method will cause the layer's state to be built, if that has not happened before. This requires that the layer will later be used with inputs that match the input shape provided here.

**Parameters** **input\_shape** – Shape tuple (tuple of integers) or *tf.TensorShape*, or structure of shape tuples / *tf.TensorShape* instances (one per output tensor of the layer). Shape tuples can include *None* for free dimensions, instead of an integer.

**Returns** A *tf.TensorShape* instance or structure of *tf.TensorShape* instances.

**compute\_output\_signature(*input\_signature*)**

Compute the output tensor signature of the layer based on the inputs.

Unlike a `TensorShape` object, a `TensorSpec` object contains both shape and dtype information for a tensor. This method allows layers to provide output dtype information if it is different from the input dtype. For any layer that doesn't implement this function, the framework will fall back to use `compute_output_shape`, and will assume that the output dtype matches the input dtype.

**Parameters** `input_signature` – Single `TensorSpec` or nested structure of `TensorSpec` objects, describing a candidate input for the layer.

#### Returns

**Single `TensorSpec` or nested structure of `TensorSpec` objects**, describing how the layer would transform the provided input.

**Raises** `TypeError` – If `input_signature` contains a non-`TensorSpec` object.

#### `count_params()`

Count the total number of scalars composing the weights.

**Returns** An integer count.

**Raises** `ValueError` – if the layer isn't yet built (in which case its weights aren't yet defined).

#### `property distribute_reduction_method`

The method employed to reduce per-replica values during training.

Unless specified, the value “auto” will be assumed, indicating that the reduction strategy should be chosen based on the current running environment. See `reduce_per_replica` function for more details.

#### `property distribute_strategy`

The `tf.distribute.Strategy` this model was created under.

#### `property dtype`

The dtype of the layer weights.

This is equivalent to `Layer.dtype_policy.variable_dtype`. Unless mixed precision is used, this is the same as `Layer.compute_dtype`, the dtype of the layer’s computations.

#### `property dtype_policy`

The dtype policy associated with this layer.

This is an instance of a `tf.keras.mixed_precision.Policy`.

#### `property dynamic`

Whether the layer is dynamic (eager-only); set in the constructor.

`evaluate(x=None, y=None, batch_size=None, verbose='auto', sample_weight=None, steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiprocessing=False, return_dict=False, **kwargs)`

Returns the loss value & metrics values for the model in test mode.

Computation is done in batches (see the `batch_size` arg.)

#### Parameters

- `x` – Input data. It could be:
  - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
  - A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
  - A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
  - A `tf.data` dataset. Should return a tuple of either `(inputs, targets)` or `(inputs, targets, sample_weights)`.

- A generator or `keras.utils.Sequence` returning `(inputs, targets)` or `(inputs, targets, sample_weights)`.

A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given in the *Unpacking behavior for iterator-like inputs* section of `Model.fit`.

- **y** – Target data. Like the input data `x`, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with `x` (you cannot have Numpy inputs and tensor targets, or inversely). If `x` is a dataset, generator or `keras.utils.Sequence` instance, `y` should not be specified (since targets will be obtained from the iterator/dataset).
- **batch\_size** – Integer or `None`. Number of samples per batch of computation. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` if your data is in the form of a dataset, generators, or `keras.utils.Sequence` instances (since they generate batches).
- **verbose** – “`auto`”, 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line. “`auto`” defaults to 1 for most cases, and to 2 when used with *ParameterServerStrategy*. Note that the progress bar is not particularly useful when logged to a file, so `verbose=2` is recommended when not running interactively (e.g. in a production environment).
- **sample\_weight** – Optional Numpy array of weights for the test samples, used for weighting the loss function. You can either pass a flat (1D) Numpy array with the same length as the input samples

**(1:1 mapping between weights and samples), or in the case of** temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample. This argument is not supported when `x` is a dataset, instead pass sample weights as the third element of `x`.

- **steps** – Integer or `None`. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of `None`. If `x` is a `tf.data` dataset and `steps` is None, ‘evaluate’ will run until the dataset is exhausted. This argument is not supported with array inputs.
- **callbacks** – List of `keras.callbacks.Callback` instances. List of callbacks to apply during evaluation. See [callbacks]([https://www.tensorflow.org/api\\_docs/python/tf/keras/callbacks](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks)).
- **max\_queue\_size** – Integer. Used for generator or `keras.utils.Sequence` input only. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers** – Integer. Used for generator or `keras.utils.Sequence` input only. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1.
- **use\_multiprocessing** – Boolean. Used for generator or `keras.utils.Sequence` input only. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can’t be passed easily to children processes.
- **return\_dict** – If `True`, loss and metric results are returned as a dict, with each key being the name of the metric. If `False`, they are returned as a list.
- **\*\*kwargs** – Unused at this time.

See the discussion of *Unpacking behavior for iterator-like inputs* for *Model.fit*.

**Returns** Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute *model.metrics\_names* will give you the display labels for the scalar outputs.

**Raises RuntimeError** – If *model.evaluate* is wrapped in a *tf.function*.

**evaluate\_generator**(*generator*, *steps=None*, *callbacks=None*, *max\_queue\_size=10*, *workers=1*, *use\_multiprocessing=False*, *verbose=0*)

Evaluates the model on a data generator.

**DEPRECATED:** *Model.evaluate* now supports generators, so there is no longer any need to use this endpoint.

**finalize\_state()**

Finalizes the layers state after updating layer weights.

This function can be subclassed in a layer and will be called after updating a layer weights. It can be overridden to finalize any additional layer state after a weight update.

This function will be called after weights of a layer have been restored from a loaded model.

**fit**(*x=None*, *y=None*, *batch\_size=None*, *epochs=1*, *verbose='auto'*, *callbacks=None*, *validation\_split=0.0*, *validation\_data=None*, *shuffle=True*, *class\_weight=None*, *sample\_weight=None*, *initial\_epoch=0*, *steps\_per\_epoch=None*, *validation\_steps=None*, *validation\_batch\_size=None*, *validation\_freq=1*, *max\_queue\_size=10*, *workers=1*, *use\_multiprocessing=False*)

Trains the model for a fixed number of epochs (iterations on a dataset).

### Parameters

- **x** – Input data. It could be:
  - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
  - A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
  - A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
  - A *tf.data* dataset. Should return a tuple of either (*inputs*, *targets*) or (*inputs*, *targets*, *sample\_weights*).
  - A generator or *keras.utils.Sequence* returning (*inputs*, *targets*) or (*inputs*, *targets*, *sample\_weights*).
  - A *tf.keras.utils.experimental.DatasetCreator*, which wraps a callable that takes a single argument of type *tf.distribute.InputContext*, and returns a *tf.data.Dataset*. *DatasetCreator* should be used when users prefer to specify the per-replica batching and sharding logic for the *Dataset*. See *tf.keras.utils.experimental.DatasetCreator* doc for more information.

A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given below. If these include *sample\_weights* as a third component, note that sample weighting applies to the *weighted\_metrics* argument but not the *metrics* argument in *compile()*. If using *tf.distribute.experimental.ParameterServerStrategy*, only *DatasetCreator* type is supported for *x*.

- **y** – Target data. Like the input data *x*, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with *x* (you cannot have Numpy inputs and tensor

targets, or inversely). If  $x$  is a dataset, generator, or `keras.utils.Sequence` instance,  $y$  should not be specified (since targets will be obtained from  $x$ ).

- **batch\_size** – Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` if your data is in the form of datasets, generators, or `keras.utils.Sequence` instances (since they generate batches).
  - **epochs** – Integer. Number of epochs to train the model. An epoch is an iteration over the entire  $x$  and  $y$  data provided (unless the `steps_per_epoch` flag is set to something other than `None`). Note that in conjunction with `initial_epoch`, `epochs` is to be understood as “final epoch”. The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.
  - **verbose** – ‘auto’, 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch. ‘auto’ defaults to 1 for most cases, but 2 when used with `ParameterServerStrategy`. Note that the progress bar is not particularly useful when logged to a file, so `verbose=2` is recommended when not running interactively (eg, in a production environment).
  - **callbacks** – List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See `tf.keras.callbacks`. Note `tf.keras.callbacks.ProgbarLogger` and `tf.keras.callbacks.History` callbacks are created automatically and need not be passed into `model.fit`. `tf.keras.callbacks.ProgbarLogger` is created or not based on `verbose` argument to `model.fit`. Callbacks with batch-level calls are currently unsupported with `tf.distribute.experimental.ParameterServerStrategy`, and users are advised to implement epoch-level calls instead with an appropriate `steps_per_epoch` value.
  - **validation\_split** – Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the  $x$  and  $y$  data provided, before shuffling. This argument is not supported when  $x$  is a dataset, generator or `keras.utils.Sequence` instance. If both `validation_data` and `validation_split` are provided, `validation_data` will override `validation_split`. `validation_split` is not yet supported with `tf.distribute.experimental.ParameterServerStrategy`.
  - **validation\_data** – Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. Thus, note the fact that the validation loss of data provided using `validation_split` or `validation_data` is not affected by regularization layers like noise and dropout. `validation_data` will override `validation_split`. `validation_data` could be:
    - A tuple  $(x_{\text{val}}, y_{\text{val}})$  of Numpy arrays or tensors.
    - A tuple  $(x_{\text{val}}, y_{\text{val}}, \text{val\_sample\_weights})$  of NumPy arrays.
    - A `tf.data.Dataset`.
    - A Python generator or `keras.utils.Sequence` returning  $(\text{inputs}, \text{targets})$  or  $(\text{inputs}, \text{targets}, \text{sample\_weights})$ .
- `validation_data` is not yet supported with `tf.distribute.experimental.ParameterServerStrategy`.
- **shuffle** – Boolean (whether to shuffle the training data before each epoch) or str (for ‘batch’). This argument is ignored when  $x$  is a generator or an object of `tf.data.Dataset`. ‘batch’ is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not `None`.

- **class\_weight** – Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to “pay more attention” to samples from an under-represented class.
- **sample\_weight** – Optional Numpy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape (*samples, sequence\_length*), to apply a different weight to every timestep of every sample. This argument is not supported when *x* is a dataset, generator, or *keras.utils.Sequence* instance, instead provide the sample\_weights as the third element of *x*. Note that sample weighting does not apply to metrics specified via the *metrics* argument in *compile()*. To apply sample weighting to your metrics, you can specify them via the *weighted\_metrics* in *compile()* instead.
- **initial\_epoch** – Integer. Epoch at which to start training (useful for resuming a previous training run).
- **steps\_per\_epoch** – Integer or *None*. Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as TensorFlow data tensors, the default *None* is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined. If *x* is a *tf.data* dataset, and ‘*steps\_per\_epoch*’ is *None*, the epoch will run until the input dataset is exhausted. When passing an infinitely repeating dataset, you must specify the *steps\_per\_epoch* argument. If *steps\_per\_epoch=-1* the training will run indefinitely with an infinitely repeating dataset. This argument is not supported with array inputs. When using *tf.distribute.experimental.ParameterServerStrategy*:
  - *steps\_per\_epoch=None* is not supported.
- **validation\_steps** – Only relevant if *validation\_data* is provided and is a *tf.data* dataset. Total number of steps (batches of samples) to draw before stopping when performing validation at the end of every epoch. If ‘*validation\_steps*’ is *None*, validation will run until the *validation\_data* dataset is exhausted. In the case of an infinitely repeated dataset, it will run into an infinite loop. If ‘*validation\_steps*’ is specified and only part of the dataset will be consumed, the evaluation will start from the beginning of the dataset at each epoch. This ensures that the same validation samples are used every time.
- **validation\_batch\_size** – Integer or *None*. Number of samples per validation batch. If unspecified, will default to *batch\_size*. Do not specify the *validation\_batch\_size* if your data is in the form of datasets, generators, or *keras.utils.Sequence* instances (since they generate batches).
- **validation\_freq** – Only relevant if validation data is provided. Integer or *collections.abc.Container* instance (e.g. list, tuple, etc.). If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. *validation\_freq=2* runs validation every 2 epochs. If a Container, specifies the epochs on which to run validation, e.g. *validation\_freq=[1, 2, 10]* runs validation at the end of the 1st, 2nd, and 10th epochs.
- **max\_queue\_size** – Integer. Used for generator or *keras.utils.Sequence* input only. Maximum size for the generator queue. If unspecified, *max\_queue\_size* will default to 10.
- **workers** – Integer. Used for generator or *keras.utils.Sequence* input only. Maximum number of processes to spin up when using process-based threading. If unspecified,

*workers* will default to 1.

- **use\_multiprocessing** – Boolean. Used for generator or *keras.utils.Sequence* input only. If *True*, use process-based threading. If unspecified, *use\_multiprocessing* will default to *False*. Note that because this implementation relies on multiprocessing, you should not pass non-pickleable arguments to the generator as they can't be passed easily to children processes.

#### Unpacking behavior for iterator-like inputs:

A common pattern is to pass a *tf.data.Dataset*, generator, or

*tf.keras.utils.Sequence* to the *x* argument of *fit*, which will in fact yield not only features (*x*) but optionally targets (*y*) and sample weights. Keras requires that the output of such iterator-likes be unambiguous. The iterator should return a tuple of length 1, 2, or 3, where the optional second and third elements will be used for *y* and *sample\_weight* respectively. Any other type provided will be wrapped in a length one tuple, effectively treating everything as '*x*'. When yielding dicts, they should still adhere to the top-level tuple structure. e.g. *{“x0”: x0, “x1”: x1}*, *y*). Keras will not attempt to separate features, targets, and weights from the keys of a single dict.

A notable unsupported data type is the namedtuple. The reason is

that it behaves like both an ordered datatype (tuple) and a mapping datatype (dict). So given a namedtuple of the form:

```
namedtuple("example_tuple", ["y", "x"])
```

it is ambiguous whether to reverse the order of the elements when interpreting the value. Even worse is a tuple of the form:

```
namedtuple("other_tuple", ["x", "y", "z"])
```

where it is unclear if the tuple was intended to be unpacked into *x*, *y*, and *sample\_weight* or passed through as a single element to *x*. As a result the data processing code will simply raise a *ValueError* if it encounters a namedtuple. (Along with instructions to remedy the issue.)

**Returns** A *History* object. Its *History.history* attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

#### Raises

- **RuntimeError** –
  1. If the model was never compiled or,
- **2. If model.fit is wrapped in tf.function.** –
- **ValueError** – In case of mismatch between the provided input data and what the model expects or when the input data is empty.

```
fit_generator(generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None,
validation_data=None, validation_steps=None, validation_freq=1, class_weight=None,
max_queue_size=10, workers=1, use_multiprocessing=False, shuffle=True,
initial_epoch=0)
```

Fits the model on data yielded batch-by-batch by a Python generator.

**DEPRECATED:** *Model.fit* now supports generators, so there is no longer any need to use this endpoint.

```
classmethod from_config(config, custom_objects=None)
```

Creates a layer from its config.

This method is the reverse of `get_config`, capable of instantiating the same layer from the config dictionary. It does not handle layer connectivity (handled by Network), nor weights (handled by `set_weights`).

**Parameters** `config` – A Python dictionary, typically the output of `get_config`.

**Returns** A layer instance.

### `get_config()`

Returns the config of the *Model*.

Config is a Python dictionary (serializable) containing the configuration of an object, which in this case is a *Model*. This allows the *Model* to be reinstated later (without its trained weights) from this configuration.

Note that `get_config()` does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Developers of subclassed *Model* are advised to override this method, and continue to update the dict from `super(MyModel, self).get_config()` to provide the proper configuration of this *Model*. The default config is an empty dict. Optionally, raise `NotImplementedError` to allow Keras to attempt a default serialization.

**Returns** Python dictionary containing the configuration of this *Model*.

### `get_input_at(node_index)`

Retrieves the input tensor(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g.  
`node_index=0` will correspond to the first input node of the layer.

**Returns** A tensor (or list of tensors if the layer has multiple inputs).

**Raises** `RuntimeError` – If called in Eager mode.

### `get_input_mask_at(node_index)`

Retrieves the input mask tensor(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g.  
`node_index=0` will correspond to the first time the layer was called.

**Returns** A mask tensor (or list of tensors if the layer has multiple inputs).

### `get_input_shape_at(node_index)`

Retrieves the input shape(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g.  
`node_index=0` will correspond to the first time the layer was called.

**Returns** A shape tuple (or list of shape tuples if the layer has multiple inputs).

**Raises** `RuntimeError` – If called in Eager mode.

### `get_layer(name=None, index=None)`

Retrieves a layer based on either its name (unique) or index.

If `name` and `index` are both provided, `index` will take precedence. Indices are based on order of horizontal graph traversal (bottom-up).

#### Parameters

- `name` – String, name of layer.
- `index` – Integer, index of layer.

**Returns** A layer instance.

**get\_metrics\_result()**

Returns the model's metrics values as a dict.

If any of the metric result is a dict (containing multiple metrics), each of them gets added to the top level returned dict of this method.

**Returns** A *dict* containing values of the metrics listed in *self.metrics*. Example: `{'loss': 0.2, 'accuracy': 0.7}`.

**get\_output\_at(node\_index)**

Retrieves the output tensor(s) of a layer at a given node.

**Parameters** **node\_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first output node of the layer.

**Returns** A tensor (or list of tensors if the layer has multiple outputs).

**Raises** **RuntimeError** – If called in Eager mode.

**get\_output\_mask\_at(node\_index)**

Retrieves the output mask tensor(s) of a layer at a given node.

**Parameters** **node\_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

**Returns** A mask tensor (or list of tensors if the layer has multiple outputs).

**get\_output\_shape\_at(node\_index)**

Retrieves the output shape(s) of a layer at a given node.

**Parameters** **node\_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

**Returns** A shape tuple (or list of shape tuples if the layer has multiple outputs).

**Raises** **RuntimeError** – If called in Eager mode.

**get\_weight\_paths()**

Retrieve all the variables and their paths for the model.

The variable path (string) is a stable key to indentify a *tf.Variable* instance owned by the model. It can be used to specify variable-specific configurations (e.g. DTensor, quantization) from a global view.

This method returns a dict with weight object paths as keys and the corresponding *tf.Variable* instances as values.

Note that if the model is a subclassed model and the weights haven't been initialized, an empty dict will be returned.

**Returns**

A dict where keys are variable paths and values are *tf.Variable* instances.

Example:

```
```python
class SubclassModel(tf.keras.Model):

    def __init__(self, name=None):
        super().__init__(name=name)
        self.d1 = tf.keras.layers.Dense(10)
        self.d2 = tf.keras.layers.Dense(20)

    def call(self, inputs):
        x = self.d1(inputs)
        return self.d2(x)

model = SubclassModel()
model(tf.zeros((10, 10)))
weight_paths = model.get_weight_paths() # weight_paths: { 'd1.kernel': model.d1.kernel, 'd1.bias': model.d1.bias, 'd2.kernel': model.d2.kernel, 'd2.bias': model.d2.bias }
```

```
# Functional model inputs = tf.keras.Input((10,), batch_size=10) x = tf.keras.layers.Dense(20, name='d1')(inputs) output = tf.keras.layers.Dense(30, name='d2')(x) model = tf.keras.Model(inputs, output) d1 = model.layers[1] d2 = model.layers[2] weight_paths = model.get_weight_paths() # weight_paths: # { # 'd1.kernel': d1.kernel, # 'd1.bias': d1.bias, # 'd2.kernel': d2.kernel, # 'd2.bias': d2.bias, # }
```

get_weights()

Retrieves the weights of the model.

Returns A flat list of Numpy arrays.

property inbound_nodes

Return Functional API nodes upstream of this layer.

property input

Retrieves the input tensor(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer.

Returns Input tensor or list of input tensors.

Raises

- **RuntimeError** – If called in Eager mode.

- **AttributeError** – If no inbound nodes are found.

property input_mask

Retrieves the input mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Input mask tensor (potentially None) or list of input mask tensors.

Raises

- **AttributeError** – if the layer is connected to

- **more than one incoming layers.** –

property input_shape

Retrieves the input shape(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer, or if all inputs have the same shape.

Returns Input shape, as an integer shape tuple (or list of shape tuples, one tuple per input tensor).

Raises

- **AttributeError** – if the layer has no defined input_shape.

- **RuntimeError** – if called in Eager mode.

property input_spec

InputSpec instance(s) describing the input format for this layer.

When you create a layer subclass, you can set *self.input_spec* to enable the layer to run input compatibility checks when it is called. Consider a *Conv2D* layer: it can only be called on a single input tensor of rank 4. As such, you can set, in *__init__()*:

```
`python self.input_spec = tf.keras.layers.InputSpec(ndim=4)`
```

Now, if you try to call the layer on an input that isn't rank 4 (for instance, an input of shape (2,), it will raise a nicely-formatted error:

```
` ValueError: Input 0 of layer conv2d is incompatible with the layer: expected
ndim=4, found ndim=1. Full shape received: [2] `
```

Input checks that can be specified via *input_spec* include:

- Structure (e.g. a single input, a list of 2 inputs, etc)
- Shape
- Rank (ndim)
- Dtype

For more information, see [tf.keras.layers.InputSpec](#).

Returns A [tf.keras.layers.InputSpec](#) instance, or nested structure thereof.

property layers

load_weights(filepath, by_name=False, skip_mismatch=False, options=None)

Loads all layer weights, either from a TensorFlow or an HDF5 weight file.

If *by_name* is False weights are loaded based on the network's topology. This means the architecture should be the same as when the weights were saved. Note that layers that don't have weights are not taken into account in the topological ordering, so adding or removing layers is fine as long as they don't have weights.

If *by_name* is True, weights are loaded into layers only if they share the same name. This is useful for fine-tuning or transfer-learning models where some of the layers have changed.

Only topological loading (*by_name=False*) is supported when loading weights from the TensorFlow format. Note that topological loading differs slightly between TensorFlow and HDF5 formats for user-defined classes inheriting from [tf.keras.Model](#): HDF5 loads based on a flattened list of weights, while the TensorFlow format loads based on the object-local names of attributes to which layers are assigned in the *Model*'s constructor.

Parameters

- **filepath** – String, path to the weights file to load. For weight files in TensorFlow format, this is the file prefix (the same as was passed to *save_weights*). This can also be a path to a SavedModel saved from *model.save*.
- **by_name** – Boolean, whether to load weights by name or by topological order. Only topological loading is supported for weight files in TensorFlow format.
- **skip_mismatch** – Boolean, whether to skip loading of layers where there is a mismatch in the number of weights, or a mismatch in the shape of the weight (only valid when *by_name=True*).
- **options** – Optional [tf.train.CheckpointOptions](#) object that specifies options for loading weights.

Returns

When loading a weight file in TensorFlow format, returns the same status object as [tf.train.Checkpoint.restore](#). When graph building, restore ops are run automatically as soon as the network is built (on first call for user-defined classes inheriting from *Model*, immediately if it is already built).

When loading weights in HDF5 format, returns *None*.

Raises

- **ImportError** – If *h5py* is not available and the weight file is in HDF5 format.
- **ValueError** – If *skip_mismatch* is set to *True* when *by_name* is *False*.

log_prob(sample: tensorflow.python.framework.ops.Tensor) → tensorflow.python.framework.ops.Tensor

property losses

List of losses added using the *add_loss()* API.

Variable regularization tensors are created when this property is accessed, so it is eager safe: accessing *losses* under a *tf.GradientTape* will propagate gradients back to the corresponding variables.

Examples:

```
>>> class MyLayer(tf.keras.layers.Layer):
...     def call(self, inputs):
...         self.add_loss(tf.abs(tf.reduce_mean(inputs)))
...         return inputs
>>> l = MyLayer()
>>> l(np.ones((10, 1)))
>>> l.losses
[1.0]
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> x = tf.keras.layers.Dense(10)(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Activity regularization.
>>> len(model.losses)
0
>>> model.add_loss(tf.abs(tf.reduce_mean(x)))
>>> len(model.losses)
1
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> d = tf.keras.layers.Dense(10, kernel_initializer='ones')
>>> x = d(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Weight regularization.
>>> model.add_loss(lambda: tf.reduce_mean(d.kernel))
>>> model.losses
[<tf.Tensor: shape=(), dtype=float32, numpy=1.0>]
```

Returns A list of tensors.

`make_predict_function(force=False)`

Creates a function that executes one step of inference.

This method can be overridden to support custom inference logic. This method is called by *Model.predict* and *Model.predict_on_batch*.

Typically, this method directly controls *tf.function* and *tf.distribute.Strategy* settings, and delegates the actual evaluation logic to *Model.predict_step*.

This function is cached the first time *Model.predict* or *Model.predict_on_batch* is called. The cache is cleared whenever *Model.compile* is called. You can skip the cache and generate again the function with *force=True*.

Parameters `force` – Whether to regenerate the predict function and skip the cached function if available.

Returns Function. The function created by this method should accept a *tf.data.Iterator*, and return the outputs of the *Model*.

make_test_function(*force=False*)

Creates a function that executes one step of evaluation.

This method can be overridden to support custom evaluation logic. This method is called by *Model.evaluate* and *Model.test_on_batch*.

Typically, this method directly controls *tf.function* and *tf.distribute.Strategy* settings, and delegates the actual evaluation logic to *Model.test_step*.

This function is cached the first time *Model.evaluate* or *Model.test_on_batch* is called. The cache is cleared whenever *Model.compile* is called. You can skip the cache and generate again the function with *force=True*.

Parameters **force** – Whether to regenerate the test function and skip the cached function if available.

Returns Function. The function created by this method should accept a *tf.data.Iterator*, and return a *dict* containing values that will be passed to *tf.keras.Callbacks.on_test_batch_end*.

make_train_function(*force=False*)

Creates a function that executes one step of training.

This method can be overridden to support custom training logic. This method is called by *Model.fit* and *Model.train_on_batch*.

Typically, this method directly controls *tf.function* and *tf.distribute.Strategy* settings, and delegates the actual training logic to *Model.train_step*.

This function is cached the first time *Model.fit* or *Model.train_on_batch* is called. The cache is cleared whenever *Model.compile* is called. You can skip the cache and generate again the function with *force=True*.

Parameters **force** – Whether to regenerate the train function and skip the cached function if available.

Returns Function. The function created by this method should accept a *tf.data.Iterator*, and return a *dict* containing values that will be passed to *tf.keras.Callbacks.on_train_batch_end*, such as `{'loss': 0.2, 'accuracy': 0.7}`.

property metrics

Returns the model's metrics added using *compile()*, *add_metric()* APIs.

Note: Metrics passed to *compile()* are available only after a *keras.Model* has been trained/evaluated on actual data.

Examples:

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> outputs = tf.keras.layers.Dense(2)(inputs)
>>> model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
>>> [m.name for m in model.metrics]
[]
```

```
>>> x = np.random.random((2, 3))
>>> y = np.random.randint(0, 2, (2, 2))
>>> model.fit(x, y)
>>> [m.name for m in model.metrics]
['loss', 'mae']
```

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2, name='out')
```

(continues on next page)

(continued from previous page)

```
>>> output_1 = d(inputs)
>>> output_2 = d(inputs)
>>> model = tf.keras.models.Model(
...     inputs=inputs, outputs=[output_1, output_2])
>>> model.add_metric(
...     tf.reduce_sum(output_2), name='mean', aggregation='mean')
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae", "acc"])
>>> model.fit(x, (y, y))
>>> [m.name for m in model.metrics]
['loss', 'out_loss', 'out_1_loss', 'out_mae', 'out_acc', 'out_1_mae',
'out_1_acc', 'mean']
```

property metrics_names

Returns the model's display labels for all outputs.

Note: *metrics_names* are available only after a *keras.Model* has been trained/evaluated on actual data.

Examples:

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> outputs = tf.keras.layers.Dense(2)(inputs)
>>> model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
>>> model.metrics_names
[]
```

```
>>> x = np.random.random((2, 3))
>>> y = np.random.randint(0, 2, (2, 2))
>>> model.fit(x, y)
>>> model.metrics_names
['loss', 'mae']
```

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2, name='out')
>>> output_1 = d(inputs)
>>> output_2 = d(inputs)
>>> model = tf.keras.models.Model(
...     inputs=inputs, outputs=[output_1, output_2])
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae", "acc"])
>>> model.fit(x, (y, y))
>>> model.metrics_names
['loss', 'out_loss', 'out_1_loss', 'out_mae', 'out_acc', 'out_1_mae',
'out_1_acc']
```

property name

Name of the layer (string), set in the constructor.

property name_scope

Returns a *tf.name_scope* instance for this class.

property non_trainable_variables

Sequence of non-trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don't expect the return value

to change.

Returns A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

property non_trainable_weights

List of all non-trainable weights tracked by this layer.

Non-trainable weights are *not* updated during training. They are expected to be updated manually in `call()`.

Returns A list of non-trainable variables.

property outbound_nodes

Return Functional API nodes downstream of this layer.

property output

Retrieves the output tensor(s) of a layer.

Only applicable if the layer has exactly one output, i.e. if it is connected to one incoming layer.

Returns Output tensor or list of output tensors.

Raises

- **AttributeError** – if the layer is connected to more than one incoming layers.
- **RuntimeError** – if called in Eager mode.

property output_mask

Retrieves the output mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Output mask tensor (potentially None) or list of output mask tensors.

Raises

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

property output_shape

Retrieves the output shape(s) of a layer.

Only applicable if the layer has one output, or if all outputs have the same shape.

Returns Output shape, as an integer shape tuple (or list of shape tuples, one tuple per output tensor).

Raises

- **AttributeError** – if the layer has no defined output shape.
- **RuntimeError** – if called in Eager mode.

predict(*x*, *batch_size=None*, *verbose='auto'*, *steps=None*, *callbacks=None*, *max_queue_size=10*,
workers=1, *use_multiprocessing=False*)

Generates output predictions for the input samples.

Computation is done in batches. This method is designed for batch processing of large numbers of inputs. It is not intended for use inside of loops that iterate over your data and process small numbers of inputs at a time.

For small numbers of inputs that fit in one batch, directly use `__call__()` for faster execution, e.g., `model(x)`, or `model(x, training=False)` if you have layers such as `tf.keras.layers.BatchNormalization` that behave differently during inference. You may pair the individual model call with a `tf.function` for additional perfor-

mance inside your inner loop. If you need access to numpy array values instead of tensors after your model call, you can use `tensor.numpy()` to get the numpy array value of an eager tensor.

Also, note the fact that test loss is not affected by regularization layers like noise and dropout.

Note: See [this FAQ entry]([https://keras.io/getting_started/faq/#whats-the-difference-between-model-methods-predict-and-call_\(\)](https://keras.io/getting_started/faq/#whats-the-difference-between-model-methods-predict-and-call_())) for more details about the difference between *Model* methods `predict()` and `__call__()`.

Parameters

- **x** – Input samples. It could be:
 - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
 - A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
 - A `tf.data` dataset.
 - A generator or `keras.utils.Sequence` instance.

A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given in the *Unpacking behavior for iterator-like inputs* section of `Model.fit`.

- **batch_size** – Integer or `None`. Number of samples per batch. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` if your data is in the form of dataset, generators, or `keras.utils.Sequence` instances (since they generate batches).
- **verbose** – “auto”, 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line. “auto” defaults to 1 for most cases, and to 2 when used with *ParameterServerStrategy*. Note that the progress bar is not particularly useful when logged to a file, so `verbose=2` is recommended when not running interactively (e.g. in a production environment).
- **steps** – Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of `None`. If `x` is a `tf.data` dataset and `steps` is `None`, `predict()` will run until the input dataset is exhausted.
- **callbacks** – List of `keras.callbacks.Callback` instances. List of callbacks to apply during prediction. See [callbacks](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks).
- **max_queue_size** – Integer. Used for generator or `keras.utils.Sequence` input only. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers** – Integer. Used for generator or `keras.utils.Sequence` input only. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1.
- **use_multiprocessing** – Boolean. Used for generator or `keras.utils.Sequence` input only. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.

See the discussion of *Unpacking behavior for iterator-like inputs* for `Model.fit`. Note that `Model.predict` uses the same interpretation rules as `Model.fit` and `Model.evaluate`, so inputs must be unambiguous for all three methods.

Returns Numpy array(s) of predictions.

Raises

- **RuntimeError** – If `model.predict` is wrapped in a `tf.function`.
- **ValueError** – In case of mismatch between the provided input data and the model's expectations, or in case a stateful model receives a number of samples that is not a multiple of the batch size.

`predict_generator(generator, steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiprocessing=False, verbose=0)`

Generates predictions for the input samples from a data generator.

DEPRECATED: `Model.predict` now supports generators, so there is no longer any need to use this endpoint.

`predict_on_batch(x)`

Returns predictions for a single batch of samples.

Parameters `x` – Input data. It could be: - A Numpy array (or array-like), or a list of arrays (in case the

model has multiple inputs).

- **A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).**

Returns Numpy array(s) of predictions.

Raises `RuntimeError` – If `model.predict_on_batch` is wrapped in a `tf.function`.

`predict_step(data)`

The logic for one inference step.

This method can be overridden to support custom inference logic. This method is called by `Model.make_predict_function`.

This method should contain the mathematical logic for one step of inference. This typically includes the forward pass.

Configuration details for *how* this logic is run (e.g. `tf.function` and `tf.distribute.Strategy` settings), should be left to `Model.make_predict_function`, which can also be overridden.

Parameters `data` – A nested structure of `Tensor`'s.

Returns The result of one inference step, typically the output of calling the `Model` on data.

`reset_metrics()`

Resets the state of all the metrics in the model.

Examples:

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> outputs = tf.keras.layers.Dense(2)(inputs)
>>> model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
```

```
>>> x = np.random.random((2, 3))
>>> y = np.random.randint(0, 2, (2, 2))
>>> _ = model.fit(x, y, verbose=0)
>>> assert all(float(m.result()) for m in model.metrics)
```

```
>>> model.reset_metrics()
>>> assert all(float(m.result()) == 0 for m in model.metrics)
```

reset_states()**property run_eagerly**

Settable attribute indicating whether the model should run eagerly.

Running eagerly means that your model will be run step by step, like Python code. Your model might run slower, but it should become easier for you to debug it by stepping into individual layer calls.

By default, we will attempt to compile your model to a static graph to deliver the best execution performance.

Returns Boolean, whether the model should run eagerly.

sample(batch_size: int) → Tuple[tensorflow.python.framework.ops.Tensor,
tensorflow.python.framework.ops.Tensor]

save(filepath, overwrite=True, include_optimizer=True, save_format=None, signatures=None,
options=None, save_traces=True)

Saves the model to Tensorflow SavedModel or a single HDF5 file.

Please see `tf.keras.models.save_model` or the [Serialization and Saving guide](https://keras.io/guides/serialization_and_saving/) for details.

Parameters

- **filepath** – String, PathLike, path to SavedModel or H5 file to save the model.
- **overwrite** – Whether to silently overwrite any existing file at the target location, or provide the user with a manual prompt.
- **include_optimizer** – If True, save optimizer's state together.
- **save_format** – Either ‘tf’ or ‘h5’, indicating whether to save the model to Tensorflow SavedModel or HDF5. Defaults to ‘tf’ in TF 2.X, and ‘h5’ in TF 1.X.
- **signatures** – Signatures to save with the SavedModel. Applicable to the ‘tf’ format only. Please see the `signatures` argument in `tf.saved_model.save` for details.
- **options** – (only applies to SavedModel format) `tf.saved_model.SaveOptions` object that specifies options for saving to SavedModel.
- **save_traces** – (only applies to SavedModel format) When enabled, the SavedModel will store the function traces for each layer. This can be disabled, so that only the configs of each layer are stored. Defaults to `True`. Disabling this will decrease serialization time and reduce file size, but it requires that all custom layers/models implement a `get_config()` method.

Example:

```
```python
from keras.models import load_model
model.save('my_model.h5') # creates a HDF5 file 'my_model.h5'
del model # deletes the existing model
returns a compiled model # identical to the previous one
model = load_model('my_model.h5')
````
```

save_spec(dynamic_batch=True)

Returns the `tf.TensorSpec` of call inputs as a tuple (`args, kwargs`).

This value is automatically defined after calling the model for the first time. Afterwards, you can use it when exporting the model for serving:

```
```python
model = tf.keras.Model(...)

@tf.function def serve(*args, **kwargs):
 outputs = model(*args, **kwargs) # Apply postprocessing steps, or add additional outputs. ...
 return outputs

arg_specs is [tf.TensorSpec(...), ...]. kwarg_specs, in this # example, is an empty dict since functional
models do not use keyword # arguments. arg_specs, kwarg_specs = model.save_spec()

model.save(path, signatures={

 'serving_default': serve.get_concrete_function(*arg_specs, **kwarg_specs)
})
```

**Parameters `dynamic_batch`** – Whether to set the batch sizes of all the returned `tf.TensorSpec` to `None`. (Note that when defining functional or Sequential models with `tf.keras.Input([...], batch_size=X)`, the batch size will always be preserved). Defaults to `True`.

**Returns** If the model inputs are defined, returns a tuple (`args, kwargs`). All elements in `args` and `kwargs` are `tf.TensorSpec`. If the model inputs are not defined, returns `None`. The model inputs are automatically set when calling the model, `model.fit`, `model.evaluate` or `model.predict`.

#### `save_weights(filepath, overwrite=True, save_format=None, options=None)`

Saves all layer weights.

Either saves in HDF5 or in TensorFlow format based on the `save_format` argument.

**When saving in HDF5 format, the weight file has:**

- **`layer_names` (attribute), a list of strings** (ordered names of model layers).
- **For every layer, a group named `layer.name`**
  - **For every such layer group, a group attribute `weight_names`**, a list of strings (ordered names of weights tensor of the layer).
  - **For every weight in the layer, a dataset** storing the weight value, named after the weight tensor.

When saving in TensorFlow format, all objects referenced by the network are saved in the same format as `tf.train.Checkpoint`, including any `Layer` instances or `Optimizer` instances assigned to object attributes. For networks constructed from inputs and outputs using `tf.keras.Model(inputs, outputs)`, `Layer` instances used by the network are tracked/saved automatically. For user-defined classes which inherit from `tf.keras.Model`, `Layer` instances must be assigned to object attributes, typically in the constructor. See the documentation of `tf.train.Checkpoint` and `tf.keras.Model` for details.

While the formats are the same, do not mix `save_weights` and `tf.train.Checkpoint`. Checkpoints saved by `Model.save_weights` should be loaded using `Model.load_weights`. Checkpoints saved using `tf.train.Checkpoint.save` should be restored using the corresponding `tf.train.Checkpoint.restore`. Prefer `tf.train.Checkpoint` over `save_weights` for training checkpoints.

The TensorFlow format matches objects and variables by starting at a root object, `self` for `save_weights`, and greedily matching attribute names. For `Model.save` this is the `Model`, and for `Checkpoint.save` this is the `Checkpoint` even if the `Checkpoint` has a model attached. This means saving a `tf.keras.Model` using `save_weights` and loading into a `tf.train.Checkpoint` with a `Model` attached (or vice versa) will not match the `Model`'s variables. See the [guide to training checkpoints](<https://www.tensorflow.org/guide/checkpoint>) for details on the TensorFlow format.

#### Parameters

- **filepath** – String or PathLike, path to the file to save the weights to. When saving in TensorFlow format, this is the prefix used for checkpoint files (multiple files are generated). Note that the ‘.h5’ suffix causes weights to be saved in HDF5 format.
- **overwrite** – Whether to silently overwrite any existing file at the target location, or provide the user with a manual prompt.
- **save\_format** – Either ‘tf’ or ‘h5’. A *filepath* ending in ‘.h5’ or ‘.keras’ will default to HDF5 if *save\_format* is *None*. Otherwise *None* defaults to ‘tf’.
- **options** – Optional *tf.train.CheckpointOptions* object that specifies options for saving weights.

**Raises** `ImportError` – If *h5py* is not available when attempting to save in HDF5 format.

#### `set_weights(weights)`

Sets the weights of the layer, from NumPy arrays.

The weights of a layer represent the state of the layer. This function sets the weight values from numpy arrays. The weight values should be passed in the order they are created by the layer. Note that the layer’s weights must be instantiated before calling this function, by calling the layer.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
... kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
 [1.],
 [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
... kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
 [2.],
 [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
 [1.],
 [1.]], dtype=float32), array([0.], dtype=float32)]
```

**Parameters** **weights** – a list of NumPy arrays. The number of arrays and their shape must match number of the dimensions of the weights of the layer (i.e. it should match the output of *get\_weights*).

**Raises** `ValueError` – If the provided weights list does not match the layer’s specifications.

#### `property state_updates`

Deprecated, do NOT use!

Returns the *updates* from all layers that are stateful.

This is useful for separating training updates and state updates, e.g. when we need to update a layer’s internal state during prediction.

**Returns** A list of update ops.

**property stateful**

**property submodules**

Sequence of all sub-modules.

Submodules are modules which are properties of this module, or found as properties of modules which are properties of this module (and so on).

```
>>> a = tf.Module()
>>> b = tf.Module()
>>> c = tf.Module()
>>> a.b = b
>>> b.c = c
>>> list(a.submodules) == [b, c]
True
>>> list(b.submodules) == [c]
True
>>> list(c.submodules) == []
True
```

**Returns** A sequence of all submodules.

**summary**(*line\_length=None*, *positions=None*, *print\_fn=None*, *expand\_nested=False*, *show\_trainable=False*, *layer\_range=None*)

Prints a string summary of the network.

#### Parameters

- **line\_length** – Total length of printed lines (e.g. set this to adapt the display to different terminal window sizes).
- **positions** – Relative or absolute positions of log elements in each line. If not provided, defaults to [.33, .55, .67, 1].
- **print\_fn** – Print function to use. Defaults to *print*. It will be called on each line of the summary. You can set it to a custom function in order to capture the string summary.
- **expand\_nested** – Whether to expand the nested models. If not provided, defaults to *False*.
- **show\_trainable** – Whether to show if a layer is trainable. If not provided, defaults to *False*.
- **layer\_range** – a list or tuple of 2 strings, which is the starting layer name and ending layer name (both inclusive) indicating the range of layers to be printed in summary. It also accepts regex patterns instead of exact name. In such case, start predicate will be the first element it matches to *layer\_range[0]* and the end predicate will be the last element it matches to *layer\_range[1]*. By default *None* which considers all layers of model.

**Raises** **ValueError** – if *summary()* is called before the model is built.

**property supports\_masking**

Whether this layer supports computing a mask using *compute\_mask*.

**test\_on\_batch**(*x*, *y=None*, *sample\_weight=None*, *reset\_metrics=True*, *return\_dict=False*)

Test the model on a single batch of samples.

#### Parameters

- **x** – Input data. It could be: - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
- A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
- A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
- **y** – Target data. Like the input data *x*, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with *x* (you cannot have Numpy inputs and tensor targets, or inversely).
- **sample\_weight** – Optional array of the same length as *x*, containing weights to apply to the model’s loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence\_length), to apply a different weight to every timestep of every sample.
- **reset\_metrics** – If *True*, the metrics returned will be only for this batch. If *False*, the metrics will be statefully accumulated across batches.
- **return\_dict** – If *True*, loss and metric results are returned as a dict, with each key being the name of the metric. If *False*, they are returned as a list.

**Returns** Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute *model.metrics\_names* will give you the display labels for the scalar outputs.

**Raises** `RuntimeError` – If *model.test\_on\_batch* is wrapped in a *tf.function*.

### `test_step(data)`

The logic for one evaluation step.

This method can be overridden to support custom evaluation logic. This method is called by *Model.make\_test\_function*.

This function should contain the mathematical logic for one step of evaluation. This typically includes the forward pass, loss calculation, and metrics updates.

Configuration details for how this logic is run (e.g. *tf.function* and *tf.distribute.Strategy* settings), should be left to *Model.make\_test\_function*, which can also be overridden.

**Parameters** `data` – A nested structure of `Tensor`’s.

**Returns** A dict containing values that will be passed to *tf.keras.callbacks.CallbackList.on\_train\_batch\_end*. Typically, the values of the *Model*’s metrics are returned.

### `to_json(**kwargs)`

Returns a JSON string containing the network configuration.

To load a network from a JSON save file, use *keras.models.model\_from\_json(json\_string, custom\_objects={})*.

**Parameters** `**kwargs` – Additional keyword arguments to be passed to `*json.dumps()`.

**Returns** A JSON string.

### `to_yaml(**kwargs)`

Returns a yaml string containing the network configuration.

Note: Since TF 2.6, this method is no longer supported and will raise a `RuntimeError`.

To load a network from a yaml save file, use `keras.models.model_from_yaml(yaml_string, custom_objects={})`.

`custom_objects` should be a dictionary mapping the names of custom losses / layers / etc to the corresponding functions / classes.

**Parameters** `**kwargs` – Additional keyword arguments to be passed to `yaml.dump()`.

**Returns** A YAML string.

**Raises** `RuntimeError` – announces that the method poses a security risk

**train\_on\_batch**(*x*, *y*=*None*, *sample\_weight*=*None*, *class\_weight*=*None*, *reset\_metrics*=*True*, *return\_dict*=*False*)

Runs a single gradient update on a single batch of data.

#### Parameters

- **x** – Input data. It could be: - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
  - **A TensorFlow tensor, or a list of tensors** (in case the model has multiple inputs).
  - **A dict mapping input names to the corresponding array/tensors**, if the model has named inputs.
- **y** – Target data. Like the input data *x*, it could be either Numpy array(s) or TensorFlow tensor(s).
- **sample\_weight** – Optional array of the same length as *x*, containing weights to apply to the model’s loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence\_length), to apply a different weight to every timestep of every sample.
- **class\_weight** – Optional dictionary mapping class indices (integers) to a weight (float) to apply to the model’s loss for the samples from this class during training. This can be useful to tell the model to “pay more attention” to samples from an under-represented class.
- **reset\_metrics** – If *True*, the metrics returned will be only for this batch. If *False*, the metrics will be statefully accumulated across batches.
- **return\_dict** – If *True*, loss and metric results are returned as a dict, with each key being the name of the metric. If *False*, they are returned as a list.

**Returns** Scalar training loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

**Raises** `RuntimeError` – If `model.train_on_batch` is wrapped in a `tf.function`.

**train\_step**(*data*)

The logic for one training step.

This method can be overridden to support custom training logic. For concrete examples of how to override this method see [Customizing what happens in fit]([https://www.tensorflow.org/guide/keras/customizing\\_what\\_happens\\_in\\_fit](https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit)). This method is called by `Model.make_train_function`.

This method should contain the mathematical logic for one step of training. This typically includes the forward pass, loss calculation, backpropagation, and metric updates.

Configuration details for *how* this logic is run (e.g. `tf.function` and `tf.distribute.Strategy` settings), should be left to `Model.make_train_function`, which can also be overridden.

**Parameters** `data` – A nested structure of `Tensor`’s.

**Returns** A `dict` containing values that will be passed to `tf.keras.callbacks.CallbackList.on_train_batch_end`. Typically, the values of the `Model`’s metrics are returned. Example: `{‘loss’: 0.2, ‘accuracy’: 0.7}`.

#### **property trainable**

#### **property trainable\_variables**

Sequence of trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don’t expect the return value to change.

**Returns** A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

#### **property trainable\_weights**

List of all trainable weights tracked by this layer.

Trainable weights are updated via gradient descent during training.

**Returns** A list of trainable variables.

#### **property updates**

#### **property variable\_dtype**

Alias of `Layer.dtype`, the dtype of the weights.

#### **property variables**

Returns the list of all layer variables/weights.

Alias of `self.weights`.

Note: This will not track the weights of nested `tf.Modules` that are not themselves Keras layers.

**Returns** A list of variables.

#### **property weights**

Returns the list of all layer variables/weights.

Note: This will not track the weights of nested `tf.Modules` that are not themselves Keras layers.

**Returns** A list of variables.

#### **classmethod with\_name\_scope(method)**

Decorator to automatically enter the module name scope.

```
>>> class MyModule(tf.Module):
... @tf.Module.with_name_scope
... def __call__(self, x):
... if not hasattr(self, 'w'):
... self.w = tf.Variable(tf.random.normal([x.shape[1], 3]))
... return tf.matmul(x, self.w)
```

Using the above module would produce `tf.Variable`’s and `tf.Tensor`’s whose names included the module name:

```
>>> mod = MyModule()
>>> mod(tf.ones([1, 2]))
<tf.Tensor: shape=(1, 3), dtype=float32, numpy=..., dtype=float32)>
>>> mod.w
<tf.Variable 'my_module/Variable:0' shape=(2, 3) dtype=float32,
numpy=..., dtype=float32)>
```

**Parameters** `method` – The method to wrap.

**Returns** The original method wrapped such that it enters the module's name scope.

```
class tensorcircuit.applications.van.PixelCNN(*args, **kwargs)
```

Bases: keras.engine.training.Model

```
__init__(spin_channel: int, depth: int, filters: int)
```

```
property activity_regularizer
```

Optional regularizer function for the output of this layer.

```
add_loss(losses, **kwargs)
```

Add loss tensor(s), potentially dependent on layer inputs.

Some losses (for instance, activity regularization losses) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs  $a$  and  $b$ , some entries in `layer.losses` may be dependent on  $a$  and some on  $b$ . This method automatically keeps track of dependencies.

This method can be used inside a subclassed layer or model's `call` function, in which case `losses` should be a Tensor or list of Tensors.

Example:

```
```python
class MyLayer(tf.keras.layers.Layer):
    def call(self, inputs):
        self.add_loss(tf.abs(tf.reduce_mean(inputs)))
        return inputs
```
```

This method can also be called directly on a Functional Model during construction. In this case, any loss Tensors passed to this Model must be symbolic and be able to be traced back to the model's `Input`'s. *These losses become part of the model's topology and are tracked in `'get_config'`.*

Example:

```
`python
inputs = tf.keras.Input(shape=(10,))
x = tf.keras.layers.Dense(10)(inputs)
outputs = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs, outputs) # Activity regularization.
model.add_loss(tf.abs(tf.reduce_mean(x)))`
```

If this is not the case for your loss (if, for example, your loss references a `Variable` of one of the model's layers), you can wrap your loss in a zero-argument lambda. These losses are not tracked as part of the model's topology since they can't be serialized.

Example:

```
`python
inputs = tf.keras.Input(shape=(10,))
d = tf.keras.layers.Dense(10)
x = d(inputs)
outputs = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs, outputs) # Weight regularization.
model.add_loss(lambda: tf.reduce_mean(d.kernel))`
```

**Parameters**

- **losses** – Loss tensor, or list/tuple of tensors. Rather than tensors, losses may also be zero-argument callables which create a loss tensor.
- **\*\*kwargs** – Used for backwards compatibility only.

**add\_metric**(*value*, *name*=*None*, *\*\*kwargs*)

Adds metric tensor to the layer.

This method can be used inside the *call()* method of a subclassed layer or model.

```
```python class MyMetricLayer(tf.keras.layers.Layer):
```

```
    def __init__(self): super(MyMetricLayer, self).__init__(name='my_metric_layer') self.mean = tf.keras.metrics.Mean(name='metric_1')

    def call(self, inputs): self.add_metric(self.mean(inputs)) self.add_metric(tf.reduce_sum(inputs), name='metric_2') return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any tensor passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These metrics become part of the model's topology and are tracked when you save the model via 'save()'*.

```
```python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) model.add_metric(tf.math_ops.reduce_sum(x), name='metric_1')``
```

Note: Calling *add\_metric()* with the result of a metric object on a Functional Model, as shown in the example below, is not supported. This is because we cannot trace the metric result tensor back to the model's inputs.

```
```python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) model.add_metric(tf.keras.metrics.Mean()(x), name='metric_1')``
```

Parameters

- **value** – Metric tensor.
- **name** – String metric name.
- ****kwargs** – Additional keyword arguments for backward compatibility. Accepted values: *aggregation* - When the *value* tensor provided is not the result of calling a *keras.Metric* instance, it will be aggregated by default using a *keras.Metric.Mean*.

add_update(*updates*)

Add update op(s), potentially dependent on layer inputs.

Weight updates (for instance, the updates of the moving mean and variance in a BatchNormalization layer) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs *a* and *b*, some entries in *layer.updates* may be dependent on *a* and some on *b*. This method automatically keeps track of dependencies.

This call is ignored when eager execution is enabled (in that case, variable updates are run on the fly and thus do not need to be tracked for later execution).

Parameters **updates** – Update op, or list/tuple of update ops, or zero-arg callable that returns an update op. A zero-arg callable should be passed in order to disable running the updates by setting *trainable=False* on this Layer, when executing in Eager mode.

add_variable(*args, **kwargs)

Deprecated, do NOT use! Alias for `add_weight`.

add_weight(*name=None*, *shape=None*, *dtype=None*, *initializer=None*, *regularizer=None*, *trainable=None*, *constraint=None*, *use_resource=None*, *synchronization=VariableSynchronization.AUTO*, *aggregation=VariableAggregationV2.NONE*, **kwargs)

Adds a new variable to the layer.

Parameters

- **name** – Variable name.
- **shape** – Variable shape. Defaults to scalar if unspecified.
- **dtype** – The type of the variable. Defaults to `self.dtype`.
- **initializer** – Initializer instance (callable).
- **regularizer** – Regularizer instance (callable).
- **trainable** – Boolean, whether the variable should be part of the layer’s “trainable_variables” (e.g. variables, biases) or “non_trainable_variables” (e.g. BatchNorm mean and variance). Note that `trainable` cannot be `True` if `synchronization` is set to `ON_READ`.
- **constraint** – Constraint instance (callable).
- **use_resource** – Whether to use a `ResourceVariable` or not. See [this guide](https://www.tensorflow.org/guide/migrate/tf1_vs_tf2#resourcevariables_instead_of_referencevariables)
for more information.
- **synchronization** – Indicates when a distributed a variable will be aggregated. Accepted values are constants defined in the class `tf.VariableSynchronization`. By default the synchronization is set to `AUTO` and the current `DistributionStrategy` chooses when to synchronize. If `synchronization` is set to `ON_READ`, `trainable` must not be set to `True`.
- **aggregation** – Indicates how a distributed variable will be aggregated. Accepted values are constants defined in the class `tf.VariableAggregation`.
- ****kwargs** – Additional keyword arguments. Accepted values are `getter`, `collections`, `experimental_autocast` and `caching_device`.

Returns The variable created.

Raises `ValueError` – When giving unsupported `dtype` and no `initializer` or when `trainable` has been set to `True` with `synchronization` set as `ON_READ`.

build(*input_shape*)

Builds the model based on input shapes received.

This is to be used for subclassed models, which do not know at instantiation time what their inputs look like.

This method only exists for users who want to call `model.build()` in a standalone way (as a substitute for calling the model on real data to build it). It will never be called by the framework (and thus it will never throw unexpected errors in an unrelated workflow).

Parameters **input_shape** – Single tuple, `TensorShape` instance, or list/dict of shapes, where shapes are tuples, integers, or `TensorShape` instances.

Raises

- **ValueError** –

1. In case of invalid user-provided data (not of type tuple, list, *TensorShape*, or dict). 2. If the model requires call arguments that are agnostic to the input shapes (positional or keyword arg in call signature). 3. If not all layers were properly built. 4. If float type inputs are not supported within the layers.

- **In each of these cases, the user should build their model by calling –**

- **it on real tensor data.** –

call(*inputs*: tensorflow.python.framework.ops.Tensor) → tensorflow.python.framework.ops.Tensor
Calls the model on new inputs and returns the outputs as tensors.

In this case *call()* just reapplyes all ops in the graph to the new inputs (e.g. build a new computational graph from the provided inputs).

Note: This method should not be called directly. It is only meant to be overridden when subclassing *tf.keras.Model*. To call a model on an input, always use the *__call__()* method, i.e. *model(inputs)*, which relies on the underlying *call()* method.

Parameters

- **inputs** – Input tensor, or dict/list/tuple of input tensors.
- **training** – Boolean or boolean scalar tensor, indicating whether to run the *Network* in training mode or inference mode.
- **mask** – A mask or list of masks. A mask can be either a boolean tensor or None (no mask). For more details, check the guide [here](https://www.tensorflow.org/guide/keras/masking_and_padding).

Returns A tensor if there is a single output, or a list of tensors if there are more than one outputs.

compile(*optimizer='rmsprop'*, *loss=None*, *metrics=None*, *loss_weights=None*, *weighted_metrics=None*, *run_eagerly=None*, *steps_per_execution=None*, *jit_compile=None*, ***kwargs*)
Configures the model for training.

Example:

```
```python
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
 loss=tf.keras.losses.BinaryCrossentropy(),
 metrics=[tf.keras.metrics.BinaryAccuracy(),
 tf.keras.metrics.FalseNegatives()])
```

```

Parameters

- **optimizer** – String (name of optimizer) or optimizer instance. See *tf.keras.optimizers*.
- **loss** – Loss function. May be a string (name of loss function), or a *tf.keras.losses.Loss* instance. See *tf.keras.losses*. A loss function is any callable with the signature *loss = fn(y_true, y_pred)*, where *y_true* are the ground truth values, and *y_pred* are the model's predictions. *y_true* should have shape (*batch_size*, *d0*, .. *dN*) (except in the case of sparse loss functions such as sparse categorical crossentropy which expects integer arrays of shape (*batch_size*, *d0*, .. *dN-1*)). *y_pred* should have shape (*batch_size*, *d0*, .. *dN*). The loss function should return a float tensor. If a custom *Loss* instance is used and reduction is set to *None*, return value has shape (*batch_size*, *d0*, .. *dN-1*) i.e. per-sample or per-timestep loss values; otherwise, it is a scalar. If the model has

multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses, unless *loss_weights* is specified.

- **metrics** – List of metrics to be evaluated by the model during training and testing. Each of this can be a string (name of a built-in function), function or a *tf.keras.metrics.Metric* instance. See *tf.keras.metrics*. Typically you will use *metrics=['accuracy']*. A function is any callable with the signature *result = fn(y_true, y_pred)*. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as *metrics={'output_a':'accuracy', 'output_b':['accuracy', 'mse']}*. You can also pass a list to specify a metric or a list of metrics for each output, such as *metrics=[[{'accuracy'}, {'accuracy', 'mse'}]]* or *metrics=[{'accuracy', 'accuracy', 'mse'}]*. When you pass the strings ‘accuracy’ or ‘acc’, we convert this to one of *tf.keras.metrics.BinaryAccuracy*, *tf.keras.metrics.CategoricalAccuracy*, *tf.keras.metrics.SparseCategoricalAccuracy* based on the shapes of the targets and of the model output. We do a similar conversion for the strings ‘crossentropy’ and ‘ce’ as well. The metrics passed here are evaluated without sample weighting; if you would like sample weighting to apply, you can specify your metrics via the *weighted_metrics* argument instead.
- **loss_weights** – Optional list or dictionary specifying scalar coefficients (Python floats) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the *weighted sum* of all individual losses, weighted by the *loss_weights* coefficients. If a list, it is expected to have a 1:1 mapping to the model’s outputs. If a dict, it is expected to map output names (strings) to scalar coefficients.
- **weighted_metrics** – List of metrics to be evaluated and weighted by *sample_weight* or *class_weight* during training and testing.
- **run_eagerly** – Bool. Defaults to *False*. If *True*, this *Model*’s logic will not be wrapped in a *tf.function*. Recommended to leave this as *None* unless your *Model* cannot be run inside a *tf.function*. *run_eagerly=True* is not supported when using *tf.distribute.experimental.ParameterServerStrategy*.
- **steps_per_execution** – Int. Defaults to 1. The number of batches to run during each *tf.function* call. Running multiple batches inside a single *tf.function* call can greatly improve performance on TPUs or small models with a large Python overhead. At most, one full epoch will be run each execution. If a number larger than the size of the epoch is passed, the execution will be truncated to the size of the epoch. Note that if *steps_per_execution* is set to *N*, *Callback.on_batch_begin* and *Callback.on_batch_end* methods will only be called every *N* batches (i.e. before/after each *tf.function* execution).
- **jit_compile** – If *True*, compile the model training step with XLA. [XLA](<https://www.tensorflow.org/xla>) is an optimizing compiler for machine learning. *jit_compile* is not enabled for by default. This option cannot be enabled with *run_eagerly=True*. Note that *jit_compile=True* may not necessarily work for all models. For more information on supported operations please refer to the [XLA documentation](<https://www.tensorflow.org/xla>). Also refer to [known XLA issues](https://www.tensorflow.org/xla/known_issues) for more details.
- ****kwargs** – Arguments supported for backwards compatibility only.

property compute_dtype

The dtype of the layer’s computations.

This is equivalent to `Layer.dtype_policy.compute_dtype`. Unless mixed precision is used, this is the same as `Layer.dtype`, the dtype of the weights.

Layers automatically cast their inputs to the compute dtype, which causes computations and the output to be in the compute dtype as well. This is done by the base Layer class in `Layer.__call__`, so you do not have to insert these casts if implementing your own layer.

Layers often perform certain internal computations in higher precision when `compute_dtype` is `float16` or `bfloat16` for numeric stability. The output will still typically be `float16` or `bfloat16` in such cases.

Returns The layer's compute dtype.

compute_loss(*x=None*, *y=None*, *y_pred=None*, *sample_weight=None*)

Compute the total loss, validate it, and return it.

Subclasses can optionally override this method to provide custom loss computation logic.

Example: `python class MyModel(tf.keras.Model):`

```
def __init__(self, *args, **kwargs): super(MyModel, self).__init__(*args, **kwargs)
    self.loss_tracker = tf.keras.metrics.Mean(name='loss')

def compute_loss(self, x, y, y_pred, sample_weight): loss =
    tf.reduce_mean(tf.math.squared_difference(y_pred, y)) loss += tf.add_n(self.losses)
    self.loss_tracker.update_state(loss) return loss

def reset_metrics(self): self.loss_tracker.reset_states()

@property def metrics(self):
    return [self.loss_tracker]

tensors = tf.random.uniform((10, 10)), tf.random.uniform((10,)) dataset =
tf.data.Dataset.from_tensor_slices(tensors).repeat().batch(1)

inputs = tf.keras.layers.Input(shape=(10,), name='my_input') outputs = tf.keras.layers.Dense(10)(inputs)
model = MyModel(inputs, outputs) model.add_loss(tf.reduce_sum(outputs))

optimizer = tf.keras.optimizers.SGD() model.compile(optimizer, loss='mse', steps_per_execution=10)
model.fit(dataset, epochs=2, steps_per_epoch=10) print('My custom loss: ', model.loss_tracker.result().numpy())
```

Parameters

- **x** – Input data.
- **y** – Target data.
- **y_pred** – Predictions returned by the model (output of `model(x)`)
- **sample_weight** – Sample weights for weighting the loss function.

Returns The total loss as a `tf.Tensor`, or `None` if no loss results (which is the case when called by `Model.test_step`).

compute_mask(*inputs*, *mask=None*)

Computes an output mask tensor.

Parameters

- **inputs** – Tensor or list of tensors.
- **mask** – Tensor or list of tensors.

Returns

None or a tensor (or list of tensors, one per output tensor of the layer).

compute_metrics(*x*, *y*, *y_pred*, *sample_weight*)

Update metric states and collect all metrics to be returned.

Subclasses can optionally override this method to provide custom metric updating and collection logic.

Example: `python class MyModel(tf.keras.Sequential):`

```
def compute_metrics(self, x, y, y_pred, sample_weight):
    # This super call updates self.compiled_metrics and returns # results for all metrics
    # listed in self.metrics. metric_results = super(MyModel, self).compute_metrics(
        x, y, y_pred, sample_weight)
    #
    # Note that self.custom_metric is not listed in self.metrics.
    # self.custom_metric.update_state(x, y, y_pred, sample_weight) metric_results['custom_metric_name'] = self.custom_metric.result() return metric_results
```

...

Parameters

- **x** – Input data.
- **y** – Target data.
- **y_pred** – Predictions returned by the model (output of *model.call(x)*)
- **sample_weight** – Sample weights for weighting the loss function.

Returns A *dict* containing values that will be passed to *tf.keras.callbacks.CallbackList.on_train_batch_end()*. Typically, the values of the metrics listed in *self.metrics* are returned. Example: `{'loss': 0.2, 'accuracy': 0.7}`.

compute_output_shape(*input_shape*)

Computes the output shape of the layer.

This method will cause the layer's state to be built, if that has not happened before. This requires that the layer will later be used with inputs that match the input shape provided here.

Parameters **input_shape** – Shape tuple (tuple of integers) or *tf.TensorShape*, or structure of shape tuples / *tf.TensorShape* instances (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns A *tf.TensorShape* instance or structure of *tf.TensorShape* instances.

compute_output_signature(*input_signature*)

Compute the output tensor signature of the layer based on the inputs.

Unlike a *TensorShape* object, a *TensorSpec* object contains both shape and dtype information for a tensor. This method allows layers to provide output dtype information if it is different from the input dtype. For any layer that doesn't implement this function, the framework will fall back to use *compute_output_shape*, and will assume that the output dtype matches the input dtype.

Parameters **input_signature** – Single *TensorSpec* or nested structure of *TensorSpec* objects, describing a candidate input for the layer.

Returns

Single TensorSpec or nested structure of TensorSpec objects, describing how the layer would transform the provided input.

Raises **TypeError** – If *input_signature* contains a non-*TensorSpec* object.

count_params()

Count the total number of scalars composing the weights.

Returns An integer count.

Raises ValueError – if the layer isn’t yet built (in which case its weights aren’t yet defined).

property distribute_reduction_method

The method employed to reduce per-replica values during training.

Unless specified, the value “auto” will be assumed, indicating that the reduction strategy should be chosen based on the current running environment. See *reduce_per_replica* function for more details.

property distribute_strategy

The *tf.distribute.Strategy* this model was created under.

property dtype

The dtype of the layer weights.

This is equivalent to *Layer.dtype_policy.variable_dtype*. Unless mixed precision is used, this is the same as *Layer.compute_dtype*, the dtype of the layer’s computations.

property dtype_policy

The dtype policy associated with this layer.

This is an instance of a *tf.keras.mixed_precision.Policy*.

property dynamic

Whether the layer is dynamic (eager-only); set in the constructor.

```
evaluate(x=None, y=None, batch_size=None, verbose='auto', sample_weight=None, steps=None,
         callbacks=None, max_queue_size=10, workers=1, use_multiprocessing=False, return_dict=False,
         **kwargs)
```

Returns the loss value & metrics values for the model in test mode.

Computation is done in batches (see the *batch_size* arg.)

Parameters

- **x** – Input data. It could be:
 - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
 - A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
 - A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
 - A *tf.data* dataset. Should return a tuple of either *(inputs, targets)* or *(inputs, targets, sample_weights)*.
 - A generator or *keras.utils.Sequence* returning *(inputs, targets)* or *(inputs, targets, sample_weights)*.

A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given in the *Unpacking behavior for iterator-like inputs* section of *Model.fit*.

- **y** – Target data. Like the input data *x*, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with *x* (you cannot have Numpy inputs and tensor targets, or inversely). If *x* is a dataset, generator or *keras.utils.Sequence* instance, *y* should not be specified (since targets will be obtained from the iterator/dataset).

- **batch_size** – Integer or *None*. Number of samples per batch of computation. If unspecified, *batch_size* will default to 32. Do not specify the *batch_size* if your data is in the form of a dataset, generators, or *keras.utils.Sequence* instances (since they generate batches).
- **verbose** – “auto”, 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line. “auto” defaults to 1 for most cases, and to 2 when used with *ParameterServerStrategy*. Note that the progress bar is not particularly useful when logged to a file, so *verbose*=2 is recommended when not running interactively (e.g. in a production environment).
- **sample_weight** – Optional Numpy array of weights for the test samples, used for weighting the loss function. You can either pass a flat (1D) Numpy array with the same length as the input samples

(1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape (*samples*, *sequence_length*), to apply a different weight to every timestep of every sample. This argument is not supported when *x* is a dataset, instead pass sample weights as the third element of *x*.
- **steps** – Integer or *None*. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of *None*. If *x* is a *tf.data* dataset and *steps* is *None*, ‘evaluate’ will run until the dataset is exhausted. This argument is not supported with array inputs.
- **callbacks** – List of *keras.callbacks.Callback* instances. List of callbacks to apply during evaluation. See [callbacks](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks).
- **max_queue_size** – Integer. Used for generator or *keras.utils.Sequence* input only. Maximum size for the generator queue. If unspecified, *max_queue_size* will default to 10.
- **workers** – Integer. Used for generator or *keras.utils.Sequence* input only. Maximum number of processes to spin up when using process-based threading. If unspecified, *workers* will default to 1.
- **use_multiprocessing** – Boolean. Used for generator or *keras.utils.Sequence* input only. If *True*, use process-based threading. If unspecified, *use_multiprocessing* will default to *False*. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can’t be passed easily to children processes.
- **return_dict** – If *True*, loss and metric results are returned as a dict, with each key being the name of the metric. If *False*, they are returned as a list.
- ****kwargs** – Unused at this time.

See the discussion of *Unpacking behavior for iterator-like inputs* for *Model.fit*.

Returns Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute *model.metrics_names* will give you the display labels for the scalar outputs.

Raises **RuntimeError** – If *model.evaluate* is wrapped in a *tf.function*.

evaluate_generator(*generator*, *steps=None*, *callbacks=None*, *max_queue_size=10*, *workers=1*,
 use_multiprocessing=False, *verbose=0*)

Evaluates the model on a data generator.

DEPRECATED: *Model.evaluate* now supports generators, so there is no longer any need to use this endpoint.

`finalize_state()`

Finalizes the layers state after updating layer weights.

This function can be subclassed in a layer and will be called after updating a layer weights. It can be overridden to finalize any additional layer state after a weight update.

This function will be called after weights of a layer have been restored from a loaded model.

```
fit(x=None, y=None, batch_size=None, epochs=1, verbose='auto', callbacks=None, validation_split=0.0,
     validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0,
     steps_per_epoch=None, validation_steps=None, validation_batch_size=None, validation_freq=1,
     max_queue_size=10, workers=1, use_multiprocessing=False)
```

Trains the model for a fixed number of epochs (iterations on a dataset).

Parameters

- **x** – Input data. It could be: - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
 - A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
 - A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
 - A *tf.data* dataset. Should return a tuple of either (*inputs*, *targets*) or (*inputs*, *targets*, *sample_weights*).
 - A generator or *keras.utils.Sequence* returning (*inputs*, *targets*) or (*inputs*, *targets*, *sample_weights*).
 - A *tf.keras.utils.experimental.DatasetCreator*, which wraps a callable that takes a single argument of type *tf.distribute.InputContext*, and returns a *tf.data.Dataset*. *DatasetCreator* should be used when users prefer to specify the per-replica batching and sharding logic for the *Dataset*. See *tf.keras.utils.experimental.DatasetCreator* doc for more information.

A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given below. If these include *sample_weights* as a third component, note that sample weighting applies to the *weighted_metrics* argument but not the *metrics* argument in *compile()*. If using *tf.distribute.experimental.ParameterServerStrategy*, only *DatasetCreator* type is supported for *x*.

- **y** – Target data. Like the input data *x*, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with *x* (you cannot have Numpy inputs and tensor targets, or inversely). If *x* is a dataset, generator, or *keras.utils.Sequence* instance, *y* should not be specified (since targets will be obtained from *x*).
- **batch_size** – Integer or *None*. Number of samples per gradient update. If unspecified, *batch_size* will default to 32. Do not specify the *batch_size* if your data is in the form of datasets, generators, or *keras.utils.Sequence* instances (since they generate batches).
- **epochs** – Integer. Number of epochs to train the model. An epoch is an iteration over the entire *x* and *y* data provided (unless the *steps_per_epoch* flag is set to something

other than None). Note that in conjunction with `initial_epoch`, `epochs` is to be understood as “final epoch”. The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.

- **verbose** – ‘auto’, 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch. ‘auto’ defaults to 1 for most cases, but 2 when used with `ParameterServerStrategy`. Note that the progress bar is not particularly useful when logged to a file, so `verbose=2` is recommended when not running interactively (eg, in a production environment).
- **callbacks** – List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See `tf.keras.callbacks`. Note `tf.keras.callbacks.ProgbarLogger` and `tf.keras.callbacks.History` callbacks are created automatically and need not be passed into `model.fit`. `tf.keras.callbacks.ProgbarLogger` is created or not based on `verbose` argument to `model.fit`. Callbacks with batch-level calls are currently unsupported with `tf.distribute.experimental.ParameterServerStrategy`, and users are advised to implement epoch-level calls instead with an appropriate `steps_per_epoch` value.
- **validation_split** – Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the `x` and `y` data provided, before shuffling. This argument is not supported when `x` is a dataset, generator or `keras.utils.Sequence` instance. If both `validation_data` and `validation_split` are provided, `validation_data` will override `validation_split`. `validation_split` is not yet supported with `tf.distribute.experimental.ParameterServerStrategy`.
- **validation_data** – Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. Thus, note the fact that the validation loss of data provided using `validation_split` or `validation_data` is not affected by regularization layers like noise and dropout. `validation_data` will override `validation_split`. `validation_data` could be:
 - A tuple (`x_val`, `y_val`) of Numpy arrays or tensors.
 - A tuple (`x_val`, `y_val`, `val_sample_weights`) of NumPy arrays.
 - A `tf.data.Dataset`.
 - A Python generator or `keras.utils.Sequence` returning (`inputs`, `targets`) or (`inputs`, `targets`, `sample_weights`).
- `validation_data` is not yet supported with `tf.distribute.experimental.ParameterServerStrategy`.
- **shuffle** – Boolean (whether to shuffle the training data before each epoch) or str (for ‘batch’). This argument is ignored when `x` is a generator or an object of `tf.data.Dataset`. ‘batch’ is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not `None`.
- **class_weight** – Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to “pay more attention” to samples from an under-represented class.
- **sample_weight** – Optional Numpy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape (`samples`, `sequence_length`), to apply a different weight to every timestep of every sample.

This argument is not supported when x is a dataset, generator, or `keras.utils.Sequence` instance, instead provide the `sample_weights` as the third element of x . Note that sample weighting does not apply to metrics specified via the `metrics` argument in `compile()`. To apply sample weighting to your metrics, you can specify them via the `weighted_metrics` in `compile()` instead.

- **initial_epoch** – Integer. Epoch at which to start training (useful for resuming a previous training run).
- **steps_per_epoch** – Integer or `None`. Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as TensorFlow data tensors, the default `None` is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined. If x is a `tf.data` dataset, and ‘`steps_per_epoch`’ is `None`, the epoch will run until the input dataset is exhausted. When passing an infinitely repeating dataset, you must specify the `steps_per_epoch` argument. If `steps_per_epoch=-1` the training will run indefinitely with an infinitely repeating dataset. This argument is not supported with array inputs. When using `tf.distribute.experimental.ParameterServerStrategy`:
 - `steps_per_epoch=None` is not supported.
- **validation_steps** – Only relevant if `validation_data` is provided and is a `tf.data` dataset. Total number of steps (batches of samples) to draw before stopping when performing validation at the end of every epoch. If ‘`validation_steps`’ is `None`, validation will run until the `validation_data` dataset is exhausted. In the case of an infinitely repeated dataset, it will run into an infinite loop. If ‘`validation_steps`’ is specified and only part of the dataset will be consumed, the evaluation will start from the beginning of the dataset at each epoch. This ensures that the same validation samples are used every time.
- **validation_batch_size** – Integer or `None`. Number of samples per validation batch. If unspecified, will default to `batch_size`. Do not specify the `validation_batch_size` if your data is in the form of datasets, generators, or `keras.utils.Sequence` instances (since they generate batches).
- **validation_freq** – Only relevant if validation data is provided. Integer or `collections.abc.Container` instance (e.g. list, tuple, etc.). If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. `validation_freq=2` runs validation every 2 epochs. If a Container, specifies the epochs on which to run validation, e.g. `validation_freq=[1, 2, 10]` runs validation at the end of the 1st, 2nd, and 10th epochs.
- **max_queue_size** – Integer. Used for generator or `keras.utils.Sequence` input only. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers** – Integer. Used for generator or `keras.utils.Sequence` input only. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1.
- **use_multiprocessing** – Boolean. Used for generator or `keras.utils.Sequence` input only. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.

Unpacking behavior for iterator-like inputs:

A common pattern is to pass a `tf.data.Dataset`, generator, or

`tf.keras.utils.Sequence` to the `x` argument of `fit`, which will in fact yield not only features (`x`) but optionally targets (`y`) and sample weights. Keras requires that the output of such iterator-likes be unambiguous. The iterator should return a tuple of length 1, 2, or 3, where the optional second and third elements will be used for `y` and `sample_weight` respectively. Any other type provided will be wrapped in a length one tuple, effectively treating everything as '`x`'. When yielding dicts, they should still adhere to the top-level tuple structure. e.g. `({"x0": x0, "x1": x1}, y)`. Keras will not attempt to separate features, targets, and weights from the keys of a single dict.

A notable unsupported data type is the `namedtuple`. The reason is that it behaves like both an ordered datatype (tuple) and a mapping datatype (dict). So given a `namedtuple` of the form:

```
namedtuple("example_tuple", ["y", "x"])
```

it is ambiguous whether to reverse the order of the elements when interpreting the value. Even worse is a tuple of the form:

```
namedtuple("other_tuple", ["x", "y", "z"])
```

where it is unclear if the tuple was intended to be unpacked into `x`, `y`, and `sample_weight` or passed through as a single element to `x`. As a result the data processing code will simply raise a `ValueError` if it encounters a `namedtuple`. (Along with instructions to remedy the issue.)

Returns A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises

- **RuntimeError** –
 1. If the model was never compiled or,
- **2. If `model.fit` is wrapped in `tf.function`.** –
- **ValueError** – In case of mismatch between the provided input data and what the model expects or when the input data is empty.

```
fit_generator(generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None,
              validation_data=None, validation_steps=None, validation_freq=1, class_weight=None,
              max_queue_size=10, workers=1, use_multiprocessing=False, shuffle=True,
              initial_epoch=0)
```

Fits the model on data yielded batch-by-batch by a Python generator.

DEPRECATED: `Model.fit` now supports generators, so there is no longer any need to use this endpoint.

```
classmethod from_config(config, custom_objects=None)
```

Creates a layer from its config.

This method is the reverse of `get_config`, capable of instantiating the same layer from the config dictionary. It does not handle layer connectivity (handled by Network), nor weights (handled by `set_weights`).

Parameters `config` – A Python dictionary, typically the output of `get_config`.

Returns A layer instance.

```
get_config()
```

Returns the config of the `Model`.

Config is a Python dictionary (serializable) containing the configuration of an object, which in this case is a `Model`. This allows the `Model` to be reinstated later (without its trained weights) from this configuration.

Note that `get_config()` does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Developers of subclassed `Model` are advised to override this method, and continue to update the dict from `super(MyModel, self).get_config()` to provide the proper configuration of this `Model`. The default config is an empty dict. Optionally, raise `NotImplementedError` to allow Keras to attempt a default serialization.

Returns Python dictionary containing the configuration of this `Model`.

`get_input_at(node_index)`

Retrieves the input tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first input node of the layer.

Returns A tensor (or list of tensors if the layer has multiple inputs).

Raises `RuntimeError` – If called in Eager mode.

`get_input_mask_at(node_index)`

Retrieves the input mask tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A mask tensor (or list of tensors if the layer has multiple inputs).

`get_input_shape_at(node_index)`

Retrieves the input shape(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A shape tuple (or list of shape tuples if the layer has multiple inputs).

Raises `RuntimeError` – If called in Eager mode.

`get_layer(name=None, index=None)`

Retrieves a layer based on either its name (unique) or index.

If `name` and `index` are both provided, `index` will take precedence. Indices are based on order of horizontal graph traversal (bottom-up).

Parameters

- `name` – String, name of layer.
- `index` – Integer, index of layer.

Returns A layer instance.

`get_metrics_result()`

Returns the model's metrics values as a dict.

If any of the metric result is a dict (containing multiple metrics), each of them gets added to the top level returned dict of this method.

Returns A `dict` containing values of the metrics listed in `self.metrics`. Example: `{'loss': 0.2, 'accuracy': 0.7}`.

`get_output_at(node_index)`

Retrieves the output tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first output node of the layer.

Returns A tensor (or list of tensors if the layer has multiple outputs).

Raises RuntimeError – If called in Eager mode.

`get_output_mask_at(node_index)`

Retrieves the output mask tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g.

`node_index=0` will correspond to the first time the layer was called.

Returns A mask tensor (or list of tensors if the layer has multiple outputs).

`get_output_shape_at(node_index)`

Retrieves the output shape(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g.

`node_index=0` will correspond to the first time the layer was called.

Returns A shape tuple (or list of shape tuples if the layer has multiple outputs).

Raises RuntimeError – If called in Eager mode.

`get_weight_paths()`

Retrieve all the variables and their paths for the model.

The variable path (string) is a stable key to identify a `tf.Variable` instance owned by the model. It can be used to specify variable-specific configurations (e.g. DTensor, quantization) from a global view.

This method returns a dict with weight object paths as keys and the corresponding `tf.Variable` instances as values.

Note that if the model is a subclassed model and the weights haven't been initialized, an empty dict will be returned.

Returns

A dict where keys are variable paths and values are `tf.Variable` instances.

Example:

```
```python
class SubclassModel(tf.keras.Model):
 def __init__(self, name=None):
 super().__init__(name=name)
 self.d1 = tf.keras.layers.Dense(10)
 self.d2 = tf.keras.layers.Dense(20)

 def call(self, inputs):
 x = self.d1(inputs)
 return self.d2(x)

model = SubclassModel()
model(tf.zeros((10, 10)))
weight_paths = model.get_weight_paths() # weight_paths: { # 'd1.kernel': model.d1.kernel, # 'd1.bias': model.d1.bias, # 'd2.kernel': model.d2.kernel, # 'd2.bias': model.d2.bias, # }

Functional model
inputs = tf.keras.Input((10,), batch_size=10)
x = tf.keras.layers.Dense(20, name='d1')(inputs)
output = tf.keras.layers.Dense(30, name='d2')(x)
model = tf.keras.Model(inputs, output)
d1 = model.layers[1]
d2 = model.layers[2]
weight_paths = model.get_weight_paths() # weight_paths: { # 'd1.kernel': d1.kernel, # 'd1.bias': d1.bias, # 'd2.kernel': d2.kernel, # 'd2.bias': d2.bias, # }
```

```

`get_weights()`

Retrieves the weights of the model.

Returns A flat list of Numpy arrays.

`property inbound_nodes`

Return Functional API nodes upstream of this layer.

property input

Retrieves the input tensor(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer.

Returns Input tensor or list of input tensors.

Raises

- **RuntimeError** – If called in Eager mode.

- **AttributeError** – If no inbound nodes are found.

property input_mask

Retrieves the input mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Input mask tensor (potentially None) or list of input mask tensors.

Raises

- **AttributeError** – if the layer is connected to

- **more than one incoming layers.** –

property input_shape

Retrieves the input shape(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer, or if all inputs have the same shape.

Returns Input shape, as an integer shape tuple (or list of shape tuples, one tuple per input tensor).

Raises

- **AttributeError** – if the layer has no defined input_shape.

- **RuntimeError** – if called in Eager mode.

property input_spec

InputSpec instance(s) describing the input format for this layer.

When you create a layer subclass, you can set *self.input_spec* to enable the layer to run input compatibility checks when it is called. Consider a *Conv2D* layer: it can only be called on a single input tensor of rank 4. As such, you can set, in *__init__()*:

```
`python self.input_spec = tf.keras.layers.InputSpec(ndim=4)`
```

Now, if you try to call the layer on an input that isn't rank 4 (for instance, an input of shape (2,), it will raise a nicely-formatted error:

```
`ValueError: Input 0 of layer conv2d is incompatible with the layer: expected ndim=4, found ndim=1. Full shape received: [2]`
```

Input checks that can be specified via *input_spec* include: - Structure (e.g. a single input, a list of 2 inputs, etc) - Shape - Rank (ndim) - Dtype

For more information, see *tf.keras.layers.InputSpec*.

Returns A *tf.keras.layers.InputSpec* instance, or nested structure thereof.

property layers

load_weights(filepath, by_name=False, skip_mismatch=False, options=None)

Loads all layer weights, either from a TensorFlow or an HDF5 weight file.

If `by_name` is False weights are loaded based on the network's topology. This means the architecture should be the same as when the weights were saved. Note that layers that don't have weights are not taken into account in the topological ordering, so adding or removing layers is fine as long as they don't have weights.

If `by_name` is True, weights are loaded into layers only if they share the same name. This is useful for fine-tuning or transfer-learning models where some of the layers have changed.

Only topological loading (`by_name=False`) is supported when loading weights from the TensorFlow format. Note that topological loading differs slightly between TensorFlow and HDF5 formats for user-defined classes inheriting from `tf.keras.Model`: HDF5 loads based on a flattened list of weights, while the TensorFlow format loads based on the object-local names of attributes to which layers are assigned in the `Model`'s constructor.

Parameters

- **filepath** – String, path to the weights file to load. For weight files in TensorFlow format, this is the file prefix (the same as was passed to `save_weights`). This can also be a path to a SavedModel saved from `model.save`.
- **by_name** – Boolean, whether to load weights by name or by topological order. Only topological loading is supported for weight files in TensorFlow format.
- **skip_mismatch** – Boolean, whether to skip loading of layers where there is a mismatch in the number of weights, or a mismatch in the shape of the weight (only valid when `by_name=True`).
- **options** – Optional `tf.train.CheckpointOptions` object that specifies options for loading weights.

Returns

When loading a weight file in TensorFlow format, returns the same status object as `tf.train.Checkpoint.restore`. When graph building, restore ops are run automatically as soon as the network is built (on first call for user-defined classes inheriting from `Model`, immediately if it is already built).

When loading weights in HDF5 format, returns `None`.

Raises

- **ImportError** – If `h5py` is not available and the weight file is in HDF5 format.
- **ValueError** – If `skip_mismatch` is set to `True` when `by_name` is `False`.

log_prob(sample: tensorflow.python.framework.ops.Tensor) → tensorflow.python.framework.ops.Tensor**property losses**

List of losses added using the `add_loss()` API.

Variable regularization tensors are created when this property is accessed, so it is eager safe: accessing `losses` under a `tf.GradientTape` will propagate gradients back to the corresponding variables.

Examples:

```
>>> class MyLayer(tf.keras.layers.Layer):
...     def call(self, inputs):
...         self.add_loss(tf.abs(tf.reduce_mean(inputs)))
...         return inputs
```

(continues on next page)

(continued from previous page)

```
>>> l = MyLayer()
>>> l(np.ones((10, 1)))
>>> l.losses
[1.0]
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> x = tf.keras.layers.Dense(10)(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Activity regularization.
>>> len(model.losses)
0
>>> model.add_loss(tf.abs(tf.reduce_mean(x)))
>>> len(model.losses)
1
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> d = tf.keras.layers.Dense(10, kernel_initializer='ones')
>>> x = d(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Weight regularization.
>>> model.add_loss(lambda: tf.reduce_mean(d.kernel))
>>> model.losses
[<tf.Tensor: shape=(), dtype=float32, numpy=1.0>]
```

Returns A list of tensors.

`make_predict_function(force=False)`

Creates a function that executes one step of inference.

This method can be overridden to support custom inference logic. This method is called by `Model.predict` and `Model.predict_on_batch`.

Typically, this method directly controls `tf.function` and `tf.distribute.Strategy` settings, and delegates the actual evaluation logic to `Model.predict_step`.

This function is cached the first time `Model.predict` or `Model.predict_on_batch` is called. The cache is cleared whenever `Model.compile` is called. You can skip the cache and generate again the function with `force=True`.

Parameters `force` – Whether to regenerate the predict function and skip the cached function if available.

Returns Function. The function created by this method should accept a `tf.data.Iterator`, and return the outputs of the `Model`.

`make_test_function(force=False)`

Creates a function that executes one step of evaluation.

This method can be overridden to support custom evaluation logic. This method is called by `Model.evaluate` and `Model.test_on_batch`.

Typically, this method directly controls `tf.function` and `tf.distribute.Strategy` settings, and delegates the actual evaluation logic to `Model.test_step`.

This function is cached the first time `Model.evaluate` or `Model.test_on_batch` is called. The cache is cleared whenever `Model.compile` is called. You can skip the cache and generate again the function with `force=True`.

Parameters force – Whether to regenerate the test function and skip the cached function if available.

Returns Function. The function created by this method should accept a `tf.data.Iterator`, and return a `dict` containing values that will be passed to `tf.keras.Callbacks.on_test_batch_end`.

`make_train_function(force=False)`

Creates a function that executes one step of training.

This method can be overridden to support custom training logic. This method is called by `Model.fit` and `Model.train_on_batch`.

Typically, this method directly controls `tf.function` and `tf.distribute.Strategy` settings, and delegates the actual training logic to `Model.train_step`.

This function is cached the first time `Model.fit` or `Model.train_on_batch` is called. The cache is cleared whenever `Model.compile` is called. You can skip the cache and generate again the function with `force=True`.

Parameters force – Whether to regenerate the train function and skip the cached function if available.

Returns Function. The function created by this method should accept a `tf.data.Iterator`, and return a `dict` containing values that will be passed to `tf.keras.Callbacks.on_train_batch_end`, such as `{'loss': 0.2, 'accuracy': 0.7}`.

`property metrics`

Returns the model's metrics added using `compile()`, `add_metric()` APIs.

Note: Metrics passed to `compile()` are available only after a `keras.Model` has been trained/evaluated on actual data.

Examples:

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> outputs = tf.keras.layers.Dense(2)(inputs)
>>> model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
>>> [m.name for m in model.metrics]
[]
```

```
>>> x = np.random.random((2, 3))
>>> y = np.random.randint(0, 2, (2, 2))
>>> model.fit(x, y)
>>> [m.name for m in model.metrics]
['loss', 'mae']
```

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2, name='out')
>>> output_1 = d(inputs)
>>> output_2 = d(inputs)
>>> model = tf.keras.models.Model(
...     inputs=inputs, outputs=[output_1, output_2])
>>> model.add_metric(
...     tf.reduce_mean(output_2), name='mean', aggregation='mean')
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae", "acc"])
```

(continues on next page)

(continued from previous page)

```
>>> model.fit(x, (y, y))
>>> [m.name for m in model.metrics]
['loss', 'out_loss', 'out_1_loss', 'out_mae', 'out_acc', 'out_1_mae',
'out_1_acc', 'mean']
```

property metrics_names

Returns the model's display labels for all outputs.

Note: *metrics_names* are available only after a *keras.Model* has been trained/evaluated on actual data.

Examples:

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> outputs = tf.keras.layers.Dense(2)(inputs)
>>> model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
>>> model.metrics_names
[]
```

```
>>> x = np.random.random((2, 3))
>>> y = np.random.randint(0, 2, (2, 2))
>>> model.fit(x, y)
>>> model.metrics_names
['loss', 'mae']
```

```
>>> inputs = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2, name='out')
>>> output_1 = d(inputs)
>>> output_2 = d(inputs)
>>> model = tf.keras.models.Model(
...     inputs=inputs, outputs=[output_1, output_2])
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae", "acc"])
>>> model.fit(x, (y, y))
>>> model.metrics_names
['loss', 'out_loss', 'out_1_loss', 'out_mae', 'out_acc', 'out_1_mae',
'out_1_acc']
```

property name

Name of the layer (string), set in the constructor.

property name_scope

Returns a *tf.name_scope* instance for this class.

property non_trainable_variables

Sequence of non-trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don't expect the return value to change.

Returns A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

property non_trainable_weights

List of all non-trainable weights tracked by this layer.

Non-trainable weights are *not* updated during training. They are expected to be updated manually in `call()`.

Returns A list of non-trainable variables.

property `outbound_nodes`

Return Functional API nodes downstream of this layer.

property `output`

Retrieves the output tensor(s) of a layer.

Only applicable if the layer has exactly one output, i.e. if it is connected to one incoming layer.

Returns Output tensor or list of output tensors.

Raises

- **AttributeError** – if the layer is connected to more than one incoming layers.
- **RuntimeError** – if called in Eager mode.

property `output_mask`

Retrieves the output mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Output mask tensor (potentially None) or list of output mask tensors.

Raises

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

property `output_shape`

Retrieves the output shape(s) of a layer.

Only applicable if the layer has one output, or if all outputs have the same shape.

Returns Output shape, as an integer shape tuple (or list of shape tuples, one tuple per output tensor).

Raises

- **AttributeError** – if the layer has no defined output shape.
- **RuntimeError** – if called in Eager mode.

**`predict`(*x*, *batch_size=None*, *verbose='auto'*, *steps=None*, *callbacks=None*, *max_queue_size=10*,
workers=1, *use_multiprocessing=False*)**

Generates output predictions for the input samples.

Computation is done in batches. This method is designed for batch processing of large numbers of inputs. It is not intended for use inside of loops that iterate over your data and process small numbers of inputs at a time.

For small numbers of inputs that fit in one batch, directly use `__call__()` for faster execution, e.g., `model(x)`, or `model(x, training=False)` if you have layers such as `tf.keras.layers.BatchNormalization` that behave differently during inference. You may pair the individual model call with a `tf.function` for additional performance inside your inner loop. If you need access to numpy array values instead of tensors after your model call, you can use `tensor.numpy()` to get the numpy array value of an eager tensor.

Also, note the fact that test loss is not affected by regularization layers like noise and dropout.

Note: See [this FAQ entry](https://keras.io/getting_started/faq/#whats-the-difference-between-model-methods-predict-and-call) for more details about the difference between *Model* methods `predict()` and `__call__()`.

Parameters

- **x** – Input samples. It could be:
 - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
 - A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
 - A *tf.data* dataset.
 - A generator or *keras.utils.Sequence* instance.

A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given in the *Unpacking behavior for iterator-like inputs* section of *Model.fit*.

- **batch_size** – Integer or *None*. Number of samples per batch. If unspecified, *batch_size* will default to 32. Do not specify the *batch_size* if your data is in the form of dataset, generators, or *keras.utils.Sequence* instances (since they generate batches).
- **verbose** – “auto”, 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line. “auto” defaults to 1 for most cases, and to 2 when used with *ParameterServerStrategy*. Note that the progress bar is not particularly useful when logged to a file, so *verbose*=2 is recommended when not running interactively (e.g. in a production environment).
- **steps** – Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of *None*. If *x* is a *tf.data* dataset and *steps* is *None*, *predict()* will run until the input dataset is exhausted.
- **callbacks** – List of *keras.callbacks.Callback* instances. List of callbacks to apply during prediction. See [callbacks](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks).
- **max_queue_size** – Integer. Used for generator or *keras.utils.Sequence* input only. Maximum size for the generator queue. If unspecified, *max_queue_size* will default to 10.
- **workers** – Integer. Used for generator or *keras.utils.Sequence* input only. Maximum number of processes to spin up when using process-based threading. If unspecified, *workers* will default to 1.
- **use_multiprocessing** – Boolean. Used for generator or *keras.utils.Sequence* input only. If *True*, use process-based threading. If unspecified, *use_multiprocessing* will default to *False*. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can’t be passed easily to children processes.

See the discussion of *Unpacking behavior for iterator-like inputs* for *Model.fit*. Note that *Model.predict* uses the same interpretation rules as *Model.fit* and *Model.evaluate*, so inputs must be unambiguous for all three methods.

Returns Numpy array(s) of predictions.

Raises

- **RuntimeError** – If *model.predict* is wrapped in a *tf.function*.
- **ValueError** – In case of mismatch between the provided input data and the model’s expectations, or in case a stateful model receives a number of samples that is not a multiple of the batch size.

```
predict_generator(generator, steps=None, callbacks=None, max_queue_size=10, workers=1,  
use_multiprocessing=False, verbose=0)
```

Generates predictions for the input samples from a data generator.

DEPRECATED: *Model.predict* now supports generators, so there is no longer any need to use this endpoint.

```
predict_on_batch(x)
```

Returns predictions for a single batch of samples.

Parameters **x** – Input data. It could be: - A Numpy array (or array-like), or a list of arrays (in case the

model has multiple inputs).

- **A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).**

Returns Numpy array(s) of predictions.

Raises **RuntimeError** – If *model.predict_on_batch* is wrapped in a *tf.function*.

```
predict_step(data)
```

The logic for one inference step.

This method can be overridden to support custom inference logic. This method is called by *Model.make_predict_function*.

This method should contain the mathematical logic for one step of inference. This typically includes the forward pass.

Configuration details for how this logic is run (e.g. *tf.function* and *tf.distribute.Strategy* settings), should be left to *Model.make_predict_function*, which can also be overridden.

Parameters **data** – A nested structure of `Tensor`'s.

Returns The result of one inference step, typically the output of calling the *Model* on data.

```
reset_metrics()
```

Resets the state of all the metrics in the model.

Examples:

```
>>> inputs = tf.keras.layers.Input(shape=(3,))  
>>> outputs = tf.keras.layers.Dense(2)(inputs)  
>>> model = tf.keras.models.Model(inputs=inputs, outputs=outputs)  
>>> model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
```

```
>>> x = np.random.random((2, 3))  
>>> y = np.random.randint(0, 2, (2, 2))  
>>> _ = model.fit(x, y, verbose=0)  
>>> assert all(float(m.result()) == 0 for m in model.metrics)
```

```
>>> model.reset_metrics()  
>>> assert all(float(m.result()) == 0 for m in model.metrics)
```

```
reset_states()
```

```
property run_eagerly
```

Settable attribute indicating whether the model should run eagerly.

Running eagerly means that your model will be run step by step, like Python code. Your model might run slower, but it should become easier for you to debug it by stepping into individual layer calls.

By default, we will attempt to compile your model to a static graph to deliver the best execution performance.

Returns Boolean, whether the model should run eagerly.

sample(batch_size: int, h: int, w: int) → Tuple[numpy.array, tensorflow.python.framework.ops.Tensor]

save(filepath, overwrite=True, include_optimizer=True, save_format=None, signatures=None, options=None, save_traces=True)

Saves the model to Tensorflow SavedModel or a single HDF5 file.

Please see `tf.keras.models.save_model` or the [Serialization and Saving guide](https://keras.io/guides/serialization_and_saving/) for details.

Parameters

- **filepath** – String, PathLike, path to SavedModel or H5 file to save the model.
- **overwrite** – Whether to silently overwrite any existing file at the target location, or provide the user with a manual prompt.
- **include_optimizer** – If True, save optimizer's state together.
- **save_format** – Either ‘tf’ or ‘h5’, indicating whether to save the model to Tensorflow SavedModel or HDF5. Defaults to ‘tf’ in TF 2.X, and ‘h5’ in TF 1.X.
- **signatures** – Signatures to save with the SavedModel. Applicable to the ‘tf’ format only. Please see the `signatures` argument in `tf.saved_model.save` for details.
- **options** – (only applies to SavedModel format) `tf.saved_model.SaveOptions` object that specifies options for saving to SavedModel.
- **save_traces** – (only applies to SavedModel format) When enabled, the SavedModel will store the function traces for each layer. This can be disabled, so that only the configs of each layer are stored. Defaults to `True`. Disabling this will decrease serialization time and reduce file size, but it requires that all custom layers/models implement a `get_config()` method.

Example:

```
```python
from keras.models import load_model
model.save('my_model.h5') # creates a HDF5 file 'my_model.h5' del model # deletes the existing model
returns a compiled model # identical to the previous one
model = load_model('my_model.h5')
```

```

save_spec(dynamic_batch=True)

Returns the `tf.TensorSpec` of call inputs as a tuple (`args, kwargs`).

This value is automatically defined after calling the model for the first time. Afterwards, you can use it when exporting the model for serving:

```
```python
model = tf.keras.Model(...)
@tf.function def serve(*args, **kwargs):
 outputs = model(*args, **kwargs) # Apply postprocessing steps, or add additional outputs. ...
 return outputs
arg_specs is [tf.TensorSpec(...), ...]. kwarg_specs, in this # example, is an empty dict since functional
models do not use keyword # arguments.
arg_specs, kwarg_specs = model.save_spec()
model.save(path, signatures={})
```

```

'serving_default': serve.get_concrete_function(*arg_specs, **kwarg_specs)

Parameters **dynamic_batch** – Whether to set the batch sizes of all the returned *tf.TensorSpec* to *None*. (Note that when defining functional or Sequential models with *tf.keras.Input(..., batch_size=X)*, the batch size will always be preserved). Defaults to *True*.

Returns If the model inputs are defined, returns a tuple (*args*, *kwargs*). All elements in *args* and *kwargs* are *tf.TensorSpec*. If the model inputs are not defined, returns *None*. The model inputs are automatically set when calling the model, *model.fit*, *model.evaluate* or *model.predict*.

save_weights(*filepath*, *overwrite=True*, *save_format=None*, *options=None*)

Saves all layer weights.

Either saves in HDF5 or in TensorFlow format based on the *save_format* argument.

When saving in HDF5 format, the weight file has:

- ***layer_names* (attribute), a list of strings** (ordered names of model layers).
- **For every layer, a group named *layer.name***
 - **For every such layer group, a group attribute *weight_names***, a list of strings (ordered names of weights tensor of the layer).
 - **For every weight in the layer, a dataset** storing the weight value, named after the weight tensor.

When saving in TensorFlow format, all objects referenced by the network are saved in the same format as *tf.train.Checkpoint*, including any *Layer* instances or *Optimizer* instances assigned to object attributes. For networks constructed from inputs and outputs using *tf.keras.Model(inputs, outputs)*, *Layer* instances used by the network are tracked/saved automatically. For user-defined classes which inherit from *tf.keras.Model*, *Layer* instances must be assigned to object attributes, typically in the constructor. See the documentation of *tf.train.Checkpoint* and *tf.keras.Model* for details.

While the formats are the same, do not mix *save_weights* and *tf.train.Checkpoint*. Checkpoints saved by *Model.save_weights* should be loaded using *Model.load_weights*. Checkpoints saved using *tf.train.Checkpoint.save* should be restored using the corresponding *tf.train.Checkpoint.restore*. Prefer *tf.train.Checkpoint* over *save_weights* for training checkpoints.

The TensorFlow format matches objects and variables by starting at a root object, *self* for *save_weights*, and greedily matching attribute names. For *Model.save* this is the *Model*, and for *Checkpoint.save* this is the *Checkpoint* even if the *Checkpoint* has a model attached. This means saving a *tf.keras.Model* using *save_weights* and loading into a *tf.train.Checkpoint* with a *Model* attached (or vice versa) will not match the *Model*'s variables. See the [guide to training checkpoints](<https://www.tensorflow.org/guide/checkpoint>) for details on the TensorFlow format.

Parameters

- ***filepath*** – String or PathLike, path to the file to save the weights to. When saving in TensorFlow format, this is the prefix used for checkpoint files (multiple files are generated). Note that the '.h5' suffix causes weights to be saved in HDF5 format.
- ***overwrite*** – Whether to silently overwrite any existing file at the target location, or provide the user with a manual prompt.
- ***save_format*** – Either 'tf' or 'h5'. A *filepath* ending in '.h5' or '.keras' will default to HDF5 if *save_format* is *None*. Otherwise *None* defaults to 'tf'.

- **options** – Optional `tf.train.CheckpointOptions` object that specifies options for saving weights.

Raises ImportError – If `h5py` is not available when attempting to save in HDF5 format.

`set_weights(weights)`

Sets the weights of the layer, from NumPy arrays.

The weights of a layer represent the state of the layer. This function sets the weight values from numpy arrays. The weight values should be passed in the order they are created by the layer. Note that the layer's weights must be instantiated before calling this function, by calling the layer.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
       [2.],
       [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
```

Parameters **weights** – a list of NumPy arrays. The number of arrays and their shape must match number of the dimensions of the weights of the layer (i.e. it should match the output of `get_weights`).

Raises ValueError – If the provided weights list does not match the layer's specifications.

`property state_updates`

Deprecated, do NOT use!

Returns the *updates* from all layers that are stateful.

This is useful for separating training updates and state updates, e.g. when we need to update a layer's internal state during prediction.

Returns A list of update ops.

`property stateful`

`property submodules`

Sequence of all sub-modules.

Submodules are modules which are properties of this module, or found as properties of modules which are properties of this module (and so on).

```
>>> a = tf.Module()
>>> b = tf.Module()
>>> c = tf.Module()
>>> a.b = b
>>> b.c = c
>>> list(a.submodules) == [b, c]
True
>>> list(b.submodules) == [c]
True
>>> list(c.submodules) == []
True
```

Returns A sequence of all submodules.

summary(*line_length=None*, *positions=None*, *print_fn=None*, *expand_nested=False*, *show_trainable=False*, *layer_range=None*)

Prints a string summary of the network.

Parameters

- **line_length** – Total length of printed lines (e.g. set this to adapt the display to different terminal window sizes).
- **positions** – Relative or absolute positions of log elements in each line. If not provided, defaults to [.33, .55, .67, 1].
- **print_fn** – Print function to use. Defaults to *print*. It will be called on each line of the summary. You can set it to a custom function in order to capture the string summary.
- **expand_nested** – Whether to expand the nested models. If not provided, defaults to *False*.
- **show_trainable** – Whether to show if a layer is trainable. If not provided, defaults to *False*.
- **layer_range** – a list or tuple of 2 strings, which is the starting layer name and ending layer name (both inclusive) indicating the range of layers to be printed in summary. It also accepts regex patterns instead of exact name. In such case, start predicate will be the first element it matches to *layer_range[0]* and the end predicate will be the last element it matches to *layer_range[1]*. By default *None* which considers all layers of model.

Raises **ValueError** – if *summary()* is called before the model is built.

property supports_masking

Whether this layer supports computing a mask using *compute_mask*.

test_on_batch(*x*, *y=None*, *sample_weight=None*, *reset_metrics=True*, *return_dict=False*)

Test the model on a single batch of samples.

Parameters

- **x** – Input data. It could be: - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
 - **A TensorFlow tensor, or a list of tensors (in case the model has** multiple inputs).

- A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
- **y** – Target data. Like the input data x , it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with x (you cannot have Numpy inputs and tensor targets, or inversely).
- **sample_weight** – Optional array of the same length as x , containing weights to apply to the model’s loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample.
- **reset_metrics** – If *True*, the metrics returned will be only for this batch. If *False*, the metrics will be statefully accumulated across batches.
- **return_dict** – If *True*, loss and metric results are returned as a dict, with each key being the name of the metric. If *False*, they are returned as a list.

Returns Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute *model.metrics_names* will give you the display labels for the scalar outputs.

Raises `RuntimeError` – If *model.test_on_batch* is wrapped in a *tf.function*.

test_step(*data*)

The logic for one evaluation step.

This method can be overridden to support custom evaluation logic. This method is called by *Model.make_test_function*.

This function should contain the mathematical logic for one step of evaluation. This typically includes the forward pass, loss calculation, and metrics updates.

Configuration details for how this logic is run (e.g. *tf.function* and *tf.distribute.Strategy* settings), should be left to *Model.make_test_function*, which can also be overridden.

Parameters **data** – A nested structure of `Tensor`’s.

Returns A *dict* containing values that will be passed to *tf.keras.callbacks.CallbackList.on_train_batch_end*. Typically, the values of the *Model*’s metrics are returned.

to_json(***kwargs*)

Returns a JSON string containing the network configuration.

To load a network from a JSON save file, use *keras.models.model_from_json(json_string, custom_objects={})*.

Parameters ****kwargs** – Additional keyword arguments to be passed to **json.dumps()*.

Returns A JSON string.

to_yaml(***kwargs*)

Returns a yaml string containing the network configuration.

Note: Since TF 2.6, this method is no longer supported and will raise a *RuntimeError*.

To load a network from a yaml save file, use *keras.models.model_from_yaml(yaml_string, custom_objects={})*.

custom_objects should be a dictionary mapping the names of custom losses / layers / etc to the corresponding functions / classes.

Parameters ****kwargs** – Additional keyword arguments to be passed to *yaml.dump()*.

Returns A YAML string.

Raises RuntimeError – announces that the method poses a security risk

train_on_batch(*x*, *y*=*None*, *sample_weight*=*None*, *class_weight*=*None*, *reset_metrics*=*True*,
return_dict=*False*)

Runs a single gradient update on a single batch of data.

Parameters

- **x** – Input data. It could be: - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
 - **A TensorFlow tensor, or a list of tensors** (in case the model has multiple inputs).
 - **A dict mapping input names to the corresponding array/tensors**, if the model has named inputs.
- **y** – Target data. Like the input data *x*, it could be either Numpy array(s) or TensorFlow tensor(s).
- **sample_weight** – Optional array of the same length as *x*, containing weights to apply to the model’s loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample.
- **class_weight** – Optional dictionary mapping class indices (integers) to a weight (float) to apply to the model’s loss for the samples from this class during training. This can be useful to tell the model to “pay more attention” to samples from an under-represented class.
- **reset_metrics** – If *True*, the metrics returned will be only for this batch. If *False*, the metrics will be statefully accumulated across batches.
- **return_dict** – If *True*, loss and metric results are returned as a dict, with each key being the name of the metric. If *False*, they are returned as a list.

Returns Scalar training loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute *model.metrics_names* will give you the display labels for the scalar outputs.

Raises RuntimeError – If *model.train_on_batch* is wrapped in a *tf.function*.

train_step(*data*)

The logic for one training step.

This method can be overridden to support custom training logic. For concrete examples of how to override this method see [Customizing what happens in fit](https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit). This method is called by *Model.make_train_function*.

This method should contain the mathematical logic for one step of training. This typically includes the forward pass, loss calculation, backpropagation, and metric updates.

Configuration details for *how* this logic is run (e.g. *tf.function* and *tf.distribute.Strategy* settings), should be left to *Model.make_train_function*, which can also be overridden.

Parameters **data** – A nested structure of `Tensor`’s.

Returns A *dict* containing values that will be passed to *tf.keras.callbacks.CallbackList.on_train_batch_end*. Typically, the values of the *Model*’s metrics are returned. Example: {‘loss’: 0.2, ‘accuracy’: 0.7}.

property trainable**property trainable_variables**

Sequence of trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don't expect the return value to change.

Returns A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

property trainable_weights

List of all trainable weights tracked by this layer.

Trainable weights are updated via gradient descent during training.

Returns A list of trainable variables.

property updates**property variable_dtype**

Alias of *Layer.dtype*, the dtype of the weights.

property variables

Returns the list of all layer variables/weights.

Alias of *self.weights*.

Note: This will not track the weights of nested *tf.Modules* that are not themselves Keras layers.

Returns A list of variables.

property weights

Returns the list of all layer variables/weights.

Note: This will not track the weights of nested *tf.Modules* that are not themselves Keras layers.

Returns A list of variables.

classmethod with_name_scope(method)

Decorator to automatically enter the module name scope.

```
>>> class MyModule(tf.Module):
...     @tf.Module.with_name_scope
...     def __call__(self, x):
...         if not hasattr(self, 'w'):
...             self.w = tf.Variable(tf.random.normal([x.shape[1], 3]))
...         return tf.matmul(x, self.w)
```

Using the above module would produce `tf.Variable`'s and `tf.Tensor`'s whose names included the module name:

```
>>> mod = MyModule()
>>> mod(tf.ones([1, 2]))
<tf.Tensor: shape=(1, 3), dtype=float32, numpy=..., dtype=float32>
>>> mod.w
<tf.Variable 'my_module/Variable:0' shape=(2, 3) dtype=float32,
numpy=..., dtype=float32>
```

Parameters **method** – The method to wrap.

Returns The original method wrapped such that it enters the module's name scope.

```
class tensorcircuit.applications.van.ResidualBlock(*args, **kwargs)
```

Bases: keras.engine.base_layer.Layer

```
__init__(layers: List[Any])
```

```
property activity_regularizer
```

Optional regularizer function for the output of this layer.

```
add_loss(losses, **kwargs)
```

Add loss tensor(s), potentially dependent on layer inputs.

Some losses (for instance, activity regularization losses) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs a and b , some entries in $layer.losses$ may be dependent on a and some on b . This method automatically keeps track of dependencies.

This method can be used inside a subclassed layer or model's *call* function, in which case *losses* should be a Tensor or list of Tensors.

Example:

```
```python class MyLayer(tf.keras.layers.Layer):
```

```
 def call(self, inputs): self.add_loss(tf.abs(tf.reduce_mean(inputs))) return inputs
```

```
```
```

This method can also be called directly on a Functional Model during construction. In this case, any loss Tensors passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These losses become part of the model's topology and are tracked in 'get_config'.*

Example:

```
```python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) # Activity regularization. model.add_loss(tf.abs(tf.reduce_mean(x)))``
```

If this is not the case for your loss (if, for example, your loss references a *Variable* of one of the model's layers), you can wrap your loss in a zero-argument lambda. These losses are not tracked as part of the model's topology since they can't be serialized.

Example:

```
```python inputs = tf.keras.Input(shape=(10,)) d = tf.keras.layers.Dense(10) x = d(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) # Weight regularization. model.add_loss(lambda: tf.reduce_mean(d.kernel))``
```

Parameters

- **losses** – Loss tensor, or list/tuple of tensors. Rather than tensors, losses may also be zero-argument callables which create a loss tensor.
- ****kwargs** – Used for backwards compatibility only.

```
add_metric(value, name=None, **kwargs)
```

Adds metric tensor to the layer.

This method can be used inside the *call()* method of a subclassed layer or model.

```
```python class MyMetricLayer(tf.keras.layers.Layer):
```

```
def __init__(self): super(MyMetricLayer, self).__init__(name='my_metric_layer') self.mean = tf.keras.metrics.Mean(name='metric_1')

def call(self, inputs): self.add_metric(self.mean(inputs)) self.add_metric(tf.reduce_sum(inputs), name='metric_2') return inputs
```

...

This method can also be called directly on a Functional Model during construction. In this case, any tensor passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These metrics become part of the model's topology and are tracked when you save the model via `save()`.*

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) model.add_metric(math_ops.reduce_sum(x), name='metric_1')`
```

Note: Calling *add\_metric()* with the result of a metric object on a Functional Model, as shown in the example below, is not supported. This is because we cannot trace the metric result tensor back to the model's inputs.

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) model.add_metric(tf.keras.metrics.Mean()(x), name='metric_1')`
```

#### Parameters

- **value** – Metric tensor.
- **name** – String metric name.
- **\*\*kwargs** – Additional keyword arguments for backward compatibility. Accepted values: *aggregation* - When the *value* tensor provided is not the result of calling a *keras.Metric* instance, it will be aggregated by default using a *keras.Metric.Mean*.

#### add\_update(updates)

Add update op(s), potentially dependent on layer inputs.

Weight updates (for instance, the updates of the moving mean and variance in a BatchNormalization layer) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs *a* and *b*, some entries in *layer.updates* may be dependent on *a* and some on *b*. This method automatically keeps track of dependencies.

This call is ignored when eager execution is enabled (in that case, variable updates are run on the fly and thus do not need to be tracked for later execution).

**Parameters** **updates** – Update op, or list/tuple of update ops, or zero-arg callable that returns an update op. A zero-arg callable should be passed in order to disable running the updates by setting *trainable=False* on this Layer, when executing in Eager mode.

#### add\_variable(\*args, \*\*kwargs)

Deprecated, do NOT use! Alias for *add\_weight*.

#### add\_weight(name=None, shape=None, dtype=None, initializer=None, regularizer=None, trainable=None, constraint=None, use\_resource=None, synchronization=VariableSynchronization.AUTO, aggregation=VariableAggregationV2.NONE, \*\*kwargs)

Adds a new variable to the layer.

#### Parameters

- **name** – Variable name.

- **shape** – Variable shape. Defaults to scalar if unspecified.
- **dtype** – The type of the variable. Defaults to `self.dtype`.
- **initializer** – Initializer instance (callable).
- **regularizer** – Regularizer instance (callable).
- **trainable** – Boolean, whether the variable should be part of the layer’s “trainable\_variables” (e.g. variables, biases) or “non\_trainable\_variables” (e.g. BatchNorm mean and variance). Note that `trainable` cannot be `True` if `synchronization` is set to `ON_READ`.
- **constraint** – Constraint instance (callable).
- **use\_resource** – Whether to use a `ResourceVariable` or not. See [this guide]([https://www.tensorflow.org/guide/migrate/tf1\\_vs\\_tf2#resourcevariables\\_instead\\_of\\_referencevariables](https://www.tensorflow.org/guide/migrate/tf1_vs_tf2#resourcevariables_instead_of_referencevariables))  
for more information.

- **synchronization** – Indicates when a distributed a variable will be aggregated. Accepted values are constants defined in the class `tf.VariableSynchronization`. By default the synchronization is set to `AUTO` and the current `DistributionStrategy` chooses when to synchronize. If `synchronization` is set to `ON_READ`, `trainable` must not be set to `True`.
- **aggregation** – Indicates how a distributed variable will be aggregated. Accepted values are constants defined in the class `tf.VariableAggregation`.
- **\*\*kwargs** – Additional keyword arguments. Accepted values are `getter`, `collections`, `experimental_autocast` and `caching_device`.

**Returns** The variable created.

**Raises ValueError** – When giving unsupported dtype and no initializer or when trainable has been set to True with synchronization set as `ON_READ`.

### `build(input_shape)`

Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of `Layer` or `Model` can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of `call()`.

This is typically used to create the weights of `Layer` subclasses (at the discretion of the subclass implementer).

**Parameters** `input_shape` – Instance of `TensorShape`, or list of instances of `TensorShape` if the layer expects a list of inputs (one instance per input).

### `call(inputs: tensorflow.python.framework.ops.Tensor) → tensorflow.python.framework.ops.Tensor`

This is where the layer’s logic lives.

The `call()` method may not create state (except in its first invocation, wrapping the creation of variables or other resources in `tf.init_scope()`). It is recommended to create state, including `tf.Variable` instances and nested `Layer` instances,

in `__init__()`, or in the `build()` method that is

called automatically before `call()` executes for the first time.

### Parameters

- **inputs** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero
  - arguments, and *inputs* cannot be provided via the default value of a keyword argument.
  - NumPy array or Python scalar values in *inputs* get cast as tensors.
  - Keras mask metadata is only collected from *inputs*.
  - Layers are built (*build(input\_shape)* method) using shape info from *inputs* only.
  - *input\_spec* compatibility is only checked against *inputs*.
  - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in *\*args* or *\*\*kwargs*, their casting behavior in mixed precision should be handled manually.
  - The SavedModel input specification is generated using *inputs* only.
  - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.
- **\*args** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- **\*\*kwargs** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved:
  - *training*: Boolean scalar tensor of Python boolean indicating whether the *call* is meant for training or inference.
  - *mask*: Boolean input mask. If the layer’s *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

**Returns** A tensor or list/tuple of tensors.

#### **property compute\_dtype**

The dtype of the layer’s computations.

This is equivalent to *Layer.dtype\_policy.compute\_dtype*. Unless mixed precision is used, this is the same as *Layer.dtype*, the dtype of the weights.

Layers automatically cast their inputs to the compute dtype, which causes computations and the output to be in the compute dtype as well. This is done by the base Layer class in *Layer.\_\_call\_\_*, so you do not have to insert these casts if implementing your own layer.

Layers often perform certain internal computations in higher precision when *compute\_dtype* is float16 or bfloat16 for numeric stability. The output will still typically be float16 or bfloat16 in such cases.

**Returns** The layer’s compute dtype.

#### **compute\_mask(inputs, mask=None)**

Computes an output mask tensor.

#### **Parameters**

- **inputs** – Tensor or list of tensors.
- **mask** – Tensor or list of tensors.

**Returns**

**None or a tensor (or list of tensors,** one per output tensor of the layer).

**compute\_output\_shape(*input\_shape*)**

Computes the output shape of the layer.

This method will cause the layer's state to be built, if that has not happened before. This requires that the layer will later be used with inputs that match the input shape provided here.

**Parameters** **input\_shape** – Shape tuple (tuple of integers) or *tf.TensorShape*, or structure of shape tuples / *tf.TensorShape* instances (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

**Returns** A *tf.TensorShape* instance or structure of *tf.TensorShape* instances.

**compute\_output\_signature(*input\_signature*)**

Compute the output tensor signature of the layer based on the inputs.

Unlike a *TensorShape* object, a *TensorSpec* object contains both shape and dtype information for a tensor. This method allows layers to provide output dtype information if it is different from the input dtype. For any layer that doesn't implement this function, the framework will fall back to use *compute\_output\_shape*, and will assume that the output dtype matches the input dtype.

**Parameters** **input\_signature** – Single *TensorSpec* or nested structure of *TensorSpec* objects, describing a candidate input for the layer.

**Returns**

**Single *TensorSpec* or nested structure of *TensorSpec* objects,** describing how the layer would transform the provided input.

**Raises** **TypeError** – If *input\_signature* contains a non-*TensorSpec* object.

**count\_params()**

Count the total number of scalars composing the weights.

**Returns** An integer count.

**Raises** **ValueError** – if the layer isn't yet built (in which case its weights aren't yet defined).

**property dtype**

The dtype of the layer weights.

This is equivalent to *Layer.dtype\_policy.variable\_dtype*. Unless mixed precision is used, this is the same as *Layer.compute\_dtype*, the dtype of the layer's computations.

**property dtype\_policy**

The dtype policy associated with this layer.

This is an instance of a *tf.keras.mixed\_precision.Policy*.

**property dynamic**

Whether the layer is dynamic (eager-only); set in the constructor.

**finalize\_state()**

Finalizes the layers state after updating layer weights.

This function can be subclassed in a layer and will be called after updating a layer weights. It can be overridden to finalize any additional layer state after a weight update.

This function will be called after weights of a layer have been restored from a loaded model.

**classmethod from\_config(*config*)**

Creates a layer from its config.

This method is the reverse of `get_config`, capable of instantiating the same layer from the config dictionary. It does not handle layer connectivity (handled by Network), nor weights (handled by `set_weights`).

**Parameters** `config` – A Python dictionary, typically the output of `get_config`.

**Returns** A layer instance.

### `get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that `get_config()` does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

**Returns** Python dictionary.

### `get_input_at(node_index)`

Retrieves the input tensor(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g.  
`node_index=0` will correspond to the first input node of the layer.

**Returns** A tensor (or list of tensors if the layer has multiple inputs).

**Raises** `RuntimeError` – If called in Eager mode.

### `get_input_mask_at(node_index)`

Retrieves the input mask tensor(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g.  
`node_index=0` will correspond to the first time the layer was called.

**Returns** A mask tensor (or list of tensors if the layer has multiple inputs).

### `get_input_shape_at(node_index)`

Retrieves the input shape(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g.  
`node_index=0` will correspond to the first time the layer was called.

**Returns** A shape tuple (or list of shape tuples if the layer has multiple inputs).

**Raises** `RuntimeError` – If called in Eager mode.

### `get_output_at(node_index)`

Retrieves the output tensor(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g.  
`node_index=0` will correspond to the first output node of the layer.

**Returns** A tensor (or list of tensors if the layer has multiple outputs).

**Raises** `RuntimeError` – If called in Eager mode.

### `get_output_mask_at(node_index)`

Retrieves the output mask tensor(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g.  
`node_index=0` will correspond to the first time the layer was called.

**Returns** A mask tensor (or list of tensors if the layer has multiple outputs).

**get\_output\_shape\_at(node\_index)**

Retrieves the output shape(s) of a layer at a given node.

**Parameters** `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

**Returns** A shape tuple (or list of shape tuples if the layer has multiple outputs).

**Raises** `RuntimeError` – If called in Eager mode.

**get\_weights()**

Returns the current weights of the layer, as NumPy arrays.

The weights of a layer represent the state of the layer. This function returns both trainable and non-trainable weight values associated with this layer as a list of NumPy arrays, which can in turn be used to load state into similarly parameterized layers.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
... kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
 [1.],
 [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
... kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
 [2.],
 [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
 [1.],
 [1.]], dtype=float32), array([0.], dtype=float32)]
```

**Returns** Weights values as a list of NumPy arrays.

**property inbound\_nodes**

Return Functional API nodes upstream of this layer.

**property input**

Retrieves the input tensor(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer.

**Returns** Input tensor or list of input tensors.

**Raises**

- `RuntimeError` – If called in Eager mode.

- `AttributeError` – If no inbound nodes are found.

**property input\_mask**

Retrieves the input mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

**Returns** Input mask tensor (potentially None) or list of input mask tensors.

**Raises**

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

#### property `input_shape`

Retrieves the input shape(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer, or if all inputs have the same shape.

**Returns** Input shape, as an integer shape tuple (or list of shape tuples, one tuple per input tensor).

**Raises**

- **AttributeError** – if the layer has no defined `input_shape`.
- **RuntimeError** – if called in Eager mode.

#### property `input_spec`

`InputSpec` instance(s) describing the input format for this layer.

When you create a layer subclass, you can set `self.input_spec` to enable the layer to run input compatibility checks when it is called. Consider a `Conv2D` layer: it can only be called on a single input tensor of rank 4. As such, you can set, in `__init__()`:

```
`python self.input_spec = tf.keras.layers.InputSpec(ndim=4)`
```

Now, if you try to call the layer on an input that isn't rank 4 (for instance, an input of shape (2, ), it will raise a nicely-formatted error:

```
`ValueError: Input 0 of layer conv2d is incompatible with the layer: expected ndim=4, found ndim=1. Full shape received: [2]`
```

Input checks that can be specified via `input_spec` include:

- Structure (e.g. a single input, a list of 2 inputs, etc)
- Shape
- Rank (ndim)
- Dtype

For more information, see `tf.keras.layers.InputSpec`.

**Returns** A `tf.keras.layers.InputSpec` instance, or nested structure thereof.

#### property `losses`

List of losses added using the `add_loss()` API.

Variable regularization tensors are created when this property is accessed, so it is eager safe: accessing `losses` under a `tf.GradientTape` will propagate gradients back to the corresponding variables.

Examples:

```
>>> class MyLayer(tf.keras.layers.Layer):
... def call(self, inputs):
... self.add_loss(tf.abs(tf.reduce_mean(inputs)))
... return inputs
>>> l = MyLayer()
>>> l(np.ones((10, 1)))
>>> l.losses
[1.0]
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> x = tf.keras.layers.Dense(10)(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Activity regularization.
>>> len(model.losses)
0
>>> model.add_loss(tf.abs(tf.reduce_mean(x)))
>>> len(model.losses)
1

>>> inputs = tf.keras.Input(shape=(10,))
>>> d = tf.keras.layers.Dense(10, kernel_initializer='ones')
>>> x = d(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Weight regularization.
>>> model.add_loss(lambda: tf.reduce_mean(d.kernel))
>>> model.losses
[<tf.Tensor: shape=(), dtype=float32, numpy=1.0>]
```

**Returns** A list of tensors.

#### property metrics

List of metrics added using the `add_metric()` API.

Example:

```
>>> input = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2)
>>> output = d(input)
>>> d.add_metric(tf.reduce_max(output), name='max')
>>> d.add_metric(tf.reduce_min(output), name='min')
>>> [m.name for m in d.metrics]
['max', 'min']
```

**Returns** A list of *Metric* objects.

#### property name

Name of the layer (string), set in the constructor.

#### property name\_scope

Returns a `tf.name_scope` instance for this class.

#### property non\_trainable\_variables

Sequence of non-trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don't expect the return value to change.

**Returns** A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

**property non\_trainable\_weights**

List of all non-trainable weights tracked by this layer.

Non-trainable weights are *not* updated during training. They are expected to be updated manually in `call()`.

**Returns** A list of non-trainable variables.

**property outbound\_nodes**

Return Functional API nodes downstream of this layer.

**property output**

Retrieves the output tensor(s) of a layer.

Only applicable if the layer has exactly one output, i.e. if it is connected to one incoming layer.

**Returns** Output tensor or list of output tensors.

**Raises**

- **AttributeError** – if the layer is connected to more than one incoming layers.
- **RuntimeError** – if called in Eager mode.

**property output\_mask**

Retrieves the output mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

**Returns** Output mask tensor (potentially None) or list of output mask tensors.

**Raises**

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

**property output\_shape**

Retrieves the output shape(s) of a layer.

Only applicable if the layer has one output, or if all outputs have the same shape.

**Returns** Output shape, as an integer shape tuple (or list of shape tuples, one tuple per output tensor).

**Raises**

- **AttributeError** – if the layer has no defined output shape.
- **RuntimeError** – if called in Eager mode.

**set\_weights(weights)**

Sets the weights of the layer, from NumPy arrays.

The weights of a layer represent the state of the layer. This function sets the weight values from numpy arrays. The weight values should be passed in the order they are created by the layer. Note that the layer's weights must be instantiated before calling this function, by calling the layer.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
... kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.]],
```

(continues on next page)

(continued from previous page)

```
[1.],
[1.], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
... kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
 [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
 [1.]], dtype=float32), array([0.], dtype=float32)]
```

**Parameters** `weights` – a list of NumPy arrays. The number of arrays and their shape must match number of the dimensions of the weights of the layer (i.e. it should match the output of `get_weights`).

**Raises** `ValueError` – If the provided weights list does not match the layer’s specifications.

#### property stateful

#### property submodules

Sequence of all sub-modules.

Submodules are modules which are properties of this module, or found as properties of modules which are properties of this module (and so on).

```
>>> a = tf.Module()
>>> b = tf.Module()
>>> c = tf.Module()
>>> a.b = b
>>> b.c = c
>>> list(a.submodules) == [b, c]
True
>>> list(b.submodules) == [c]
True
>>> list(c.submodules) == []
True
```

**Returns** A sequence of all submodules.

#### property supports\_masking

Whether this layer supports computing a mask using `compute_mask`.

#### property trainable

#### property trainable\_variables

Sequence of trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don’t expect the return value to change.

**Returns** A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

**property trainable\_weights**

List of all trainable weights tracked by this layer.

Trainable weights are updated via gradient descent during training.

**Returns** A list of trainable variables.

**property updates**

**property variable\_dtype**

Alias of *Layer.dtype*, the dtype of the weights.

**property variables**

Returns the list of all layer variables/weights.

Alias of *self.weights*.

Note: This will not track the weights of nested *tf.Modules* that are not themselves Keras layers.

**Returns** A list of variables.

**property weights**

Returns the list of all layer variables/weights.

**Returns** A list of variables.

**classmethod with\_name\_scope(method)**

Decorator to automatically enter the module name scope.

```
>>> class MyModule(tf.Module):
... @tf.Module.with_name_scope
... def __call__(self, x):
... if not hasattr(self, 'w'):
... self.w = tf.Variable(tf.random.normal([x.shape[1], 3]))
... return tf.matmul(x, self.w)
```

Using the above module would produce `tf.Variable`'s and `tf.Tensor`'s whose names included the module name:

```
>>> mod = MyModule()
>>> mod(tf.ones([1, 2]))
<tf.Tensor: shape=(1, 3), dtype=float32, numpy=..., dtype=float32>
>>> mod.w
<tf.Variable 'my_module/Variable:0' shape=(2, 3) dtype=float32,
numpy=..., dtype=float32>
```

**Parameters** **method** – The method to wrap.

**Returns** The original method wrapped such that it enters the module's name scope.

**tensorcircuit.applications.vqes**

VQNHE application

```
class tensorcircuit.applications.vqes.JointSchedule(steps: int = 300, pre_rate: float = 0.1,
 pre_decay: int = 400, post_rate: float = 0.001,
 post_decay: int = 4000, dtype: Any = tf.float64)
```

Bases: keras.optimizers.schedules.learning\_rate\_schedule.LearningRateSchedule

```
__init__(steps: int = 300, pre_rate: float = 0.1, pre_decay: int = 400, post_rate: float = 0.001, post_decay:
 int = 4000, dtype: Any = tf.float64) → None
```

```
classmethod from_config(config)
```

Instantiates a *LearningRateSchedule* from its config.

**Parameters** **config** – Output of *get\_config()*.

**Returns** A *LearningRateSchedule* instance.

```
abstract get_config()
```

```
class tensorcircuit.applications.vqes.Linear(*args, **kwargs)
```

Bases: keras.engine.base\_layer.Layer

Dense layer but with complex weights, used for building complex RBM

```
__init__(units: int, input_dim: int, stddev: float = 0.1) → None
```

```
property activity_regularizer
```

Optional regularizer function for the output of this layer.

```
add_loss(losses, **kwargs)
```

Add loss tensor(s), potentially dependent on layer inputs.

Some losses (for instance, activity regularization losses) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs *a* and *b*, some entries in *layer.losses* may be dependent on *a* and some on *b*. This method automatically keeps track of dependencies.

This method can be used inside a subclassed layer or model's *call* function, in which case *losses* should be a Tensor or list of Tensors.

Example:

```
```python
class MyLayer(tf.keras.layers.Layer):
    def call(self, inputs):
        self.add_loss(tf.abs(tf.reduce_mean(inputs)))
        return inputs
```

```

This method can also be called directly on a Functional Model during construction. In this case, any loss Tensors passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These losses become part of the model's topology and are tracked in 'get\_config'.*

Example:

```
`python
inputs = tf.keras.Input(shape=(10,))
x = tf.keras.layers.Dense(10)(inputs)
outputs = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs, outputs) # Activity regularization.
model.add_loss(tf.abs(tf.reduce_mean(x)))`
```

If this is not the case for your loss (if, for example, your loss references a *Variable* of one of the model's layers), you can wrap your loss in a zero-argument lambda. These losses are not tracked as part of the model's topology since they can't be serialized.

Example:

```
`python inputs = tf.keras.Input(shape=(10,)) d = tf.keras.layers.Dense(10) x = d(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) # Weight regularization. model.add_loss(lambda: tf.reduce_mean(d.kernel))`
```

#### Parameters

- **losses** – Loss tensor, or list/tuple of tensors. Rather than tensors, losses may also be zero-argument callables which create a loss tensor.
- **\*\*kwargs** – Used for backwards compatibility only.

**add\_metric**(*value*, *name*=*None*, **\*\*kwargs**)

Adds metric tensor to the layer.

This method can be used inside the *call()* method of a subclassed layer or model.

```python class MyMetricLayer(tf.keras.layers.Layer):

```
def __init__(self): super(MyMetricLayer, self).__init__(name='my_metric_layer') self.mean = tf.keras.metrics.Mean(name='metric_1')

def call(self, inputs): self.add_metric(self.mean(inputs)) self.add_metric(tf.reduce_sum(inputs), name='metric_2') return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any tensor passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These metrics become part of the model's topology and are tracked when you save the model via 'save()'*.

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) model.add_metric(tf.math_ops.reduce_sum(x), name='metric_1')`
```

Note: Calling *add_metric()* with the result of a metric object on a Functional Model, as shown in the example below, is not supported. This is because we cannot trace the metric result tensor back to the model's inputs.

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs, outputs) model.add_metric(tf.keras.metrics.Mean()(x), name='metric_1')`
```

Parameters

- **value** – Metric tensor.
- **name** – String metric name.
- ****kwargs** – Additional keyword arguments for backward compatibility. Accepted values: *aggregation* - When the *value* tensor provided is not the result of calling a *keras.Metric* instance, it will be aggregated by default using a *keras.Metric.Mean*.

add_update(*updates*)

Add update op(s), potentially dependent on layer inputs.

Weight updates (for instance, the updates of the moving mean and variance in a BatchNormalization layer) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs *a* and *b*, some entries in *layer.updates* may be dependent on *a* and some on *b*. This method automatically keeps track of dependencies.

This call is ignored when eager execution is enabled (in that case, variable updates are run on the fly and thus do not need to be tracked for later execution).

Parameters `updates` – Update op, or list/tuple of update ops, or zero-arg callable that returns an update op. A zero-arg callable should be passed in order to disable running the updates by setting `trainable=False` on this Layer, when executing in Eager mode.

`add_variable(*args, **kwargs)`

Deprecated, do NOT use! Alias for `add_weight`.

`add_weight(name=None, shape=None, dtype=None, initializer=None, regularizer=None, trainable=None, constraint=None, use_resource=None, synchronization=VariableSynchronization.AUTO, aggregation=VariableAggregationV2.NONE, **kwargs)`

Adds a new variable to the layer.

Parameters

- `name` – Variable name.
- `shape` – Variable shape. Defaults to scalar if unspecified.
- `dtype` – The type of the variable. Defaults to `self.dtype`.
- `initializer` – Initializer instance (callable).
- `regularizer` – Regularizer instance (callable).
- `trainable` – Boolean, whether the variable should be part of the layer’s “trainable_variables” (e.g. variables, biases) or “non_trainable_variables” (e.g. BatchNorm mean and variance). Note that `trainable` cannot be `True` if `synchronization` is set to `ON_READ`.
- `constraint` – Constraint instance (callable).
- `use_resource` – Whether to use a `ResourceVariable` or not. See [this guide](https://www.tensorflow.org/guide/migrate/tf1_vs_tf2#resourcevariables_instead_of_referencevariables)

for more information.

- `synchronization` – Indicates when a distributed a variable will be aggregated. Accepted values are constants defined in the class `tf.VariableSynchronization`. By default the synchronization is set to `AUTO` and the current `DistributionStrategy` chooses when to synchronize. If `synchronization` is set to `ON_READ`, `trainable` must not be set to `True`.
- `aggregation` – Indicates how a distributed variable will be aggregated. Accepted values are constants defined in the class `tf.VariableAggregation`.
- `**kwargs` – Additional keyword arguments. Accepted values are `getter`, `collections`, `experimental_autocast` and `caching_device`.

Returns The variable created.

Raises `ValueError` – When giving unsupported `dtype` and no `initializer` or when `trainable` has been set to `True` with `synchronization` set as `ON_READ`.

`build(input_shape)`

Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of `Layer` or `Model` can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of `call()`.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters `input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call(inputs: Any) → Any`

This is where the layer's logic lives.

The `call()` method may not create state (except in its first invocation, wrapping the creation of variables or other resources in `tf.init_scope()`). It is recommended to create state, including `tf.Variable` instances and nested *Layer* instances,

in `__init__()`, or in the `build()` method that is

called automatically before `call()` executes for the first time.

Parameters

- **inputs** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero arguments, and *inputs* cannot be provided via the default value of a keyword argument.
 - NumPy array or Python scalar values in *inputs* get cast as tensors.
 - Keras mask metadata is only collected from *inputs*.
 - Layers are built (`build(input_shape)` method) using shape info from *inputs* only.
 - `input_spec` compatibility is only checked against *inputs*.
 - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in `*args` or `**kwargs`, their casting behavior in mixed precision should be handled manually.
 - The SavedModel input specification is generated using *inputs* only.
 - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.
- ***args** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- ****kwargs** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - `training`: Boolean scalar tensor or Python boolean indicating

whether the `call` is meant for training or inference.

- `mask`: Boolean input mask. If the layer's `call()` method takes a `mask` argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns A tensor or list/tuple of tensors.

property `compute_dtype`

The dtype of the layer's computations.

This is equivalent to `Layer.dtype_policy.compute_dtype`. Unless mixed precision is used, this is the same as `Layer.dtype`, the dtype of the weights.

Layers automatically cast their inputs to the compute dtype, which causes computations and the output to be in the compute dtype as well. This is done by the base Layer class in `Layer.__call__`, so you do not have to insert these casts if implementing your own layer.

Layers often perform certain internal computations in higher precision when `compute_dtype` is float16 or bfloat16 for numeric stability. The output will still typically be float16 or bfloat16 in such cases.

Returns The layer's compute dtype.

compute_mask(*inputs*, *mask=None*)

Computes an output mask tensor.

Parameters

- **inputs** – Tensor or list of tensors.
- **mask** – Tensor or list of tensors.

Returns

None or a tensor (or list of tensors, one per output tensor of the layer).

compute_output_shape(*input_shape*)

Computes the output shape of the layer.

This method will cause the layer's state to be built, if that has not happened before. This requires that the layer will later be used with inputs that match the input shape provided here.

Parameters **input_shape** – Shape tuple (tuple of integers) or `tf.TensorShape`, or structure of shape tuples / `tf.TensorShape` instances (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns A `tf.TensorShape` instance or structure of `tf.TensorShape` instances.

compute_output_signature(*input_signature*)

Compute the output tensor signature of the layer based on the inputs.

Unlike a `TensorShape` object, a `TensorSpec` object contains both shape and dtype information for a tensor. This method allows layers to provide output dtype information if it is different from the input dtype. For any layer that doesn't implement this function, the framework will fall back to use `compute_output_shape`, and will assume that the output dtype matches the input dtype.

Parameters **input_signature** – Single `TensorSpec` or nested structure of `TensorSpec` objects, describing a candidate input for the layer.

Returns

Single TensorSpec or nested structure of TensorSpec objects, describing how the layer would transform the provided input.

Raises **TypeError** – If *input_signature* contains a non-`TensorSpec` object.

count_params()

Count the total number of scalars composing the weights.

Returns An integer count.

Raises **ValueError** – if the layer isn't yet built (in which case its weights aren't yet defined).

property **dtype**

The dtype of the layer weights.

This is equivalent to `Layer.dtype_policy.variable_dtype`. Unless mixed precision is used, this is the same as `Layer.compute_dtype`, the dtype of the layer's computations.

property `dtype_policy`

The dtype policy associated with this layer.

This is an instance of a `tf.keras.mixed_precision.Policy`.

property `dynamic`

Whether the layer is dynamic (eager-only); set in the constructor.

`finalize_state()`

Finalizes the layers state after updating layer weights.

This function can be subclassed in a layer and will be called after updating a layer weights. It can be overridden to finalize any additional layer state after a weight update.

This function will be called after weights of a layer have been restored from a loaded model.

`classmethod from_config(config)`

Creates a layer from its config.

This method is the reverse of `get_config`, capable of instantiating the same layer from the config dictionary. It does not handle layer connectivity (handled by Network), nor weights (handled by `set_weights`).

Parameters `config` – A Python dictionary, typically the output of `get_config`.

Returns A layer instance.

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by `Network` (one layer of abstraction above).

Note that `get_config()` does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns Python dictionary.

`get_input_at(node_index)`

Retrieves the input tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first input node of the layer.

Returns A tensor (or list of tensors if the layer has multiple inputs).

Raises `RuntimeError` – If called in Eager mode.

`get_input_mask_at(node_index)`

Retrieves the input mask tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A mask tensor (or list of tensors if the layer has multiple inputs).

`get_input_shape_at(node_index)`

Retrieves the input shape(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A shape tuple (or list of shape tuples if the layer has multiple inputs).

Raises `RuntimeError` – If called in Eager mode.

`get_output_at(node_index)`

Retrieves the output tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first output node of the layer.

Returns A tensor (or list of tensors if the layer has multiple outputs).

Raises `RuntimeError` – If called in Eager mode.

`get_output_mask_at(node_index)`

Retrieves the output mask tensor(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A mask tensor (or list of tensors if the layer has multiple outputs).

`get_output_shape_at(node_index)`

Retrieves the output shape(s) of a layer at a given node.

Parameters `node_index` – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A shape tuple (or list of shape tuples if the layer has multiple outputs).

Raises `RuntimeError` – If called in Eager mode.

`get_weights()`

Returns the current weights of the layer, as NumPy arrays.

The weights of a layer represent the state of the layer. This function returns both trainable and non-trainable weight values associated with this layer as a list of NumPy arrays, which can in turn be used to load state into similarly parameterized layers.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
       [2.],
       [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
```

Returns Weights values as a list of NumPy arrays.

property inbound_nodes

Return Functional API nodes upstream of this layer.

property input

Retrieves the input tensor(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer.

Returns Input tensor or list of input tensors.

Raises

- **RuntimeError** – If called in Eager mode.
- **AttributeError** – If no inbound nodes are found.

property input_mask

Retrieves the input mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Input mask tensor (potentially None) or list of input mask tensors.

Raises

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

property input_shape

Retrieves the input shape(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer, or if all inputs have the same shape.

Returns Input shape, as an integer shape tuple (or list of shape tuples, one tuple per input tensor).

Raises

- **AttributeError** – if the layer has no defined input_shape.
- **RuntimeError** – if called in Eager mode.

property input_spec

InputSpec instance(s) describing the input format for this layer.

When you create a layer subclass, you can set *self.input_spec* to enable the layer to run input compatibility checks when it is called. Consider a *Conv2D* layer: it can only be called on a single input tensor of rank 4. As such, you can set, in *__init__()*:

```
`python self.input_spec = tf.keras.layers.InputSpec(ndim=4)`
```

Now, if you try to call the layer on an input that isn't rank 4 (for instance, an input of shape (2,), it will raise a nicely-formatted error:

```
`ValueError: Input 0 of layer conv2d is incompatible with the layer: expected ndim=4, found ndim=1. Full shape received: [2]`
```

Input checks that can be specified via *input_spec* include: - Structure (e.g. a single input, a list of 2 inputs, etc) - Shape - Rank (ndim) - Dtype

For more information, see *tf.keras.layers.InputSpec*.

Returns A *tf.keras.layers.InputSpec* instance, or nested structure thereof.

property losses

List of losses added using the `add_loss()` API.

Variable regularization tensors are created when this property is accessed, so it is eager safe: accessing `losses` under a `tf.GradientTape` will propagate gradients back to the corresponding variables.

Examples:

```
>>> class MyLayer(tf.keras.layers.Layer):
...     def call(self, inputs):
...         self.add_loss(tf.abs(tf.reduce_mean(inputs)))
...         return inputs
>>> l = MyLayer()
>>> l(np.ones((10, 1)))
>>> l.losses
[1.0]
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> x = tf.keras.layers.Dense(10)(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Activity regularization.
>>> len(model.losses)
0
>>> model.add_loss(tf.abs(tf.reduce_mean(x)))
>>> len(model.losses)
1
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> d = tf.keras.layers.Dense(10, kernel_initializer='ones')
>>> x = d(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Weight regularization.
>>> model.add_loss(lambda: tf.reduce_mean(d.kernel))
>>> model.losses
[<tf.Tensor: shape=(), dtype=float32, numpy=1.0>]
```

Returns A list of tensors.

property metrics

List of metrics added using the `add_metric()` API.

Example:

```
>>> input = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2)
>>> output = d(input)
>>> d.add_metric(tf.reduce_max(output), name='max')
>>> d.add_metric(tf.reduce_min(output), name='min')
>>> [m.name for m in d.metrics]
['max', 'min']
```

Returns A list of `Metric` objects.

property name

Name of the layer (string), set in the constructor.

property name_scope

Returns a `tf.name_scope` instance for this class.

property non_trainable_variables

Sequence of non-trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don't expect the return value to change.

Returns A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

property non_trainable_weights

List of all non-trainable weights tracked by this layer.

Non-trainable weights are *not* updated during training. They are expected to be updated manually in `call()`.

Returns A list of non-trainable variables.

property outbound_nodes

Return Functional API nodes downstream of this layer.

property output

Retrieves the output tensor(s) of a layer.

Only applicable if the layer has exactly one output, i.e. if it is connected to one incoming layer.

Returns Output tensor or list of output tensors.

Raises

- **AttributeError** – if the layer is connected to more than one incoming layers.
- **RuntimeError** – if called in Eager mode.

property output_mask

Retrieves the output mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Output mask tensor (potentially None) or list of output mask tensors.

Raises

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

property output_shape

Retrieves the output shape(s) of a layer.

Only applicable if the layer has one output, or if all outputs have the same shape.

Returns Output shape, as an integer shape tuple (or list of shape tuples, one tuple per output tensor).

Raises

- **AttributeError** – if the layer has no defined output shape.
- **RuntimeError** – if called in Eager mode.

set_weights(weights)

Sets the weights of the layer, from NumPy arrays.

The weights of a layer represent the state of the layer. This function sets the weight values from numpy arrays. The weight values should be passed in the order they are created by the layer. Note that the layer's weights must be instantiated before calling this function, by calling the layer.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([1.],
      [1.],
      [1.]), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([2.],
      [2.],
      [2.]), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([1.],
      [1.],
      [1.]), array([0.], dtype=float32)]
```

Parameters **weights** – a list of NumPy arrays. The number of arrays and their shape must match number of the dimensions of the weights of the layer (i.e. it should match the output of *get_weights*).

Raises **ValueError** – If the provided weights list does not match the layer's specifications.

property stateful**property submodules**

Sequence of all sub-modules.

Submodules are modules which are properties of this module, or found as properties of modules which are properties of this module (and so on).

```
>>> a = tf.Module()
>>> b = tf.Module()
>>> c = tf.Module()
>>> a.b = b
>>> b.c = c
>>> list(a.submodules) == [b, c]
True
>>> list(b.submodules) == [c]
True
>>> list(c.submodules) == []
True
```

Returns A sequence of all submodules.

property supports_masking

Whether this layer supports computing a mask using *compute_mask*.

property trainable

property trainable_variables

Sequence of trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don't expect the return value to change.

Returns A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

property trainable_weights

List of all trainable weights tracked by this layer.

Trainable weights are updated via gradient descent during training.

Returns A list of trainable variables.

property updates

property variable_dtype

Alias of *Layer.dtype*, the dtype of the weights.

property variables

Returns the list of all layer variables/weights.

Alias of *self.weights*.

Note: This will not track the weights of nested *tf.Modules* that are not themselves Keras layers.

Returns A list of variables.

property weights

Returns the list of all layer variables/weights.

Returns A list of variables.

classmethod with_name_scope(method)

Decorator to automatically enter the module name scope.

```
>>> class MyModule(tf.Module):
...     @tf.Module.with_name_scope
...     def __call__(self, x):
...         if not hasattr(self, 'w'):
...             self.w = tf.Variable(tf.random.normal([x.shape[1], 3]))
...         return tf.matmul(x, self.w)
```

Using the above module would produce `tf.Variable`'s and `tf.Tensor`'s whose names included the module name:

```
>>> mod = MyModule()
>>> mod(tf.ones([1, 2]))
<tf.Tensor: shape=(1, 2), dtype=float32, numpy=..., dtype=float32>
>>> mod.w
<tf.Variable 'my_module/Variable:0' shape=(2, 2) dtype=float32,
numpy=..., dtype=float32>
```

Parameters `method` – The method to wrap.

Returns The original method wrapped such that it enters the module's name scope.

```
class tensorcircuit.applications.vqes.VQNHE(n: int, hamiltonian: List[List[float]], model_params:  
                                              Optional[Dict[str, Any]] = None, circuit_params:  
                                              Optional[Dict[str, Any]] = None, shortcut: bool = False)  
  
Bases: object  
  
__init__(n: int, hamiltonian: List[List[float]], model_params: Optional[Dict[str, Any]] = None,  
        circuit_params: Optional[Dict[str, Any]] = None, shortcut: bool = False) → None  
  
assign(c: Optional[List[Any]] = None, q: Optional[Any] = None) → None  
  
create_circuit(choose: str = 'hea', **kws: Any) → Callable[[Any], Any]  
  
create_complex_model(init_value: float = 0.0, max_value: float = 1.0, min_value: float = 0, **kws: Any)  
    → Any  
  
create_complex_rbm_model(stddev: float = 0.1, width: int = 2, **kws: Any) → Any  
  
create_hea2_circuit(epochs: int = 2, filled_qubit: Optional[List[int]] = None, **kws: Any) →  
    Callable[[Any], Any]  
  
create_hea_circuit(epochs: int = 2, filled_qubit: Optional[List[int]] = None, **kws: Any) →  
    Callable[[Any], Any]  
  
create_hn_circuit(**kws: Any) → Callable[[Any], Any]  
  
create_model(choose: str = 'real', **kws: Any) → Any  
  
create_real_model(init_value: float = 0.0, max_value: float = 1.0, min_value: float = 0, depth: int = 2,  
                  width: int = 2, **kws: Any) → Any  
  
create_real_rbm_model(stddev: float = 0.1, width: int = 2, **kws: Any) → Any  
  
evaluation(cv: Any) → Tuple[Any, Any, Any]  
  
VQNHE  
  
    Parameters cv (Tensor) – [description]  
    Returns [description]  
    Return type Tuple[Tensor, Tensor, Tensor]  
  
load(path: str) → None  
  
multi_training(maxiter: int = 5000, optq: Optional[Callable[[int], Any]] = None, optc:  
               Optional[Callable[[int], Any]] = None, threshold: float = 1e-08, debug: int = 150, onlyq:  
               int = 0, checkpoints: Optional[List[Tuple[int, float]]] = None, tries: int = 10,  
               initialization_func: Optional[Callable[[int], Dict[str, Any]]] = None) → List[Dict[str,  
                                         Any]]  
  
plain_evaluation(cv: Any) → Any  
  
VQE  
  
    Parameters cv (Tensor) – [description]  
    Returns [description]  
    Return type Tensor  
  
recover() → None  
  
save(path: str) → None
```

```

training(maxiter: int = 5000, optq: Optional[Callable[[int], Any]] = None, optc: Optional[Callable[[int], Any]] = None, threshold: float = 1e-08, debug: int = 100, onlyq: int = 0, checkpoints: Optional[List[Tuple[int, float]]] = None) → Tuple[Any, Any, Any, int, List[float]]

tensorcircuit.applications.vqes.construct_matrix(ham: List[List[float]]) → Any
tensorcircuit.applications.vqes.construct_matrix_tf(ham: List[List[float]], dtype: Any = tf.complex128) → Any
tensorcircuit.applications.vqes.construct_matrix_v2(ham: List[List[float]], dtype: Any = tf.complex128) → Any
tensorcircuit.applications.vqes.construct_matrix_v3(ham: List[List[float]], dtype: Any = tf.complex128) → Any
tensorcircuit.applications.vqes.paulistring(term: Tuple[int, ...]) → Any
tensorcircuit.applications.vqes.vqe_energy(c: tensorcircuit.circuit.Circuit, h: List[List[float]], reuse: bool = True) → Any
tensorcircuit.applications.vqes.vqe_energy_shortcut(c: tensorcircuit.circuit.Circuit, h: Any) → Any

```

4.1.3 tensorcircuit.backends

`tensorcircuit.backends.backend_factory`

Backend register

`tensorcircuit.backends.backend_factory.get_backend(backend: Union[str, Any]) → Any`

Get the `tc.backend` object.

Parameters `backend` (`Union[Text, tnbbackend]`) – “numpy”, “tensorflow”, “jax”, “pytorch”

Raises `ValueError` – Backend doesn’t exist for `backend` argument.

Returns The `tc.backend` object that with all registered universal functions.

Return type backend object

`tensorcircuit.backends.jax_backend`

Backend magic inherited from tensornetwork: jax backend

`class tensorcircuit.backends.jax_backend.JaxBackend`

Bases: `tensornetwork.backends.jax.jax_backend.JaxBackend`, `tensorcircuit.backends.abstract_backend.ExtendedBackend`

See the original backend API at `jax backend`

`__init__()` → None

`abs(a: Any)` → Any

Returns the elementwise absolute value of tensor. :param tensor: An input tensor.

Returns Its elementwise absolute value.

Return type tensor

`acos(a: Any)` → Any

Return the acos of a tensor a.

Parameters `a` (`Tensor`) – tensor in matrix form

Returns `acos` of `a`

Return type `Tensor`

acosh(`a: Any`) → `Any`

Return the `acosh` of a tensor `a`.

Parameters `a (Tensor)` – tensor in matrix form

Returns `acosh` of `a`

Return type `Tensor`

addition(`tensor1: Any, tensor2: Any`) → `Any`

Return the default addition of `tensor`. A backend can override such implementation. :param `tensor1`: A tensor. :param `tensor2`: A tensor.

Returns `Tensor`

adjoint(`a: Any`) → `Any`

Return the conjugate and transpose of a tensor `a`

Parameters `a (Tensor)` – Input tensor

Returns adjoint tensor of `a`

Return type `Tensor`

arange(`start: int, stop: Optional[int] = None, step: int = 1`) → `Any`

Values are generated within the half-open interval [start, stop)

Parameters

- **start** (`int`) – start index
- **stop** (`Optional[int]`, `optional`) – end index, defaults to `None`
- **step** (`Optional[int]`, `optional`) – steps, defaults to `1`

Returns _description_

Return type `Tensor`

argmax(`a: Any, axis: int = 0`) → `Any`

Return the index of maximum of an array an axis.

Parameters

- **a** (`Tensor`) – [description]
- **axis** (`int`) – [description], defaults to `0`, different behavior from numpy defaults!

Returns [description]

Return type `Tensor`

argmin(`a: Any, axis: int = 0`) → `Any`

Return the index of minimum of an array an axis.

Parameters

- **a** (`Tensor`) – [description]
- **axis** (`int`) – [description], defaults to `0`, different behavior from numpy defaults!

Returns [description]

Return type `Tensor`

asin(*a*: Any) → Any

Return the acos of a tensor *a*.

Parameters *a* (Tensor) – tensor in matrix form

Returns asin of *a*

Return type Tensor

asinh(*a*: Any) → Any

Return the asinh of a tensor *a*.

Parameters *a* (Tensor) – tensor in matrix form

Returns asinh of *a*

Return type Tensor

atan(*a*: Any) → Any

Return the atan of a tensor *a*.

Parameters *a* (Tensor) – tensor in matrix form

Returns atan of *a*

Return type Tensor

atan2(*y*: Any, *x*: Any) → Any

Return the atan of a tensor *y/x*.

Parameters *a* (Tensor) – tensor in matrix form

Returns atan2 of *a*

Return type Tensor

atanh(*a*: Any) → Any

Return the atanh of a tensor *a*.

Parameters *a* (Tensor) – tensor in matrix form

Returns atanh of *a*

Return type Tensor

broadcast_left_multiplication(*tensor1*: Any, *tensor2*: Any) → Any

Perform broadcasting for multiplication of *tensor1* onto *tensor2*, i.e. *tensor1 * tensor2*, where *tensor2* is an arbitrary tensor and *tensor1* is a one-dimensional tensor. The broadcasting is applied to the first index of *tensor2*. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

Returns The result of multiplying *tensor1* onto *tensor2*.

Return type Tensor

broadcast_right_multiplication(*tensor1*: Any, *tensor2*: Any) → Any

Perform broadcasting for multiplication of *tensor2* onto *tensor1*, i.e. *tensor1 * tensor2*, where *tensor1* is an arbitrary tensor and *tensor2* is a one-dimensional tensor. The broadcasting is applied to the last index of *tensor1*. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

Returns The result of multiplying *tensor1* onto *tensor2*.

Return type Tensor

cast(*a*: Any, *dtype*: str) → Any

Cast the tensor *dtype* of a *a*.

Parameters

- **a** (*Tensor*) – tensor
- **dtype** (*str*) – “float32”, “float64”, “complex64”, “complex128”

Returns a of new dtype

Return type Tensor

cholesky(*tensor*: Any, *pivot_axis*: int = -1, *non_negative_diagonal*: bool = False) → Tuple[Any, Any]

concat(*a*: Sequence[Any], *axis*: int = 0) → Any

Join a sequence of arrays along an existing axis.

Parameters

- **a** (Sequence[*Tensor*]) – [description]
- **axis** (*int, optional*) – [description], defaults to 0

cond(*pred*: bool, *true_fun*: Callable[[], Any], *false_fun*: Callable[[], Any]) → Any

The native cond for XLA compiling, wrapper for `tf.cond` and limited functionality of `jax.lax.cond`.

Parameters

- **pred** (*bool*) – [description]
- **true_fun** (*Callable[[], Tensor]*) – [description]
- **false_fun** (*Callable[[], Tensor]*) – [description]

Returns [description]

Return type Tensor

conj(*tensor*: Any) → Any

Return the complex conjugate of *tensor* :param tensor: A tensor.

Returns Tensor

convert_to_tensor(*tensor*: Any) → Any

Convert a np.array or a tensor to a tensor type for the backend.

coo_sparse_matrix(*indices*: Any, *values*: Any, *shape*: Any) → Any

Generate the coo format sparse matrix from indices and values, which is the only sparse format supported in different ML backends.

Parameters

- **indices** (*Tensor*) – shape [n, 2] for n non zero values in the returned matrix
- **values** (*Tensor*) – shape [n]
- **shape** (*Tensor*) – Tuple[int, ...]

Returns [description]

Return type Tensor

coo_sparse_matrix_from_numpy(*a*: Any) → Any

Generate the coo format sparse matrix from scipy coo sparse matrix.

Parameters **a** (*Tensor*) – Scipy coo format sparse matrix

Returns SparseTensor in backend format

Return type Tensor

copy(*tensor*: Any) → Any

Return the copy of a, matrix exponential.

Parameters `a` (*Tensor*) – tensor in matrix form

Returns matrix exponential of matrix `a`

Return type Tensor

`cos(a: Any) → Any`

Return cos of *tensor*. :param `tensor`: A tensor.

Returns Tensor

`cosh(a: Any) → Any`

Return the cosh of a tensor `a`.

Parameters `a` (*Tensor*) – tensor in matrix form

Returns cosh of `a`

Return type Tensor

`cumsum(a: Any, axis: Optional[int] = None) → Any`

Return the cumulative sum of the elements along a given axis.

Parameters

- `a` (*Tensor*) – [description]

- `axis` (*Optional[int]*, *optional*) – The default behavior is the same as numpy, different from tf/torch as cumsum of the flatten 1D array, defaults to None

Returns [description]

Return type Tensor

`deserialize_tensor(s: str) → Any`

Return a tensor given a serialized tensor string.

Parameters `s` – The input string representing a serialized tensor.

Returns The tensor object represented by the string.

`device(a: Any) → str`

get the universal device str for the tensor, in the format of tf

Parameters `a` (*Tensor*) – the tensor

Returns device str where the tensor lives on

Return type str

`device_move(a: Any, dev: Any) → Any`

move tensor `a` to device `dev`

Parameters

- `a` (*Tensor*) – the tensor

- `dev` (*Any*) – device str or device obj in corresponding backend

Returns the tensor on new device

Return type Tensor

`diagflat(tensor: Any, k: int = 0) → Any`

Flattens tensor and creates a new matrix of zeros with its elements on the `k`'th diagonal. :param `tensor`: A tensor. :param `k`: The diagonal upon which to place its elements.

Returns A new tensor with all zeros save the specified diagonal.

Return type tensor

diagonal(*tensor*: Any, *offset* = 0, *axis1* = - 2, *axis2* = - 1) → Any
Return specified diagonals.

If tensor is 2-D, returns the diagonal of tensor with the given offset, i.e., the collection of elements of the form $a[i, i+off]$. If *a* has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing *axis1* and *axis2* and appending an index to the right equal to the size of the resulting diagonals.

This function only extracts diagonals. If you wish to create diagonal matrices from vectors, use `diagflat`.

Parameters

- **tensor** – A tensor.
- **offset** – Offset of the diagonal from the main diagonal.
- **axis1** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second last/last axis.
- **axis2** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second last/last axis.

Returns

A **dim = min(1, tensor.ndim - 2)** tensor storing the batched diagonals.

Return type array_of_diagonals

divide(*tensor1*: Any, *tensor2*: Any) → Any

Return the default divide of *tensor*. A backend can override such implementation. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

Returns Tensor

dtype(*a*: Any) → str

Obtain dtype string for tensor *a*

Parameters **a** (Tensor) – The tensor

Returns dtype str, such as “complex64”

Return type str

eigh(*matrix*: Any) → Tuple[Any, Any]

Compute eigenvectors and eigenvalues of a hermitian matrix.

Parameters **matrix** – A symetric matrix.

Returns The eigenvalues in ascending order. Tensor: The eigenvectors.

Return type Tensor

eigs(*A*: Callable, *args*: Optional[List] = None, *initial_state*: Optional[Any] = None, *shape*:

Optional[Tuple[int, ...]] = None, *dtype*: Optional[Type[numumpy.number]] = None, *num_krylov_vecs*: int = 50, *numeig*: int = 6, *tol*: float = 1e-08, *which*: str = 'LR', *maxiter*: int = 20) → Tuple[Any, List]

Implicitly restarted Arnoldi method for finding the lowest eigenvector-eigenvalue pairs of a linear operator *A*. *A* is a function implementing the matrix-vector product.

WARNING: This routine uses jax.jit to reduce runtimes. jitting is triggered at the first invocation of *eigs*, and on any subsequent calls if the python *id* of *A* changes, even if the formal definition of *A* stays the same. Example: the following will jit once at the beginning, and then never again:

```python import jax import numpy as np def *A*(*H*,*x*):

```
return jax.numpy.dot(H,x)
```

```
for n in range(100): H = jax.numpy.array(np.random.rand(10,10)) x = jax.numpy.array(np.random.rand(10,10))
res = eigs(A, [H],x) #jitting is triggered only at n=0
```

```

The following code triggers jitting at every iteration, which results in considerably reduced performance

```
python import jax import numpy as np for n in range(100):
```

```
def A(H,x): return jax.numpy.dot(H,x)
```

```
H = jax.numpy.array(np.random.rand(10,10)) x = jax.numpy.array(np.random.rand(10,10)) res =
eigs(A, [H],x) #jitting is triggered at every step n
```

```

## Parameters

- **A** – A (sparse) implementation of a linear operator. Call signature of *A* is *res = A(vector, \*args)*, where *vector* can be an arbitrary *Tensor*, and *res.shape* has to be *vector.shape*.
- **args** – A list of arguments to *A*. *A* will be called as *res = A(initial\_state, \*args)*.
- **initial\_state** – An initial vector for the algorithm. If *None*, a random initial *Tensor* is created using the *backend.randn* method
- **shape** – The shape of the input-dimension of *A*.
- **dtype** – The dtype of the input *A*. If no *initial\_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.
- **num\_krylov\_vecs** – The number of iterations (number of krylov vectors).
- **numeig** – The number of eigenvector-eigenvalue pairs to be computed.
- **tol** – The desired precision of the eigenvalues. For the jax backend this has currently no effect, and precision of eigenvalues is not guaranteed. This feature may be added at a later point. To increase precision the caller can either increase *maxiter* or *num\_krylov\_vecs*.
- **which** – Flag for targetting different types of eigenvalues. Currently supported are *which = 'LR'* (larges real part) and *which = 'LM'* (larges magnitude).
- **maxiter** – Maximum number of restarts. For *maxiter=0* the routine becomes equivalent to a simple Arnoldi method.

## Returns

**(eigvals, eigvecs)** *eigvals*: A list of *numeig* eigenvalues *eigvecs*: A list of *numeig* eigenvectors

**eigsh**(*A*: *Callable*, *args*: *Optional[List]* = *None*, *initial\_state*: *Optional[Any]* = *None*, *shape*: *Optional[Tuple[int, ...]]* = *None*, *dtype*: *Optional[Type[numpy.number]]* = *None*, *num\_krylov\_vecs*: *int* = 50, *numeig*: *int* = 6, *tol*: *float* = 1e-08, *which*: *str* = 'SA', *maxiter*: *int* = 20) → *Tuple[Any, List]*  
Implicitly restarted Lanczos method for finding the lowest eigenvector-eigenvalue pairs of a symmetric (hermitian) linear operator *A*. *A* is a function implementing the matrix-vector product.

WARNING: This routine uses jax.jit to reduce runtimes. jitting is triggered at the first invocation of *eigsh*, and on any subsequent calls if the python *id* of *A* changes, even if the formal definition of *A* stays the same. Example: the following will jit once at the beginning, and then never again:

```
python import jax import numpy as np def A(H,x):
```

```
 return jax.np.dot(H,x)
```

```
for n in range(100): H = jax.np.array(np.random.rand(10,10)) x = jax.np.array(np.random.rand(10,10))
 res = eigsh(A, [H],x) #jitting is triggered only at n=0
```

```

The following code triggers jitting at every iteration, which results in considerably reduced performance

```
```python import jax import numpy as np for n in range(100):
```

```
 def A(H,x): return jax.np.dot(H,x)
```

```
 H = jax.np.array(np.random.rand(10,10)) x = jax.np.array(np.random.rand(10,10)) res =
 eigsh(A, [H],x) #jitting is triggered at every step n
```

```

Parameters

- **A** – A (sparse) implementation of a linear operator. Call signature of *A* is *res = A(vector, *args)*, where *vector* can be an arbitrary *Tensor*, and *res.shape* has to be *vector.shape*.
- **args** – A list of arguments to *A*. *A* will be called as *res = A(initial_state, *args)*.
- **initial_state** – An initial vector for the algorithm. If *None*, a random initial *Tensor* is created using the *backend.randn* method
- **shape** – The shape of the input-dimension of *A*.
- **dtype** – The dtype of the input *A*. If no *initial_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.
- **num_krylov_vecs** – The number of iterations (number of krylov vectors).
- **numeig** – The number of eigenvector-eigenvalue pairs to be computed.
- **tol** – The desired precision of the eigenvalues. For the jax backend this has currently no effect, and precision of eigenvalues is not guaranteed. This feature may be added at a later point. To increase precision the caller can either increase *maxiter* or *num_krylov_vecs*.
- **which** – Flag for targetting different types of eigenvalues. Currently supported are *which = 'LR'* (larges real part) and *which = 'LM'* (larges magnitude).
- **maxiter** – Maximum number of restarts. For *maxiter=0* the routine becomes equivalent to a simple Arnoldi method.

Returns

(eigvals, eigvecs) *eigvals*: A list of *numeig* eigenvalues *eigvecs*: A list of *numeig* eigenvectors

eigsh_lanczos(*A*: *Callable*, *args*: *Optional[List[Any]]* = *None*, *initial_state*: *Optional[Any]* = *None*, *shape*: *Optional[Tuple]* = *None*, *dtype*: *Optional[Type[numumpy.number]]* = *None*, *num_krylov_vecs*: *int* = 20, *numeig*: *int* = 1, *tol*: *float* = 1e-08, *delta*: *float* = 1e-08, *ndiag*: *int* = 10, *reorthogonalize*: *Optional[bool]* = *False*) → *Tuple[Any, List]*

Lanczos method for finding the lowest eigenvector-eigenvalue pairs of a hermitian linear operator *A*. *A* is a function implementing the matrix-vector product. WARNING: This routine uses jax.jit to reduce runtimes. jitting is triggered at the first invocation of *eigsh_lanczos*, and on any subsequent calls if the python *id* of *A* changes, even if the formal definition of *A* stays the same. Example: the following will jit once at the beginning, and then never again:

```
```python import jax import numpy as np def A(H,x):
```

```

 return jax.np.dot(H,x)

for n in range(100): H = jax.np.array(np.random.rand(10,10)) x = jax.np.array(np.random.rand(10,10))
 res = eigsh_lanczos(A, [H],x) #jitting is triggered only at n=0

```

The following code triggers jitting at every iteration, which results in considerably reduced performance

```
python import jax import numpy as np for n in range(100):
```

```

def A(H,x): return jax.np.dot(H,x)

H = jax.np.array(np.random.rand(10,10)) x = jax.np.array(np.random.rand(10,10)) res =
eigsh_lanczos(A, [H],x) #jitting is triggered at every step n

```

## Parameters

- **A** – A (sparse) implementation of a linear operator. Call signature of  $A$  is  $res = A(vector, *args)$ , where  $vector$  can be an arbitrary *Tensor*, and  $res.shape$  has to be  $vector.shape$ .
- **args** – A list of arguments to  $A$ .  $A$  will be called as  $res = A(initial\_state, *args)$ .
- **initial\_state** – An initial vector for the Lanczos algorithm. If *None*, a random initial *Tensor* is created using the *backend.randn* method
- **shape** – The shape of the input-dimension of  $A$ .
- **dtype** – The dtype of the input  $A$ . If no *initial\_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.
- **num\_krylov\_vecs** – The number of iterations (number of krylov vectors).
- **numeig** – The number of eigenvector-eigenvalue pairs to be computed. If  $numeig > 1$ , *reorthogonalize* has to be *True*.
- **tol** – The desired precision of the eigenvalues. For the jax backend this has currently no effect, and precision of eigenvalues is not guaranteed. This feature may be added at a later point. To increase precision the caller can increase *num\_krylov\_vecs*.
- **delta** – Stopping criterion for Lanczos iteration. If a Krylov vector :math:  $x_n$  has an L2 norm  $\|x_n\| < delta$ , the iteration is stopped. It means that an (approximate) invariant subspace has been found.
- **ndiag** – The tridiagonal Operator is diagonalized every *ndiag* iterations to check convergence. This has currently no effect for the jax backend, but may be added at a later point.
- **reorthogonalize** – If *True*, Krylov vectors are kept orthogonal by explicit orthogonalization (more costly than *reorthogonalize=False*)

## Returns

**(eigvals, eigvecs)** eigvals: A jax-array containing *numeig* lowest eigenvalues eigvecs: A list of *numeig* lowest eigenvectors

**eigvalsh(a: Any) → Any**

Get the eigenvalues of matrix  $a$ .

**Parameters a (Tensor)** – tensor in matrix form

**Returns** eigenvalues of  $a$

**Return type** Tensor

**einsum**(*expression: str, \*tensors: Any, optimize: bool = True*) → Any  
Calculate sum of products of tensors according to expression.

**eps**(*dtype: Type[numpy.number]*) → float  
Return machine epsilon for given *dtype*

**Parameters** **dtype** – A dtype.

**Returns** Machine epsilon.

**Return type** float

**exp**(*tensor: Any*) → Any  
Return elementwise exp of *tensor*. :param tensor: A tensor.

**Returns** Tensor

**expm**(*a: Any*) → Any  
Return expm log of *matrix*, matrix exponential. :param matrix: A tensor.

**Returns** Tensor

**eye**(*N: int, dtype: Optional[str] = None, M: Optional[int] = None*) → Any

**Return an identity matrix of dimension dim** Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported

**Parameters**

- **N (int)** – The dimension of the returned matrix.
- **dtype** – The dtype of the returned matrix.
- **M (int)** – The dimension of the returned matrix.

**from\_dlpark**(*a: Any*) → Any  
Transform a dlpark capsule to a tensor

**Parameters** **a (Any)** – the dlpark capsule

**Returns** \_description\_

**Return type** Tensor

**gather1d**(*operand: Any, indices: Any*) → Any  
Return *operand[indices]*, both *operand* and *indices* are rank-1 tensor.

**Parameters**

- **operand (Tensor)** – rank-1 tensor
- **indices (Tensor)** – rank-1 tensor with int dtype

**Returns** *operand[indices]*

**Return type** Tensor

**get\_random\_state**(*seed: Optional[int] = None*) → Any  
Get the backend specific random state object.

**Parameters** **seed (Optional[int], optional)** – [description], defaults to be None

:return:the backend specific random state object :rtype: Any

---

**gmres**(*A\_mv*: Callable, *b*: Any, *A\_args*: Optional[List] = None, *A\_kwargs*: Optional[dict] = None, *x0*: Optional[Any] = None, *tol*: float = 1e-05, *atol*: Optional[float] = None, *num\_krylov\_vectors*: int = 20, *maxiter*: Optional[int] = 1, *M*: Optional[Callable] = None) → Tuple[Any, int]

GMRES solves the linear system  $A @ x = b$  for  $x$  given a vector  $b$  and a general (not necessarily symmetric/Hermitian) linear operator  $A$ .

As a Krylov method, GMRES does not require a concrete matrix representation of the  $n$  by  $n$   $A$ , but only a function  $vector1 = A_mv(vector0, *A_args, **A_kwargs)$  prescribing a one-to-one linear map from  $vector0$  to  $vector1$  (that is,  $A$  must be square, and thus  $vector0$  and  $vector1$  the same size). If  $A$  is a dense matrix, or if it is a symmetric/Hermitian operator, a different linear solver will usually be preferable.

GMRES works by first constructing the Krylov basis  $K = (x0, A_mv@x0, A_mv@A_mv@x0, \dots, (A_mv^num\_krylov\_vectors)@x_0)$  and then solving a certain dense linear system  $K @ q0 = q1$  from whose solution  $x$  can be approximated. For  $num\_krylov\_vectors = n$  the solution is provably exact in infinite precision, but the expense is cubic in  $num\_krylov\_vectors$  so one is typically interested in the  $num\_krylov\_vectors << n$  case. The solution can in this case be repeatedly improved, to a point, by restarting the Arnoldi iterations each time  $num\_krylov\_vectors$  is reached. Unfortunately the optimal parameter choices balancing expense and accuracy are difficult to predict in advance, so applying this function requires a degree of experimentation.

In a tensor network code one is typically interested in  $A_mv$  implementing some tensor contraction. This implementation thus allows  $b$  and  $x0$  to be of whatever arbitrary, though identical, shape  $b = A_mv(x0, \dots)$  expects. Reshaping to and from a matrix problem is handled internally.

### Parameters

- ***A\_mv*** – A function  $v0 = A_mv(v, *A_args, **A_kwargs)$  where  $v0$  and  $v$  have the same shape.
- ***b*** – The  $b$  in  $A @ x = b$ ; it should be of the shape  $A_mv$  operates on.
- ***A\_args*** – Positional arguments to  $A_mv$ , supplied to this interface as a list. Default: None.
- ***A\_kwargs*** – Keyword arguments to  $A_mv$ , supplied to this interface as a dictionary. Default: None.
- ***x0*** – An optional guess solution. Zeros are used by default. If  $x0$  is supplied, its shape and dtype must match those of  $b$ , or an error will be thrown. Default: zeros.
- ***tol*** – Solution tolerance to achieve,  $\text{norm}(\text{residual}) \leq \max(\text{tol} * \text{norm}(b), \text{atol})$ . Default: tol=1E-05  
atol=tol
- ***atol*** – Solution tolerance to achieve,  $\text{norm}(\text{residual}) \leq \max(\text{tol} * \text{norm}(b), \text{atol})$ . Default: tol=1E-05  
atol=tol
- ***num\_krylov\_vectors*** –
  - : **Size of the Krylov space to build at each restart.** Expense is cubic in this parameter. It must be positive. If greater than  $b.size$ , it will be set to  $b.size$ . Default: 20
  - ***maxiter*** – The Krylov space will be repeatedly rebuilt up to this many times. Large values of this argument should be used only with caution, since especially for nearly symmetric matrices and small  $num\_krylov\_vectors$  convergence might well freeze at a value significantly larger than  $tol$ . Default: 1.

- **M** – Inverse of the preconditioner of A; see the docstring for `scipy.sparse.linalg.gmres`. This is only supported in the numpy backend. Supplying this argument to other backends will trigger `NotImplementedError`. Default: None.

**Raises ValueError** – -if `x0` is supplied but its shape differs from that of `b`. -in NumPy, if the ARPACK solver reports a breakdown (which usually indicates some kind of floating point issue). -if `num_krylov_vectors` is 0 or exceeds `b.size`. -if `tol` was negative. -if `M` was supplied with any backend but NumPy.

**Returns** The converged solution. It has the same shape as `b`. `info` : 0 if convergence was achieved, the number of restarts otherwise.

**Return type** `x`

**grad**(`f: Callable[..., Any]`, `argnums: Union[int, Sequence[int]] = 0`, `has_aux: bool = False`) → `Any`  
Return the function which is the grad function of input `f`.

**Example**

```
>>> f = lambda x,y: x**2+2*y
>>> g = tc.backend.grad(f)
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
2
>>> g = tc.backend.grad(f, argnums=(0,1))
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
[2, 2]
```

**Parameters**

- **f** (`Callable[..., Any]`) – the function to be differentiated
- **argnums** (`Union[int, Sequence[int]]`, optional) – the position of args in `f` that are to be differentiated, defaults to be 0

**Returns** the grad function of `f` with the same set of arguments as `f`

**Return type** `Callable[..., Any]`

**hessian**(`f: Callable[..., Any]`, `argnums: Union[int, Sequence[int]] = 0`) → `Any`

**i**(`dtype: Optional[Any] = None`) → `Any`

Return 1.j in as a tensor compatible with the backend.

**Parameters** `dtype` (`str`) – “complex64” or “complex128”

**Returns** 1.j tensor

**Return type** Tensor

**imag**(`a: Any`) → `Any`

Return the elementwise imaginary value of a tensor `a`.

**Parameters** `a` (`Tensor`) – tensor

**Returns** imaginary value of `a`

**Return type** Tensor

**implicit\_randc**(`a: Union[int, Sequence[int], Any]`, `shape: Union[int, Sequence[int]], p: Optional[Union[Sequence[float], Any]] = None`) → `Any`

[summary]

**Parameters**

- **g** (*Any*) – [description]
- **a** (*Union[int, Sequence[int], Tensor]*) – The possible options
- **shape** (*Union[int, Sequence[int]]*) – Sampling output shape
- **p** (*Optional[Union[Sequence[float], Tensor]]*, *optional*) – probability for each option in a, defaults to None, as equal probability distribution

**Returns** [description]

**Return type** Tensor

**implicit\_rndn**(*shape: Union[int, Sequence[int]] = 1, mean: float = 0, stddev: float = 1, dtype: str = '32'*) → Any

Call the random normal function with the random state management behind the scene.

**Parameters**

- **shape** (*Union[int, Sequence[int]]*, *optional*) – [description], defaults to 1
- **mean** (*float*, *optional*) – [description], defaults to 0
- **stddev** (*float*, *optional*) – [description], defaults to 1
- **dtype** (*str*, *optional*) – [description], defaults to “32”

**Returns** [description]

**Return type** Tensor

**implicit\_randu**(*shape: Union[int, Sequence[int]] = 1, low: float = 0, high: float = 1, dtype: str = '32'*) → Any

Call the random uniform function with the random state management behind the scene.

**Parameters**

- **shape** (*Union[int, Sequence[int]]*, *optional*) – [description], defaults to 1
- **mean** (*float*, *optional*) – [description], defaults to 0
- **stddev** (*float*, *optional*) – [description], defaults to 1
- **dtype** (*str*, *optional*) – [description], defaults to “32”

**Returns** [description]

**Return type** Tensor

**index\_update**(*tensor: Any, mask: Any, assignee: Any*) → Any

Update *tensor* at elements defined by *mask* with value *assignee*.

**Parameters**

- **tensor** – A *Tensor* object.
- **mask** – A boolean mask.
- **assignee** – A scalar *Tensor*. The values to assigned to *tensor* at positions where *mask* is *True*.

**inv**(*matrix: Any*) → Any

Compute the matrix inverse of *matrix*.

**Parameters** **matrix** – A matrix.

**Returns** The inverse of *matrix*

**Return type** Tensor

**is\_sparse**(*a*: Any) → boolDetermine whether the type of input *a* is sparse.**Parameters** **a** (*Tensor*) – input matrix *a***Returns** a bool indicating whether the matrix *a* is sparse**Return type** bool**is\_tensor**(*a*: Any) → boolReturn a boolean on whether *a* is a tensor in backend package.**Parameters** **a** (*Tensor*) – a tensor to be determined**Returns** whether *a* is a tensor**Return type** bool**item**(*tensor*)

Return the item of a 1-element tensor.

**Parameters** **tensor** – A 1-element tensor**Returns** The value in tensor.**jacbwd**(*f*: Callable[...], Any], *argnums*: Union[int, Sequence[int]] = 0) → AnyCompute the Jacobian of *f* using reverse mode AD.**Parameters**

- **f** (Callable[..., Any]) – The function whose Jacobian is required
- **argnums** (Union[int, Sequence[int]], optional) – the position of the arg as Jacobian input, defaults to 0

**Returns** outer tuple for output, inner tuple for input args**Return type** Tensor**jacfwd**(*f*: Callable[...], Any], *argnums*: Union[int, Sequence[int]] = 0) → AnyCompute the Jacobian of *f* using the forward mode AD.**Parameters**

- **f** (Callable[..., Any]) – the function whose Jacobian is required
- **argnums** (Union[int, Sequence[int]], optional) – the position of the arg as Jacobian input, defaults to 0

**Returns** outer tuple for input args, inner tuple for outputs**Return type** Tensor**jacrev**(*f*: Callable[...], Any], *argnums*: Union[int, Sequence[int]] = 0) → AnyCompute the Jacobian of *f* using reverse mode AD.**Parameters**

- **f** (Callable[..., Any]) – The function whose Jacobian is required
- **argnums** (Union[int, Sequence[int]], optional) – the position of the arg as Jacobian input, defaults to 0

**Returns** outer tuple for output, inner tuple for input args**Return type** Tensor

**jit**(*f*: Callable[[...], Any], static\_argnums: Optional[Union[int, Sequence[int]]] = None, jit\_compile: Optional[bool] = None) → Any  
 Return a jitted or graph-compiled version of *fun* for JAX backend. For all other backends returns *fun*.  
 :param fun: Callable :param args: Arguments to *fun*. :param kwargs: Keyword arguments to *fun*.

**Returns** jitted/graph-compiled version of *fun*, or just *fun*.

**Return type** Callable

**jvp**(*f*: Callable[[...], Any], inputs: Union[Any, Sequence[Any]], v: Union[Any, Sequence[Any]]) → Tuple[Union[Any, Sequence[Any]], Union[Any, Sequence[Any]]]  
 Function that computes a (forward-mode) Jacobian-vector product of *f*. Strictly speaking, this function is value\_and\_jvp.

**Parameters**

- **f** (Callable[..., Any]) – The function to compute jvp
- **inputs** (Union[Tensor, Sequence[Tensor]]) – input for *f*
- **v** (Union[Tensor, Sequence[Tensor]]) – tangents

**Returns** (*f*(\*inputs), jvp\_tensor), where jvp\_tensor is the same shape as the output of *f*

**Return type** Tuple[Union[Tensor, Sequence[Tensor]], Union[Tensor, Sequence[Tensor]]]

**kron**(*a*: Any, *b*: Any) → Any

Return the kronecker product of two matrices *a* and *b*.

**Parameters**

- **a** (Tensor) – tensor in matrix form
- **b** (Tensor) – tensor in matrix form

**Returns** kronecker product of *a* and *b*

**Return type** Tensor

**left\_shift**(*x*: Any, *y*: Any) → Any

Shift the bits of an integer *x* to the left *y* bits.

**Parameters**

- **x** (Tensor) – input values
- **y** (Tensor) – Number of bits shift to *x*

**Returns** result with the same shape as *x*

**Return type** Tensor

**log**(*tensor*: Any) → Any

Return elementwise natural logarithm of *tensor*. :param tensor: A tensor.

**Returns** Tensor

**matmul**(*tensor1*: Any, *tensor2*: Any) → Any

Perform a possibly batched matrix-matrix multiplication between *tensor1* and *tensor2*. The following behaviour is similar to *numpy.matmul*: - If both arguments are 2-D they are multiplied like conventional matrices.

- If either argument is N-D, N > 2, it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.

Both arguments to *matmul* have to be tensors of order  $\geq 2$ . :param tensor1: An input tensor. :param tensor2: An input tensor.

**Returns** The result of performing the matmul.

**Return type** tensor

**max**(*a*: Any, *axis*: Optional[int] = None) → Any

Return the maximum of an array or maximum along an axis.

**Parameters**

- **a** (Tensor) – [description]
- **axis** (Optional[int], optional) – [description], defaults to None

**Returns** [description]

**Return type** Tensor

**mean**(*a*: Any, *axis*: Optional[Sequence[int]] = None, *keepdims*: bool = False) → Any

Compute the arithmetic mean for *a* along the specified *axis*.

**Parameters**

- **a** (Tensor) – tensor to take average
- **axis** (Optional[Sequence[int]], optional) – the axis to take mean, defaults to None indicating sum over flatten array
- **keepdims** (bool, optional) – \_description\_, defaults to False

**Returns** \_description\_

**Return type** Tensor

**min**(*a*: Any, *axis*: Optional[int] = None) → Any

Return the minimum of an array or minimum along an axis.

**Parameters**

- **a** (Tensor) – [description]
- **axis** (Optional[int], optional) – [description], defaults to None

**Returns** [description]

**Return type** Tensor

**mod**(*x*: Any, *y*: Any) → Any

Compute *y*-mod of *x* (negative number behavior is not guaranteed to be consistent)

**Parameters**

- **x** (Tensor) – input values
- **y** (Tensor) – mod *y*

**Returns** results

**Return type** Tensor

**multiply**(*tensor1*: Any, *tensor2*: Any) → Any

Return the default multiplication of *tensor*.

A backend can override such implementation. :param tensor1: A tensor. :param tensor2: A tensor.

**Returns** Tensor

**norm(tensor: Any) → Any**

Calculate the L2-norm of the elements of *tensor*

**numpy(a: Any) → Any**

Return the numpy array of a tensor a, but may not work in a jitted function.

**Parameters** **a** (*Tensor*) – tensor in matrix form

**Returns** numpy array of a

**Return type** Tensor

**one\_hot(a: Any, num: int) → Any**

See doc for [onehot\(\)](#)

**onehot(a: Any, num: int) → Any**

One-hot encodes the given a. Each index in the input a is encoded as a vector of zeros of length num with the element at index set to one:

**Parameters**

- **a** (*Tensor*) – input tensor
- **num** (*int*) – number of features in onehot dimension

**Returns** onehot tensor with the last extra dimension

**Return type** Tensor

**ones(shape: Tuple[int, ...], dtype: Optional[str] = None) → Any**

Return an ones-matrix of dimension dim Depending on specific backends, dim has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported :param shape: The dimension of the returned matrix. :type shape: int :param dtype: The dtype of the returned matrix.

**optimizer**

alias of [tensorcircuit.backends.jax\\_backend.optax\\_optimizer](#)

**outer\_product(tensor1: Any, tensor2: Any) → Any**

Calculate the outer product of the two given tensors.

**pivot(tensor: Any, pivot\_axis: int = -1) → Any**

Reshapes a tensor into a matrix, whose columns (rows) are the vectorized dimensions to the left (right) of pivot\_axis.

In other words, with tensor.shape = (1, 2, 4, 5) and pivot\_axis=2, this function returns an (8, 5) matrix.

**Parameters**

- **tensor** – The tensor to pivot.
- **pivot\_axis** – The axis about which to pivot.

**Returns** The pivoted tensor.

**power(a: Any, b: Union[Any, float]) → Any**

Returns the power of tensor a to the value of b. In the case b is a tensor, then the power is by element with a as the base and b as the exponent.

**In the case b is a scalar, then the power of each value in a** is raised to the exponent of b.

**Parameters**

- **a** – The tensor that contains the base.

- **b** – The tensor that contains the exponent or a single scalar.

**probability\_sample**(shots: int, p: Any, status: Optional[Any] = None, g: Optional[Any] = None) → Any

Drawn shots samples from probability distribution p, given the external randomness determined by uniform distributed status tensor or backend random generator g. This method is similar with stateful\_randc, but it supports status beyond g, which is convenient when jit or vmap

**Parameters**

- **shots (int)** – Number of samples to draw with replacement
- **p (Tensor)** – prbability vector
- **status (Optional [Tensor], optional)** – external randomness as a tensor with each element drawn uniformly from [0, 1], defaults to None
- **g (Any, optional)** – backend random generator, defaults to None

**Returns** The drawn sample as an int tensor

**Return type** Tensor

**qr**(tensor: Any, pivot\_axis: int = -1, non\_negative\_diagonal: bool = False) → Tuple[Any, Any]

Computes the QR decomposition of a tensor. See tensornetwork.backends.tensorflow.decompositions for details.

**randn**(shape: Tuple[int, ...], dtype: Optional[numpy.dtype] = None, seed: Optional[int] = None) → Any

Return a random-normal-matrix of dimension dim Depending on specific backends, dim has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends).

Block-sparse behavior is currently not supported :param shape: The dimension of the returned matrix. :type shape: int :param dtype: The dtype of the returned matrix. :param seed: The seed for the random number generator

**random\_split**(key: Any) → Tuple[Any, Any]

A jax like split API, but it doesn't split the key generator for other backends. It is just for a consistent interface of random code; make sure you know what the function actually does. This function is mainly a utility to write backend agnostic code instead of doing magic things.

**Parameters** **key (Any)** – [description]

**Returns** [description]

**Return type** Tuple[Any, Any]

**random\_uniform**(shape: Tuple[int, ...], boundaries: Optional[Tuple[float, float]] = (0.0, 1.0), dtype:

Optional[numpy.dtype] = None, seed: Optional[int] = None) → Any

Return a random uniform matrix of dimension dim.

Depending on specific backends, dim has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported :param shape: The dimension of the returned matrix. :type shape: int :param boundaries: The boundaries of the uniform distribution. :type boundaries: tuple :param dtype: The dtype of the returned matrix. :param seed: The seed for the random number generator

**Returns** random uniform initialized tensor.

**Return type** Tensor

**real**(a: Any) → Any

Return the elementwise real value of a tensor a.

**Parameters** **a (Tensor)** – tensor

**Returns** real value of a

**Return type** Tensor

**relu**(*a*: Any) → Any

Rectified linear unit activation function. Computes the element-wise function:

$$\text{relu}(x) = \max(x, 0)$$

**Parameters** *a* (Tensor) – Input tensor

**Returns** Tensor after relu

**Return type** Tensor

**reshape**(*tensor*: Any, *shape*: Any) → Any

Reshape tensor to the given shape.

**Parameters** *tensor* – A tensor.

**Returns** The reshaped tensor.

**reshape2**(*a*: Any) → Any

Reshape a tensor to the [2, 2, ...] shape.

**Parameters** *a* (Tensor) – Input tensor

**Returns** the reshaped tensor

**Return type** Tensor

**reshapem**(*a*: Any) → Any

Reshape a tensor to the [1, 1] shape.

**Parameters** *a* (Tensor) – Input tensor

**Returns** the reshaped tensor

**Return type** Tensor

**reverse**(*a*: Any) → Any

return *a*[::-1], only 1D tensor is guaranteed for consistent behavior

**Parameters** *a* (Tensor) – 1D tensor

**Returns** 1D tensor in reverse order

**Return type** Tensor

**right\_shift**(*x*: Any, *y*: Any) → Any

Shift the bits of an integer *x* to the right *y* bits.

**Parameters**

- **x** (Tensor) – input values
- **y** (Tensor) – Number of bits shift to *x*

**Returns** result with the same shape as *x*

**Return type** Tensor

**rq**(*tensor*: Any, *pivot\_axis*: int = -1, *non\_negative\_diagonal*: bool = False) → Tuple[Any, Any]

Computes the RQ (reversed QR) decomposition of a tensor. See `tensornetwork.backends.tensorflow.decompositions` for details.

**scatter**(*operand*: Any, *indices*: Any, *updates*: Any) → Any

Roughly equivalent to *operand*[*indices*] = *updates*, indices only support shape with rank 2 for now.

**Parameters**

- **operand** (*Tensor*) – [description]
- **indices** (*Tensor*) – [description]
- **updates** (*Tensor*) – [description]

**Returns** [description]**Return type** Tensor**searchsorted**(*a*: Any, *v*: Any, *side*: str = 'left') → Any

Find indices where elements should be inserted to maintain order.

**Parameters**

- **a** (*Tensor*) – input array sorted in ascending order
- **v** (*Tensor*) – value to inserted
- **side** (str, optional) – If ‘left’, the index of the first suitable location found is given. If ‘right’, return the last such index. If there is no suitable index, return either 0 or N (where N is the length of a), defaults to “left”

**Returns** Array of insertion points with the same shape as v, or an integer if v is a scalar.**Return type** Tensor**serialize\_tensor**(*tensor*: Any) → str

Return a string that serializes the given tensor.

**Parameters** **tensor** – The input tensor.**Returns** A string representing the serialized tensor.**set\_random\_state**(*seed*: Optional[Union[int, Any]] = None, *get\_only*: bool = False) → Any

Set the random state attached to the backend.

**Parameters**

- **seed** (Optional[int], optional) – the random seed, defaults to be None
- **get\_only** (bool, defaults to be False) – If set to be true, only get the random state in return instead of setting the state on the backend

**shape\_concat**(*values*: Any, *axis*: int) → Any

Concatenate a sequence of tensors together about the given axis.

**shape\_prod**(*values*: Any) → Any

Take the product of all of the elements in values

**shape\_tensor**(*tensor*: Any) → Any

Get the shape of a tensor.

**Parameters** **tensor** – A tensor.**Returns** The shape of the input tensor returned as another tensor.**shape\_tuple**(*tensor*: Any) → Tuple[Optional[int], ...]

Get the shape of a tensor as a tuple of integers.

**Parameters** **tensor** – A tensor.**Returns** The shape of the input tensor returned as a tuple of ints.**sigmoid**(*a*: Any) → Any

Compute sigmoid of input a

**Parameters** `a` (*Tensor*) – [description]

**Returns** [description]

**Return type** Tensor

**sign**(*tensor*: Any) → Any

Returns an elementwise tensor with entries  $y[i] = 1, 0, -1$  where  $\text{tensor}[i] > 0$ ,  $= 0$ , and  $< 0$  respectively.

For complex input the behaviour of this function may depend on the backend. The Jax backend version returns  $y[i] = x[i]/\sqrt{|x[i]|^2}$ .

**Parameters** `tensor` – The input tensor.

**sin**(*a*: Any) → Any

Return sin of *tensor*. :param *tensor*: A tensor.

**Returns** Tensor

**sinh**(*a*: Any) → Any

Return the sinh of a tensor *a*.

**Parameters** `a` (*Tensor*) – tensor in matrix form

**Returns** sinh of *a*

**Return type** Tensor

**size**(*a*: Any) → Any

Return the total number of elements in *a* in tensor form.

**Parameters** `a` (*Tensor*) – tensor

**Returns** the total number of elements in *a*

**Return type** Tensor

**sizen**(*a*: Any) → int

Return the total number of elements in tensor *a*, but in integer form.

**Parameters** `a` (*Tensor*) – tensor

**Returns** the total number of elements in tensor *a*

**Return type** int

**slice**(*tensor*: Any, *start\_indices*: Tuple[int, ...], *slice\_sizes*: Tuple[int, ...]) → Any

Obtains a slice of a tensor based on *start\_indices* and *slice\_sizes*.

**Parameters**

- **tensor** – A tensor.
- **start\_indices** – Tuple of integers denoting start indices of slice.
- **slice\_sizes** – Tuple of integers denoting size of slice along each axis.

**softmax**(*a*: Sequence[Any], *axis*: Optional[int] = None) → Any

Softmax function. Computes the function which rescales elements to the range [0,1] such that the elements along axis sum to 1.

$$\text{softmax}(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

**Parameters**

- **a** (Sequence[*Tensor*]) – Tensor

- **axis** (*int, optional*) – A dimension along which Softmax will be computed , defaults to None for all axis sum.

**Returns** concatenated tensor

**Return type** Tensor

**solve**(*A: Any, b: Any, assume\_a: str = 'gen'*) → Any

Solve the linear system Ax=b and return the solution x.

**Parameters**

- **A** (*Tensor*) – The multiplied matrix.
- **b** (*Tensor*) – The resulted matrix.

**Returns** The solution of the linear system.

**Return type** Tensor

**sparse\_dense\_matmul**(*sp\_a: Any, b: Any*) → Any

A sparse matrix multiplies a dense matrix.

**Parameters**

- **sp\_a** (*Tensor*) – a sparse matrix
- **b** (*Tensor*) – a dense matrix

**Returns** dense matrix

**Return type** Tensor

**sparse\_shape**(*tensor: Any*) → Tuple[Optional[int], ...]

**sqrt**(*tensor: Any*) → Any

Take the square root (element wise) of a given tensor.

**sqrtmh**(*a: Any*) → Any

Return the sqrtm of a Hermitian matrix a.

**Parameters** **a** (*Tensor*) – tensor in matrix form

**Returns** sqrtm of a

**Return type** Tensor

**stack**(*a: Sequence[Any], axis: int = 0*) → Any

Concatenates a sequence of tensors a along a new dimension axis.

**Parameters**

- **a** (*Sequence[Tensor]*) – List of tensors in the same shape
- **axis** (*int, optional*) – the stack axis, defaults to 0

**Returns** concatenated tensor

**Return type** Tensor

**stateful\_randc**(*g: Any, a: Union[int, Sequence[int], Any], shape: Union[int, Sequence[int]], p: Optional[Union[Sequence[float], Any]] = None*) → Any

[summary]

**Parameters**

- **g** (*Any*) – [description]
- **a** (*Union[int, Sequence[int], Tensor]*) – The possible options

- **shape** (*Union[int, Sequence[int]]*) – Sampling output shape
- **p** (*Optional[Union[Sequence[float], Tensor]], optional*) – probability for each option in a, defaults to None, as equal probability distribution

**Returns** [description]

**Return type** Tensor

**stateful\_randn**(*g: Any, shape: Union[int, Sequence[int]] = 1, mean: float = 0, stddev: float = 1, dtype: str = '32'*) → Any

[summary]

**Parameters**

- **self** (*Any*) – [description]
- **g** (*Any*) – stateful register for each package
- **shape** (*Union[int, Sequence[int]]*) – shape of output sampling tensor
- **mean** (*float, optional*) – [description], defaults to 0
- **stddev** (*float, optional*) – [description], defaults to 1
- **dtype** (*str, optional*) – only real data type is supported, “32” or “64”, defaults to “32”

**Returns** [description]

**Return type** Tensor

**stateful\_randu**(*g: Any, shape: Union[int, Sequence[int]] = 1, low: float = 0, high: float = 1, dtype: str = '32'*) → Any

Uniform random sampler from low to high.

**Parameters**

- **g** (*Any*) – stateful register for each package
- **shape** (*Union[int, Sequence[int]], optional*) – shape of output sampling tensor, defaults to 1
- **low** (*float, optional*) – [description], defaults to 0
- **high** (*float, optional*) – [description], defaults to 1
- **dtype** (*str, optional*) – only real data type is supported, “32” or “64”, defaults to “32”

**Returns** [description]

**Return type** Tensor

**std**(*a: Any, axis: Optional[Sequence[int]] = None, keepdims: bool = False*) → Any

Compute the standard deviation along the specified axis.

**Parameters**

- **a** (*Tensor*) – \_description\_
- **axis** (*Optional[Sequence[int]], optional*) – Axis or axes along which the standard deviation is computed, defaults to None, implying all axis
- **keepdims** (*bool, optional*) – If this is set to True, the axes which are reduced are left in the result as dimensions with size one, defaults to False

**Returns** \_description\_

**Return type** Tensor

**stop\_gradient**(*a*: Any) → Any  
Stop backpropagation from *a*.

**Parameters** *a* (Tensor) – [description]

**Returns** [description]

**Return type** Tensor

**subtraction**(*tensor1*: Any, *tensor2*: Any) → Any  
Return the default subtraction of *tensor*. A backend can override such implementation.  
:param *tensor1*: A tensor.  
:param *tensor2*: A tensor.

**Returns** Tensor

**sum**(*tensor*: Any, *axis*: Optional[Sequence[int]] = None, *keepdims*: bool = False) → Any  
Sum elements of *tensor* along the specified *axis*. Results in a new Tensor with the summed axis removed.  
:param *tensor*: An input tensor.

**Returns**

The result of performing the summation. The order of the tensor will be reduced by 1.

**Return type** tensor

**svd**(*tensor*: Any, *pivot\_axis*: int = -1, *max\_singular\_values*: Optional[int] = None, *max\_truncation\_error*: Optional[float] = None, *relative*: Optional[bool] = False) → Tuple[Any, Any, Any, Any]  
Computes the singular value decomposition (SVD) of a tensor.

The SVD is performed by treating the tensor as a matrix, with an effective left (row) index resulting from combining the axes *tensor.shape[:pivot\_axis]* and an effective right (column) index resulting from combining the axes *tensor.shape[pivot\_axis:]*.

For example, if *tensor* had a shape (2, 3, 4, 5) and *pivot\_axis* was 2, then *u* would have shape (2, 3, 6), *s* would have shape (6), and *vh* would have shape (6, 4, 5).

If *max\_singular\_values* is set to an integer, the SVD is truncated to keep at most this many singular values.

If *max\_truncation\_error* > 0, as many singular values will be truncated as possible, so that the truncation error (the norm of discarded singular values) is at most *max\_truncation\_error*. If *relative* is set *True* then *max\_truncation\_err* is understood relative to the largest singular value.

If both *max\_singular\_values* and *max\_truncation\_error* are specified, the number of retained singular values will be *min(max\_singular\_values, nsv\_auto\_trunc)*, where *nsv\_auto\_trunc* is the number of singular values that must be kept to maintain a truncation error smaller than *max\_truncation\_error*.

The output consists of three tensors *u*, *s*, *vh* such that:

``` python  
u[i1,...,iN, j] * *s*[j] * *vh*[j, k1,...,kM] == *tensor*[i1,...,iN, k1,...,kM]

``` Note that the output ordering matches numpy.linalg.svd rather than tf.svd.

**Parameters**

- **tensor** – A tensor to be decomposed.
- **pivot\_axis** – Where to split the tensor's axes before flattening into a matrix.
- **max\_singular\_values** – The number of singular values to keep, or *None* to keep them all.
- **max\_truncation\_error** – The maximum allowed truncation error or *None* to not do any truncation.

- **relative** – Multiply  $\max\_truncation\_err$  with the largest singular value.

**Returns**

Left tensor factor. s: Vector of ordered singular values from largest to smallest. vh: Right tensor factor. s\_rest: Vector of discarded singular values (length zero if no truncation).

**Return type** u

**switch**(*index*: Any, *branches*: Sequence[Callable[], Any]) → Any  
*branches*[*index*]()

**Parameters**

- **index** (Tensor) – [description]
- **branches** (Sequence[Callable[], Tensor]) – [description]

**Returns** [description]**Return type** Tensor

**tan**(*a*: Any) → Any

Return the tan of a tensor a.

**Parameters** **a** (Tensor) – tensor in matrix form**Returns** tan of a**Return type** Tensor

**tanh**(*a*: Any) → Any

Return the tanh of a tensor a.

**Parameters** **a** (Tensor) – tensor in matrix form**Returns** tanh of a**Return type** Tensor

**tensordot**(*a*: Any, *b*: Any, *axes*: Union[int, Sequence[Sequence[int]]) → Any

Do a tensordot of tensors *a* and *b* over the given axes.

**Parameters**

- **a** – A tensor.
- **b** – Another tensor.
- **axes** – Two lists of integers. These values are the contraction axes.

**tile**(*a*: Any, *rep*: Any) → Any

Constructs a tensor by tiling a given tensor.

**Parameters**

- **a** (Tensor) – [description]
- **rep** (Tensor) – 1d tensor with length the same as the rank of a

**Returns** [description]**Return type** Tensor

**to\_dense**(*sp\_a*: Any) → Any

Convert a sparse matrix to dense tensor.

**Parameters** `sp_a` (*Tensor*) – a sparse matrix

**Returns** the resulted dense matrix

**Return type** Tensor

**to\_dlpack**(*a*: Any) → Any

Transform the tensor *a* as a dlpack capsule

**Parameters** `a` (*Tensor*) – \_description\_

**Returns** \_description\_

**Return type** Any

**trace**(*tensor*: Any, *offset*: int = 0, *axis1*: int = - 2, *axis2*: int = - 1) → Any

Return summed entries along diagonals.

If tensor is 2-D, the sum is over the diagonal of tensor with the given offset, i.e., the collection of elements of the form *a*[*i*, *i*+*offset*]. If *a* has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-array whose diagonal is summed.

**Parameters**

- **tensor** – A tensor.
- **offset** – Offset of the diagonal from the main diagonal.
- **axis1** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second last/last axis.
- **axis2** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second last/last axis.

**Returns** The batched summed diagonals.

**Return type** array\_of\_diagonals

**transpose**(*tensor*, *perm*=None) → Any

Transpose a tensor according to a given permutation. By default the axes are reversed. :param tensor: A tensor. :param perm: The permutation of the axes.

**Returns** The transposed tensor

**tree\_flatten**(*pytree*: Any) → Tuple[Any, Any]

Flatten python structure to 1D list

**Parameters** `pytree` (Any) – python structure to be flattened

**Returns** The 1D list of flattened structure and treedef which can be used for later unflatten

**Return type** Tuple[Any, Any]

**tree\_map**(*f*: Callable[[], Any], \**pytrees*: Any) → Any

Return the new tree map with multiple arg function *f* through pytrees.

**Parameters**

- **f** (Callable[[], Any]) – The function
- **pytrees** (Any) – inputs as any python structure

**Raises** `NotImplementedError` – raise when neither tensorflow or jax is installed.

**Returns** The new tree map with the same structure but different values.

**Return type** Any

**tree\_unflatten**(*treedef*: Any, *leaves*: Any) → AnyPack 1D list to pytree defined via *treedef***Parameters**

- **treedef** (Any) – Def of pytree structure, the second return from `tree_flatten`
- **leaves** (Any) – the 1D list of flattened data structure

**Returns** Packed pytree**Return type** Any**unique\_with\_counts**(*a*: Any, \*, *size*: Optional[int] = None, *fill\_value*: Optional[int] = None) → Tuple[Any, Any]Find the unique elements and their corresponding counts of the given tensor *a*.**Parameters** *a* (Tensor) – [description]**Returns** Unique elements, corresponding counts**Return type** Tuple[Tensor, Tensor]**value\_and\_grad**(*f*: Callable[..., Any], *argnums*: Union[int, Sequence[int]] = 0, *has\_aux*: bool = False) → Callable[..., Tuple[Any, Any]]Return the function which returns the value and grad of *f*.**Example**

```
>>> f = lambda x,y: x**2+2*y
>>> g = tc.backend.value_and_grad(f)
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
5, 2
>>> g = tc.backend.value_and_grad(f, argnums=(0,1))
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
5, [2, 2]
```

**Parameters**

- **f** (Callable[..., Any]) – the function to be differentiated
- **argnums** (Union[int, Sequence[int]], optional) – the position of args in *f* that are to be differentiated, defaults to be 0

**Returns** the value and grad function of *f* with the same set of arguments as *f***Return type** Callable[..., Tuple[Any, Any]]**vectorized\_value\_and\_grad**(*f*: Callable[..., Any], *argnums*: Union[int, Sequence[int]] = 0, *vectorized\_argnums*: Union[int, Sequence[int]] = 0, *has\_aux*: bool = False) → Callable[..., Tuple[Any, Any]]Return the VVAG function of *f*. The inputs for *f* is (*args*[0], *args*[1], *args*[2], ...), and the output of *f* is a scalar. Suppose VVAG(*f*) is a function with inputs in the form (*vargs*[0], *args*[1], *args*[2], ...), where *vargs*[0] has one extra dimension than *args*[0] in the first axis and consistent with *args*[0] in shape for remaining dimensions, i.e.  $\text{shape}(\text{vargs}[0]) = [\text{batch}] + \text{shape}(\text{args}[0])$ . (We only cover cases where *vectorized\_argnums* defaults to 0 here for demonstration). VVAG(*f*) returns a tuple as a value tensor with shape [batch, 1] and a gradient tuple with shape: ([batch]+ $\text{shape}(\text{args}[\text{argnum}])$  for *argnum* in *argnums*). The gradient for *argnums*=*k* is defined as

$$g^k = \frac{\partial \sum_{i \in \text{batch}} f(\text{vargs}[0][i], \text{args}[1], \dots)}{\partial \text{args}[k]}$$

Therefore, if argnums=0, the gradient is reduced to

$$g_i^0 = \frac{\partial f(vargs[0][i])}{\partial vargs[0][i]}$$

, which is specifically suitable for batched VQE optimization, where args[0] is the circuit parameters.

And if argnums=1, the gradient is like

$$g_i^1 = \frac{\partial \sum_j f(vargs[0][j], args[1])}{\partial args[1][i]}$$

, which is suitable for quantum machine learning scenarios, where `f` is the loss function, args[0] corresponds to the input data and args[1] corresponds to the weights in the QML model.

#### Parameters

- `f(Callable[..., Any])` – [description]
- `argnums(Union[int, Sequence[int]], optional)` – [description], defaults to 0
- `vectorized_argnums(Union[int, Sequence[int]], defaults to 0)` – the args to be vectorized, these arguments should share the same batch shape in the first dimension

#### Returns

 [description]

**Return type** Callable[..., Tuple[Any, Any]]

**vjp**(`f: Callable[..., Any], inputs: Union[Any, Sequence[Any]], v: Union[Any, Sequence[Any]]`) → Tuple[Union[Any, Sequence[Any]], Union[Any, Sequence[Any]]]  
Function that computes the dot product between a vector `v` and the Jacobian of the given function at the point given by the inputs. (reverse mode AD relevant) Strictly speaking, this function is value\_and\_vjp.

#### Parameters

- `f(Callable[..., Any])` – the function to carry out vjp calculation
- `inputs(Union[Tensor, Sequence[Tensor]])` – input for `f`
- `v(Union[Tensor, Sequence[Tensor]])` – value vector or gradient from downstream in reverse mode AD the same shape as return of function `f`

**Returns** (`f(*inputs), vjp_tensor`), where `vjp_tensor` is the same shape as `inputs`

**Return type** Tuple[Union[Tensor, Sequence[Tensor]], Union[Tensor, Sequence[Tensor]]]

**vmap**(`f: Callable[..., Any], vectorized_argnums: Union[int, Sequence[int]] = 0`) → Any  
Return the vectorized map or batched version of `f` on the first extra axis. The general interface supports `f` with multiple arguments and broadcast in the first dimension.

#### Parameters

- `f(Callable[..., Any])` – function to be broadcasted.
- `vectorized_argnums(Union[int, Sequence[int]], defaults to 0)` – the args to be vectorized, these arguments should share the same batch shape in the first dimension

**Returns** vmap version of `f`

**Return type** Any

**vvag**(*f*: Callable[...], *Any*], *argnums*: Union[int, Sequence[int]] = 0, *vectorized\_argnums*: Union[int, Sequence[int]] = 0, *has\_aux*: bool = False) → Callable[...], Tuple[*Any*, *Any*]]

Return the VVAG function of *f*. The inputs for *f* is (*args*[0], *args*[1], *args*[2], ...), and the output of *f* is a scalar. Suppose VVAG(*f*) is a function with inputs in the form (*vargs*[0], *args*[1], *args*[2], ...), where *vargs*[0] has one extra dimension than *args*[0] in the first axis and consistent with *args*[0] in shape for remaining dimensions, i.e.  $\text{shape}(\text{vargs}[0]) = [\text{batch}] + \text{shape}(\text{args}[0])$ . (We only cover cases where *vectorized\_argnums* defaults to 0 here for demonstration). VVAG(*f*) returns a tuple as a value tensor with shape [batch, 1] and a gradient tuple with shape: ([batch]+shape(*args*[*argnum*])) for *argnum* in *argnums*). The gradient for *argnums*=*k* is defined as

$$g^k = \frac{\partial \sum_{i \in \text{batch}} f(\text{vargs}[0][i], \text{args}[1], \dots)}{\partial \text{args}[k]}$$

Therefore, if *argnums*=0, the gradient is reduced to

$$g_i^0 = \frac{\partial f(\text{vargs}[0][i])}{\partial \text{vargs}[0][i]}$$

, which is specifically suitable for batched VQE optimization, where *args*[0] is the circuit parameters.

And if *argnums*=1, the gradient is like

$$g_i^1 = \frac{\partial \sum_j f(\text{vargs}[0][j], \text{args}[1])}{\partial \text{args}[1][i]}$$

, which is suitable for quantum machine learning scenarios, where *f* is the loss function, *args*[0] corresponds to the input data and *args*[1] corresponds to the weights in the QML model.

### Parameters

- **f** (Callable[..., *Any*]) – [description]
- **argnums** (Union[int, Sequence[int]], optional) – [description], defaults to 0
- **vectorized\_argnums** (Union[int, Sequence[int]], defaults to 0) – the args to be vectorized, these arguments should share the same batch shape in the fist dimension

### Returns

[description]

**Return type** Callable[..., Tuple[*Any*, *Any*]]

**zeros**(*shape*: Tuple[int, ...], *dtype*: Optional[str] = None) → Any

Return a zeros-matrix of dimension *dim* Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends).

Block-sparse behavior is currently not supported :param *shape*: The dimension of the returned matrix. :type *shape*: int :param *dtype*: The dtype of the returned matrix.

tensorcircuit.backends.jax\_backend.**jnp**: Any

tensorcircuit.backends.jax\_backend.**jsp**: Any

tensorcircuit.backends.jax\_backend.**libjax**: Any

tensorcircuit.backends.jax\_backend.**optax**: Any

**class** tensorcircuit.backends.jax\_backend.**optax\_optimizer**(*optimizer*: Any)

Bases: object

**\_\_init\_\_**(*optimizer*: Any) → None

**update**(*grads*: Any, *params*: Any) → Any

**tensorcircuit.backends.numpy\_backend**

Backend magic inherited from tensornetwork: numpy backend

**class tensorcircuit.backends.numpy\_backend.NumpyBackend**

Bases: `tensornetwork.backends.numpy.numpy_backend.NumPyBackend`, `tensorcircuit.backends.abstract_backend.ExtendedBackend`

see the original backend API at [numpy backend](#)

**\_\_init\_\_()** → None

**abs**(*a*: Any) → Any

Returns the elementwise absolute value of tensor. :param tensor: An input tensor.

**Returns** Its elementwise absolute value.

**Return type** tensor

**acos**(*a*: Any) → Any

Return the acos of a tensor *a*.

**Parameters** *a* (Tensor) – tensor in matrix form

**Returns** acos of *a*

**Return type** Tensor

**acosh**(*a*: Any) → Any

Return the acosh of a tensor *a*.

**Parameters** *a* (Tensor) – tensor in matrix form

**Returns** acosh of *a*

**Return type** Tensor

**addition**(*tensor1*: Any, *tensor2*: Any) → Any

Return the default addition of *tensor*. A backend can override such implementation. :param tensor1: A tensor. :param tensor2: A tensor.

**Returns** Tensor

**adjoint**(*a*: Any) → Any

Return the conjugate and transpose of a tensor *a*

**Parameters** *a* (Tensor) – Input tensor

**Returns** adjoint tensor of *a*

**Return type** Tensor

**arange**(*start*: int, *stop*: Optional[int] = None, *step*: int = 1) → Any

Values are generated within the half-open interval [start, stop)

**Parameters**

- **start** (int) – start index
- **stop** (Optional[int], optional) – end index, defaults to None
- **step** (Optional[int], optional) – steps, defaults to 1

**Returns** \_description\_

**Return type** Tensor

**argmax**(*a*: Any, *axis*: int = 0) → Any

Return the index of maximum of an array an axis.

**Parameters**

- **a** (*Tensor*) – [description]
- **axis** (*int*) – [description], defaults to 0, different behavior from numpy defaults!

**Returns** [description]

**Return type** Tensor

**argmin**(*a*: Any, *axis*: int = 0) → Any

Return the index of minimum of an array an axis.

**Parameters**

- **a** (*Tensor*) – [description]
- **axis** (*int*) – [description], defaults to 0, different behavior from numpy defaults!

**Returns** [description]

**Return type** Tensor

**asin**(*a*: Any) → Any

Return the acos of a tensor *a*.

**Parameters** **a** (*Tensor*) – tensor in matrix form

**Returns** asin of *a*

**Return type** Tensor

**asinh**(*a*: Any) → Any

Return the asinh of a tensor *a*.

**Parameters** **a** (*Tensor*) – tensor in matrix form

**Returns** asinh of *a*

**Return type** Tensor

**atan**(*a*: Any) → Any

Return the atan of a tensor *a*.

**Parameters** **a** (*Tensor*) – tensor in matrix form

**Returns** atan of *a*

**Return type** Tensor

**atan2**(*y*: Any, *x*: Any) → Any

Return the atan of a tensor *y/x*.

**Parameters** **a** (*Tensor*) – tensor in matrix form

**Returns** atan2 of *a*

**Return type** Tensor

**atanh**(*a*: Any) → Any

Return the atanh of a tensor *a*.

**Parameters** **a** (*Tensor*) – tensor in matrix form

**Returns** atanh of *a*

**Return type** Tensor

**broadcast\_left\_multiplication**(*tensor1*: Any, *tensor2*: Any) → Any

Perform broadcasting for multiplication of *tensor1* onto *tensor2*, i.e. *tensor1* \* *tensor2`*, where *tensor2* is an arbitrary tensor and *tensor1* is a one-dimensional tensor. The broadcasting is applied to the first index of *tensor2*. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

**Returns** The result of multiplying *tensor1* onto *tensor2*.

**Return type** Tensor

**broadcast\_right\_multiplication**(*tensor1*: Any, *tensor2*: Any) → Any

Perform broadcasting for multiplication of *tensor2* onto *tensor1*, i.e. *tensor1* \* *tensor2`*, where *tensor1* is an arbitrary tensor and *tensor2* is a one-dimensional tensor. The broadcasting is applied to the last index of *tensor1*. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

**Returns** The result of multiplying *tensor1* onto *tensor2*.

**Return type** Tensor

**cast**(*a*: Any, *dtype*: str) → Any

Cast the tensor *dtype* of a *a*.

**Parameters**

- **a** (Tensor) – tensor
- **dtype** (str) – “float32”, “float64”, “complex64”, “complex128”

**Returns** *a* of new *dtype*

**Return type** Tensor

**cholesky**(*tensor*: Any, *pivot\_axis*: int = -1, *non\_negative\_diagonal*: bool = False) → Tuple[Any, Any]

**concat**(*a*: Sequence[Any], *axis*: int = 0) → Any

Join a sequence of arrays along an existing axis.

**Parameters**

- **a** (Sequence[Tensor]) – [description]
- **axis** (int, optional) – [description], defaults to 0

**cond**(*pred*: bool, *true\_fun*: Callable[[], Any], *false\_fun*: Callable[[], Any]) → Any

The native cond for XLA compiling, wrapper for tf.cond and limited functionality of jax.lax.cond.

**Parameters**

- **pred** (bool) – [description]
- **true\_fun** (Callable[[], Tensor]) – [description]
- **false\_fun** (Callable[[], Tensor]) – [description]

**Returns** [description]

**Return type** Tensor

**conj**(*tensor*: Any) → Any

Return the complex conjugate of *tensor* :param *tensor*: A tensor.

**Returns** Tensor

**convert\_to\_tensor**(*a*: Any) → Any

Convert a np.array or a tensor to a tensor type for the backend.

**coo\_sparse\_matrix**(*indices*: Any, *values*: Any, *shape*: Any) → Any

Generate the coo format sparse matrix from indices and values, which is the only sparse format supported in different ML backends.

**Parameters**

- **indices** (*Tensor*) – shape [n, 2] for n non zero values in the returned matrix
- **values** (*Tensor*) – shape [n]
- **shape** (*Tensor*) – Tuple[int, ...]

**Returns** [description]

**Return type** Tensor

**coo\_sparse\_matrix\_from\_numpy**(*a*: Any) → Any

Generate the coo format sparse matrix from scipy coo sparse matrix.

**Parameters** *a* (*Tensor*) – Scipy coo format sparse matrix

**Returns** SparseTensor in backend format

**Return type** Tensor

**copy**(*a*: Any) → Any

Return the copy of *a*, matrix exponential.

**Parameters** *a* (*Tensor*) – tensor in matrix form

**Returns** matrix exponential of matrix *a*

**Return type** Tensor

**cos**(*a*: Any) → Any

Return cos of *tensor*. :param tensor: A tensor.

**Returns** Tensor

**cosh**(*a*: Any) → Any

Return the cosh of a tensor *a*.

**Parameters** *a* (*Tensor*) – tensor in matrix form

**Returns** cosh of *a*

**Return type** Tensor

**cumsum**(*a*: Any, *axis*: Optional[int] = None) → Any

Return the cumulative sum of the elements along a given axis.

**Parameters**

- **a** (*Tensor*) – [description]
- **axis** (Optional[int], optional) – The default behavior is the same as numpy, different from tf/torch as cumsum of the flatten 1D array, defaults to None

**Returns** [description]

**Return type** Tensor

**deserialize\_tensor**(*s*: str) → Any

Return a tensor given a serialized tensor string.

**Parameters** *s* – The input string representing a serialized tensor.

**Returns** The tensor object represented by the string.

**device**(*a*: Any) → str  
get the universal device str for the tensor, in the format of tf

**Parameters** *a* (Tensor) – the tensor

**Returns** device str where the tensor lives on

**Return type** str

**device\_move**(*a*: Any, *dev*: Any) → Any  
move tensor *a* to device *dev*

**Parameters**

- *a* (Tensor) – the tensor

- *dev* (Any) – device str or device obj in corresponding backend

**Returns** the tensor on new device

**Return type** Tensor

**diagflat**(*tensor*: Any, *k*: int = 0) → Any

Flattens tensor and creates a new matrix of zeros with its elements on the *k*'th diagonal. :param *tensor*: A tensor. :param *k*: The diagonal upon which to place its elements.

**Returns** A new tensor with all zeros save the specified diagonal.

**Return type** tensor

**diagonal**(*tensor*: Any, *offset*: int = 0, *axis1*: int = - 2, *axis2*: int = - 1) → Any  
Return specified diagonals.

If tensor is 2-D, returns the diagonal of tensor with the given offset, i.e., the collection of elements of the form *a*[*i*, *i*+*offset*]. If *a* has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing *axis1* and *axis2* and appending an index to the right equal to the size of the resulting diagonals.

This function only extracts diagonals. If you wish to create diagonal matrices from vectors, use diagflat.

**Parameters**

- **tensor** – A tensor.

- **offset** – Offset of the diagonal from the main diagonal.

- **axis1** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second-last/last axis.

- **axis2** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second-last/last axis.

**Returns**

A dim = min(1, tensor.ndim - 2) tensor storing the batched diagonals.

**Return type** array\_of\_diagonals

**divide**(*tensor1*: Any, *tensor2*: Any) → Any

Return the default divide of *tensor*. A backend can override such implementation. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

**Returns** Tensor

**dtype**(*a*: Any) → str

Obtain dtype string for tensor *a*

**Parameters** `a` (*Tensor*) – The tensor

**Returns** `dtype` str, such as “complex64”

**Return type** str

**eigh**(*matrix*: Any) → Tuple[Any, Any]

Compute eigenvectors and eigenvalues of a hermitian matrix.

**Parameters** `matrix` – A symmetric matrix.

**Returns** The eigenvalues in ascending order. Tensor: The eigenvectors.

**Return type** Tensor

**eigs**(*A*: Callable, *args*: Optional[List] = *None*, *initial\_state*: Optional[Any] = *None*, *shape*:

`Optional[Tuple[int, ...]]` = *None*, `dtype`: `Optional[Type[numpy.number]]` = *None*, `num_krylov_vecs`: int = 50, `numeig`: int = 6, `tol`: float = 1e-08, `which`: str = ‘LR’, `maxiter`: `Optional[int]` = *None*) → Tuple[Any, List]

Arnoldi method for finding the lowest eigenvector-eigenvalue pairs of a linear operator *A*. If no *initial\_state* is provided then *shape* and *dtype* are required so that a suitable initial state can be randomly generated. This is a wrapper for `scipy.sparse.linalg.eigs` which only supports a subset of the arguments of `scipy.sparse.linalg.eigs`.

**Parameters**

- `A` – A (sparse) implementation of a linear operator
- `args` – A list of arguments to *A*. *A* will be called as `res = A(initial_state, *args)`.
- `initial_state` – An initial vector for the algorithm. If *None*, a random initial *Tensor* is created using the `numpy.random.randn` method.
- `shape` – The shape of the input-dimension of *A*.
- `dtype` – The dtype of the input *A*. If both no *initial\_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.
- `num_krylov_vecs` – The number of iterations (number of krylov vectors).
- `numeig` – The nummber of eigenvector-eigenvalue pairs to be computed. If `numeig > 1`, `reorthogonalize` has to be *True*.
- `tol` – The desired precision of the eigenvalus. Uses
- `which` – [‘LM’ | ‘SM’ | ‘LR’ | ‘SR’ | ‘LI’] Which *k* eigenvectors and eigenvalues to find:  
 ‘LM’ : largest magnitude ‘SM’ : smallest magnitude ‘LR’ : largest real part ‘SR’ : smallest real part ‘LI’ : largest imaginary part
- `maxiter` – The maximum number of iterations.

**Returns** An array of *numeig* lowest eigenvalues *list*: A list of *numeig* lowest eigenvectors

**Return type** np.ndarray

**eigsh**(*A*: Callable, *args*: `Optional[List[Any]]` = *None*, *initial\_state*: `Optional[Any]` = *None*, *shape*:

`Optional[Tuple[int, ...]]` = *None*, `dtype`: `Optional[Type[numpy.number]]` = *None*, `num_krylov_vecs`: int = 50, `numeig`: int = 1, `tol`: float = 1e-08, `which`: str = ‘LR’, `maxiter`: `Optional[int]` = *None*) → Tuple[Any, List]

Lanczos method for finding the lowest eigenvector-eigenvalue pairs of a symmetric (hermitian) linear operator *A*. *A* is a callable implementing the matrix-vector product. If no *initial\_state* is provided then *shape* and *dtype* have to be passed so that a suitable initial state can be randomly generated. :param *A*: A (sparse) implementation of a linear operator :param *args*: A list of arguments to *A*. *A* will be called as

*res* = *A*(*initial\_state*, \**args*).

#### Parameters

- **initial\_state** – An initial vector for the algorithm. If *None*, a random initial *Tensor* is created using the *numpy.random.randn* method.
- **shape** – The shape of the input-dimension of *A*.
- **dtype** – The dtype of the input *A*. If both no *initial\_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.
- **num\_krylov\_vecs** – The number of iterations (number of krylov vectors).
- **numeig** – The nummber of eigenvector-eigenvalue pairs to be computed. If *numeig* > 1, *reorthogonalize* has to be *True*.
- **tol** – The desired precision of the eigenvalues. Uses
- **which** – [‘LM’ | ‘SM’ | ‘LR’ | ‘SR’ | ‘LI’ | ‘SI’] Which *k* eigenvectors and eigenvalues to find:

‘LM’ : largest magnitude ‘SM’ : smallest magnitude ‘LR’ : largest real part ‘SR’  
: smallest real part ‘LI’ : largest imaginary part ‘SI’ : smallest imaginary part

Note that not all of those might be supported by specialized backends.

- **maxiter** – The maximum number of iterations.

**Returns** An array of *numeig* lowest eigenvalues *list*: A list of *numeig* lowest eigenvectors

**Return type** *Tensor*

**eigsh\_lanczos**(*A*: *Callable*, *args*: *Optional[List[Any]]* = *None*, *initial\_state*: *Optional[Any]* = *None*, *shape*: *Optional[Tuple]* = *None*, *dtype*: *Optional[Type[numpy.number]]* = *None*, *num\_krylov\_vecs*: *int* = 20, *numeig*: *int* = 1, *tol*: *float* = 1e-08, *delta*: *float* = 1e-08, *ndiag*: *int* = 20, *reorthogonalize*: *bool* = *False*) → *Tuple[Any, List]*

Lanczos method for finding the lowest eigenvector-eigenvalue pairs of a linear operator *A*. :param *A*: A (sparse) implementation of a linear operator.

Call signature of *A* is *res* = *A*(*vector*, \**args*), where *vector* can be an arbitrary *Tensor*, and *res.shape* has to be *vector.shape*.

#### Parameters

- **args** – A list of arguments to *A*. *A* will be called as *res* = *A*(*initial\_state*, \**args*).
- **initial\_state** – An initial vector for the Lanczos algorithm. If *None*, a random initial *Tensor* is created using the *backend.randn* method
- **shape** – The shape of the input-dimension of *A*.
- **dtype** – The dtype of the input *A*. If both no *initial\_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.
- **num\_krylov\_vecs** – The number of iterations (number of krylov vectors).
- **numeig** – The nummber of eigenvector-eigenvalue pairs to be computed. If *numeig* > 1, *reorthogonalize* has to be *True*.
- **tol** – The desired precision of the eigenvalues. Uses *np.linalg.norm(eigvalsnew[0:numeig] - eigvalsold[0:numeig]) < tol* as stopping criterion between two diagonalization steps of the tridiagonal operator.

- **delta** – Stopping criterion for Lanczos iteration. If a Krylov vector :math: x\_n has an L2 norm  $\|x_n\| < \delta$ , the iteration is stopped. It means that an (approximate) invariant subspace has been found.
- **ndiag** – The tridiagonal Operator is diagonalized every  $ndiag$  iterations to check convergence.
- **reorthogonalize** – If *True*, Krylov vectors are kept orthogonal by explicit orthogonalization (more costly than *reorthogonalize=False*)

**Returns**

**(eigvals, eigvecs)** eigvals: A list of  $numeig$  lowest eigenvalues eigvecs: A list of  $numeig$  lowest eigenvectors

**eigvalsh**(*a*: Any) → Any

Get the eigenvalues of matrix *a*.

**Parameters** **a** (*Tensor*) – tensor in matrix form

**Returns** eigenvalues of *a*

**Return type** Tensor

**einsum**(*expression*: str, \**tensors*: Any, *optimize*: bool = *True*) → Any

Calculate sum of products of tensors according to expression.

**eps**(*dtype*: Type[*numpy.number*]) → float

Return machine epsilon for given *dtype*

**Parameters** **dtype** – A dtype.

**Returns** Machine epsilon.

**Return type** float

**exp**(*tensor*: Any) → Any

Return elementwise exp of *tensor*. :param *tensor*: A tensor.

**Returns** Tensor

**expm**(*a*: Any) → Any

Return expm log of *matrix*, matrix exponential. :param *matrix*: A tensor.

**Returns** Tensor

**eye**(*N*: int, *dtype*: Optional[str] = *None*, *M*: Optional[int] = *None*) → Any

**Return an identity matrix of dimension dim** Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported

**Parameters**

- **N** (int) – The dimension of the returned matrix.
- **dtype** – The dtype of the returned matrix.
- **M** (int) – The dimension of the returned matrix.

**from\_dlpac**(*a*: Any) → Any

Transform a dlpac capsule to a tensor

**Parameters** **a** (Any) – the dlpac capsule

**Returns** `_description_`

**Return type** Tensor

**gather1d**(*operand*: Any, *indices*: Any) → Any

Return *operand*[*indices*], both *operand* and *indices* are rank-1 tensor.

**Parameters**

- **operand** (Tensor) – rank-1 tensor

- **indices** (Tensor) – rank-1 tensor with int dtype

**Returns** *operand*[*indices*]

**Return type** Tensor

**get\_random\_state**(*seed*: Optional[int] = None) → Any

Get the backend specific random state object.

**Parameters** **seed** (Optional[int], optional) – [description], defaults to be None

:return:the backend specific random state object :rtype: Any

**gmres**(*A\_mv*: Callable, *b*: Any, *A\_args*: Optional[List] = None, *A\_kwargs*: Optional[dict] = None, *x0*: Optional[Any] = None, *tol*: float =  $1e-05$ , *atol*: Optional[float] = None, *num\_krylov\_vectors*: int = 20, *maxiter*: Optional[int] = 1, *M*: Optional[Callable] = None) → Tuple[Any, int]

GMRES solves the linear system  $A @ x = b$  for  $x$  given a vector  $b$  and a general (not necessarily symmetric/Hermitian) linear operator  $A$ .

As a Krylov method, GMRES does not require a concrete matrix representation of the  $n$  by  $n$   $A$ , but only a function  $\text{vector1} = A\_mv(\text{vector0}, *A\_args, **A\_kwargs)$  prescribing a one-to-one linear map from  $\text{vector0}$  to  $\text{vector1}$  (that is,  $A$  must be square, and thus  $\text{vector0}$  and  $\text{vector1}$  the same size). If  $A$  is a dense matrix, or if it is a symmetric/Hermitian operator, a different linear solver will usually be preferable.

GMRES works by first constructing the Krylov basis  $K = (x0, A\_mv@x0, A\_mv@A\_mv@x0, \dots, (A\_mv^{\text{num\_krylov\_vectors}})@x0)$  and then solving a certain dense linear system  $K @ q0 = q1$  from whose solution  $x$  can be approximated. For  $\text{num\_krylov\_vectors} = n$  the solution is provably exact in infinite precision, but the expense is cubic in  $\text{num\_krylov\_vectors}$  so one is typically interested in the  $\text{num\_krylov\_vectors} << n$  case. The solution can in this case be repeatedly improved, to a point, by restarting the Arnoldi iterations each time  $\text{num\_krylov\_vectors}$  is reached. Unfortunately the optimal parameter choices balancing expense and accuracy are difficult to predict in advance, so applying this function requires a degree of experimentation.

In a tensor network code one is typically interested in  $A\_mv$  implementing some tensor contraction. This implementation thus allows  $b$  and  $x0$  to be of whatever arbitrary, though identical, shape  $b = A\_mv(x0, \dots)$  expects. Reshaping to and from a matrix problem is handled internally.

**Parameters**

- **A\_mv** – A function  $v0 = A\_mv(v, *A\_args, **A\_kwargs)$  where  $v0$  and  $v$  have the same shape.
- **b** – The  $b$  in  $A @ x = b$ ; it should be of the shape  $A\_mv$  operates on.
- **A\_args** – Positional arguments to  $A\_mv$ , supplied to this interface as a list. Default: None.
- **A\_kwargs** – Keyword arguments to  $A\_mv$ , supplied to this interface as a dictionary. Default: None.
- **x0** – An optional guess solution. Zeros are used by default. If  $x0$  is supplied, its shape and dtype must match those of  $b$ , or an error will be thrown. Default: zeros.

- **tol** – Solution tolerance to achieve,  $\text{norm}(\text{residual}) \leq \max(\text{tol} * \text{norm}(b), \text{atol})$ . Default: tol=1E-05  
atol=tol
- **atol** – Solution tolerance to achieve,  $\text{norm}(\text{residual}) \leq \max(\text{tol} * \text{norm}(b), \text{atol})$ . Default: tol=1E-05  
atol=tol
- **num\_krylov\_vectors** –  
: Size of the Krylov space to build at each restart. Expense is cubic in this parameter. It must be positive. If greater than b.size, it will be set to b.size. Default: 20
- **maxiter** – The Krylov space will be repeatedly rebuilt up to this many times. Large values of this argument should be used only with caution, since especially for nearly symmetric matrices and small num\_krylov\_vectors convergence might well freeze at a value significantly larger than tol. Default: 1.
- **M** – Inverse of the preconditioner of A; see the docstring for `scipy.sparse.linalg.gmres`. This is only supported in the numpy backend. Supplying this argument to other backends will trigger `NotImplementedError`. Default: None.

**Raises ValueError** – -if  $x_0$  is supplied but its shape differs from that of  $b$ . -in NumPy, if the ARPACK solver reports a breakdown (which usually indicates some kind of floating point issue). -if num\_krylov\_vectors is 0 or exceeds b.size. -if tol was negative. -if M was supplied with any backend but NumPy.

**Returns** The converged solution. It has the same shape as  $b$ . info : 0 if convergence was achieved, the number of restarts otherwise.

#### Return type x

**grad**( $f: \text{Callable}[\dots, \text{Any}]$ , argnums:  $\text{Union}[\text{int}, \text{Sequence}[\text{int}]] = 0$ , has\_aux:  $\text{bool} = \text{False}$ ) →  $\text{Callable}[\dots, \text{Any}]$   
Return the function which is the grad function of input f.

#### Example

```
>>> f = lambda x,y: x**2+2*y
>>> g = tc.backend.grad(f)
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
2
>>> g = tc.backend.grad(f, argnums=(0,1))
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
[2, 2]
```

#### Parameters

- **f** ( $\text{Callable}[\dots, \text{Any}]$ ) – the function to be differentiated
- **argnums** ( $\text{Union}[\text{int}, \text{Sequence}[\text{int}]]$ , optional) – the position of args in f that are to be differentiated, defaults to be 0

**Returns** the grad function of f with the same set of arguments as f

#### Return type Callable[..., Any]

**hessian**( $f: \text{Callable}[\dots, \text{Any}]$ , argnums:  $\text{Union}[\text{int}, \text{Sequence}[\text{int}]] = 0$ ) → Any

**i**(*dtype: Optional[Any] = None*) → Any

Return 1.j in as a tensor compatible with the backend.

**Parameters** **dtype** (*str*) – “complex64” or “complex128”

**Returns** 1.j tensor

**Return type** Tensor

**imag**(*a: Any*) → Any

Return the elementwise imaginary value of a tensor a.

**Parameters** **a** (*Tensor*) – tensor

**Returns** imaginary value of a

**Return type** Tensor

**implicit\_randc**(*a: Union[int, Sequence[int], Any], shape: Union[int, Sequence[int]], p: Optional[Union[Sequence[float], Any]] = None*) → Any

[summary]

**Parameters**

- **g** (*Any*) – [description]
- **a** (*Union[int, Sequence[int], Tensor]*) – The possible options
- **shape** (*Union[int, Sequence[int]]*) – Sampling output shape
- **p** (*Optional[Union[Sequence[float], Tensor]], optional*) – probability for each option in a, defaults to None, as equal probability distribution

**Returns** [description]

**Return type** Tensor

**implicit\_randn**(*shape: Union[int, Sequence[int]] = 1, mean: float = 0, stddev: float = 1, dtype: str = '32'*) → Any

Call the random normal function with the random state management behind the scene.

**Parameters**

- **shape** (*Union[int, Sequence[int]], optional*) – [description], defaults to 1
- **mean** (*float, optional*) – [description], defaults to 0
- **stddev** (*float, optional*) – [description], defaults to 1
- **dtype** (*str, optional*) – [description], defaults to “32”

**Returns** [description]

**Return type** Tensor

**implicit\_randu**(*shape: Union[int, Sequence[int]] = 1, low: float = 0, high: float = 1, dtype: str = '32'*) → Any

Call the random uniform function with the random state management behind the scene.

**Parameters**

- **shape** (*Union[int, Sequence[int]], optional*) – [description], defaults to 1
- **mean** (*float, optional*) – [description], defaults to 0
- **stddev** (*float, optional*) – [description], defaults to 1
- **dtype** (*str, optional*) – [description], defaults to “32”

**Returns** [description]

**Return type** Tensor

**index\_update**(*tensor*: Any, *mask*: Any, *assignee*: Any) → Any

Update *tensor* at elements defined by *mask* with value *assignee*.

**Parameters**

- **tensor** – A *Tensor* object.
- **mask** – A boolean mask.
- **assignee** – A scalar *Tensor*. The values to assigned to *tensor* at positions where *mask* is *True*.

**inv**(*matrix*: Any) → Any

Compute the matrix inverse of *matrix*.

**Parameters** **matrix** – A matrix.

**Returns** The inverse of *matrix*

**Return type** Tensor

**is\_sparse**(*a*: Any) → bool

Determine whether the type of input *a* is sparse.

**Parameters** **a** (*Tensor*) – input matrix *a*

**Returns** a bool indicating whether the matrix *a* is sparse

**Return type** bool

**is\_tensor**(*a*: Any) → bool

Return a boolean on whether *a* is a tensor in backend package.

**Parameters** **a** (*Tensor*) – a tensor to be determined

**Returns** whether *a* is a tensor

**Return type** bool

**item**(*tensor*)

Return the item of a 1-element tensor.

**Parameters** **tensor** – A 1-element tensor

**Returns** The value in tensor.

**jacbwd**(*f*: Callable[...], Any], *argnums*: Union[int, Sequence[int]] = 0) → Any

Compute the Jacobian of *f* using reverse mode AD.

**Parameters**

- **f** (*Callable*[..., Any]) – The function whose Jacobian is required
- **argnums** (*Union*[int, Sequence[int]], optional) – the position of the arg as Jacobian input, defaults to 0

**Returns** outer tuple for output, inner tuple for input args

**Return type** Tensor

**jacfwd**(*f*: Callable[...], Any], *argnums*: Union[int, Sequence[int]] = 0) → Any

Compute the Jacobian of *f* using the forward mode AD.

**Parameters**

- **f** (*Callable[..., Any]*) – the function whose Jacobian is required
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – the position of the arg as Jacobian input, defaults to 0

**Returns** outer tuple for input args, inner tuple for outputs

**Return type** Tensor

**jacrev**(*f*: *Callable[..., Any]*, *argnums*: *Union[int, Sequence[int]]* = 0) → Any

Compute the Jacobian of *f* using reverse mode AD.

**Parameters**

- **f** (*Callable[..., Any]*) – The function whose Jacobian is required
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – the position of the arg as Jacobian input, defaults to 0

**Returns** outer tuple for output, inner tuple for input args

**Return type** Tensor

**jit**(*f*: *Callable[..., Any]*, *static\_argnums*: *Optional[Union[int, Sequence[int]]]* = None, *jit\_compile*: *Optional[bool]* = None) → Callable[*...*, Any]

Return a jitted or graph-compiled version of *fun* for JAX backend. For all other backends returns *fun*.  
:param fun: Callable :param args: Arguments to *fun*. :param kwargs: Keyword arguments to *fun*.

**Returns** jitted/graph-compiled version of *fun*, or just *fun*.

**Return type** Callable

**jvp**(*f*: *Callable[..., Any]*, *inputs*: *Union[Any, Sequence[Any]]*, *v*: *Union[Any, Sequence[Any]]*) → Tuple[*Union[Any, Sequence[Any]]*, *Union[Any, Sequence[Any]]*]

Function that computes a (forward-mode) Jacobian-vector product of *f*. Strictly speaking, this function is value\_and\_jvp.

**Parameters**

- **f** (*Callable[..., Any]*) – The function to compute jvp
- **inputs** (*Union[Tensor, Sequence[Tensor]]*) – input for *f*
- **v** (*Union[Tensor, Sequence[Tensor]]*) – tangents

**Returns** (*f(\*inputs*), *jvp\_tensor*), where *jvp\_tensor* is the same shape as the output of *f*

**Return type** Tuple[*Union[Tensor, Sequence[Tensor]]*, *Union[Tensor, Sequence[Tensor]]*]

**kron**(*a*: Any, *b*: Any) → Any

Return the kronecker product of two matrices *a* and *b*.

**Parameters**

- **a** (*Tensor*) – tensor in matrix form
- **b** (*Tensor*) – tensor in matrix form

**Returns** kronecker product of *a* and *b*

**Return type** Tensor

**left\_shift**(*x*: Any, *y*: Any) → Any

Shift the bits of an integer *x* to the left *y* bits.

**Parameters**

- **x** (*Tensor*) – input values

- **y (Tensor)** – Number of bits shift to x

**Returns** result with the same shape as x

**Return type** Tensor

**log**(*tensor*: Any) → Any

Return elementwise natural logarithm of *tensor*. :param *tensor*: A tensor.

**Returns** Tensor

**matmul**(*tensor1*: Any, *tensor2*: Any) → Any

Perform a possibly batched matrix-matrix multiplication between *tensor1* and *tensor2*. The following behaviour is similar to *numpy.matmul*: - If both arguments are 2-D they are multiplied like conventional matrices.

- If either argument is N-D, N > 2, it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.

Both arguments to *matmul* have to be tensors of order  $\geq 2$ . :param *tensor1*: An input tensor. :param *tensor2*: An input tensor.

**Returns** The result of performing the matmul.

**Return type** tensor

**max**(*a*: Any, *axis*: Optional[int] = None) → Any

Return the maximum of an array or maximum along an axis.

**Parameters**

- **a (Tensor)** – [description]
- **axis (Optional[int], optional)** – [description], defaults to None

**Returns** [description]

**Return type** Tensor

**mean**(*a*: Any, *axis*: Optional[Sequence[int]] = None, *keepdims*: bool = False) → Any

Compute the arithmetic mean for *a* along the specified *axis*.

**Parameters**

- **a (Tensor)** – tensor to take average
- **axis (Optional[Sequence[int]], optional)** – the axis to take mean, defaults to None indicating sum over flatten array
- **keepdims (bool, optional)** – \_description\_, defaults to False

**Returns** \_description\_

**Return type** Tensor

**min**(*a*: Any, *axis*: Optional[int] = None) → Any

Return the minimum of an array or minimum along an axis.

**Parameters**

- **a (Tensor)** – [description]
- **axis (Optional[int], optional)** – [description], defaults to None

**Returns** [description]

**Return type** Tensor**mod**(*x*: Any, *y*: Any) → Any

Compute y-mod of x (negative number behavior is not guaranteed to be consistent)

**Parameters**

- **x** (Tensor) – input values
- **y** (Tensor) – mod y

**Returns** results**Return type** Tensor**multiply**(*tensor1*: Any, *tensor2*: Any) → AnyReturn the default multiplication of *tensor*.

A backend can override such implementation. :param tensor1: A tensor. :param tensor2: A tensor.

**Returns** Tensor**norm**(*tensor*: Any) → AnyCalculate the L2-norm of the elements of *tensor***numpy**(*a*: Any) → AnyReturn the numpy array of a tensor *a*, but may not work in a jitted function.**Parameters** **a** (Tensor) – tensor in matrix form**Returns** numpy array of *a***Return type** Tensor**one\_hot**(*a*: Any, *num*: int) → AnySee doc for [onehot\(\)](#)**onehot**(*a*: Any, *num*: int) → AnyOne-hot encodes the given *a*. Each index in the input *a* is encoded as a vector of zeros of length *num* with the element at index set to one:**Parameters**

- **a** (Tensor) – input tensor
- **num** (int) – number of features in onehot dimension

**Returns** onehot tensor with the last extra dimension**Return type** Tensor**ones**(*shape*: Sequence[int], *dtype*: Optional[str] = None) → AnyReturn an ones-matrix of dimension *dim* Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported :param shape: The dimension of the returned matrix. :type shape: int :param dtype: The dtype of the returned matrix.**outer\_product**(*tensor1*: Any, *tensor2*: Any) → Any

Calculate the outer product of the two given tensors.

**pivot**(*tensor*: Any, *pivot\_axis*: int = -1) → AnyReshapes a tensor into a matrix, whose columns (rows) are the vectorized dimensions to the left (right) of *pivot\_axis*.In other words, with *tensor.shape* = (1, 2, 4, 5) and *pivot\_axis*=2, this function returns an (8, 5) matrix.**Parameters**

- **tensor** – The tensor to pivot.
- **pivot\_axis** – The axis about which to pivot.

**Returns** The pivoted tensor.

**power**(*a*: Any, *b*: Union[Any, float]) → Any

**Returns the exponentiation of tensor a raised to b.**

**If b is a tensor, then the exponentiation is element-wise** between the two tensors, with *a* as the base and *b* as the power. Note that *a* and *b* must be broadcastable to the same shape if *b* is a tensor.

**If b is a scalar, then the exponentiation is each value in a raised to the power of b.**

#### Parameters

- **a** – The tensor containing the bases.
- **b** – The tensor containing the powers; or a single scalar as the power.

**Returns**

**The tensor that is each element of a raised to the power of b.** Note that the shape of the returned tensor is that produced by the broadcast of *a* and *b*.

**probability\_sample**(*shots*: int, *p*: Any, *status*: Optional[Any] = None, *g*: Optional[Any] = None) → Any

Drawn *shots* samples from probability distribution *p*, given the external randomness determined by uniform distributed *status* tensor or backend random generator *g*. This method is similar with *stateful\_randc*, but it supports *status* beyond *g*, which is convenient when *jit* or *vmap*

#### Parameters

- **shots (int)** – Number of samples to draw with replacement
- **p (Tensor)** – prbability vector
- **status (Optional[Tensor], optional)** – external randomness as a tensor with each element drawn uniformly from [0, 1], defaults to None
- **g (Any, optional)** – backend random generator, defaults to None

**Returns** The drawn sample as an int tensor

**Return type** Tensor

**qr**(*tensor*: Any, *pivot\_axis*: int = -1, *non\_negative\_diagonal*: bool = False) → Tuple[Any, Any]

Computes the QR decomposition of a tensor.

**randn**(*shape*: Tuple[int, ...], *dtype*: Optional[numpy.dtype] = None, *seed*: Optional[int] = None) → Any

Return a random-normal-matrix of dimension *dim* Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends).

Block-sparse behavior is currently not supported :param *shape*: The dimension of the returned matrix. :type *shape*: int :param *dtype*: The dtype of the returned matrix. :param *seed*: The seed for the random number generator

**random\_split**(*key*: Any) → Tuple[Any, Any]

A jax like split API, but it doesn't split the key generator for other backends. It is just for a consistent interface of random code; make sure you know what the function actually does. This function is mainly a utility to write backend agnostic code instead of doing magic things.

**Parameters** **key** (Any) – [description]

**Returns** [description]

**Return type** Tuple[Any, Any]

**random\_uniform**(*shape*: Tuple[int, ...], *boundaries*: Optional[Tuple[float, float]] = (0.0, 1.0), *dtype*: Optional[numpy.dtype] = None, *seed*: Optional[int] = None) → Any

Return a random uniform matrix of dimension *dim*.

Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported :param *shape*: The dimension of the returned matrix. :type *shape*: int :param *boundaries*: The boundaries of the uniform distribution. :type *boundaries*: tuple :param *dtype*: The dtype of the returned matrix. :param *seed*: The seed for the random number generator

**Returns** random uniform initialized tensor.

**Return type** Tensor

**real**(*a*: Any) → Any

Return the elementwise real value of a tensor *a*.

**Parameters** *a* (Tensor) – tensor

**Returns** real value of *a*

**Return type** Tensor

**relu**(*a*: Any) → Any

Rectified linear unit activation function. Computes the element-wise function:

$$\text{relu}(x) = \max(x, 0)$$

**Parameters** *a* (Tensor) – Input tensor

**Returns** Tensor after relu

**Return type** Tensor

**reshape**(*tensor*: Any, *shape*: Any) → Any

Reshape tensor to the given shape.

**Parameters** *tensor* – A tensor.

**Returns** The reshaped tensor.

**reshape2**(*a*: Any) → Any

Reshape a tensor to the [2, 2, ...] shape.

**Parameters** *a* (Tensor) – Input tensor

**Returns** the reshaped tensor

**Return type** Tensor

**reshapem**(*a*: Any) → Any

Reshape a tensor to the [l, l] shape.

**Parameters** *a* (Tensor) – Input tensor

**Returns** the reshaped tensor

**Return type** Tensor

**reverse**(*a*: Any) → Any

return *a*[::-1], only 1D tensor is guaranteed for consistent behavior

**Parameters** *a* (Tensor) – 1D tensor

**Returns** 1D tensor in reverse order

**Return type** Tensor

**right\_shift**(*x*: Any, *y*: Any) → Any

Shift the bits of an integer *x* to the right *y* bits.

**Parameters**

- **x** (*Tensor*) – input values
- **y** (*Tensor*) – Number of bits shift to *x*

**Returns** result with the same shape as *x*

**Return type** Tensor

**rq**(*tensor*: Any, *pivot\_axis*: int = -1, *non\_negative\_diagonal*: bool = False) → Tuple[Any, Any]

Computes the RQ (reversed QR) decomposition of a tensor.

**scatter**(*operand*: Any, *indices*: Any, *updates*: Any) → Any

Roughly equivalent to *operand*[*indices*] = *updates*, indices only support shape with rank 2 for now.

**Parameters**

- **operand** (*Tensor*) – [description]
- **indices** (*Tensor*) – [description]
- **updates** (*Tensor*) – [description]

**Returns** [description]

**Return type** Tensor

**searchsorted**(*a*: Any, *v*: Any, *side*: str = 'left') → Any

Find indices where elements should be inserted to maintain order.

**Parameters**

- **a** (*Tensor*) – input array sorted in ascending order
- **v** (*Tensor*) – value to inserted
- **side** (str, optional) – If ‘left’, the index of the first suitable location found is given. If ‘right’, return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *a*), defaults to “left”

**Returns** Array of insertion points with the same shape as *v*, or an integer if *v* is a scalar.

**Return type** Tensor

**serialize\_tensor**(*tensor*: Any) → str

Return a string that serializes the given tensor.

**Parameters** **tensor** – The input tensor.

**Returns** A string representing the serialized tensor.

**set\_random\_state**(*seed*: Optional[int] = None, *get\_only*: bool = False) → Any

Set the random state attached to the backend.

**Parameters**

- **seed** (Optional[int], optional) – the random seed, defaults to be None
- **get\_only** (bool, defaults to be False) – If set to be true, only get the random state in return instead of setting the state on the backend

**shape\_concat**(*values*: Any, *axis*: int) → Any  
Concatenate a sequence of tensors together about the given axis.

**shape\_prod**(*values*: Any) → Any  
Take the product of all of the elements in *values*

**shape\_tensor**(*tensor*: Any) → Any  
Get the shape of a tensor.

**Parameters** **tensor** – A tensor.

**Returns** The shape of the input tensor returned as another tensor.

**shape\_tuple**(*tensor*: Any) → Tuple[Optional[int], ...]  
Get the shape of a tensor as a tuple of integers.

**Parameters** **tensor** – A tensor.

**Returns** The shape of the input tensor returned as a tuple of ints.

**sigmoid**(*a*: Any) → Any  
Compute sigmoid of input *a*

**Parameters** **a** (Tensor) – [description]

**Returns** [description]

**Return type** Tensor

**sign**(*tensor*: Any) → Any  
Returns an elementwise tensor with entries  $y[i] = 1, 0, -1$  if  $\text{tensor}[i] > 0, == 0$ , and  $< 0$  respectively.

For complex input the behaviour of this function may depend on the backend. The NumPy version returns  $y[i] = x[i]/\sqrt{|x[i]|^2}$ .

**Parameters** **tensor** – The input tensor.

**sin**(*a*: Any) → Any  
Return sin of *tensor*. :param *tensor*: A tensor.

**Returns** Tensor

**sinh**(*a*: Any) → Any  
Return the sinh of a tensor *a*.

**Parameters** **a** (Tensor) – tensor in matrix form

**Returns** sinh of *a*

**Return type** Tensor

**size**(*a*: Any) → Any  
Return the total number of elements in *a* in tensor form.

**Parameters** **a** (Tensor) – tensor

**Returns** the total number of elements in *a*

**Return type** Tensor

**sizen**(*a*: Any) → int  
Return the total number of elements in tensor *a*, but in integer form.

**Parameters** **a** (Tensor) – tensor

**Returns** the total number of elements in tensor *a*

**Return type** int

**slice**(*tensor*: Any, *start\_indices*: Tuple[int, ...], *slice\_sizes*: Tuple[int, ...]) → Any

Obtains a slice of a tensor based on *start\_indices* and *slice\_sizes*.

#### Parameters

- **tensor** – A tensor.
- **start\_indices** – Tuple of integers denoting start indices of slice.
- **slice\_sizes** – Tuple of integers denoting size of slice along each axis.

**softmax**(*a*: Sequence[Any], *axis*: Optional[int] = None) → Any

Softmax function. Computes the function which rescales elements to the range [0,1] such that the elements along axis sum to 1.

$$\text{softmax}(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

#### Parameters

- **a** (Sequence[*Tensor*]) – Tensor
- **axis** (int, optional) – A dimension along which Softmax will be computed , de-faults to None for all axis sum.

**Returns** concatenated tensor

**Return type** Tensor

**solve**(*A*: Any, *b*: Any, *assume\_a*: str = 'gen') → Any

Solve the linear system Ax=b and return the solution x.

#### Parameters

- **A** (*Tensor*) – The multiplied matrix.
- **b** (*Tensor*) – The resulted matrix.

**Returns** The solution of the linear system.

**Return type** Tensor

**sparse\_dense\_matmul**(*sp\_a*: Any, *b*: Any) → Any

A sparse matrix multiplies a dense matrix.

#### Parameters

- **sp\_a** (*Tensor*) – a sparse matrix
- **b** (*Tensor*) – a dense matrix

**Returns** dense matrix

**Return type** Tensor

**sparse\_shape**(*tensor*: Any) → Tuple[Optional[int], ...]

**sqrt**(*tensor*: Any) → Any

Take the square root (element wise) of a given tensor.

**sqrtmh**(*a*: Any) → Any

Return the sqrtm of a Hermitian matrix a.

**Parameters** **a** (*Tensor*) – tensor in matrix form

**Returns** sqrtm of a

**Return type** Tensor

**stack**(*a*: Sequence[*Any*], *axis*: int = 0) → *Any*  
Concatenates a sequence of tensors *a* along a new dimension *axis*.

**Parameters**

- **a** (Sequence[Tensor]) – List of tensors in the same shape
- **axis** (int, optional) – the stack axis, defaults to 0

**Returns** concatenated tensor

**Return type** Tensor

**stateful\_randc**(*g*: numpy.random.\_generator.Generator, *a*: Union[int, Sequence[int], Any], *shape*: Union[int, Sequence[int]], *p*: Optional[Union[Sequence[float], Any]] = None) → *Any*  
[summary]

**Parameters**

- **g** (*Any*) – [description]
- **a** (Union[int, Sequence[int], Tensor]) – The possible options
- **shape** (Union[int, Sequence[int]]) – Sampling output shape
- **p** (Optional[Union[Sequence[float], Tensor]], optional) – probability for each option in *a*, defaults to None, as equal probability distribution

**Returns** [description]

**Return type** Tensor

**stateful\_randn**(*g*: numpy.random.\_generator.Generator, *shape*: Union[int, Sequence[int]] = 1, *mean*: float = 0, *stddev*: float = 1, *dtype*: str = '32') → *Any*  
[summary]

**Parameters**

- **self** (*Any*) – [description]
- **g** (*Any*) – stateful register for each package
- **shape** (Union[int, Sequence[int]]) – shape of output sampling tensor
- **mean** (float, optional) – [description], defaults to 0
- **stddev** (float, optional) – [description], defaults to 1
- **dtype** (str, optional) – only real data type is supported, "32" or "64", defaults to "32"

**Returns** [description]

**Return type** Tensor

**stateful\_randu**(*g*: numpy.random.\_generator.Generator, *shape*: Union[int, Sequence[int]] = 1, *low*: float = 0, *high*: float = 1, *dtype*: str = '32') → *Any*  
Uniform random sampler from *low* to *high*.

**Parameters**

- **g** (*Any*) – stateful register for each package
- **shape** (Union[int, Sequence[int]], optional) – shape of output sampling tensor, defaults to 1
- **low** (float, optional) – [description], defaults to 0

- **high** (*float, optional*) – [description], defaults to 1
- **dtype** (*str, optional*) – only real data type is supported, “32” or “64”, defaults to “32”

**Returns** [description]

**Return type** Tensor

**std**(*a: Any, axis: Optional[Sequence[int]] = None, keepdims: bool = False*) → Any

Compute the standard deviation along the specified axis.

**Parameters**

- **a** (*Tensor*) – \_description\_
- **axis** (*Optional[Sequence[int]], optional*) – Axis or axes along which the standard deviation is computed, defaults to None, implying all axis
- **keepdims** (*bool, optional*) – If this is set to True, the axes which are reduced are left in the result as dimensions with size one, defaults to False

**Returns** \_description\_

**Return type** Tensor

**stop\_gradient**(*a: Any*) → Any

Stop backpropagation from *a*.

**Parameters** **a** (*Tensor*) – [description]

**Returns** [description]

**Return type** Tensor

**subtraction**(*tensor1: Any, tensor2: Any*) → Any

Return the default subtraction of *tensor*. A backend can override such implementation.  
:param tensor1: A tensor. :param tensor2: A tensor.

**Returns** Tensor

**sum**(*a: Any, axis: Optional[Sequence[int]] = None, keepdims: bool = False*) → Any

Sum elements of *tensor* along the specified *axis*. Results in a new Tensor with the summed axis removed.  
:param tensor: An input tensor.

**Returns**

The result of performing the summation. The order of the tensor will be reduced by 1.

**Return type** tensor

**svd**(*tensor: Any, pivot\_axis: int = -1, max\_singular\_values: Optional[int] = None, max\_truncation\_error: Optional[float] = None, relative: Optional[bool] = False*) → Tuple[Any, Any, Any]

Computes the singular value decomposition (SVD) of a tensor.

The SVD is performed by treating the tensor as a matrix, with an effective left (row) index resulting from combining the axes *tensor.shape[:pivot\_axis]* and an effective right (column) index resulting from combining the axes *tensor.shape[pivot\_axis:]*.

For example, if *tensor* had a shape (2, 3, 4, 5) and *pivot\_axis* was 2, then *u* would have shape (2, 3, 6), *s* would have shape (6), and *vh* would have shape (6, 4, 5).

If *max\_singular\_values* is set to an integer, the SVD is truncated to keep at most this many singular values.

If `max_truncation_error > 0`, as many singular values will be truncated as possible, so that the truncation error (the norm of discarded singular values) is at most `max_truncation_error`. If `relative` is set `True` then `max_truncation_err` is understood relative to the largest singular value.

If both `max_singular_values` and `max_truncation_error` are specified, the number of retained singular values will be `min(max_singular_values, nsv_auto_trunc)`, where `nsv_auto_trunc` is the number of singular values that must be kept to maintain a truncation error smaller than `max_truncation_error`.

The output consists of three tensors `u`, `s`, `vh` such that: ```` python`

```
u[i1,...,iN, j] * s[j] * vh[j, k1,...,kM] == tensor[i1,...,iN, k1,...,kM]
```

````` Note that the output ordering matches `numpy.linalg.svd` rather than `tf.svd`.

Parameters

- **tensor** – A tensor to be decomposed.
- **pivot_axis** – Where to split the tensor’s axes before flattening into a matrix.
- **max_singular_values** – The number of singular values to keep, or `None` to keep them all.
- **max_truncation_error** – The maximum allowed truncation error or `None` to not do any truncation.
- **relative** – Multiply `max_truncation_err` with the largest singular value.

Returns

Left tensor factor. `s`: Vector of ordered singular values from largest to smallest. `vh`: Right tensor factor. `s_rest`: Vector of discarded singular values (length zero if no truncation).

Return type

```
switch(index: Any, branches: Sequence[Callable[[], Any]]) → Any  
branches[index]()
```

Parameters

- **index** (`Tensor`) – [description]
- **branches** (`Sequence[Callable[[], Tensor]]`) – [description]

Returns

[description]

Return type

```
tan(a: Any) → Any  
Return the tan of a tensor a.
```

Parameters `a` (`Tensor`) – tensor in matrix form

Returns tan of a

Return type Tensor

```
tanh(a: Any) → Any  
Return the tanh of a tensor a.
```

Parameters `a` (`Tensor`) – tensor in matrix form

Returns tanh of a

Return type Tensor

tensoordot(*a*: Any, *b*: Any, *axes*: Union[int, Sequence[Sequence[int]]]) → Any

Do a tensordot of tensors *a* and *b* over the given axes.

Parameters

- **a** – A tensor.
- **b** – Another tensor.
- **axes** – Two lists of integers. These values are the contraction axes.

tile(*a*: Any, *rep*: Any) → Any

Constructs a tensor by tiling a given tensor.

Parameters

- **a** (*Tensor*) – [description]
- **rep** (*Tensor*) – 1d tensor with length the same as the rank of *a*

Returns [description]

Return type Tensor

to_dense(*sp_a*: Any) → Any

Convert a sparse matrix to dense tensor.

Parameters **sp_a** (*Tensor*) – a sparse matrix

Returns the resulted dense matrix

Return type Tensor

to_dlpack(*a*: Any) → Any

Transform the tensor *a* as a dlpack capsule

Parameters **a** (*Tensor*) – _description_

Returns _description_

Return type Any

trace(*tensor*: Any, *offset*: int = 0, *axis1*: int = - 2, *axis2*: int = - 1) → Any

Return summed entries along diagonals.

If tensor is 2-D, the sum is over the diagonal of tensor with the given offset, i.e., the collection of elements of the form *a*[*i*, *i*+*offset*]. If *a* has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-array whose diagonal is summed.

Parameters

- **tensor** – A tensor.
- **offset** – Offset of the diagonal from the main diagonal.
- **axis1** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second-last/last axis.
- **axis2** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second-last/last axis.

Returns The batched summed diagonals.

Return type array_of_diagonals

transpose(*tensor*: Any, *perm*: Optional[Sequence] = None) → Any

Transpose a tensor according to a given permutation. By default the axes are reversed. :param tensor: A tensor. :param perm: The permutation of the axes.

Returns The transposed tensor

tree_flatten(*pytree*: Any) → Tuple[Any, Any]

Flatten python structure to 1D list

Parameters *pytree* (Any) – python structure to be flattened

Returns The 1D list of flattened structure and treedef which can be used for later unflatten

Return type Tuple[Any, Any]

tree_map(*f*: Callable[[...], Any], **pytrees*: Any) → Any

Return the new tree map with multiple arg function *f* through pytrees.

Parameters

- *f* (Callable[..., Any]) – The function

- *pytrees* (Any) – inputs as any python structure

Raises `NotImplementedError` – raise when neither tensorflow or jax is installed.

Returns The new tree map with the same structure but different values.

Return type Any

tree_unflatten(*treedef*: Any, *leaves*: Any) → Any

Pack 1D list to pytree defined via *treedef*

Parameters

- *treedef* (Any) – Def of pytree structure, the second return from `tree_flatten`

- *leaves* (Any) – the 1D list of flattened data structure

Returns Packed pytree

Return type Any

unique_with_counts(*a*: Any, ***kws*: Any) → Tuple[Any, Any]

Find the unique elements and their corresponding counts of the given tensor *a*.

Parameters *a* (Tensor) – [description]

Returns Unique elements, corresponding counts

Return type Tuple[Tensor, Tensor]

value_and_grad(*f*: Callable[[...], Any], *argnums*: Union[int, Sequence[int]] = 0, *has_aux*: bool = False)

→ Callable[[...], Tuple[Any, Any]]]

Return the function which returns the value and grad of *f*.

Example

```
>>> f = lambda x,y: x**2+2*y
>>> g = tc.backend.value_and_grad(f)
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
5, 2
>>> g = tc.backend.value_and_grad(f, argnums=(0,1))
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
5, [2, 2]
```

Parameters

- *f* (Callable[..., Any]) – the function to be differentiated

- **argnums** (*Union[int, Sequence[int]]*, *optional*) – the position of args in **f** that are to be differentiated, defaults to be 0

Returns the value and grad function of **f** with the same set of arguments as **f**

Return type Callable[..., Tuple[Any, Any]]

vectorized_value_and_grad(*f: Callable[..., Any]*, *argnums: Union[int, Sequence[int]] = 0*, *vectorized_argnums: Union[int, Sequence[int]] = 0*, *has_aux: bool = False*) → Callable[..., Tuple[Any, Any]]

Return the VVAG function of **f**. The inputs for **f** is (*args[0]*, *args[1]*, *args[2]*, ...), and the output of **f** is a scalar. Suppose VVAG(**f**) is a function with inputs in the form (*vargs[0]*, *args[1]*, *args[2]*, ...), where *vargs[0]* has one extra dimension than *args[0]* in the first axis and consistent with *args[0]* in shape for remaining dimensions, i.e. $\text{shape}(\text{vargs}[0]) = [\text{batch}] + \text{shape}(\text{args}[0])$. (We only cover cases where **vectorized_argnums** defaults to 0 here for demonstration). VVAG(**f**) returns a tuple as a value tensor with shape [batch, 1] and a gradient tuple with shape: ([batch]+shape(args[argnum]) for argnum in **argnums**). The gradient for **argnums=k** is defined as

$$g^k = \frac{\partial \sum_{i \in \text{batch}} f(\text{vargs}[0][i], \text{args}[1], \dots)}{\partial \text{args}[k]}$$

Therefore, if **argnums=0**, the gradient is reduced to

$$g_i^0 = \frac{\partial f(\text{vargs}[0][i])}{\partial \text{vargs}[0][i]}$$

, which is specifically suitable for batched VQE optimization, where *args[0]* is the circuit parameters.

And if **argnums=1**, the gradient is like

$$g_i^1 = \frac{\partial \sum_j f(\text{vargs}[0][j], \text{args}[1])}{\partial \text{args}[1][i]}$$

, which is suitable for quantum machine learning scenarios, where **f** is the loss function, *args[0]* corresponds to the input data and *args[1]* corresponds to the weights in the QML model.

Parameters

- **f** (*Callable[..., Any]*) – [description]
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – [description], defaults to 0
- **vectorized_argnums** (*Union[int, Sequence[int]]*, *defaults to 0*) – the args to be vectorized, these arguments should share the same batch shape in the fist dimension

Returns [description]

Return type Callable[..., Tuple[Any, Any]]

vjp(*f: Callable[..., Any]*, *inputs: Union[Any, Sequence[Any]]*, *v: Union[Any, Sequence[Any]]*) → Tuple[Union[Any, Sequence[Any]], Union[Any, Sequence[Any]]]

Function that computes the dot product between a vector **v** and the Jacobian of the given function at the point given by the inputs. (reverse mode AD relevant) Strictly speaking, this function is **value_and_vjp**.

Parameters

- **f** (*Callable[..., Any]*) – the function to carry out vjp calculation
- **inputs** (*Union[Tensor, Sequence[Tensor]]*) – input for **f**

- **v** (*Union[Tensor, Sequence[Tensor]]*) – value vector or gradient from downstream in reverse mode AD the same shape as return of function **f**

Returns (*f(*inputs*), *vjp_tensor*), where *vjp_tensor* is the same shape as *inputs*

Return type *Tuple[Union[Tensor, Sequence[Tensor]], Union[Tensor, Sequence[Tensor]]]*

vmap(*f: Callable[...], Any], vectorized_argnums: Union[int, Sequence[int]] = 0*) → *Any*

Return the vectorized map or batched version of *f* on the first extra axis. The general interface supports *f* with multiple arguments and broadcast in the fist dimension.

Parameters

- **f** (*Callable[..., Any]*) – function to be broadcasted.
- **vectorized_argnums** (*Union[int, Sequence[int]]*, *defaults to 0*) – the args to be vectorized, these arguments should share the same batch shape in the fist dimension

Returns vmap version of *f*

Return type *Any*

vvag(*f: Callable[...], Any], argnums: Union[int, Sequence[int]] = 0, vectorized_argnums: Union[int,*

Sequence[int]] = 0, has_aux: bool = False) → *Callable[...], Tuple[Any, Any]]*

Return the VVAG function of *f*. The inputs for *f* is (*args[0]*, *args[1]*, *args[2]*, ...), and the output of *f* is a scalar. Suppose VVAG(*f*) is a function with inputs in the form (*vargs[0]*, *args[1]*, *args[2]*, ...), where *vargs[0]* has one extra dimension than *args[0]* in the first axis and consistent with *args[0]* in shape for remaining dimensions, i.e. *shape(vargs[0]) = [batch] + shape(args[0])*. (We only cover cases where *vectorized_argnums* defaults to 0 here for demonstration). VVAG(*f*) returns a tuple as a value tensor with shape [batch, 1] and a gradient tuple with shape: ([batch]+shape(args[argnum])) for argnum in *argnums*). The gradient for *argnums=k* is defined as

$$g^k = \frac{\partial \sum_{i \in \text{batch}} f(\text{vargs}[0][i], \text{args}[1], \dots)}{\partial \text{args}[k]}$$

Therefore, if *argnums=0*, the gradient is reduced to

$$g_i^0 = \frac{\partial f(\text{vargs}[0][i])}{\partial \text{vargs}[0][i]}$$

, which is specifically suitable for batched VQE optimization, where *args[0]* is the circuit parameters.

And if *argnums=1*, the gradient is like

$$g_i^1 = \frac{\partial \sum_j f(\text{vargs}[0][j], \text{args}[1])}{\partial \text{args}[1][i]}$$

, which is suitable for quantum machine learning scenarios, where *f* is the loss function, *args[0]* corresponds to the input data and *args[1]* corresponds to the weights in the QML model.

Parameters

- **f** (*Callable[..., Any]*) – [description]
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – [description], defaults to 0
- **vectorized_argnums** (*Union[int, Sequence[int]]*, *defaults to 0*) – the args to be vectorized, these arguments should share the same batch shape in the fist dimension

Returns [description]

Return type Callable[..., Tuple[Any, Any]]

zeros(*shape*: Sequence[int], *dtype*: Optional[str] = None) → Any

Return a zeros-matrix of dimension *dim*. Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends).

Block-sparse behavior is currently not supported :param *shape*: The dimension of the returned matrix.
:type *shape*: int :param *dtype*: The dtype of the returned matrix.

tensorcircuit.backends.pytorch_backend

Backend magic inherited from tensornetwork: pytorch backend

class tensorcircuit.backends.pytorch_backend.PyTorchBackend

Bases: tensornetwork.backends.pytorch.pytorch_backend.PyTorchBackend, tensorcircuit.backends.abstract_backend.ExtendedBackend

See the original backend API at [pytorch backend](#)

Note the functionality provided by pytorch backend is incomplete, it currently lacks native efficient jit and vmap support.

__init__() → None

abs(*tensor*: Any) → Any

Returns the elementwise absolute value of tensor. :param *tensor*: An input tensor.

Returns Its elementwise absolute value.

Return type tensor

acos(*a*: Any) → Any

Return the acos of a tensor *a*.

Parameters *a* (Tensor) – tensor in matrix form

Returns acos of *a*

Return type Tensor

acosh(*a*: Any) → Any

Return the acosh of a tensor *a*.

Parameters *a* (Tensor) – tensor in matrix form

Returns acosh of *a*

Return type Tensor

addition(*tensor1*: Any, *tensor2*: Any) → Any

Return the default addition of *tensor*. A backend can override such implementation. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

Returns Tensor

adjoint(*a*: Any) → Any

Return the conjugate and transpose of a tensor *a*

Parameters *a* (Tensor) – Input tensor

Returns adjoint tensor of *a*

Return type Tensor

arange(*start: int, stop: Optional[int] = None, step: int = 1*) → Any

Values are generated within the half-open interval [start, stop)

Parameters

- **start** (*int*) – start index
- **stop** (*Optional[int], optional*) – end index, defaults to None
- **step** (*Optional[int], optional*) – steps, defaults to 1

Returns _description_

Return type Tensor

argmax(*a: Any, axis: int = 0*) → Any

Return the index of maximum of an array an axis.

Parameters

- **a** (*Tensor*) – [description]
- **axis** (*int*) – [description], defaults to 0, different behavior from numpy defaults!

Returns [description]

Return type Tensor

argmin(*a: Any, axis: int = 0*) → Any

Return the index of minimum of an array an axis.

Parameters

- **a** (*Tensor*) – [description]
- **axis** (*int*) – [description], defaults to 0, different behavior from numpy defaults!

Returns [description]

Return type Tensor

asin(*a: Any*) → Any

Return the acos of a tensor a.

Parameters **a** (*Tensor*) – tensor in matrix form

Returns asin of a

Return type Tensor

asinh(*a: Any*) → Any

Return the asinh of a tensor a.

Parameters **a** (*Tensor*) – tensor in matrix form

Returns asinh of a

Return type Tensor

atan(*a: Any*) → Any

Return the atan of a tensor a.

Parameters **a** (*Tensor*) – tensor in matrix form

Returns atan of a

Return type Tensor

atan2(*y*: Any, *x*: Any) → Any

Return the atan of a tensor *y/x*.

Parameters **a** (Tensor) – tensor in matrix form

Returns atan2 of a

Return type Tensor

atanh(*a*: Any) → Any

Return the atanh of a tensor *a*.

Parameters **a** (Tensor) – tensor in matrix form

Returns atanh of a

Return type Tensor

broadcast_left_multiplication(*tensor1*: Any, *tensor2*: Any) → Any

Perform broadcasting for multiplication of *tensor1* onto *tensor2*, i.e. *tensor1* * *tensor2`*, where *tensor2* is an arbitrary tensor and *tensor1* is a one-dimensional tensor. The broadcasting is applied to the first index of *tensor2*. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

Returns The result of multiplying *tensor1* onto *tensor2*.

Return type Tensor

broadcast_right_multiplication(*tensor1*: Any, *tensor2*: Any) → Any

Perform broadcasting for multiplication of *tensor2* onto *tensor1*, i.e. *tensor1* * *tensor2`*, where *tensor1* is an arbitrary tensor and *tensor2* is a one-dimensional tensor. The broadcasting is applied to the last index of *tensor1*. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

Returns The result of multiplying *tensor1* onto *tensor2*.

Return type Tensor

cast(*a*: Any, *dtype*: str) → Any

Cast the tensor *dtype* of a *a*.

Parameters

- **a** (Tensor) – tensor
- **dtype** (str) – “float32”, “float64”, “complex64”, “complex128”

Returns *a* of new *dtype*

Return type Tensor

cholesky(*tensor*: Any, *pivot_axis*: int = -1, *non_negative_diagonal*: bool = False) → Tuple[Any, Any]

concat(*a*: Sequence[Any], *axis*: int = 0) → Any

Join a sequence of arrays along an existing axis.

Parameters

- **a** (Sequence[*Tensor*]) – [description]
- **axis** (int, optional) – [description], defaults to 0

cond(*pred*: bool, *true_fun*: Callable[[*Tensor*]], *false_fun*: Callable[[*Tensor*]]) → Any

The native cond for XLA compiling, wrapper for `tf.cond` and limited functionality of `jax.lax.cond`.

Parameters

- **pred** (bool) – [description]
- **true_fun** (Callable[[*Tensor*]], *Tensor*) – [description]

- **false_fun** (*Callable*[[], *Tensor*]) – [description]

Returns [description]

Return type Tensor

conj(*tensor*: Any) → Any

Return the complex conjugate of *tensor* :param tensor: A tensor.

Returns Tensor

convert_to_tensor(*tensor*: Any) → Any

Convert a np.array or a tensor to a tensor type for the backend.

coo_sparse_matrix(*indices*: Any, *values*: Any, *shape*: Any) → Any

Generate the coo format sparse matrix from indices and values, which is the only sparse format supported in different ML backends.

Parameters

- **indices** (*Tensor*) – shape [n, 2] for n non zero values in the returned matrix
- **values** (*Tensor*) – shape [n]
- **shape** (*Tensor*) – Tuple[int, ...]

Returns [description]

Return type Tensor

coo_sparse_matrix_from_numpy(*a*: Any) → Any

Generate the coo format sparse matrix from scipy coo sparse matrix.

Parameters *a* (*Tensor*) – Scipy coo format sparse matrix

Returns SparseTensor in backend format

Return type Tensor

copy(*a*: Any) → Any

Return the copy of *a*, matrix exponential.

Parameters *a* (*Tensor*) – tensor in matrix form

Returns matrix exponential of matrix *a*

Return type Tensor

cos(*a*: Any) → Any

Return cos of *tensor*. :param tensor: A tensor.

Returns Tensor

cosh(*a*: Any) → Any

Return the cosh of a tensor *a*.

Parameters *a* (*Tensor*) – tensor in matrix form

Returns cosh of *a*

Return type Tensor

cumsum(*a*: Any, *axis*: *Optional[int] = None*) → Any

Return the cumulative sum of the elements along a given axis.

Parameters

- ***a* (*Tensor*)** – [description]

- **axis** (*Optional[int], optional*) – The default behavior is the same as numpy, different from tf/torch as cumsum of the flatten 1D array, defaults to None

Returns [description]

Return type Tensor

deserialize_tensor(*s: str*) → Any

Return a tensor given a serialized tensor string.

Parameters **s** – The input string representing a serialized tensor.

Returns The tensor object represented by the string.

device(*a: Any*) → str

get the universal device str for the tensor, in the format of tf

Parameters **a** (*Tensor*) – the tensor

Returns device str where the tensor lives on

Return type str

device_move(*a: Any, dev: Any*) → Any

move tensor *a* to device *dev*

Parameters

- **a** (*Tensor*) – the tensor

- **dev** (*Any*) – device str or device obj in corresponding backend

Returns the tensor on new device

Return type Tensor

diagflat(*tensor: Any, k: int = 0*) → Any

Flattens tensor and creates a new matrix of zeros with its elements on the *k*'th diagonal. :param tensor: A tensor. :param k: The diagonal upon which to place its elements.

Returns A new tensor with all zeros save the specified diagonal.

Return type tensor

diagonal(*tensor: Any, offset: int = 0, axis1: int = - 2, axis2: int = - 1*) → Any

Return specified diagonals.

If tensor is 2-D, returns the diagonal of tensor with the given offset, i.e., the collection of elements of the form *a[i, i+offset]*. If *a* has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing *axis1* and *axis2* and appending an index to the right equal to the size of the resulting diagonals.

This function only extracts diagonals. If you wish to create diagonal matrices from vectors, use `diagflat`.

Parameters

- **tensor** – A tensor.
- **offset** – Offset of the diagonal from the main diagonal.
- **axis1** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second-last and last axis (note this differs from the NumPy defaults).

- **axis2** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second-last and last axis (note this differs from the NumPy defaults).

Returns

A **dim = min(1, tensor.ndim - 2)** tensor storing the batched diagonals.

Return type array_of_diagonals

divide(*tensor1*: Any, *tensor2*: Any) → Any

Return the default divide of *tensor*. A backend can override such implementation. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

Returns Tensor

dtype(*a*: Any) → str

Obtain dtype string for tensor *a*

Parameters *a* (Tensor) – The tensor

Returns dtype str, such as “complex64”

Return type str

eigh(*matrix*: Any) → Tuple[Any, Any]

Compute eigenvectors and eigenvalues of a hermitian matrix.

Parameters *matrix* – A symetric matrix.

Returns The eigenvalues in ascending order. Tensor: The eigenvectors.

Return type Tensor

eigs(*A*: Callable, *args*: Optional[List[Any]] = None, *initial_state*: Optional[Any] = None, *shape*: Optional[Tuple[int, ...]] = None, *dtype*: Optional[Type[numumpy.number]] = None, *num_krylov_vecs*: int = 50, *numeig*: int = 1, *tol*: float = 1e-08, *which*: str = 'LR', *maxiter*: Optional[int] = None) → Tuple[Any, List]

Arnoldi method for finding the lowest eigenvector-eigenvalue pairs of a linear operator *A*. *A* is a callable implementing the matrix-vector product. If no *initial_state* is provided then *shape* and *dtype* have to be passed so that a suitable initial state can be randomly generated. :param *A*: A (sparse) implementation of a linear operator :param *args*: A list of arguments to *A*. *A* will be called as

res = *A*(*initial_state*, **args*).

Parameters

- **initial_state** – An initial vector for the algorithm. If *None*, a random initial *Tensor* is created using the *numpy.random.randn* method.
- **shape** – The shape of the input-dimension of *A*.
- **dtype** – The dtype of the input *A*. If both no *initial_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.
- **num_krylov_vecs** – The number of iterations (number of krylov vectors).
- **numeig** – The nummber of eigenvector-eigenvalue pairs to be computed. If *numeig* > 1, *reorthogonalize* has to be *True*.
- **tol** – The desired precision of the eigenvalu. Uses
- **which** – ['LM' | 'SM' | 'LR' | 'SR' | 'LI' | 'SI'] Which *k* eigenvectors and eigenvalues to find:

'LM' : largest magnitude 'SM' : smallest magnitude 'LR' : largest real part 'SR' : smallest real part 'LI' : largest imaginary part 'SI' : smallest imaginary part

Note that not all of those might be supported by specialized backends.

- **maxiter** – The maximum number of iterations.

Returns An array of *numeig* lowest eigenvalues *list*: A list of *numeig* lowest eigenvectors

Return type *Tensor*

eigsh(*A*: *Callable*, *args*: *Optional[List[Any]]* = *None*, *initial_state*: *Optional[Any]* = *None*, *shape*: *Optional[Tuple[int, ...]]* = *None*, *dtype*: *Optional[Type[numpy.number]]* = *None*, *num_krylov_vecs*: *int* = 50, *numeig*: *int* = 1, *tol*: *float* = 1e-08, *which*: *str* = 'LR', *maxiter*: *Optional[int]* = *None*) → *Tuple[Any, List]*

Lanczos method for finding the lowest eigenvector-eigenvalue pairs of a symmetric (hermitian) linear operator *A*. *A* is a callable implementing the matrix-vector product. If no *initial_state* is provided then *shape* and *dtype* have to be passed so that a suitable initial state can be randomly generated. :param *A*: A (sparse) implementation of a linear operator :param *args*: A list of arguments to *A*. *A* will be called as

res = *A*(*initial_state*, **args*).

Parameters

- **initial_state** – An initial vector for the algorithm. If *None*, a random initial *Tensor* is created using the *numpy.random.randn* method.
- **shape** – The shape of the input-dimension of *A*.
- **dtype** – The dtype of the input *A*. If both no *initial_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.
- **num_krylov_vecs** – The number of iterations (number of krylov vectors).
- **numeig** – The nummber of eigenvector-eigenvalue pairs to be computed. If *numeig* > 1, *reorthogonalize* has to be *True*.
- **tol** – The desired precision of the eigenvaluus. Uses
- **which** – ['LM' | 'SM' | 'LR' | 'SR' | 'LI' | 'SI'] Which *k* eigenvectors and eigenvalues to find:

'LM' : largest magnitude 'SM' : smallest magnitude 'LR' : largest real part 'SR' : smallest real part 'LI' : largest imaginary part 'SI' : smallest imaginary part

Note that not all of those might be supported by specialized backends.

- **maxiter** – The maximum number of iterations.

Returns An array of *numeig* lowest eigenvalues *list*: A list of *numeig* lowest eigenvectors

Return type *Tensor*

eigsh_lanczos(*A*: *Callable*, *args*: *Optional[List[Any]]* = *None*, *initial_state*: *Optional[Any]* = *None*, *shape*: *Optional[Tuple]* = *None*, *dtype*: *Optional[Type[numpy.number]]* = *None*, *num_krylov_vecs*: *int* = 20, *numeig*: *int* = 1, *tol*: *float* = 1e-08, *delta*: *float* = 1e-08, *ndiag*: *int* = 20, *reorthogonalize*: *bool* = *False*) → *Tuple[Any, List]*

Lanczos method for finding the lowest eigenvector-eigenvalue pairs of a *LinearOperator* *A*. :param *A*: A (sparse) implementation of a linear operator.

Call signature of *A* is *res* = *A*(*vector*, **args*), where *vector* can be an arbitrary *Tensor*, and *res.shape* has to be *vector.shape*.

Parameters

- **args** – A list of arguments to *A*. *A* will be called as *res = A(initial_state, *args)*.
- **initial_state** – An initial vector for the Lanczos algorithm. If *None*, a random initial *Tensor* is created using the *torch.randn* method
- **shape** – The shape of the input-dimension of *A*.
- **dtype** – The dtype of the input *A*. If both no *initial_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.
- **num_krylov_vecs** – The number of iterations (number of krylov vectors).
- **numeig** – The nummber of eigenvector-eigenvalue pairs to be computed. If *numeig > 1*, *reorthogonalize* has to be *True*.
- **tol** – The desired precision of the eigenvalues. Uses *torch.norm(eigvalsnew[0:numeig] - eigvalsold[0:numeig]) < tol* as stopping criterion between two diagonalization steps of the tridiagonal operator.
- **delta** – Stopping criterion for Lanczos iteration. If a Krylov vector :math: x_n has an L2 norm $\|x_n\| < \delta$, the iteration is stopped. It means that an (approximate) invariant subspace has been found.
- **ndiag** – The tridiagonal Operator is diagonalized every *ndiag* iterations to check convergence.
- **reorthogonalize** – If *True*, Krylov vectors are kept orthogonal by explicit orthogonalization (more costly than *reorthogonalize=False*)

Returns

(eigvals, eigvecs) *eigvals*: A list of *numeig* lowest eigenvalues *eigvecs*: A list of *numeig* lowest eigenvectors

eigvalsh(*a*: Any) → Any

Get the eigenvalues of matrix *a*.

Parameters **a** (*Tensor*) – tensor in matrix form

Returns eigenvalues of *a*

Return type Tensor

einsum(*expression*: str, **tensors*: Any, *optimize*: bool = True) → Any

Calculate sum of products of tensors according to expression.

eps(*dtype*: Type[numpy.number]) → float

Return machine epsilon for given *dtype*

Parameters **dtype** – A dtype.

Returns Machine epsilon.

Return type float

exp(*tensor*: Any) → Any

Return elementwise exp of *tensor*. :param *tensor*: A tensor.

Returns Tensor

expm(*a*: Any) → Any

Return expm log of *matrix*, matrix exponential. :param *matrix*: A tensor.

Returns Tensor

eye(*N*: int, *dtype*: Optional[str] = None, *M*: Optional[int] = None) → Any

Return an identity matrix of dimension dim Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported

Parameters

- **N** (int) – The dimension of the returned matrix.
- **dtype** – The dtype of the returned matrix.
- **M** (int) – The dimension of the returned matrix.

from_dlpark(*a*: Any) → Any

Transform a dlpark capsule to a tensor

Parameters **a** (Any) – the dlpark capsule

Returns _description_

Return type Tensor

gather1d(*operand*: Any, *indices*: Any) → Any

Return *operand*[*indices*], both *operand* and *indices* are rank-1 tensor.

Parameters

- **operand** (Tensor) – rank-1 tensor
- **indices** (Tensor) – rank-1 tensor with int dtype

Returns *operand*[*indices*]

Return type Tensor

get_random_state(*seed*: Optional[int] = None) → Any

Get the backend specific random state object.

Parameters **seed** (Optional[int], optional) – [description], defaults to be None

:return:the backend specific random state object :rtype: Any

gmres(*A_mv*: Callable, *b*: Any, *A_args*: Optional[List] = None, *A_kwargs*: Optional[dict] = None, *x0*: Optional[Any] = None, *tol*: float = 1e-05, *atol*: Optional[float] = None, *num_krylov_vectors*: int = 20, *maxiter*: Optional[int] = 1, *M*: Optional[Callable] = None) → Tuple[Any, int]

GMRES solves the linear system *A* @ *x* = *b* for *x* given a vector *b* and a general (not necessarily symmetric/Hermitian) linear operator *A*.

As a Krylov method, GMRES does not require a concrete matrix representation of the n by n *A*, but only a function *vector1* = *A_mv*(*vector0*, **A_args*, ***A_kwargs*) prescribing a one-to-one linear map from *vector0* to *vector1* (that is, *A* must be square, and thus *vector0* and *vector1* the same size). If *A* is a dense matrix, or if it is a symmetric/Hermitian operator, a different linear solver will usually be preferable.

GMRES works by first constructing the Krylov basis *K* = (*x0*, *A_mv*@*x0*, *A_mv*@*A_mv*@*x0*, ..., (*A_mv*ⁿ*num_krylov_vectors*)@*x0*) and then solving a certain dense linear system *K* @ *q0* = *q1* from whose solution *x* can be approximated. For *num_krylov_vectors* = *n* the solution is provably exact in infinite precision, but the expense is cubic in *num_krylov_vectors* so one is typically interested in the *num_krylov_vectors* << *n* case. The solution can in this case be repeatedly improved, to a point, by restarting the Arnoldi iterations each time *num_krylov_vectors* is reached. Unfortunately the optimal parameter choices balancing expense and accuracy are difficult to predict in advance, so applying this function requires a degree of experimentation.

In a tensor network code one is typically interested in `A_mv` implementing some tensor contraction. This implementation thus allows `b` and `x0` to be of whatever arbitrary, though identical, shape `b = A_mv(x0, ...)` expects. Reshaping to and from a matrix problem is handled internally.

Parameters

- **`A_mv`** – A function `v0 = A_mv(v, *A_args, **A_kwargs)` where `v0` and `v` have the same shape.
- **`b`** – The `b` in $A @ x = b$; it should be of the shape `A_mv` operates on.
- **`A_args`** – Positional arguments to `A_mv`, supplied to this interface as a list. Default: None.
- **`A_kwargs`** – Keyword arguments to `A_mv`, supplied to this interface as a dictionary. Default: None.
- **`x0`** – An optional guess solution. Zeros are used by default. If `x0` is supplied, its shape and dtype must match those of `b`, or an error will be thrown. Default: zeros.
- **`tol`** – Solution tolerance to achieve, $\text{norm}(\text{residual}) \leq \max(\text{tol} * \text{norm}(b), \text{atol})$. Default: `tol=1E-05`
atol=tol
- **`atol`** – Solution tolerance to achieve, $\text{norm}(\text{residual}) \leq \max(\text{tol} * \text{norm}(b), \text{atol})$. Default: `tol=1E-05`
atol=tol
- **`num_krylov_vectors`** –
 - : **Size of the Krylov space to build at each restart.** Expense is cubic in this parameter. It must be positive. If greater than `b.size`, it will be set to `b.size`. Default: 20
 - **`maxiter`** – The Krylov space will be repeatedly rebuilt up to this many times. Large values of this argument should be used only with caution, since especially for nearly symmetric matrices and small `num_krylov_vectors` convergence might well freeze at a value significantly larger than `tol`. Default: 1.
 - **`M`** – Inverse of the preconditioner of `A`; see the docstring for `scipy.sparse.linalg.gmres`. This is only supported in the numpy backend. Supplying this argument to other backends will trigger `NotImplementedError`. Default: None.

Raises `ValueError` – -if `x0` is supplied but its shape differs from that of `b`. -in NumPy, if the ARPACK solver reports a breakdown (which usually indicates some kind of floating point issue). -if `num_krylov_vectors` is 0 or exceeds `b.size`. -if `tol` was negative. -if `M` was supplied with any backend but NumPy.

Returns The converged solution. It has the same shape as `b`. `info : 0` if convergence was achieved, the number of restarts otherwise.

Return type `x`

`grad(f: Callable[..., Any], arngums: Union[int, Sequence[int]] = 0, has_aux: bool = False) → Callable[..., Any]`
Return the function which is the grad function of input f.

Example

```
>>> f = lambda x,y: x**2+2*y
>>> g = tc.backend.grad(f)
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
2
>>> g = tc.backend.grad(f, argnums=(0,1))
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
[2, 2]
```

Parameters

- **f** (*Callable[..., Any]*) – the function to be differentiated
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – the position of args in **f** that are to be differentiated, defaults to be 0

Returns the grad function of **f** with the same set of arguments as **f****Return type** Callable[..., Any]**hessian**(*f: Callable[[...], Any]*, *argnums: Union[int, Sequence[int]] = 0*) → Any**i**(*dtype: Optional[Any] = None*) → Any

Return 1.j in as a tensor compatible with the backend.

Parameters **dtype** (*str*) – “complex64” or “complex128”**Returns** 1.j tensor**Return type** Tensor**imag**(*a: Any*) → AnyReturn the elementwise imaginary value of a tensor **a**.**Parameters** **a** (*Tensor*) – tensor**Returns** imaginary value of **a****Return type** Tensor**implicit_randc**(*a: Union[int, Sequence[int], Any]*, *shape: Union[int, Sequence[int]]*, *p:**Optional[Union[Sequence[float], Any]] = None*) → Any

[summary]

Parameters

- **g** (*Any*) – [description]
- **a** (*Union[int, Sequence[int], Tensor]*) – The possible options
- **shape** (*Union[int, Sequence[int]]*) – Sampling output shape
- **p** (*Optional[Union[Sequence[float], Tensor]]*, *optional*) – probability for each option in **a**, defaults to None, as equal probability distribution

Returns [description]**Return type** Tensor**implicit_randn**(*shape: Union[int, Sequence[int]] = 1*, *mean: float = 0*, *stddev: float = 1*, *dtype: str = '32'*) → Any

Call the random normal function with the random state management behind the scene.

Parameters

- **shape** (*Union[int, Sequence[int]]*, *optional*) – [description], defaults to 1
- **mean** (*float*, *optional*) – [description], defaults to 0
- **stddev** (*float*, *optional*) – [description], defaults to 1
- **dtype** (*str*, *optional*) – [description], defaults to “32”

Returns [description]

Return type Tensor

implicit_randu(*shape: Union[int, Sequence[int]] = 1, low: float = 0, high: float = 1, dtype: str = '32'*) → Any

Call the random normal function with the random state management behind the scene.

Parameters

- **shape** (*Union[int, Sequence[int]]*, *optional*) – [description], defaults to 1
- **mean** (*float*, *optional*) – [description], defaults to 0
- **stddev** (*float*, *optional*) – [description], defaults to 1
- **dtype** (*str*, *optional*) – [description], defaults to “32”

Returns [description]

Return type Tensor

index_update(*tensor: Any, mask: Any, assignee: Any*) → Any

Update *tensor* at elements defined by *mask* with value *assignee*.

Parameters

- **tensor** – A *Tensor* object.
- **mask** – A boolean mask.
- **assignee** – A scalar *Tensor*. The values to assigned to *tensor* at positions where *mask* is *True*.

inv(*matrix: Any*) → Any

Compute the matrix inverse of *matrix*.

Parameters **matrix** – A matrix.

Returns The inverse of *matrix*

Return type Tensor

is_sparse(*a: Any*) → bool

Determine whether the type of input *a* is sparse.

Parameters **a** (*Tensor*) – input matrix *a*

Returns a bool indicating whether the matrix *a* is sparse

Return type bool

is_tensor(*a: Any*) → bool

Return a boolean on whether *a* is a tensor in backend package.

Parameters **a** (*Tensor*) – a tensor to be determined

Returns whether *a* is a tensor

Return type bool

item(*tensor*)

Return the item of a 1-element tensor.

Parameters **tensor** – A 1-element tensor

Returns The value in tensor.

jacbwd(*f*: *Callable*[..., *Any*], *argnums*: *Union[int, Sequence[int]]* = 0) → *Any*

Compute the Jacobian of *f* using reverse mode AD.

Parameters

- **f** (*Callable*[..., *Any*]) – The function whose Jacobian is required
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – the position of the arg as Jacobian input, defaults to 0

Returns outer tuple for output, inner tuple for input args

Return type Tensor

jacfwd(*f*: *Callable*[..., *Any*], *argnums*: *Union[int, Sequence[int]]* = 0) → *Any*

Compute the Jacobian of *f* using the forward mode AD.

Parameters

- **f** (*Callable*[..., *Any*]) – the function whose Jacobian is required
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – the position of the arg as Jacobian input, defaults to 0

Returns outer tuple for input args, inner tuple for outputs

Return type Tensor

jacrev(*f*: *Callable*[..., *Any*], *argnums*: *Union[int, Sequence[int]]* = 0) → *Any*

Compute the Jacobian of *f* using reverse mode AD.

Parameters

- **f** (*Callable*[..., *Any*]) – The function whose Jacobian is required
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – the position of the arg as Jacobian input, defaults to 0

Returns outer tuple for output, inner tuple for input args

Return type Tensor

jit(*f*: *Callable*[..., *Any*], *static_argnums*: *Optional[Union[int, Sequence[int]]]* = None, *jit_compile*:

Optional[bool] = None) → *Any*

Return a jitted or graph-compiled version of *fun* for JAX backend. For all other backends returns *fun*.
:param fun: Callable :param args: Arguments to *fun*. :param kwargs: Keyword arguments to *fun*.

Returns jitted/graph-compiled version of *fun*, or just *fun*.

Return type Callable

jvp(*f*: *Callable*[..., *Any*], *inputs*: *Union[Any, Sequence[Any]]*, *v*: *Union[Any, Sequence[Any]]*) →

Tuple[Union[Any, Sequence[Any]], Union[Any, Sequence[Any]]]

Function that computes a (forward-mode) Jacobian-vector product of *f*. Strictly speaking, this function is value_and_jvp.

Parameters

- **f** (*Callable*[..., *Any*]) – The function to compute jvp

- **inputs** (*Union[Tensor, Sequence[Tensor]]*) – input for **f**
- **v** (*Union[Tensor, Sequence[Tensor]]*) – tangents

Returns (*f(*inputs)*, *jvp_tensor*), where *jvp_tensor* is the same shape as the output of **f**

Return type *Tuple[Union[Tensor, Sequence[Tensor]], Union[Tensor, Sequence[Tensor]]]*

kron(*a: Any, b: Any*) → *Any*

Return the kronecker product of two matrices *a* and *b*.

Parameters

- **a** (*Tensor*) – tensor in matrix form
- **b** (*Tensor*) – tensor in matrix form

Returns kronecker product of *a* and *b*

Return type *Tensor*

left_shift(*x: Any, y: Any*) → *Any*

Shift the bits of an integer *x* to the left *y* bits.

Parameters

- **x** (*Tensor*) – input values
- **y** (*Tensor*) – Number of bits shift to **x**

Returns result with the same shape as *x*

Return type *Tensor*

log(*tensor: Any*) → *Any*

Return elementwise natural logarithm of *tensor*. :param *tensor*: A tensor.

Returns *Tensor*

matmul(*tensor1: Any, tensor2: Any*) → *Any*

Perform a possibly batched matrix-matrix multiplication between *tensor1* and *tensor2*. The following behaviour is similar to *numpy.matmul*: - If both arguments are 2-D they are multiplied like conventional matrices.

- If either argument is N-D, N > 2, it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.

Both arguments to *matmul* have to be tensors of order ≥ 2 . :param *tensor1*: An input tensor. :param *tensor2*: An input tensor.

Returns The result of performing the matmul.

Return type *tensor*

max(*a: Any, axis: Optional[int] = None*) → *Any*

Return the maximum of an array or maximum along an axis.

Parameters

- **a** (*Tensor*) – [description]
- **axis** (*Optional[int], optional*) – [description], defaults to None

Returns [description]

Return type *Tensor*

mean(*a*: Any, *axis*: Optional[Sequence[int]] = None, *keepdims*: bool = False) → Any
Compute the arithmetic mean for *a* along the specified *axis*.

Parameters

- **a** (*Tensor*) – tensor to take average
- **axis** (*Optional[Sequence[int]]*, *optional*) – the axis to take mean, defaults to None indicating sum over flatten array
- **keepdims** (*bool*, *optional*) – _description_, defaults to False

Returns _description_**Return type** Tensor

min(*a*: Any, *axis*: Optional[int] = None) → Any
Return the minimum of an array or minimum along an *axis*.

Parameters

- **a** (*Tensor*) – [description]
- **axis** (*Optional[int]*, *optional*) – [description], defaults to None

Returns [description]**Return type** Tensor

mod(*x*: Any, *y*: Any) → Any
Compute *y*-mod of *x* (negative number behavior is not guaranteed to be consistent)

Parameters

- **x** (*Tensor*) – input values
- **y** (*Tensor*) – mod *y*

Returns results**Return type** Tensor

multiply(*tensor1*: Any, *tensor2*: Any) → Any
Return the default multiplication of *tensor*.

A backend can override such implementation. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

Returns Tensor

norm(*tensor*: Any) → Any
Calculate the L2-norm of the elements of *tensor*

numpy(*a*: Any) → Any
Return the numpy array of a tensor *a*, but may not work in a jitted function.

Parameters **a** (*Tensor*) – tensor in matrix form**Returns** numpy array of *a***Return type** Tensor

one_hot(*a*: Any, *num*: int) → Any
See doc for [onehot\(\)](#)

onehot(*a*: Any, *num*: int) → Any
One-hot encodes the given *a*. Each index in the input *a* is encoded as a vector of zeros of length *num* with the element at index set to one:

Parameters

- **a** (*Tensor*) – input tensor
- **num** (*int*) – number of features in onehot dimension

Returns onehot tensor with the last extra dimension**Return type** Tensor**ones**(*shape: Tuple[int, ...]*, *dtype: Optional[str] = None*) → Any

Return an ones-matrix of dimension *dim*. Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported :param *shape*: The dimension of the returned matrix. :type *shape*: int :param *dtype*: The dtype of the returned matrix.

optimizer

alias of `tensorcircuit.backends.pytorch_backend.torch_optimizer`

outer_product(*tensor1: Any*, *tensor2: Any*) → Any

Calculate the outer product of the two given tensors.

pivot(*tensor: Any*, *pivot_axis: int = -1*) → Any

Reshapes a tensor into a matrix, whose columns (rows) are the vectorized dimensions to the left (right) of *pivot_axis*.

In other words, with *tensor.shape = (1, 2, 4, 5)* and *pivot_axis=2*, this function returns an (8, 5) matrix.

Parameters

- **tensor** – The tensor to pivot.
- **pivot_axis** – The axis about which to pivot.

Returns The pivoted tensor.**power**(*a: Any*, *b: Union[Any, float]*) → Any**Returns the exponentiation of tensor a raised to b.**

If b is a tensor, then the exponentiation is element-wise between the two tensors, with *a* as the base and *b* as the power. Note that *a* and *b* must be broadcastable to the same shape if *b* is a tensor.

If b is a scalar, then the exponentiation is each value in a raised to the power of b.

Parameters

- **a** – The tensor containing the bases.
- **b** – The tensor containing the powers; or a single scalar as the power.

Returns

The tensor that is each element of a raised to the power of b. Note that the shape of the returned tensor is that produced by the broadcast of *a* and *b*.

probability_sample(*shots: int*, *p: Any*, *status: Optional[Any] = None*, *g: Optional[Any] = None*) → Any

Drawn shots samples from probability distribution *p*, given the external randomness determined by uniform distributed *status* tensor or backend random generator *g*. This method is similar with *stateful_randc*, but it supports *status* beyond *g*, which is convenient when *jit* or *vmap*

Parameters

- **shots** (*int*) – Number of samples to draw with replacement
- **p** (*Tensor*) – probability vector
- **status** (*Optional[Tensor]*, *optional*) – external randomness as a tensor with each element drawn uniformly from [0, 1], defaults to None
- **g** (*Any*, *optional*) – backend random generator, defaults to None

Returns The drawn sample as an int tensor

Return type Tensor

qr(*tensor*: *Any*, *pivot_axis*: *int* = -1, *non_negative_diagonal*: *bool* = *False*) → Tuple[*Any*, *Any*]

Computes the QR decomposition of a tensor. The QR decomposition is performed by treating the tensor as a matrix, with an effective left (row) index resulting from combining the axes *tensor.shape[:pivot_axis]* and an effective right (column) index resulting from combining the axes *tensor.shape[pivot_axis:]*.

Example

If *tensor* had a shape (2, 3, 4, 5) and *pivot_axis* was 2, then *q* would have shape (2, 3, 6), and *r* would have shape (6, 4, 5). The output consists of two tensors *Q*, *R* such that:

$$Q[i_1, \dots, i_N, j] * R[j, k_1, \dots, k_M] == tensor[i_1, \dots, i_N, k_1, \dots, k_M]$$

Note that the output ordering matches numpy.linalg.svd rather than tf.svd.

Parameters

- **tensor** (*Tensor*) – A tensor to be decomposed.
- **pivot_axis** (*int*, *optional*) – Where to split the tensor's axes before flattening into a matrix.
- **non_negative_diagonal** (*bool*, *optional*) – a bool indicating whether the tensor is diagonal non-negative matrix.

Returns *Q*, the left tensor factor, and *R*, the right tensor factor.

Return type Tuple[*Tensor*, *Tensor*]

randn(*shape*: *Tuple[int, ...]*, *dtype*: *Optional[Any]* = *None*, *seed*: *Optional[int]* = *None*) → *Any*

Return a random-normal-matrix of dimension *dim* Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends).

Block-sparse behavior is currently not supported :param *shape*: The dimension of the returned matrix. :type *shape*: int :param *dtype*: The dtype of the returned matrix. :param *seed*: The seed for the random number generator

random_split(*key*: *Any*) → Tuple[*Any*, *Any*]

A jax like split API, but it doesn't split the key generator for other backends. It is just for a consistent interface of random code; make sure you know what the function actually does. This function is mainly a utility to write backend agnostic code instead of doing magic things.

Parameters **key** (*Any*) – [description]

Returns [description]

Return type Tuple[*Any*, *Any*]

random_uniform(*shape*: *Tuple[int, ...]*, *boundaries*: *Optional[Tuple[float, float]]* = (0.0, 1.0), *dtype*: *Optional[Any]* = *None*, *seed*: *Optional[int]* = *None*) → *Any*

Return a random uniform matrix of dimension *dim*.

Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported :param *shape*: The

dimension of the returned matrix. :type shape: int :param boundaries: The boundaries of the uniform distribution. :type boundaries: tuple :param dtype: The dtype of the returned matrix. :param seed: The seed for the random number generator

Returns random uniform initialized tensor.

Return type Tensor

real(*a*: Any) → Any

Return the elementwise real value of a tensor *a*.

Parameters *a* (Tensor) – tensor

Returns real value of *a*

Return type Tensor

relu(*a*: Any) → Any

Rectified linear unit activation function. Computes the element-wise function:

$$\text{relu}(x) = \max(x, 0)$$

Parameters *a* (Tensor) – Input tensor

Returns Tensor after relu

Return type Tensor

reshape(*tensor*: Any, *shape*: Any) → Any

Reshape tensor to the given shape.

Parameters *tensor* – A tensor.

Returns The reshaped tensor.

reshape2(*a*: Any) → Any

Reshape a tensor to the [2, 2, ...] shape.

Parameters *a* (Tensor) – Input tensor

Returns the reshaped tensor

Return type Tensor

reshapem(*a*: Any) → Any

Reshape a tensor to the [1, 1] shape.

Parameters *a* (Tensor) – Input tensor

Returns the reshaped tensor

Return type Tensor

reverse(*a*: Any) → Any

return *a*[::-1], only 1D tensor is guaranteed for consistent behavior

Parameters *a* (Tensor) – 1D tensor

Returns 1D tensor in reverse order

Return type Tensor

right_shift(*x*: Any, *y*: Any) → Any

Shift the bits of an integer *x* to the right *y* bits.

Parameters

- *x* (Tensor) – input values

- **y (Tensor)** – Number of bits shift to x

Returns result with the same shape as x

Return type Tensor

rq(*tensor*: Any, *pivot_axis*: int = 1, *non_negative_diagonal*: bool = False) → Tuple[Any, Any]

Computes the RQ decomposition of a tensor. The QR decomposition is performed by treating the tensor as a matrix, with an effective left (row) index resulting from combining the axes *tensor.shape[:pivot_axis]* and an effective right (column) index resulting from combining the axes *tensor.shape[pivot_axis:]*.

Example

If *tensor* had a shape (2, 3, 4, 5) and *pivot_axis* was 2, then *r* would have shape (2, 3, 6), and *q* would have shape (6, 4, 5). The output consists of two tensors *Q*, *R* such that:

$$Q[i_1, \dots, i_N, j] * R[j, k_1, \dots, k_M] == tensor[i_1, \dots, i_N, k_1, \dots, k_M]$$

Note that the output ordering matches numpy.linalg.svd rather than tf.svd.

Parameters

- **tensor (Tensor)** – A tensor to be decomposed.
- **pivot_axis (int, optional)** – Where to split the tensor’s axes before flattening into a matrix.
- **non_negative_diagonal (bool, optional)** – a bool indicating whether the tensor is diagonal non-negative matrix.

Returns Q, the left tensor factor, and R, the right tensor factor.

Return type Tuple[Tensor, Tensor]

scatter(*operand*: Any, *indices*: Any, *updates*: Any) → Any

Roughly equivalent to *operand[indices] = updates*, indices only support shape with rank 2 for now.

Parameters

- **operand (Tensor)** – [description]
- **indices (Tensor)** – [description]
- **updates (Tensor)** – [description]

Returns [description]

Return type Tensor

searchsorted(*a*: Any, *v*: Any, *side*: str = 'left') → Any

Find indices where elements should be inserted to maintain order.

Parameters

- **a (Tensor)** – input array sorted in ascending order
- **v (Tensor)** – value to inserted
- **side (str, optional)** – If ‘left’, the index of the first suitable location found is given. If ‘right’, return the last such index. If there is no suitable index, return either 0 or N (where N is the length of a), defaults to “left”

Returns Array of insertion points with the same shape as v, or an integer if v is a scalar.

Return type Tensor

serialize_tensor(*tensor*: Any) → str

Return a string that serializes the given tensor.

Parameters `tensor` – The input tensor.

Returns A string representing the serialized tensor.

set_random_state(*seed: Optional[int] = None, get_only: bool = False*) → Any

Set the random state attached to the backend.

Parameters

- **seed** (*Optional[int], optional*) – the random seed, defaults to be None
- **get_only** (*bool, defaults to be False*) – If set to be true, only get the random state in return instead of setting the state on the backend

shape_concat(*values: Any, axis: int*) → Any

Concatenate a sequence of tensors together about the given axis.

shape_prod(*values: Any*) → int

Take the product of all of the elements in values

shape_tensor(*tensor: Any*) → Any

Get the shape of a tensor.

Parameters `tensor` – A tensor.

Returns The shape of the input tensor returned as another tensor.

shape_tuple(*tensor: Any*) → Tuple[*Optional[int], ...*]

Get the shape of a tensor as a tuple of integers.

Parameters `tensor` – A tensor.

Returns The shape of the input tensor returned as a tuple of ints.

sigmoid(*a: Any*) → Any

Compute sigmoid of input a

Parameters `a` (*Tensor*) – [description]

Returns [description]

Return type Tensor

sign(*tensor: Any*) → Any

Returns an elementwise tensor with entries $y[i] = 1, 0, -1$ where $\text{tensor}[i] > 0, == 0$, and < 0 respectively.

For complex input the behaviour of this function may depend on the backend. The PyTorch version is not implemented in this case.

Parameters `tensor` – The input tensor.

sin(*a: Any*) → Any

Return sin of *tensor*. :param tensor: A tensor.

Returns Tensor

sinh(*a: Any*) → Any

Return the sinh of a tensor a.

Parameters `a` (*Tensor*) – tensor in matrix form

Returns sinh of a

Return type Tensor

size(*a: Any*) → Any

Return the total number of elements in a in tensor form.

Parameters `a` (*Tensor*) – tensor

Returns the total number of elements in a

Return type Tensor

`sizen(a: Any) → int`

Return the total number of elements in tensor a, but in integer form.

Parameters `a` (*Tensor*) – tensor

Returns the total number of elements in tensor a

Return type int

`slice(tensor: Any, start_indices: Tuple[int, ...], slice_sizes: Tuple[int, ...]) → Any`

Obtains a slice of a tensor based on start_indices and slice_sizes.

Parameters

- `tensor` – A tensor.
- `start_indices` – Tuple of integers denoting start indices of slice.
- `slice_sizes` – Tuple of integers denoting size of slice along each axis.

`softmax(a: Sequence[Any], axis: Optional[int] = None) → Any`

Softmax function. Computes the function which rescales elements to the range [0,1] such that the elements along axis sum to 1.

$$\text{softmax}(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Parameters

- `a` (*Sequence*[*Tensor*]) – Tensor
- `axis` (*int, optional*) – A dimension along which Softmax will be computed , defaults to None for all axis sum.

Returns concatenated tensor

Return type Tensor

`solve(A: Any, b: Any, **kws: Any) → Any`

Solve the linear system Ax=b and return the solution x.

Parameters

- `A` (*Tensor*) – The multiplied matrix.
- `b` (*Tensor*) – The resulted matrix.

Returns The solution of the linear system.

Return type Tensor

`sparse_dense_matmul(sp_a: Any, b: Any) → Any`

A sparse matrix multiplies a dense matrix.

Parameters

- `sp_a` (*Tensor*) – a sparse matrix
- `b` (*Tensor*) – a dense matrix

Returns dense matrix

Return type Tensor

sparse_shape(*tensor*: Any) → Tuple[Optional[int], ...]

sqrt(*tensor*: Any) → Any

Take the square root (element wise) of a given tensor.

sqrtmh(*a*: Any) → Any

Return the sqrtm of a Hermitian matrix *a*.

Parameters *a* (Tensor) – tensor in matrix form

Returns sqrtm of *a*

Return type Tensor

stack(*a*: Sequence[Any], *axis*: int = 0) → Any

Concatenates a sequence of tensors *a* along a new dimension axis.

Parameters

- **a** (Sequence[*Tensor*]) – List of tensors in the same shape
- **axis** (int, optional) – the stack axis, defaults to 0

Returns concatenated tensor

Return type Tensor

stateful_randc(*g*: Any, *a*: Union[int, Sequence[int], Any], *shape*: Union[int, Sequence[int]], *p*:

Optional[Union[Sequence[float], Any]] = None) → Any

[summary]

Parameters

- **g** (Any) – [description]
- **a** (Union[int, Sequence[int], Tensor]) – The possible options
- **shape** (Union[int, Sequence[int]]) – Sampling output shape
- **p** (Optional[Union[Sequence[float], Tensor]], optional) – probability for each option in *a*, defaults to None, as equal probability distribution

Returns [description]

Return type Tensor

stateful_randn(*g*: Any, *shape*: Union[int, Sequence[int]] = 1, *mean*: float = 0, *stddev*: float = 1, *dtype*: str = '32') → Any

[summary]

Parameters

- **self** (Any) – [description]
- **g** (Any) – stateful register for each package
- **shape** (Union[int, Sequence[int]]) – shape of output sampling tensor
- **mean** (float, optional) – [description], defaults to 0
- **stddev** (float, optional) – [description], defaults to 1
- **dtype** (str, optional) – only real data type is supported, “32” or “64”, defaults to “32”

Returns [description]

Return type Tensor

stateful_randu(*g: Any, shape: Union[int, Sequence[int]] = 1, low: float = 0, high: float = 1, dtype: str = '32'*) → Any

Uniform random sampler from `low` to `high`.

Parameters

- **g** (*Any*) – stateful register for each package
- **shape** (*Union[int, Sequence[int]], optional*) – shape of output sampling tensor, defaults to 1
- **low** (*float, optional*) – [description], defaults to 0
- **high** (*float, optional*) – [description], defaults to 1
- **dtype** (*str, optional*) – only real data type is supported, “32” or “64”, defaults to “32”

Returns [description]

Return type Tensor

std(*a: Any, axis: Optional[Sequence[int]] = None, keepdims: bool = False*) → Any

Compute the standard deviation along the specified axis.

Parameters

- **a** (*Tensor*) – _description_
- **axis** (*Optional[Sequence[int]], optional*) – Axis or axes along which the standard deviation is computed, defaults to None, implying all axis
- **keepdims** (*bool, optional*) – If this is set to True, the axes which are reduced are left in the result as dimensions with size one, defaults to False

Returns _description_

Return type Tensor

stop_gradient(*a: Any*) → Any

Stop backpropagation from `a`.

Parameters **a** (*Tensor*) – [description]

Returns [description]

Return type Tensor

subtraction(*tensor1: Any, tensor2: Any*) → Any

Return the default subtraction of `tensor`. A backend can override such implementation. :param `tensor1`: A tensor. :param `tensor2`: A tensor.

Returns Tensor

sum(*tensor: Any, axis: Optional[Sequence[int]] = None, keepdims: bool = False*) → Any

Sum elements of `tensor` along the specified `axis`. Results in a new Tensor with the summed axis removed. :param `tensor`: An input tensor.

Returns

The result of performing the summation. The order of the tensor will be reduced by 1.

Return type tensor

svd(*tensor*: Any, *pivot_axis*: int = -1, *max_singular_values*: Optional[int] = None, *max_truncation_error*: Optional[float] = None, *relative*: Optional[bool] = False) → Tuple[Any, Any, Any]

Computes the singular value decomposition (SVD) of a tensor.

The SVD is performed by treating the tensor as a matrix, with an effective left (row) index resulting from combining the axes *tensor.shape[:pivot_axis]* and an effective right (column) index resulting from combining the axes *tensor.shape[pivot_axis:]*.

For example, if *tensor* had a shape (2, 3, 4, 5) and *pivot_axis* was 2, then *u* would have shape (2, 3, 6), *s* would have shape (6), and *vh* would have shape (6, 4, 5).

If *max_singular_values* is set to an integer, the SVD is truncated to keep at most this many singular values.

If *max_truncation_error* > 0, as many singular values will be truncated as possible, so that the truncation error (the norm of discarded singular values) is at most *max_truncation_error*. If *relative* is set *True* then *max_truncation_err* is understood relative to the largest singular value.

If both *max_singular_values* and *max_truncation_error* are specified, the number of retained singular values will be *min(max_singular_values, nsv_auto_trunc)*, where *nsv_auto_trunc* is the number of singular values that must be kept to maintain a truncation error smaller than *max_truncation_error*.

The output consists of three tensors *u*, *s*, *vh* such that: ````python`

*u[i1,...,iN, j] * s[j] * vh[j, k1,...,kM] == tensor[i1,...,iN, k1,...,kM]*

````` Note that the output ordering matches numpy.linalg.svd rather than tf.svd.

#### Parameters

- **tensor** – A tensor to be decomposed.
- **pivot\_axis** – Where to split the tensor’s axes before flattening into a matrix.
- **max\_singular\_values** – The number of singular values to keep, or *None* to keep them all.
- **max\_truncation\_error** – The maximum allowed truncation error or *None* to not do any truncation.
- **relative** – Multiply *max\_truncation\_err* with the largest singular value.

#### Returns

Left tensor factor. *s*: Vector of ordered singular values from largest to smallest. *vh*: Right tensor factor. *s\_rest*: Vector of discarded singular values (length zero if no truncation).

#### Return type u

**switch**(*index*: Any, *branches*: Sequence[Callable[[], Any]]) → Any  
*branches*[*index*]()

#### Parameters

- **index** (*Tensor*) – [description]
- **branches** (Sequence[Callable[[], *Tensor*]]) – [description]

#### Returns

 [description]

#### Return type

*Tensor*

**tan**(*a*: Any) → Any  
Return the tan of a tensor *a*.

**Parameters** *a* (*Tensor*) – tensor in matrix form

**Returns** tan of a

**Return type** Tensor

**tanh**(*a*: Any) → Any

Return the tanh of a tensor a.

**Parameters** **a** (Tensor) – tensor in matrix form

**Returns** tanh of a

**Return type** Tensor

**tensordot**(*a*: Any, *b*: Any, *axes*: Union[int, Sequence[Sequence[int]]]) → Any

Do a tensordot of tensors *a* and *b* over the given axes.

**Parameters**

- **a** – A tensor.
- **b** – Another tensor.
- **axes** – Two lists of integers. These values are the contraction axes.

**tile**(*a*: Any, *rep*: Any) → Any

Constructs a tensor by tiling a given tensor.

**Parameters**

- **a** (Tensor) – [description]
- **rep** (Tensor) – 1d tensor with length the same as the rank of a

**Returns** [description]

**Return type** Tensor

**to\_dense**(*sp\_a*: Any) → Any

Convert a sparse matrix to dense tensor.

**Parameters** **sp\_a** (Tensor) – a sparse matrix

**Returns** the resulted dense matrix

**Return type** Tensor

**to\_dlpark**(*a*: Any) → Any

Transform the tensor *a* as a dlpark capsule

**Parameters** **a** (Tensor) – \_description\_

**Returns** \_description\_

**Return type** Any

**trace**(*tensor*: Any, *offset*: int = 0, *axis1*: int = - 2, *axis2*: int = - 1) → Any

Return summed entries along diagonals.

If tensor is 2-D, the sum is over the diagonal of tensor with the given offset, i.e., the collection of elements of the form *a*[*i*, *i*+*offset*]. If *a* has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-array whose diagonal is summed.

In the PyTorch backend the trace is always over the main diagonal of the last two entries.

**Parameters**

- **tensor** – A tensor.

- **offset** – Offset of the diagonal from the main diagonal. This argument is not supported by the PyTorch backend and an error will be raised if they are specified.
- **axis1** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to first/second axis. These arguments are not supported by the PyTorch backend and an error will be raised if they are specified.
- **axis2** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to first/second axis. These arguments are not supported by the PyTorch backend and an error will be raised if they are specified.

**Returns** The batched summed diagonals.

**Return type** array\_of\_diagonals

**transpose**(*tensor*, *perm=None*) → Any

Transpose a tensor according to a given permutation. By default the axes are reversed. :param *tensor*: A tensor. :param *perm*: The permutation of the axes.

**Returns** The transposed tensor

**tree\_flatten**(*pytree*: Any) → Tuple[Any, Any]

Flatten python structure to 1D list

**Parameters** *pytree* (Any) – python structure to be flattened

**Returns** The 1D list of flattened structure and treedef which can be used for later unflatten

**Return type** Tuple[Any, Any]

**tree\_map**(*f*: Callable[[...], Any], \**pytrees*: Any) → Any

Return the new tree map with multiple arg function *f* through pytrees.

**Parameters**

- **f** (Callable[..., Any]) – The function
- **pytrees** (Any) – inputs as any python structure

**Raises** **NotImplementedError** – raise when neither tensorflow or jax is installed.

**Returns** The new tree map with the same structure but different values.

**Return type** Any

**tree\_unflatten**(*treedef*: Any, *leaves*: Any) → Any

Pack 1D list to pytree defined via *treedef*

**Parameters**

- **treedef** (Any) – Def of pytree structure, the second return from **tree\_flatten**
- **leaves** (Any) – the 1D list of flattened data structure

**Returns** Packed pytree

**Return type** Any

**unique\_with\_counts**(*a*: Any, \*\**kws*: Any) → Tuple[Any, Any]

Find the unique elements and their corresponding counts of the given tensor *a*.

**Parameters** *a* (Tensor) – [description]

**Returns** Unique elements, corresponding counts

**Return type** Tuple[Tensor, Tensor]

**value\_and\_grad**(*f*: Callable[...], Any], argnums: Union[int, Sequence[int]] = 0, has\_aux: bool = False)  
 $\rightarrow$  Callable[...], Tuple[Any, Any]]

Return the function which returns the value and grad of *f*.

### Example

```
>>> f = lambda x,y: x**2+2*y
>>> g = tc.backend.value_and_grad(f)
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
5, 2
>>> g = tc.backend.value_and_grad(f, argnums=(0,1))
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
5, [2, 2]
```

### Parameters

- **f** (Callable[..., Any]) – the function to be differentiated
- **argnums** (Union[int, Sequence[int]], optional) – the position of args in *f* that are to be differentiated, defaults to be 0

**Returns** the value and grad function of *f* with the same set of arguments as *f*

**Return type** Callable[..., Tuple[Any, Any]]

**vectorized\_value\_and\_grad**(*f*: Callable[...], Any], argnums: Union[int, Sequence[int]] = 0, vectorized\_argnums: Union[int, Sequence[int]] = 0, has\_aux: bool = False)  $\rightarrow$  Callable[...], Tuple[Any, Any]]

Return the VVAG function of *f*. The inputs for *f* is (args[0], args[1], args[2], ...), and the output of *f* is a scalar. Suppose VVAG(*f*) is a function with inputs in the form (vargs[0], args[1], args[2], ...), where vargs[0] has one extra dimension than args[0] in the first axis and consistent with args[0] in shape for remaining dimensions, i.e.  $\text{shape}(\text{vargs}[0]) = [\text{batch}] + \text{shape}(\text{args}[0])$ . (We only cover cases where **vectorized\_argnums** defaults to 0 here for demonstration). VVAG(*f*) returns a tuple as a value tensor with shape [batch, 1] and a gradient tuple with shape: ( $[\text{batch}] + \text{shape}(\text{args}[\text{argnum}])$  for argnum in **argnums**). The gradient for argnums=k is defined as

$$g^k = \frac{\partial \sum_{i \in \text{batch}} f(\text{vargs}[0][i], \text{args}[1], \dots)}{\partial \text{args}[k]}$$

Therefore, if **argnums**=0, the gradient is reduced to

$$g_i^0 = \frac{\partial f(\text{vargs}[0][i])}{\partial \text{vargs}[0][i]}$$

, which is specifically suitable for batched VQE optimization, where args[0] is the circuit parameters.

And if **argnums**=1, the gradient is like

$$g_i^1 = \frac{\partial \sum_j f(\text{vargs}[0][j], \text{args}[1])}{\partial \text{args}[1][i]}$$

, which is suitable for quantum machine learning scenarios, where *f* is the loss function, args[0] corresponds to the input data and args[1] corresponds to the weights in the QML model.

### Parameters

- **f** (Callable[..., Any]) – [description]
- **argnums** (Union[int, Sequence[int]], optional) – [description], defaults to 0

- **vectorized\_argnums** (*Union[int, Sequence[int]]*, defaults to 0) – the args to be vectorized, these arguments should share the same batch shape in the first dimension

**Returns** [description]

**Return type** Callable[..., Tuple[Any, Any]]

**vjp**(*f*: Callable[..., Any], *inputs*: Union[Any, Sequence[Any]], *v*: Union[Any, Sequence[Any]]) →

Tuple[Union[Any, Sequence[Any]], Union[Any, Sequence[Any]]]

Function that computes the dot product between a vector *v* and the Jacobian of the given function at the point given by the inputs. (reverse mode AD relevant) Strictly speaking, this function is value\_and\_vjp.

**Parameters**

- **f** (*Callable[..., Any]*) – the function to carry out vjp calculation
- **inputs** (*Union[Tensor, Sequence[Tensor]]*) – input for *f*
- **v** (*Union[Tensor, Sequence[Tensor]]*) – value vector or gradient from downstream in reverse mode AD the same shape as return of function *f*

**Returns** (*f(\*inputs*), vjp\_tensor), where vjp\_tensor is the same shape as inputs

**Return type** Tuple[Union[Tensor, Sequence[Tensor]], Union[Tensor, Sequence[Tensor]]]

**vmap**(*f*: Callable[..., Any], *vectorized\_argnums*: Optional[Union[int, Sequence[int]]] = None) → Any

Return the vectorized map or batched version of *f* on the first extra axis. The general interface supports *f* with multiple arguments and broadcast in the first dimension.

**Parameters**

- **f** (*Callable[..., Any]*) – function to be broadcasted.
- **vectorized\_argnums** (*Union[int, Sequence[int]]*, defaults to 0) – the args to be vectorized, these arguments should share the same batch shape in the first dimension

**Returns** vmap version of *f*

**Return type** Any

**vvag**(*f*: Callable[..., Any], *argnums*: Union[int, Sequence[int]] = 0, *vectorized\_argnums*: Union[int, Sequence[int]] = 0, *has\_aux*: bool = False) → Callable[..., Tuple[Any, Any]]

Return the VVAG function of *f*. The inputs for *f* is (*args*[0], *args*[1], *args*[2], ...), and the output of *f* is a scalar. Suppose VVAG(*f*) is a function with inputs in the form (*vargs*[0], *args*[1], *args*[2], ...), where *vargs*[0] has one extra dimension than *args*[0] in the first axis and consistent with *args*[0] in shape for remaining dimensions, i.e. *shape(vargs*[0]) = [batch] + *shape(args*[0]). (We only cover cases where *vectorized\_argnums* defaults to 0 here for demonstration). VVAG(*f*) returns a tuple as a value tensor with shape [batch, 1] and a gradient tuple with shape: ([batch]+*shape(args[argnum])* for *argnum* in *argnums*). The gradient for *argnums*=*k* is defined as

$$g^k = \frac{\partial \sum_{i \in \text{batch}} f(\text{vargs}[0][i], \text{args}[1], \dots)}{\partial \text{args}[k]}$$

Therefore, if *argnums*=0, the gradient is reduced to

$$g_i^0 = \frac{\partial f(\text{vargs}[0][i])}{\partial \text{vargs}[0][i]}$$

, which is specifically suitable for batched VQE optimization, where *args*[0] is the circuit parameters.

And if argnums=1, the gradient is like

$$g_i^1 = \frac{\partial \sum_j f(vargs[0][j], args[1])}{\partial args[1][i]}$$

, which is suitable for quantum machine learning scenarios, where  $f$  is the loss function, args[0] corresponds to the input data and args[1] corresponds to the weights in the QML model.

#### Parameters

- **f** (*Callable[..., Any]*) – [description]
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – [description], defaults to 0
- **vectorized\_argnums** (*Union[int, Sequence[int]]*, *defaults to 0*) – the args to be vectorized, these arguments should share the same batch shape in the first dimension

#### Returns

[description]

**Return type** *Callable[..., Tuple[Any, Any]]*

**zeros**(*shape: Tuple[int, ...]*, *dtype: Optional[str] = None*) → *Any*

Return a zeros-matrix of dimension *dim* Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends).

Block-sparse behavior is currently not supported :param *shape*: The dimension of the returned matrix.  
:type *shape*: int :param *dtype*: The dtype of the returned matrix.

**class** `tensorcircuit.backends.pytorch_backend.torch_optimizer`(*optimizer: Any*)

Bases: `object`

**\_\_init\_\_**(*optimizer: Any*) → *None*

**update**(*grads: Any, params: Any*) → *Any*

`tensorcircuit.backends.pytorch_backend.torchlib: Any`

## `tensorcircuit.backends.tensorflow_backend`

Backend magic inherited from tensornetwork: tensorflow backend

**class** `tensorcircuit.backends.tensorflow_backend.TensorFlowBackend`

Bases: `tensornetwork.backends.tensorflow.tensorflow_backend.TensorFlowBackend, tensorcircuit.backends.abstract_backend.ExtendedBackend`

See the original backend API at `tensorflow backend`

**\_\_init\_\_**() → *None*

**abs**(*a: Any*) → *Any*

Returns the elementwise absolute value of tensor. :param *tensor*: An input tensor.

**Returns** Its elementwise absolute value.

**Return type** `tensor`

**acos**(*a: Any*) → *Any*

Return the acos of a tensor *a*.

**Parameters** *a* (`Tensor`) – tensor in matrix form

**Returns** *acos* of *a*

**Return type** Tensor

**acosh**(*a*: Any) → Any

Return the acosh of a tensor *a*.

**Parameters** **a** (Tensor) – tensor in matrix form

**Returns** acosh of *a*

**Return type** Tensor

**addition**(*tensor1*: Any, *tensor2*: Any) → Any

Return the default addition of *tensor*. A backend can override such implementation. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

**Returns** Tensor

**adjoint**(*a*: Any) → Any

Return the conjugate and transpose of a tensor *a*

**Parameters** **a** (Tensor) – Input tensor

**Returns** adjoint tensor of *a*

**Return type** Tensor

**arange**(*start*: int, *stop*: Optional[int] = None, *step*: int = 1) → Any

Values are generated within the half-open interval [start, stop)

**Parameters**

- **start** (int) – start index
- **stop** (Optional[int], optional) – end index, defaults to None
- **step** (Optional[int], optional) – steps, defaults to 1

**Returns** \_description\_

**Return type** Tensor

**argmax**(*a*: Any, *axis*: int = 0) → Any

Return the index of maximum of an array an axis.

**Parameters**

- **a** (Tensor) – [description]
- **axis** (int) – [description], defaults to 0, different behavior from numpy defaults!

**Returns** [description]

**Return type** Tensor

**argmin**(*a*: Any, *axis*: int = 0) → Any

Return the index of minimum of an array an axis.

**Parameters**

- **a** (Tensor) – [description]
- **axis** (int) – [description], defaults to 0, different behavior from numpy defaults!

**Returns** [description]

**Return type** Tensor

**asin**(*a*: Any) → Any

Return the acos of a tensor *a*.

**Parameters** `a` (`Tensor`) – tensor in matrix form

**Returns** asin of `a`

**Return type** Tensor

`asinh(a: Any) → Any`

Return the asinh of a tensor `a`.

**Parameters** `a` (`Tensor`) – tensor in matrix form

**Returns** asinh of `a`

**Return type** Tensor

`atan(a: Any) → Any`

Return the atan of a tensor `a`.

**Parameters** `a` (`Tensor`) – tensor in matrix form

**Returns** atan of `a`

**Return type** Tensor

`atan2(y: Any, x: Any) → Any`

Return the atan of a tensor `y/x`.

**Parameters** `a` (`Tensor`) – tensor in matrix form

**Returns** atan2 of `a`

**Return type** Tensor

`atanh(a: Any) → Any`

Return the atanh of a tensor `a`.

**Parameters** `a` (`Tensor`) – tensor in matrix form

**Returns** atanh of `a`

**Return type** Tensor

`broadcast_left_multiplication(tensor1: Any, tensor2: Any) → Any`

Perform broadcasting for multiplication of `tensor1` onto `tensor2`, i.e. `tensor1 * tensor2``, where `tensor2` is an arbitrary tensor and `tensor1` is a one-dimensional tensor. The broadcasting is applied to the first index of `tensor2`. :param `tensor1`: A tensor. :param `tensor2`: A tensor.

**Returns** The result of multiplying `tensor1` onto `tensor2`.

**Return type** Tensor

`broadcast_right_multiplication(tensor1: Any, tensor2: Any) → Any`

Perform broadcasting for multiplication of `tensor2` onto `tensor1`, i.e. `tensor1 * tensor2``, where `tensor1` is an arbitrary tensor and `tensor2` is a one-dimensional tensor. The broadcasting is applied to the last index of `tensor1`. :param `tensor1`: A tensor. :param `tensor2`: A tensor.

**Returns** The result of multiplying `tensor1` onto `tensor2`.

**Return type** Tensor

`cast(a: Any, dtype: str) → Any`

Cast the tensor `dtype` of a `a`.

**Parameters**

- `a` (`Tensor`) – tensor

- `dtype` (`str`) – “float32”, “float64”, “complex64”, “complex128”

**Returns** a of new dtype

**Return type** Tensor

**cholesky**(*tensor*: Any, *pivot\_axis*: int = -1, *non\_negative\_diagonal*: bool = False) → Tuple[Any, Any]

**concat**(*a*: Sequence[Any], *axis*: int = 0) → Any

Join a sequence of arrays along an existing axis.

**Parameters**

- **a** (Sequence[*Tensor*]) – [description]
- **axis** (int, optional) – [description], defaults to 0

**cond**(*pred*: bool, *true\_fun*: Callable[[], Any], *false\_fun*: Callable[[], Any]) → Any

The native cond for XLA compiling, wrapper for `tf.cond` and limited functionality of `jax.lax.cond`.

**Parameters**

- **pred** (bool) – [description]
- **true\_fun** (Callable[[], *Tensor*]) – [description]
- **false\_fun** (Callable[[], *Tensor*]) – [description]

**Returns** [description]

**Return type** Tensor

**conj**(*tensor*: Any) → Any

Return the complex conjugate of *tensor* :param tensor: A tensor.

**Returns** Tensor

**convert\_to\_tensor**(*tensor*: Any) → Any

Convert a np.array or a tensor to a tensor type for the backend.

**coo\_sparse\_matrix**(*indices*: Any, *values*: Any, *shape*: Any) → Any

Generate the coo format sparse matrix from indices and values, which is the only sparse format supported in different ML backends.

**Parameters**

- **indices** (*Tensor*) – shape [n, 2] for n non zero values in the returned matrix
- **values** (*Tensor*) – shape [n]
- **shape** (*Tensor*) – Tuple[int, ...]

**Returns** [description]

**Return type** Tensor

**coo\_sparse\_matrix\_from\_numpy**(*a*: Any) → Any

Generate the coo format sparse matrix from scipy coo sparse matrix.

**Parameters** **a** (*Tensor*) – Scipy coo format sparse matrix

**Returns** SparseTensor in backend format

**Return type** Tensor

**copy**(*a*: Any) → Any

Return the copy of a, matrix exponential.

**Parameters** **a** (*Tensor*) – tensor in matrix form

**Returns** matrix exponential of matrix a

**Return type** Tensor

**cos**(*a*: Any) → Any

Return cos of *tensor*. :param tensor: A tensor.

**Returns** Tensor

**cosh**(*a*: Any) → Any

Return the cosh of a tensor *a*.

**Parameters** *a* (*Tensor*) – tensor in matrix form

**Returns** cosh of *a*

**Return type** Tensor

**cumsum**(*a*: Any, *axis*: *Optional[int] = None*) → Any

Return the cumulative sum of the elements along a given axis.

**Parameters**

- *a* (*Tensor*) – [description]

- **axis** (*Optional[int], optional*) – The default behavior is the same as numpy, different from tf/torch as cumsum of the flatten 1D array, defaults to None

**Returns** [description]

**Return type** Tensor

**deserialize\_tensor**(*s*: str) → Any

Return a tensor given a serialized tensor string.

**Parameters** *s* – The input string representing a serialized tensor.

**Returns** The tensor object represented by the string.

**device**(*a*: Any) → str

get the universal device str for the tensor, in the format of tf

**Parameters** *a* (*Tensor*) – the tensor

**Returns** device str where the tensor lives on

**Return type** str

**device\_move**(*a*: Any, *dev*: Any) → Any

move tensor *a* to device *dev*

**Parameters**

- *a* (*Tensor*) – the tensor

- **dev** (Any) – device str or device obj in corresponding backend

**Returns** the tensor on new device

**Return type** Tensor

**diagflat**(*tensor*: Any, *k*: int = 0) → Any

Flattens tensor and creates a new matrix of zeros with its elements on the *k*'th diagonal. :param tensor: A tensor. :param k: The diagonal upon which to place its elements.

**Returns** A new tensor with all zeros save the specified diagonal.

**Return type** tensor

**diagonal**(*tensor*: Any, *offset*: int = 0, *axis1*: int = - 2, *axis2*: int = - 1) → Any

Return specified diagonals.

If tensor is 2-D, returns the diagonal of tensor with the given offset, i.e., the collection of elements of the form  $a[i, i+offest]$ . If a has more than two dimensions, then the axes specified by axis1 and axis2 are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing axis1 and axis2 and appending an index to the right equal to the size of the resulting diagonals.

This function only extracts diagonals. If you wish to create diagonal matrices from vectors, use diagflat.

#### Parameters

- **tensor** – A tensor.
- **offset** – Offset of the diagonal from the main diagonal.
- **axis1** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second-last and last axis (note this differs from the NumPy defaults).

These arguments are not supported in the TensorFlow backend and an error will be raised if they are specified.

- **axis2** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to second-last and last axis (note this differs from the NumPy defaults).

These arguments are not supported in the TensorFlow backend and an error will be raised if they are specified.

#### Returns

**A dim = min(1, tensor.ndim - 2) tensor storing** the batched diagonals.

**Return type** array\_of\_diagonals

**divide**(*tensor1*: Any, *tensor2*: Any) → Any

Return the default divide of *tensor*. A backend can override such implementation. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

#### Returns Tensor

**dtype**(*a*: Any) → str

Obtain dtype string for tensor *a*

**Parameters** *a* (*Tensor*) – The tensor

**Returns** dtype str, such as “complex64”

**Return type** str

**eigh**(*matrix*: Any) → Tuple[Any, Any]

Compute eigenvectors and eigenvalues of a hermitian matrix.

**Parameters** *matrix* – A symmetric matrix.

**Returns** The eigenvalues in ascending order. Tensor: The eigenvectors.

**Return type** Tensor

**eigs**(*A*: *Callable*, *args*: *Optional[List[Any]]* = *None*, *initial\_state*: *Optional[Any]* = *None*, *shape*: *Optional[Tuple[int, ...]]* = *None*, *dtype*: *Optional[Type[numpy.number]]* = *None*, *num\_krylov\_vecs*: *int* = 50, *numeig*: *int* = 1, *tol*: *float* = 1e-08, *which*: *str* = 'LR', *maxiter*: *Optional[int]* = *None*) → *Tuple[Any, List]*

Arnoldi method for finding the lowest eigenvector-eigenvalue pairs of a linear operator *A*. *A* is a callable implementing the matrix-vector product. If no *initial\_state* is provided then *shape* and *dtype* have to be passed so that a suitable initial state can be randomly generated. :param *A*: A (sparse) implementation of a linear operator :param *args*: A list of arguments to *A*. *A* will be called as

*res* = *A*(*initial\_state*, \**args*).

### Parameters

- **initial\_state** – An initial vector for the algorithm. If *None*, a random initial *Tensor* is created using the *numpy.random.randn* method.
- **shape** – The shape of the input-dimension of *A*.
- **dtype** – The dtype of the input *A*. If both no *initial\_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.
- **num\_krylov\_vecs** – The number of iterations (number of krylov vectors).
- **numeig** – The nummber of eigenvector-eigenvalue pairs to be computed. If *numeig* > 1, *reorthogonalize* has to be *True*.
- **tol** – The desired precision of the eigenvalues. Uses  
'LM' : largest magnitude 'SM' : smallest magnitude 'LR' : largest real part 'SR'  
: smallest real part 'LI' : largest imaginary part 'SI' : smallest imaginary part
- Note that not all of those might be supported by specialized backends.
- **maxiter** – The maximum number of iterations.

**Returns** An array of *numeig* lowest eigenvalues *list*: A list of *numeig* lowest eigenvectors

**Return type** *Tensor*

**eigsh**(*A*: *Callable*, *args*: *Optional[List[Any]]* = *None*, *initial\_state*: *Optional[Any]* = *None*, *shape*: *Optional[Tuple[int, ...]]* = *None*, *dtype*: *Optional[Type[numpy.number]]* = *None*, *num\_krylov\_vecs*: *int* = 50, *numeig*: *int* = 1, *tol*: *float* = 1e-08, *which*: *str* = 'LR', *maxiter*: *Optional[int]* = *None*) → *Tuple[Any, List]*

Lanczos method for finding the lowest eigenvector-eigenvalue pairs of a symmetric (hermitian) linear operator *A*. *A* is a callable implementing the matrix-vector product. If no *initial\_state* is provided then *shape* and *dtype* have to be passed so that a suitable initial state can be randomly generated. :param *A*: A (sparse) implementation of a linear operator :param *args*: A list of arguments to *A*. *A* will be called as

*res* = *A*(*initial\_state*, \**args*).

### Parameters

- **initial\_state** – An initial vector for the algorithm. If *None*, a random initial *Tensor* is created using the *numpy.random.randn* method.
- **shape** – The shape of the input-dimension of *A*.
- **dtype** – The dtype of the input *A*. If both no *initial\_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.

- **num\_krylov\_vecs** – The number of iterations (number of krylov vectors).
- **numeig** – The nummber of eigenvector-eigenvalue pairs to be computed. If  $numeig > 1$ , `reorthogonalize` has to be `True`.
- **tol** – The desired precision of the eigenvalus. Uses
- **which** – ['LM' | 'SM' | 'LR' | 'SR' | 'LI' | 'SI'] Which  $k$  eigenvectors and eigenvalues to find:
  - 'LM' : largest magnitude
  - 'SM' : smallest magnitude
  - 'LR' : largest real part
  - 'SR' : smallest real part
  - 'LI' : largest imaginary part
  - 'SI' : smallest imaginary part

Note that not all of those might be supported by specialized backends.

- **maxiter** – The maximum number of iterations.

**Returns** An array of  $numeig$  lowest eigenvalues *list*: A list of  $numeig$  lowest eigenvectors

**Return type** *Tensor*

**eigsh\_lanczos**(*A*: *Callable*, *args*: *Optional[List[Any]]* = *None*, *initial\_state*: *Optional[Any]* = *None*, *shape*: *Optional[Tuple[int, ...]]* = *None*, *dtype*: *Optional[Type[numpy.number]]* = *None*, *num\_krylov\_vecs*: *int* = 20, *numeig*: *int* = 1, *tol*: *float* = 1e-08, *delta*: *float* = 1e-08, *ndiag*: *int* = 20, *reorthogonalize*: *bool* = *False*) → *Tuple[Any, List]*

Lanczos method for finding the lowest eigenvector-eigenvalue pairs of *A*. :param *A*: A (sparse) implementation of a linear operator.

Call signature of *A* is *res* = *A*(*vector*, \**args*), where *vector* can be an arbitrary *Tensor*, and *res.shape* has to be *vector.shape*.

### Parameters

- **args** – A list of arguments to *A*. *A* will be called as *res* = *A*(*initial\_state*, \**args*).
- **initial\_state** – An initial vector for the Lanczos algorithm. If *None*, a random initial *Tensor* is created using the *backend.randn* method
- **shape** – The shape of the input-dimension of *A*.
- **dtype** – The dtype of the input *A*. If both no *initial\_state* is provided, a random initial state with shape *shape* and dtype *dtype* is created.
- **num\_krylov\_vecs** – The number of iterations (number of krylov vectors).
- **numeig** – The nummber of eigenvector-eigenvalue pairs to be computed. If  $numeig > 1$ , `reorthogonalize` has to be `True`.
- **tol** – The desired precision of the eigenvalus. Uses *backend.norm*(*eigvalsnew*[0:*numeig*] - *eigvalsold*[0:*numeig*]) < *tol* as stopping criterion between two diagonalization steps of the tridiagonal operator.
- **delta** – Stopping criterion for Lanczos iteration. If a Krylov vector :math: x\_n has an L2 norm  $\|x_n\| < \delta$ , the iteration is stopped. It means that an (approximate) invariant subspace has been found.
- **ndiag** – The tridiagonal Operator is diagonalized every *ndiag* iterations to check convergence.
- **reorthogonalize** – If `True`, Krylov vectors are kept orthogonal by explicit orthogonalization (more costly than `reorthogonalize=False`)

**Returns**

**(eigvals, eigvecs)** eigvals: A list of *numeig* lowest eigenvalues eigvecs: A list of *numeig* lowest eigenvectors

**eigvalsh**(*a*: Any) → Any

Get the eigenvalues of matrix *a*.

**Parameters** **a** (*Tensor*) – tensor in matrix form

**Returns** eigenvalues of *a*

**Return type** Tensor

**einsum**(*expression*: str, \**tensors*: Any, *optimize*: bool = True) → Any

Calculate sum of products of tensors according to expression.

**eps**(*dtype*: Type[numpy.number]) → float

Return machine epsilon for given *dtype*

**Parameters** **dtype** – A *dtype*.

**Returns** Machine epsilon.

**Return type** float

**exp**(*tensor*: Any) → Any

Return elementwise exp of *tensor*. :param *tensor*: A tensor.

**Returns** Tensor

**expm**(*a*: Any) → Any

Return expm log of *matrix*, matrix exponential. :param *matrix*: A tensor.

**Returns** Tensor

**eye**(*N*: int, *dtype*: Optional[str] = None, *M*: Optional[int] = None) → Any

**Return an identity matrix of dimension dim** Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported

**Parameters**

- **N** (int) – The dimension of the returned matrix.
- **dtype** – The *dtype* of the returned matrix.
- **M** (int) – The dimension of the returned matrix.

**from\_dlpack**(*a*: Any) → Any

Transform a dlpack capsule to a tensor

**Parameters** **a** (Any) – the dlpack capsule

**Returns** \_description\_

**Return type** Tensor

**gather1d**(*operand*: Any, *indices*: Any) → Any

Return *operand*[*indices*], both *operand* and *indices* are rank-1 tensor.

**Parameters**

- **operand** (*Tensor*) – rank-1 tensor
- **indices** (*Tensor*) – rank-1 tensor with int *dtype*

**Returns** `operand[indices]`

**Return type** `Tensor`

**get\_random\_state**(`seed: Optional[int] = None`) → Any

Get the backend specific random state object.

**Parameters** `seed (Optional[int], optional)` – [description], defaults to be `None`

:return:the backend specific random state object :rtype: Any

**gmres**(`A_mv: Callable, b: Any, A_args: Optional[List] = None, A_kwargs: Optional[dict] = None, x0: Optional[Any] = None, tol: float = 1e-05, atol: Optional[float] = None, num_krylov_vectors: int = 20, maxiter: Optional[int] = 1, M: Optional[Callable] = None`) → Tuple[Any, int]

GMRES solves the linear system  $A @ x = b$  for  $x$  given a vector  $b$  and a general (not necessarily symmetric/Hermitian) linear operator  $A$ .

As a Krylov method, GMRES does not require a concrete matrix representation of the  $n$  by  $n$   $A$ , but only a function `vector1 = A_mv(vector0, *A_args, **A_kwargs)` prescribing a one-to-one linear map from `vector0` to `vector1` (that is,  $A$  must be square, and thus `vector0` and `vector1` the same size). If  $A$  is a dense matrix, or if it is a symmetric/Hermitian operator, a different linear solver will usually be preferable.

GMRES works by first constructing the Krylov basis  $K = (x_0, A_mv @ x_0, A_mv @ A_mv @ x_0, \dots, (A_mv^{num\_krylov\_vectors}) @ x_0)$  and then solving a certain dense linear system  $K @ q_0 = q_1$  from whose solution  $x$  can be approximated. For `num_krylov_vectors = n` the solution is provably exact in infinite precision, but the expense is cubic in `num_krylov_vectors` so one is typically interested in the `num_krylov_vectors << n` case. The solution can in this case be repeatedly improved, to a point, by restarting the Arnoldi iterations each time `num_krylov_vectors` is reached. Unfortunately the optimal parameter choices balancing expense and accuracy are difficult to predict in advance, so applying this function requires a degree of experimentation.

In a tensor network code one is typically interested in `A_mv` implementing some tensor contraction. This implementation thus allows `b` and `x0` to be of whatever arbitrary, though identical, shape `b = A_mv(x0, ...)` expects. Reshaping to and from a matrix problem is handled internally.

#### Parameters

- **A\_mv** – A function `v0 = A_mv(v, *A_args, **A_kwargs)` where `v0` and `v` have the same shape.
- **b** – The  $b$  in  $A @ x = b$ ; it should be of the shape `A_mv` operates on.
- **A\_args** – Positional arguments to `A_mv`, supplied to this interface as a list. Default: `None`.
- **A\_kwargs** – Keyword arguments to `A_mv`, supplied to this interface as a dictionary. Default: `None`.
- **x0** – An optional guess solution. Zeros are used by default. If `x0` is supplied, its shape and `dtype` must match those of `b`, or an error will be thrown. Default: `zeros`.
- **tol** – Solution tolerance to achieve,  $\text{norm}(\text{residual}) \leq \max(\text{tol} * \text{norm}(b), \text{atol})$ . Default: `tol=1E-05`  
atol=tol
- **atol** – Solution tolerance to achieve,  $\text{norm}(\text{residual}) \leq \max(\text{tol} * \text{norm}(b), \text{atol})$ . Default: `tol=1E-05`  
atol=tol
- **num\_krylov\_vectors** –

- : **Size of the Krylov space to build at each restart.** Expense is cubic in this parameter. It must be positive. If greater than `b.size`, it will be set to `b.size`. Default: 20
- **maxiter** – The Krylov space will be repeatedly rebuilt up to this many times. Large values of this argument should be used only with caution, since especially for nearly symmetric matrices and small `num_krylov_vectors` convergence might well freeze at a value significantly larger than `tol`. Default: 1.
- **M** – Inverse of the preconditioner of A; see the docstring for `scipy.sparse.linalg.gmres`. This is only supported in the numpy backend. Supplying this argument to other backends will trigger `NotImplementedError`. Default: None.

**Raises ValueError** – -if `x0` is supplied but its shape differs from that of `b`. -in NumPy, if the ARPACK solver reports a breakdown (which usually indicates some kind of floating point issue). -if `num_krylov_vectors` is 0 or exceeds `b.size`. -if `tol` was negative. -if `M` was supplied with any backend but NumPy.

**Returns** The converged solution. It has the same shape as `b`. `info` : 0 if convergence was achieved, the number of restarts otherwise.

**Return type** `x`

**grad**(`f: Callable[..., Any], argnums: Union[int, Sequence[int]] = 0, has_aux: bool = False`) → `Callable[..., Any]`  
Return the function which is the grad function of input `f`.

**Example**

```
>>> f = lambda x,y: x**2+2*y
>>> g = tc.backend.grad(f)
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
2
>>> g = tc.backend.grad(f, argnums=(0,1))
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
[2, 2]
```

**Parameters**

- **f** (`Callable[..., Any]`) – the function to be differentiated
- **argnums** (`Union[int, Sequence[int]]`, optional) – the position of args in `f` that are to be differentiated, defaults to be 0

**Returns** the grad function of `f` with the same set of arguments as `f`

**Return type** `Callable[..., Any]`

**hessian**(`f: Callable[..., Any], argnums: Union[int, Sequence[int]] = 0`) → `Any`

**i**(`dtype: Optional[Any] = None`) → `Any`

Return `1.j` in as a tensor compatible with the backend.

**Parameters** `dtype` (`str`) – “complex64” or “complex128”

**Returns** `1.j` tensor

**Return type** `Tensor`

**imag**(`a: Any`) → `Any`

Return the elementwise imaginary value of a tensor `a`.

**Parameters** `a` (*Tensor*) – tensor

**Returns** imaginary value of `a`

**Return type** Tensor

**implicit\_randc**(`a: Union[int, Sequence[int], Any], shape: Union[int, Sequence[int]], p: Optional[Union[Sequence[float], Any]] = None`) → Any  
[summary]

**Parameters**

- `g` (*Any*) – [description]
- `a` (*Union[int, Sequence[int], Tensor]*) – The possible options
- `shape` (*Union[int, Sequence[int]]*) – Sampling output shape
- `p` (*Optional[Union[Sequence[float], Tensor]], optional*) – probability for each option in `a`, defaults to `None`, as equal probability distribution

**Returns** [description]

**Return type** Tensor

**implicit\_randn**(`shape: Union[int, Sequence[int]] = 1, mean: float = 0, stddev: float = 1, dtype: str = '32'`) → Any

Call the random normal function with the random state management behind the scene.

**Parameters**

- `shape` (*Union[int, Sequence[int]], optional*) – [description], defaults to 1
- `mean` (*float, optional*) – [description], defaults to 0
- `stddev` (*float, optional*) – [description], defaults to 1
- `dtype` (*str, optional*) – [description], defaults to “32”

**Returns** [description]

**Return type** Tensor

**implicit\_randu**(`shape: Union[int, Sequence[int]] = 1, low: float = 0, high: float = 1, dtype: str = '32'`) → Any

Call the random uniform function with the random state management behind the scene.

**Parameters**

- `shape` (*Union[int, Sequence[int]], optional*) – [description], defaults to 1
- `mean` (*float, optional*) – [description], defaults to 0
- `stddev` (*float, optional*) – [description], defaults to 1
- `dtype` (*str, optional*) – [description], defaults to “32”

**Returns** [description]

**Return type** Tensor

**index\_update**(`tensor: Any, mask: Any, assignee: Any`) → Any

Update `tensor` at elements defined by `mask` with value `assignee`.

**Parameters**

- `tensor` – A *Tensor* object.
- `mask` – A boolean mask.

- **assignee** – A scalar *Tensor*. The values to assigned to *tensor* at positions where *mask* is *True*.

**inv**(*matrix*: Any) → Any

Compute the matrix inverse of *matrix*.

**Parameters** **matrix** – A matrix.

**Returns** The inverse of *matrix*

**Return type** Tensor

**is\_sparse**(*a*: Any) → bool

Determine whether the type of input *a* is sparse.

**Parameters** **a** (*Tensor*) – input matrix *a*

**Returns** a bool indicating whether the matrix *a* is sparse

**Return type** bool

**is\_tensor**(*a*: Any) → bool

Return a boolean on whether *a* is a tensor in backend package.

**Parameters** **a** (*Tensor*) – a tensor to be determined

**Returns** whether *a* is a tensor

**Return type** bool

**item**(*tensor*)

Return the item of a 1-element tensor.

**Parameters** **tensor** – A 1-element tensor

**Returns** The value in tensor.

**jacbwD**(*f*: Callable[...], Any], *argnums*: Union[int, Sequence[int]] = 0) → Any

Compute the Jacobian of *f* using reverse mode AD.

**Parameters**

- **f** (*Callable[..., Any]*) – The function whose Jacobian is required
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – the position of the arg as Jacobian input, defaults to 0

**Returns** outer tuple for output, inner tuple for input args

**Return type** Tensor

**jacfwd**(*f*: Callable[...], Any], *argnums*: Union[int, Sequence[int]] = 0) → Any

Compute the Jacobian of *f* using the forward mode AD.

**Parameters**

- **f** (*Callable[..., Any]*) – the function whose Jacobian is required
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – the position of the arg as Jacobian input, defaults to 0

**Returns** outer tuple for input args, inner tuple for outputs

**Return type** Tensor

**jacrev**(*f*: Callable[...], Any], *argnums*: Union[int, Sequence[int]] = 0) → Any

Compute the Jacobian of *f* using reverse mode AD.

**Parameters**

- **f** (*Callable[..., Any]*) – The function whose Jacobian is required
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – the position of the arg as Jacobian input, defaults to 0

**Returns** outer tuple for output, inner tuple for input args

**Return type** Tensor

**jit**(*f*: *Callable[..., Any]*, *static\_argnums*: *Optional[Union[int, Sequence[int]]] = None*, *jit\_compile*: *Optional[bool] = None*) → Any

Return a jitted or graph-compiled version of *fun* for JAX backend. For all other backends returns *fun*.  
:param fun: Callable :param args: Arguments to *fun*. :param kwargs: Keyword arguments to *fun*.

**Returns** jitted/graph-compiled version of *fun*, or just *fun*.

**Return type** Callable

**jvp**(*f*: *Callable[..., Any]*, *inputs*: *Union[Any, Sequence[Any]]*, *v*: *Union[Any, Sequence[Any]]*) →

*Tuple[Union[Any, Sequence[Any]], Union[Any, Sequence[Any]]]*

Function that computes a (forward-mode) Jacobian-vector product of *f*. Strictly speaking, this function is value\_and\_jvp.

**Parameters**

- **f** (*Callable[..., Any]*) – The function to compute jvp
- **inputs** (*Union[Tensor, Sequence[Tensor]]*) – input for *f*
- **v** (*Union[Tensor, Sequence[Tensor]]*) – tangents

**Returns** (*f(\*inputs)*, *jvp\_tensor*), where *jvp\_tensor* is the same shape as the output of *f*

**Return type** *Tuple[Union[Tensor, Sequence[Tensor]], Union[Tensor, Sequence[Tensor]]]*

**kron**(*a*: *Any*, *b*: *Any*) → Any

Return the kronecker product of two matrices *a* and *b*.

**Parameters**

- **a** (*Tensor*) – tensor in matrix form
- **b** (*Tensor*) – tensor in matrix form

**Returns** kronecker product of *a* and *b*

**Return type** Tensor

**left\_shift**(*x*: *Any*, *y*: *Any*) → Any

Shift the bits of an integer *x* to the left *y* bits.

**Parameters**

- **x** (*Tensor*) – input values
- **y** (*Tensor*) – Number of bits shift to *x*

**Returns** result with the same shape as *x*

**Return type** Tensor

**log**(*tensor*: *Any*) → Any

Return elementwise natural logarithm of *tensor*. :param tensor: A tensor.

**Returns** Tensor

**matmul**(*tensor1*: Any, *tensor2*: Any) → Any

Perform a possibly batched matrix-matrix multiplication between *tensor1* and *tensor2*. The following behaviour is similar to *numpy.matmul*: - If both arguments are 2-D they are multiplied like conventional matrices.

- If either argument is N-D, N > 2, it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.

Both arguments to *matmul* have to be tensors of order  $\geq 2$ . :param *tensor1*: An input tensor. :param *tensor2*: An input tensor.

**Returns** The result of performing the matmul.

**Return type** tensor

**max**(*a*: Any, *axis*: Optional[int] = None) → Any

Return the maximum of an array or maximum along an axis.

**Parameters**

- **a** (Tensor) – [description]
- **axis** (Optional[int], optional) – [description], defaults to None

**Returns** [description]

**Return type** Tensor

**mean**(*a*: Any, *axis*: Optional[Sequence[int]] = None, *keepdims*: bool = False) → Any

Compute the arithmetic mean for *a* along the specified *axis*.

**Parameters**

- **a** (Tensor) – tensor to take average
- **axis** (Optional[Sequence[int]], optional) – the axis to take mean, defaults to None indicating sum over flatten array
- **keepdims** (bool, optional) – \_description\_, defaults to False

**Returns** \_description\_

**Return type** Tensor

**min**(*a*: Any, *axis*: Optional[int] = None) → Any

Return the minimum of an array or minimum along an axis.

**Parameters**

- **a** (Tensor) – [description]
- **axis** (Optional[int], optional) – [description], defaults to None

**Returns** [description]

**Return type** Tensor

**mod**(*x*: Any, *y*: Any) → Any

Compute *y*-mod of *x* (negative number behavior is not guaranteed to be consistent)

**Parameters**

- **x** (Tensor) – input values
- **y** (Tensor) – mod y

**Returns** results

**Return type** Tensor

**multiply**(*tensor1*: Any, *tensor2*: Any) → Any

Return the default multiplication of *tensor*.

A backend can override such implementation. :param *tensor1*: A tensor. :param *tensor2*: A tensor.

**Returns** Tensor

**norm**(*tensor*: Any) → Any

Calculate the L2-norm of the elements of *tensor*

**numpy**(*a*: Any) → Any

Return the numpy array of a tensor *a*, but may not work in a jitted function.

**Parameters** *a* (Tensor) – tensor in matrix form

**Returns** numpy array of *a*

**Return type** Tensor

**one\_hot**(*a*: Any, *num*: int) → Any

See doc for [onehot\(\)](#)

**onehot**(*a*: Any, *num*: int) → Any

One-hot encodes the given *a*. Each index in the input *a* is encoded as a vector of zeros of length *num* with the element at index set to one:

**Parameters**

- *a* (Tensor) – input tensor
- *num* (int) – number of features in onehot dimension

**Returns** onehot tensor with the last extra dimension

**Return type** Tensor

**ones**(*shape*: Tuple[int, ...], *dtype*: Optional[str] = None) → Any

Return an ones-matrix of dimension *dim* Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported :param *shape*: The dimension of the returned matrix. :type *shape*: int :param *dtype*: The dtype of the returned matrix.

**optimizer**

alias of [tensorcircuit.backends.tensorflow\\_backend.keras\\_optimizer](#)

**outer\_product**(*tensor1*: Any, *tensor2*: Any) → Any

Calculate the outer product of the two given tensors.

**pivot**(*tensor*: Any, *pivot\_axis*: int = -1) → Any

Reshapes a tensor into a matrix, whose columns (rows) are the vectorized dimensions to the left (right) of *pivot\_axis*.

In other words, with *tensor.shape* = (1, 2, 4, 5) and *pivot\_axis*=2, this function returns an (8, 5) matrix.

**Parameters**

- **tensor** – The tensor to pivot.
- **pivot\_axis** – The axis about which to pivot.

**Returns** The pivoted tensor.

**power**(*a*: Any, *b*: Union[Any, float]) → Any

**Returns the exponentiation of tensor a raised to b.**

**If b is a tensor, then the exponentiation is element-wise** between the two tensors, with *a* as the base and *b* as the power. Note that *a* and *b* must be broadcastable to the same shape if *b* is a tensor.

**If b is a scalar, then the exponentiation is each value in a raised to the power of b.**

#### Parameters

- **a** – The tensor containing the bases.
- **b** – The tensor containing the powers; or a single scalar as the power.

#### Returns

**The tensor that is each element of a raised to the power of b.** Note that the shape of the returned tensor is that produced by the broadcast of *a* and *b*.

**probability\_sample**(*shots*: int, *p*: Any, *status*: Optional[Any] = None, *g*: Optional[Any] = None) → Any

Drawn *shots* samples from probability distribution *p*, given the external randomness determined by uniform distributed *status* tensor or backend random generator *g*. This method is similar with *stateful\_randc*, but it supports *status* beyond *g*, which is convenient when *jit* or *vmap*

#### Parameters

- **shots (int)** – Number of samples to draw with replacement
- **p (Tensor)** – prbability vector
- **status (Optional[Tensor], optional)** – external randomness as a tensor with each element drawn uniformly from [0, 1], defaults to None
- **g (Any, optional)** – backend random generator, defaults to None

**Returns** The drawn sample as an int tensor

**Return type** Tensor

**qr**(*tensor*: Any, *pivot\_axis*: int = -1, *non\_negative\_diagonal*: bool = False) → Tuple[Any, Any]

Computes the QR decomposition of a tensor. The QR decomposition is performed by treating the tensor as a matrix, with an effective left (row) index resulting from combining the axes *tensor.shape[:pivot\_axis]* and an effective right (column) index resulting from combining the axes *tensor.shape[pivot\_axis:]*.

#### Example

If *tensor* had a shape (2, 3, 4, 5) and *pivot\_axis* was 2, then *q* would have shape (2, 3, 6), and *r* would have shape (6, 4, 5). The output consists of two tensors *Q*, *R* such that:

$$Q[i_1, \dots, i_N, j] * R[j, k_1, \dots, k_M] == tensor[i_1, \dots, i_N, k_1, \dots, k_M]$$

Note that the output ordering matches numpy.linalg.svd rather than tf.svd.

#### Parameters

- **tensor (Tensor)** – A tensor to be decomposed.
- **pivot\_axis (int, optional)** – Where to split the tensor's axes before flattening into a matrix.
- **non\_negative\_diagonal (bool, optional)** – a bool indicating whether the tenor is diagonal non-negative matrix.

**Returns** Q, the left tensor factor, and R, the right tensor factor.

**Return type** Tuple[Tensor, Tensor]

**randn**(*shape*: Tuple[int, ...], *dtype*: Optional[Type[numpy.number]] = None, *seed*: Optional[int] = None) → Any

Return a random-normal-matrix of dimension *dim* Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends).

Block-sparse behavior is currently not supported :param *shape*: The dimension of the returned matrix. :type *shape*: int :param *dtype*: The dtype of the returned matrix. :param *seed*: The seed for the random number generator

**random\_split**(*key*: Any) → Tuple[Any, Any]

A jax like split API, but it doesn't split the key generator for other backends. It is just for a consistent interface of random code; make sure you know what the function actually does. This function is mainly a utility to write backend agnostic code instead of doing magic things.

**Parameters** *key* (Any) – [description]

**Returns** [description]

**Return type** Tuple[Any, Any]

**random\_uniform**(*shape*: Tuple[int, ...], *boundaries*: Optional[Tuple[float, float]] = (0.0, 1.0), *dtype*: Optional[Type[numpy.number]] = None, *seed*: Optional[int] = None) → Any

Return a random uniform matrix of dimension *dim*.

Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends). Block-sparse behavior is currently not supported :param *shape*: The dimension of the returned matrix. :type *shape*: int :param *boundaries*: The boundaries of the uniform distribution. :type *boundaries*: tuple :param *dtype*: The dtype of the returned matrix. :param *seed*: The seed for the random number generator

**Returns** random uniform initialized tensor.

**Return type** Tensor

**real**(*a*: Any) → Any

Return the elementwise real value of a tensor *a*.

**Parameters** *a* (Tensor) – tensor

**Returns** real value of *a*

**Return type** Tensor

**relu**(*a*: Any) → Any

Rectified linear unit activation function. Computes the element-wise function:

$$\text{relu}(x) = \max(x, 0)$$

**Parameters** *a* (Tensor) – Input tensor

**Returns** Tensor after relu

**Return type** Tensor

**reshape**(*tensor*: Any, *shape*: Any) → Any

Reshape tensor to the given shape.

**Parameters** *tensor* – A tensor.

**Returns** The reshaped tensor.

**reshape2**(*a*: Any) → Any

Reshape a tensor to the [2, 2, ...] shape.

**Parameters** *a* (*Tensor*) – Input tensor

**Returns** the reshaped tensor

**Return type** Tensor

**reshapem**(*a*: Any) → Any

Reshape a tensor to the [1, 1] shape.

**Parameters** *a* (*Tensor*) – Input tensor

**Returns** the reshaped tensor

**Return type** Tensor

**reverse**(*a*: Any) → Any

return *a*[:::-1], only 1D tensor is guaranteed for consistent behavior

**Parameters** *a* (*Tensor*) – 1D tensor

**Returns** 1D tensor in reverse order

**Return type** Tensor

**right\_shift**(*x*: Any, *y*: Any) → Any

Shift the bits of an integer *x* to the right *y* bits.

**Parameters**

- **x** (*Tensor*) – input values
- **y** (*Tensor*) – Number of bits shift to *x*

**Returns** result with the same shape as *x*

**Return type** Tensor

**rq**(*tensor*: Any, *pivot\_axis*: int = 1, *non\_negative\_diagonal*: bool = False) → Tuple[Any, Any]

Computes the RQ decomposition of a tensor. The QR decomposition is performed by treating the tensor as a matrix, with an effective left (row) index resulting from combining the axes *tensor.shape[:pivot\_axis]* and an effective right (column) index resulting from combining the axes *tensor.shape[pivot\_axis:]*.

**Example**

If *tensor* had a shape (2, 3, 4, 5) and *pivot\_axis* was 2, then *r* would have shape (2, 3, 6), and *q* would have shape (6, 4, 5). The output consists of two tensors *Q*, *R* such that:

$$Q[i_1, \dots, i_N, j] * R[j, k_1, \dots, k_M] == tensor[i_1, \dots, i_N, k_1, \dots, k_M]$$

Note that the output ordering matches numpy.linalg.svd rather than tf.svd.

**Parameters**

- **tensor** (*Tensor*) – A tensor to be decomposed.
- **pivot\_axis** (*int, optional*) – Where to split the tensor's axes before flattening into a matrix.
- **non\_negative\_diagonal** (*bool, optional*) – a bool indicating whether the tensor is diagonal non-negative matrix.

**Returns** *Q*, the left tensor factor, and *R*, the right tensor factor.

**Return type** Tuple[*Tensor*, *Tensor*]

**scatter**(*operand*: Any, *indices*: Any, *updates*: Any) → AnyRoughly equivalent to *operand*[*indices*] = *updates*, *indices* only support shape with rank 2 for now.**Parameters**

- **operand** (*Tensor*) – [description]
- **indices** (*Tensor*) – [description]
- **updates** (*Tensor*) – [description]

**Returns** [description]**Return type** Tensor**searchsorted**(*a*: Any, *v*: Any, *side*: str = 'left') → Any

Find indices where elements should be inserted to maintain order.

**Parameters**

- **a** (*Tensor*) – input array sorted in ascending order
- **v** (*Tensor*) – value to inserted
- **side** (str, optional) – If ‘left’, the index of the first suitable location found is given. If ‘right’, return the last such index. If there is no suitable index, return either 0 or N (where N is the length of a), defaults to “left”

**Returns** Array of insertion points with the same shape as v, or an integer if v is a scalar.**Return type** Tensor**serialize\_tensor**(*tensor*: Any) → str

Return a string that serializes the given tensor.

**Parameters** **tensor** – The input tensor.**Returns** A string representing the serialized tensor.**set\_random\_state**(*seed*: Optional[Union[int, Any]] = None, *get\_only*: bool = False) → Any

Set the random state attached to the backend.

**Parameters**

- **seed** (Optional[int], optional) – the random seed, defaults to be None
- **get\_only** (bool, defaults to be False) – If set to be true, only get the random state in return instead of setting the state on the backend

**shape\_concat**(*values*: Any, *axis*: int) → Any

Concatenate a sequence of tensors together about the given axis.

**shape\_prod**(*values*: Any) → Any

Take the product of all of the elements in values

**shape\_tensor**(*tensor*: Any) → Any

Get the shape of a tensor.

**Parameters** **tensor** – A tensor.**Returns** The shape of the input tensor returned as another tensor.**shape\_tuple**(*tensor*: Any) → Tuple[Optional[int], ...]

Get the shape of a tensor as a tuple of integers.

**Parameters** **tensor** – A tensor.**Returns** The shape of the input tensor returned as a tuple of ints.

**sigmoid**(*a*: Any) → AnyCompute sigmoid of input *a***Parameters** *a* (Tensor) – [description]**Returns** [description]**Return type** Tensor**sign**(*tensor*: Any) → AnyReturns an elementwise tensor with entries  $y[i] = 1, 0, -1$  where  $\text{tensor}[i] > 0, == 0$ , and  $< 0$  respectively.For complex input the behaviour of this function may depend on the backend. The TensorFlow version returns  $y[i] = x[i] / \text{abs}(x[i])$ .**Parameters** *tensor* – The input tensor.**sin**(*a*: Any) → AnyReturn sin of *tensor*. :param *tensor*: A tensor.**Returns** Tensor**sinh**(*a*: Any) → AnyReturn the sinh of a tensor *a*.**Parameters** *a* (Tensor) – tensor in matrix form**Returns** sinh of *a***Return type** Tensor**size**(*a*: Any) → AnyReturn the total number of elements in *a* in tensor form.**Parameters** *a* (Tensor) – tensor**Returns** the total number of elements in *a***Return type** Tensor**sizen**(*a*: Any) → intReturn the total number of elements in tensor *a*, but in integer form.**Parameters** *a* (Tensor) – tensor**Returns** the total number of elements in tensor *a***Return type** int**slice**(*tensor*: Any, *start\_indices*: Tuple[int, ...], *slice\_sizes*: Tuple[int, ...]) → AnyObtains a slice of a tensor based on *start\_indices* and *slice\_sizes*.**Parameters**

- **tensor** – A tensor.
- **start\_indices** – Tuple of integers denoting start indices of slice.
- **slice\_sizes** – Tuple of integers denoting size of slice along each axis.

**softmax**(*a*: Sequence[Any], *axis*: Optional[int] = None) → Any

Softmax function. Computes the function which rescales elements to the range [0,1] such that the elements along axis sum to 1.

$$\text{softmax}(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

**Parameters**

- **a** (*Sequence[Tensor]*) – Tensor
- **axis** (*int, optional*) – A dimension along which Softmax will be computed , defaults to None for all axis sum.

**Returns** concatenated tensor**Return type** Tensor

**solve**(*A*: Any, *b*: Any, \*\**kws*: Any) → Any  
Solve the linear system Ax=b and return the solution x.

**Parameters**

- **A** (*Tensor*) – The multiplied matrix.
- **b** (*Tensor*) – The resulted matrix.

**Returns** The solution of the linear system.**Return type** Tensor

**sparse\_dense\_matmul**(*sp\_a*: Any, *b*: Any) → Any  
A sparse matrix multiplies a dense matrix.

**Parameters**

- **sp\_a** (*Tensor*) – a sparse matrix
- **b** (*Tensor*) – a dense matrix

**Returns** dense matrix**Return type** Tensor

**sparse\_shape**(*tensor*: Any) → Tuple[Optional[int], ...]

**sqrt**(*tensor*: Any) → Any  
Take the square root (element wise) of a given tensor.

**sqrtmh**(*a*: Any) → Any  
Return the sqrtm of a Hermitian matrix a.

**Parameters** **a** (*Tensor*) – tensor in matrix form**Returns** sqrtm of a**Return type** Tensor

**stack**(*a*: Sequence[Any], *axis*: int = 0) → Any  
Concatenates a sequence of tensors a along a new dimension axis.

**Parameters**

- **a** (*Sequence[Tensor]*) – List of tensors in the same shape
- **axis** (*int, optional*) – the stack axis, defaults to 0

**Returns** concatenated tensor**Return type** Tensor

**stateful\_randc**(*g*: Any, *a*: Union[int, Sequence[int], Any], *shape*: Union[int, Sequence[int]], *p*: Optional[Union[Sequence[float], Any]] = None) → Any  
[summary]

**Parameters**

- **g** (*Any*) – [description]
- **a** (*Union[int, Sequence[int], Tensor]*) – The possible options
- **shape** (*Union[int, Sequence[int]]*) – Sampling output shape
- **p** (*Optional[Union[Sequence[float], Tensor]]*, *optional*) – probability for each option in a, defaults to None, as equal probability distribution

**Returns** [description]

**Return type** Tensor

**stateful\_randn**(*g: Any, shape: Union[int, Sequence[int]] = 1, mean: float = 0, stddev: float = 1, dtype: str = '32'*) → Any  
[summary]

**Parameters**

- **self** (*Any*) – [description]
- **g** (*Any*) – stateful register for each package
- **shape** (*Union[int, Sequence[int]]*) – shape of output sampling tensor
- **mean** (*float, optional*) – [description], defaults to 0
- **stddev** (*float, optional*) – [description], defaults to 1
- **dtype** (*str, optional*) – only real data type is supported, “32” or “64”, defaults to “32”

**Returns** [description]

**Return type** Tensor

**stateful\_randu**(*g: Any, shape: Union[int, Sequence[int]] = 1, low: float = 0, high: float = 1, dtype: str = '32'*) → Any

Uniform random sampler from low to high.

**Parameters**

- **g** (*Any*) – stateful register for each package
- **shape** (*Union[int, Sequence[int]], optional*) – shape of output sampling tensor, defaults to 1
- **low** (*float, optional*) – [description], defaults to 0
- **high** (*float, optional*) – [description], defaults to 1
- **dtype** (*str, optional*) – only real data type is supported, “32” or “64”, defaults to “32”

**Returns** [description]

**Return type** Tensor

**std**(*a: Any, axis: Optional[Sequence[int]] = None, keepdims: bool = False*) → Any

Compute the standard deviation along the specified axis.

**Parameters**

- **a** (*Tensor*) – \_description\_
- **axis** (*Optional[Sequence[int]], optional*) – Axis or axes along which the standard deviation is computed, defaults to None, implying all axis

- **keepdims** (*bool, optional*) – If this is set to True, the axes which are reduced are left in the result as dimensions with size one, defaults to False

**Returns** \_description\_

**Return type** Tensor

**stop\_gradient**(*a: Any*) → Any

Stop backpropagation from *a*.

**Parameters** *a* (*Tensor*) – [description]

**Returns** [description]

**Return type** Tensor

**subtraction**(*tensor1: Any, tensor2: Any*) → Any

Return the default subtraction of *tensor*. A backend can override such implementation.  
:param tensor1: A tensor. :param tensor2: A tensor.

**Returns** Tensor

**sum**(*tensor: Any, axis: Optional[Sequence[int]] = None, keepdims: bool = False*) → Any

Sum elements of *tensor* along the specified *axis*. Results in a new Tensor with the summed axis removed.  
:param tensor: An input tensor.

**Returns**

The result of performing the summation. The order of the tensor will be reduced by  
1.

**Return type** tensor

**svd**(*tensor: Any, pivot\_axis: int = -1, max\_singular\_values: Optional[int] = None, max\_truncation\_error: Optional[float] = None, relative: Optional[bool] = False*) → Tuple[*Any, Any, Any, Any*]

Computes the singular value decomposition (SVD) of a tensor.

The SVD is performed by treating the tensor as a matrix, with an effective left (row) index resulting from combining the axes *tensor.shape[:pivot\_axis]* and an effective right (column) index resulting from combining the axes *tensor.shape[pivot\_axis:]*.

For example, if *tensor* had a shape (2, 3, 4, 5) and *pivot\_axis* was 2, then *u* would have shape (2, 3, 6), *s* would have shape (6), and *vh* would have shape (6, 4, 5).

If *max\_singular\_values* is set to an integer, the SVD is truncated to keep at most this many singular values.

If *max\_truncation\_error > 0*, as many singular values will be truncated as possible, so that the truncation error (the norm of discarded singular values) is at most *max\_truncation\_error*. If *relative* is set *True* then *max\_truncation\_err* is understood relative to the largest singular value.

If both *max\_singular\_values* and *max\_truncation\_error* are specified, the number of retained singular values will be *min(max\_singular\_values, nsv\_auto\_trunc)*, where *nsv\_auto\_trunc* is the number of singular values that must be kept to maintain a truncation error smaller than *max\_truncation\_error*.

The output consists of three tensors *u*, *s*, *vh* such that:

```python  
u[i1,...,iN, j] * s[j] * vh[j, k1,...,kM] == tensor[i1,...,iN, k1,...,kM]

``` Note that the output ordering matches numpy.linalg.svd rather than tf.svd.

**Parameters**

- **tensor** – A tensor to be decomposed.

- **pivot\_axis** – Where to split the tensor's axes before flattening into a matrix.

- **max\_singular\_values** – The number of singular values to keep, or *None* to keep them all.
- **max\_truncation\_error** – The maximum allowed truncation error or *None* to not do any truncation.
- **relative** – Multiply *max\_truncation\_err* with the largest singular value.

**Returns**

Left tensor factor. s: Vector of ordered singular values from largest to smallest. vh: Right tensor factor. s\_rest: Vector of discarded singular values (length zero if no truncation).

**Return type** u

**switch**(index: Any, branches: Sequence[Callable[], Any]) → Any  
branches[index]()

**Parameters**

- **index** (Tensor) – [description]
- **branches** (Sequence[Callable[], Tensor]) – [description]

**Returns** [description]**Return type** Tensor

**tan**(a: Any) → Any  
Return the tan of a tensor a.

**Parameters** a (Tensor) – tensor in matrix form**Returns** tan of a**Return type** Tensor

**tanh**(a: Any) → Any  
Return the tanh of a tensor a.

**Parameters** a (Tensor) – tensor in matrix form**Returns** tanh of a**Return type** Tensor

**tensordot**(a: Any, b: Any, axes: Union[int, Sequence[Sequence[int]]]) → Any  
Do a tensordot of tensors a and b over the given axes.

**Parameters**

- **a** – A tensor.
- **b** – Another tensor.
- **axes** – Two lists of integers. These values are the contraction axes.

**tile**(a: Any, rep: Any) → Any  
Constructs a tensor by tiling a given tensor.

**Parameters**

- **a** (Tensor) – [description]
- **rep** (Tensor) – 1d tensor with length the same as the rank of a

**Returns** [description]

**Return type** Tensor

**to\_dense**(*sp\_a*: Any) → Any

Convert a sparse matrix to dense tensor.

**Parameters** *sp\_a* (Tensor) – a sparse matrix

**Returns** the resulted dense matrix

**Return type** Tensor

**to\_dlpack**(*a*: Any) → Any

Transform the tensor *a* as a dlpack capsule

**Parameters** *a* (Tensor) – \_description\_

**Returns** \_description\_

**Return type** Any

**trace**(*tensor*: Any, *offset*: int = 0, *axis1*: int = - 2, *axis2*: int = - 1) → Any

Return summed entries along diagonals.

If tensor is 2-D, the sum is over the diagonal of tensor with the given offset, i.e., the collection of elements of the form *a*[*i*, *i*+*offset*]. If *a* has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-array whose diagonal is summed.

**Parameters**

- **tensor** – A tensor.
- **offset** – Offset of the diagonal from the main diagonal. This argument is not supported in the TensorFlow backend and an error will be raised if they are specified.
- **axis1** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to first/second axis. These arguments are not supported in the TensorFlow backend and an error will be raised if they are specified.
- **axis2** – Axis to be used as the first/second axis of the 2D sub-arrays from which the diagonals should be taken. Defaults to first/second axis. These arguments are not supported in the TensorFlow backend and an error will be raised if they are specified.

**Returns** The batched summed diagonals.

**Return type** array\_of\_diagonals

**transpose**(*tensor*, *perm*=None) → Any

Transpose a tensor according to a given permutation. By default the axes are reversed. :param *tensor*: A tensor. :param *perm*: The permutation of the axes.

**Returns** The transposed tensor

**tree\_flatten**(*pytree*: Any) → Tuple[Any, Any]

Flatten python structure to 1D list

**Parameters** *pytree* (Any) – python structure to be flattened

**Returns** The 1D list of flattened structure and treedef which can be used for later unflatten

**Return type** Tuple[Any, Any]

**tree\_map**(*f*: Callable[[], Any], \**pytrees*: Any) → Any

Return the new tree map with multiple arg function *f* through pytrees.

**Parameters**

- **f** (Callable[..., Any]) – The function

- **pytrees** (*Any*) – inputs as any python structure

**Raises** `NotImplementedError` – raise when neither tensorflow or jax is installed.

**Returns** The new tree map with the same structure but different values.

**Return type** *Any*

**tree\_unflatten**(*treedef*: *Any*, *leaves*: *Any*) → *Any*

Pack 1D list to pytree defined via *treedef*

**Parameters**

- **treedef** (*Any*) – Def of pytree structure, the second return from `tree_flatten`
- **leaves** (*Any*) – the 1D list of flattened data structure

**Returns** Packed pytree

**Return type** *Any*

**unique\_with\_counts**(*a*: *Any*, \*\**kws*: *Any*) → *Tuple[Any, Any]*

Find the unique elements and their corresponding counts of the given tensor *a*.

**Parameters** **a** (*Tensor*) – [description]

**Returns** Unique elements, corresponding counts

**Return type** *Tuple[Tensor, Tensor]*

**value\_and\_grad**(*f*: *Callable[..., Any]*, *argnums*: *Union[int, Sequence[int]]* = 0, *has\_aux*: *bool* = *False*)

→ *Callable[..., Tuple[Any, Any]]*

Return the function which returns the value and grad of *f*.

**Example**

```
>>> f = lambda x,y: x**2+2*y
>>> g = tc.backend.value_and_grad(f)
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
5, 2
>>> g = tc.backend.value_and_grad(f, argnums=(0,1))
>>> g(tc.num_to_tensor(1),tc.num_to_tensor(2))
5, [2, 2]
```

**Parameters**

- **f** (*Callable[..., Any]*) – the function to be differentiated
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – the position of args in *f* that are to be differentiated, defaults to be 0

**Returns** the value and grad function of *f* with the same set of arguments as *f*

**Return type** *Callable[..., Tuple[Any, Any]]*

**vectorized\_value\_and\_grad**(*f*: *Callable[..., Any]*, *argnums*: *Union[int, Sequence[int]]* = 0, *vectorized\_argnums*: *Union[int, Sequence[int]]* = 0, *has\_aux*: *bool* = *False*) → *Callable[..., Tuple[Any, Any]]*

Return the VVAG function of *f*. The inputs for *f* is (*args*[0], *args*[1], *args*[2], ...), and the output of *f* is a scalar. Suppose VVAG(*f*) is a function with inputs in the form (*vargs*[0], *args*[1], *args*[2], ...), where *vargs*[0] has one extra dimension than *args*[0] in the first axis and consistent with *args*[0] in shape for remaining dimensions, i.e.  $\text{shape}(\text{vargs}[0]) = [\text{batch}] + \text{shape}(\text{args}[0])$ . (We only cover cases where

`vectorized_argnums` defaults to 0 here for demonstration). `VVAG(f)` returns a tuple as a value tensor with shape [batch, 1] and a gradient tuple with shape: ([batch]+shape(args[argnum]) for argnum in `argnums`). The gradient for `argnums=k` is defined as

$$g^k = \frac{\partial \sum_{i \in \text{batch}} f(\text{vargs}[0][i], \text{args}[1], \dots)}{\partial \text{args}[k]}$$

Therefore, if `argnums=0`, the gradient is reduced to

$$g_i^0 = \frac{\partial f(\text{vargs}[0][i])}{\partial \text{vargs}[0][i]}$$

, which is specifically suitable for batched VQE optimization, where `args[0]` is the circuit parameters.

And if `argnums=1`, the gradient is like

$$g_i^1 = \frac{\partial \sum_j f(\text{vargs}[0][j], \text{args}[1])}{\partial \text{args}[1][i]}$$

, which is suitable for quantum machine learning scenarios, where `f` is the loss function, `args[0]` corresponds to the input data and `args[1]` corresponds to the weights in the QML model.

#### Parameters

- `f (Callable[..., Any])` – [description]
- `argnums (Union[int, Sequence[int]], optional)` – [description], defaults to 0
- `vectorized_argnums (Union[int, Sequence[int]], defaults to 0)` – the args to be vectorized, these arguments should share the same batch shape in the first dimension

#### Returns [description]

**Return type** Callable[..., Tuple[Any, Any]]

**vjp**(`f: Callable[..., Any], inputs: Union[Any, Sequence[Any]], v: Union[Any, Sequence[Any]]`) →

`Tuple[Union[Any, Sequence[Any]], Union[Any, Sequence[Any]]]`

Function that computes the dot product between a vector `v` and the Jacobian of the given function at the point given by the inputs. (reverse mode AD relevant) Strictly speaking, this function is `value_and_vjp`.

#### Parameters

- `f (Callable[..., Any])` – the function to carry out vjp calculation
- `inputs (Union[Tensor, Sequence[Tensor]])` – input for `f`
- `v (Union[Tensor, Sequence[Tensor]])` – value vector or gradient from downstream in reverse mode AD the same shape as return of function `f`

**Returns** (`f(*inputs), vjp_tensor`), where `vjp_tensor` is the same shape as `inputs`

**Return type** Tuple[Union[Tensor, Sequence[Tensor]], Union[Tensor, Sequence[Tensor]]]

**vmap**(`f: Callable[..., Any], vectorized_argnums: Union[int, Sequence[int]] = 0`) → Any

Return the vectorized map or batched version of `f` on the first extra axis. The general interface supports `f` with multiple arguments and broadcast in the first dimension.

#### Parameters

- `f (Callable[..., Any])` – function to be broadcasted.

- **vectorized\_argnums** (*Union[int, Sequence[int]]*, defaults to 0) – the args to be vectorized, these arguments should share the same batch shape in the first dimension

**Returns** vmap version of *f*

**Return type** Any

**vvag**(*f*: Callable[...], *Any*], *argnums*: Union[int, Sequence[int]] = 0, *vectorized\_argnums*: Union[int,

Sequence[int]] = 0, *has\_aux*: bool = False) → Callable[...], Tuple[Any, Any]]

Return the VVAG function of *f*. The inputs for *f* is (*args*[0], *args*[1], *args*[2], ...), and the output of *f* is a scalar. Suppose VVAG(*f*) is a function with inputs in the form (*vargs*[0], *args*[1], *args*[2], ...), where *vargs*[0] has one extra dimension than *args*[0] in the first axis and consistent with *args*[0] in shape for remaining dimensions, i.e.  $\text{shape}(\text{vargs}[0]) = [\text{batch}] + \text{shape}(\text{args}[0])$ . (We only cover cases where *vectorized\_argnums* defaults to 0 here for demonstration). VVAG(*f*) returns a tuple as a value tensor with shape [batch, 1] and a gradient tuple with shape: ([batch]+shape(*args*[*argnum*]) for *argnum* in *argnums*). The gradient for *argnums*=*k* is defined as

$$g^k = \frac{\partial \sum_{i \in \text{batch}} f(\text{vargs}[0][i], \text{args}[1], \dots)}{\partial \text{args}[k]}$$

Therefore, if *argnums*=0, the gradient is reduced to

$$g_i^0 = \frac{\partial f(\text{vargs}[0][i])}{\partial \text{vargs}[0][i]}$$

, which is specifically suitable for batched VQE optimization, where *args*[0] is the circuit parameters.

And if *argnums*=1, the gradient is like

$$g_i^1 = \frac{\partial \sum_j f(\text{vargs}[0][j], \text{args}[1])}{\partial \text{args}[1][i]}$$

, which is suitable for quantum machine learning scenarios, where *f* is the loss function, *args*[0] corresponds to the input data and *args*[1] corresponds to the weights in the QML model.

### Parameters

- **f** (Callable[..., Any]) – [description]
- **argnums** (*Union[int, Sequence[int]]*, optional) – [description], defaults to 0
- **vectorized\_argnums** (*Union[int, Sequence[int]]*, defaults to 0) – the args to be vectorized, these arguments should share the same batch shape in the first dimension

**Returns** [description]

**Return type** Callable[..., Tuple[Any, Any]]

**zeros**(*shape*: Tuple[int, ...], *dtype*: Optional[str] = None) → Any

Return a zeros-matrix of dimension *dim* Depending on specific backends, *dim* has to be either an int (numpy, torch, tensorflow) or a *ShapeType* object (for block-sparse backends).

Block-sparse behavior is currently not supported :param *shape*: The dimension of the returned matrix.  
:type *shape*: int :param *dtype*: The dtype of the returned matrix.

**class** tensorcircuit.backends.tensorflow\_backend.**keras\_optimizer**(*optimizer*: Any)

Bases: object

**\_\_init\_\_**(*optimizer*: Any) → None

**update**(*grads*: Any, *params*: Any) → Any

#### 4.1.4 tensorcircuit.basecircuit

Quantum circuit: common methods for all circuit classes as MixIn

```
class tensorcircuit.basecircuit.BaseCircuit
 Bases: tensorcircuit.abstractcircuit.AbstractCircuit
static all_zero_nodes(n: int, d: int = 2, prefix: str = 'qb-') →
 List[tensornetwork.network_components.Node]
```

**amplitude**(l: Union[str, Any]) → Any

Returns the amplitude of the circuit given the bitstring l. For state simulator, it computes  $\langle l|\psi \rangle$ , for density matrix simulator, it computes  $Tr(\rho|l\rangle\langle l|)$  Note how these two are different up to a square operation.

**Example**

```
>>> c = tc.Circuit(2)
>>> c.X(0)
>>> c.amplitude("10")
array(1.+0.j, dtype=complex64)
>>> c.CNOT(0, 1)
>>> c.amplitude("11")
array(1.+0.j, dtype=complex64)
```

**Parameters** **l** (Union[str, Tensor]) – The bitstring of 0 and 1s.

**Returns** The amplitude of the circuit.

**Return type** tn.Node.tensor

```
append(c: tensorcircuit.abstractcircuit.AbstractCircuit, indices: Optional[List[int]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
append circuit c before
```

**Example**

```
>>> c1 = tc.Circuit(2)
>>> c1.H(0)
>>> c1.H(1)
>>> c2 = tc.Circuit(2)
>>> c2.cnot(0, 1)
>>> c1.append(c2)
<tensorcircuit.circuit.Circuit object at 0x7f8402968970>
>>> c1.draw()

q_0: H ┌─┐
 └─┘
q_1: H ┌ X ┌
 └─┘
```

**Parameters**

- **c** (**BaseCircuit**) – The other circuit to be appended
- **indices** (Optional[List[int]], optional) – the qubit indices to which c is appended on. Defaults to None, which means plain concatenation.

**Returns** The composed circuit

**Return type** *BaseCircuit*

**append\_from\_qir(qir: List[Dict[str, Any]]) → None**

Apply the ciurict in form of quantum intermediate representation after the current cirucit.

**Example**

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.to_qir()
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None,
 →'mpo': False}]
>>> c2 = tc.Circuit(3)
>>> c2.CNOT(0, 1)
>>> c2.to_qir()
[{'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None,
 →None, 'mpo': False}]
>>> c.append_from_qir(c2.to_qir())
>>> c.to_qir()
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None,
 →'mpo': False},
 {'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None,
 →None, 'mpo': False}]
```

**Parameters** *qir* (*List[Dict[str, Any]]*) – The quantum intermediate representation.

**apply(gate: Union[tensorcircuit.gates.Gate, tensorcircuit.quantum.QuOperator], \*index: int, name: Optional[str] = None, split: Optional[Dict[str, Any]] = None, mpo: bool = False, ir\_dict: Optional[Dict[str, Any]] = None) → None**

An implementation of this method should also append gate directionary to self.\_cir

**apply\_general\_gate(gate: Union[tensorcircuit.gates.Gate, tensorcircuit.quantum.QuOperator], \*index: int, name: Optional[str] = None, split: Optional[Dict[str, Any]] = None, mpo: bool = False, ir\_dict: Optional[Dict[str, Any]] = None) → None**

An implementation of this method should also append gate directionary to self.\_cir

**static apply\_general\_gate\_delayed(gatef: Callable[[], tensorcircuit.gates.Gate], name: Optional[str] = None, mpo: bool = False) → Callable[[], None]**

**static apply\_general\_variable\_gate\_delayed(gatef: Callable[..., tensorcircuit.gates.Gate], name: Optional[str] = None, mpo: bool = False) → Callable[[], None]**

**barrier\_instruction(\*index: List[int]) → None**

add a barrier instruction flag, no effect on numerical simulation

**Parameters** *index* (*List[int]*) – the corresponding qubits

**circuit\_param: Dict[str, Any]**

**static coloring\_copied\_nodes(nodes: Sequence[tensornetwork.network\_components.Node], nodes0: Sequence[tensornetwork.network\_components.Node], is\_dagger: bool = True, flag: str = 'inputs') → None**

**static coloring\_nodes(nodes: Sequence[tensornetwork.network\_components.Node], is\_dagger: bool = False, flag: str = 'inputs') → None**

**cond\_measure**(*index: int, status: Optional[float] = None*) → Any

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a int Tensor and thus maintained a jittable pipeline.

**Example**

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
two possible outputs: (1, 1) or (-1, -1)
```

---

**Note:** In terms of DMCircuit, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measuremet results

---

**Parameters** `index (int)` – the qubit for the z-basis measurement

**Returns** 0 or 1 for z measurement on up and down freedom

**Return type** Tensor

**cond\_measurement**(*index: int, status: Optional[float] = None*) → Any

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a int Tensor and thus maintained a jittable pipeline.

**Example**

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
two possible outputs: (1, 1) or (-1, -1)
```

---

**Note:** In terms of DMCircuit, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measuremet results

---

**Parameters** `index (int)` – the qubit for the z-basis measurement

**Returns** 0 or 1 for z measurement on up and down freedom

**Return type** Tensor

**conditional\_gate**(*which: Any, kraus: Sequence[tensorcircuit.gates.Gate], \*index: int*) → None

Apply which-th gate from kraus list, i.e. apply kraus[which]

**Parameters**

- **which** (`Tensor`) – Tensor of shape [] and dtype int
- **kraus** (`Sequence[Gate]`) – A list of gate in the form of `tc.gate` or `Tensor`
- **index** (`int`) – the qubit lines the gate applied on

**static copy**(nodes: Sequence[tensornetwork.network\_components.Node], dangling: Optional[Sequence[tensornetwork.network\_components.Edge]] = None, conj: Optional[bool] = False) → Tuple[List[tensornetwork.network\_components.Node], List[tensornetwork.network\_components.Edge]]  
copy all nodes and dangling edges correspondingly

**Returns**

**draw**(\*\*kws: Any) → Any

Visualise the circuit. This method receives the keywords as same as qiskit.circuit.QuantumCircuit.draw. More details can be found here: <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.draw.html>.

**Example**

```
>>> c = tc.Circuit(3)
>>> c.H(1)
>>> c.X(2)
>>> c.CNOT(0, 1)
>>> c.draw(output='text')
q_0: _____
 |
q_1: H X |
 | |
 +---+
q_2: X |
 |
 +---+
```

**expectation**(\*ops: Tuple[tensornetwork.network\_components.Node, List[int]], reuse: bool = True, noise\_conf: Optional[Any] = None, nmc: int = 1000, status: Optional[Any] = None, \*\*kws: Any) → Any

**expectation\_before**(\*ops: Tuple[tensornetwork.network\_components.Node, List[int]], reuse: bool = True, \*\*kws: Any) → List[tensornetwork.network\_components.Node]

Get the tensor network in the form of a list of nodes for the expectation calculation before the real contraction

**Parameters** **reuse** (bool, optional) – \_description\_, defaults to True

**Raises** **ValueError** – \_description\_

**Returns** \_description\_

**Return type** List[tn.Node]

**expectation\_ps**(x: Optional[Sequence[int]] = None, y: Optional[Sequence[int]] = None, z: Optional[Sequence[int]] = None, reuse: bool = True, noise\_conf: Optional[Any] = None, nmc: int = 1000, status: Optional[Any] = None, \*\*kws: Any) → Any

Shortcut for Pauli string expectation. x, y, z list are for X, Y, Z positions

**Example**

```
>>> c = tc.Circuit(2)
>>> c.X(0)
>>> c.H(1)
>>> c.expectation_ps(x=[1], z=[0])
array(-0.99999994+0.j, dtype=complex64)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> c.expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
(0.46274087-3.764033e-09j)
```

**Parameters**

- **x** (*Optional[Sequence[int]]*, *optional*) – sites to apply X gate, defaults to None
- **y** (*Optional[Sequence[int]]*, *optional*) – sites to apply Y gate, defaults to None
- **z** (*Optional[Sequence[int]]*, *optional*) – sites to apply Z gate, defaults to None
- **reuse** (*bool*, *optional*) – whether to cache and reuse the wavefunction, defaults to True
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **nmc** (*int*, *optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **status** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** Expectation value**Return type** Tensor

**classmethod** **from\_json**(*jsonstr: str*, *circuit\_params: Optional[Dict[str, Any]] = None*) → *tensorcircuit.abstractcircuit.AbstractCircuit*

load json str as a Circuit

**Parameters**

- **jsonstr** (*str*) – \_description\_
- **circuit\_params** (*Optional[Dict[str, Any]]*, *optional*) – Extra circuit parameters in the format of `__init__`, defaults to None

**Returns** \_description\_**Return type** *AbstractCircuit*

**classmethod** **from\_json\_file**(*file: str*, *circuit\_params: Optional[Dict[str, Any]] = None*) → *tensorcircuit.abstractcircuit.AbstractCircuit*

load json file and convert it to a circuit

**Parameters**

- **file** (str) – filename
- **circuit\_params** (Optional[Dict[str, Any]], optional) – \_description\_, defaults to None

**Returns** \_description\_

**Return type** *AbstractCircuit*

```
classmethod from_openqasm(qasmstr: str, circuit_params: Optional[Dict[str, Any]] = None,
 keep_measure_order: bool = False) →
 tensorcircuit.abstractcircuit.AbstractCircuit

classmethod from_openqasm_file(file: str, circuit_params: Optional[Dict[str, Any]] = None,
 keep_measure_order: bool = False) →
 tensorcircuit.abstractcircuit.AbstractCircuit

classmethod from_qir(qir: List[Dict[str, Any]], circuit_params: Optional[Dict[str, Any]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
```

Restore the circuit from the quantum intermediate representation.

#### Example

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.rx(1, theta=tc.array_to_tensor(0.7))
>>> c.expi(0, 1, unitary=tc.gates._zz_matrix, theta=tc.array_to_tensor(-0.2), ↴
 split=split)
>>> len(c)
7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)
>>> qirs = c.to_qir()
>>>
>>> c = tc.Circuit.from_qir(qirs, circuit_params={"nqubits": 3})
>>> len(c._nodes)
7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)
```

#### Parameters

- **qir** (List[Dict[str, Any]]) – The quantum intermediate representation of a circuit.
- **circuit\_params** (Optional[Dict[str, Any]]) – Extra circuit parameters.

**Returns** The circuit have same gates in the qir.

**Return type** *Circuit*

```
classmethod from_qiskit(qc: Any, n: Optional[int] = None, inputs: Optional[List[float]] = None,
 circuit_params: Optional[Dict[str, Any]] = None, binding_params:
 Optional[Union[Sequence[float], Dict[Any, float]]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
```

Import Qiskit QuantumCircuit object as a `tc.Circuit` object.

#### Example

```
>>> from qiskit import QuantumCircuit
>>> qisc = QuantumCircuit(3)
>>> qisc.h(2)
>>> qisc.cswap(1, 2, 0)
>>> qisc.swap(0, 1)
>>> c = tc.Circuit.from_qiskit(qisc)
```

**Parameters**

- **qc** (*QuantumCircuit in Qiskit*) – Qiskit Circuit object
- **n** (*int*) – The number of qubits for the circuit
- **inputs** (*Optional[List[float]]*, *optional*) – possible input wavefunction for `tc.Circuit`, defaults to None
- **circuit\_params** (*Optional[Dict[str, Any]]*) – kwargs given in `Circuit.__init__` construction function, default to None.
- **binding\_params** (*Optional[Union[Sequence[float], Dict[Any, float]]]*) – (variational) parameters for the circuit. Could be either a sequence or dictionary depending on the type of parameters in the Qiskit circuit. For `ParameterVectorElement` use sequence. For `Parameter` use dictionary

**Returns** The same circuit but as tensorcircuit object**Return type** `Circuit`

**classmethod** `from_qsim_file`(*file: str*, *circuit\_params: Optional[Dict[str, Any]] = None*) → `tensorcircuit.abstractcircuit.AbstractCircuit`

**static** `front_from_nodes`(*nodes: List[tensoretwork.network\_components.Node]*) → `List[tensoretwork.network_components.Edge]`

`gate_aliases` = `[['cnot', 'cx'], ['fredkin', 'cswap'], ['toffoli', 'ccnot'], ['toffoli', 'ccx'], ['any', 'unitary'], ['sd', 'sdg'], ['td', 'tdg']]`

**gate\_count**(*gate\_list: Optional[Sequence[str]] = None*) → *int*  
count the gate number of the circuit

**Example**

```
>>> c = tc.Circuit(3)
>>> c.h(0)
>>> c.multicontrol(0, 1, 2, ctrl=[0, 1], unitary=tc.gates._x_matrix)
>>> c.toffoli(1, 2, 0)
>>> c.gate_count()
3
>>> c.gate_count(["multicontrol", "toffoli"])
2
```

**Parameters** `gate_list` (*Optional[Sequence[str]]*, *optional*) – gate name list to be counted, defaults to None (counting all gates)

**Returns** the total number of all gates or gates in the `gate_list`**Return type** `int`

**gate\_summary()** → Dict[str, int]

return the summary dictionary on gate type - gate count pair

**Returns** the gate count dict by gate type

**Return type** Dict[str, int]

**get\_positional\_logical\_mapping()** → Dict[int, int]

Get positional logical mapping dict based on measure instruction. This function is useful when we only measure part of the qubits in the circuit, to process the count result from partial measurement, we must be aware of the mapping, i.e. for each position in the count bitstring, what is the corresponding qubits (logical) defined on the circuit

**Returns** positional\_logical\_mapping

**Return type** Dict[int, int]

**get\_quvector()** → *tensorcircuit.quantum.QuVector*

Get the representation of the output state in the form of QuVector while maintaining the circuit uncomputed

**Returns** QuVector representation of the output state from the circuit

**Return type** QuVector

**initial\_mapping**(*logical\_physical\_mapping*: Dict[int, int], *n*: Optional[int] = None, *circuit\_params*:

Optional[Dict[str, Any]] = None) → *tensorcircuit.abstractcircuit.AbstractCircuit*

generate a new circuit with the qubit mapping given by *logical\_physical\_mapping*

#### Parameters

- **logical\_physical\_mapping** (Dict[int, int]) – how to map logical qubits to the physical qubits on the new circuit
- **n** (Optional[int], optional) – number of qubit of the new circuit, can be different from the original one, defaults to None
- **circuit\_params** (Optional[Dict[str, Any]], optional) – \_description\_, defaults to None

**Returns** \_description\_

**Return type** AbstractCircuit

**inputs:** Any

**inverse**(*circuit\_params*: Optional[Dict[str, Any]] = None) → *tensorcircuit.abstractcircuit.AbstractCircuit*

inverse the circuit, return a new inversed circuit

#### EXAMPLE

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rzz(1, 2, theta=0.8)
>>> c1 = c.inverse()
```

**Parameters** **circuit\_params** (Optional[Dict[str, Any]], optional) – keywords dict for initialization the new circuit, defaults to None

**Returns** the inversed circuit

**Return type** Circuit

**is\_dm:** bool

```
is_mps: bool = False
```

**measure**(\*index: int, with\_prob: bool = False, status: Optional[Any] = None) → Tuple[Any, Any]  
Take measurement to the given quantum lines. This method is jittable is and about 100 times faster than unjit version!

**Parameters**

- **index** (int) – Measure on which quantum line.
- **with\_prob** (bool, optional) – If true, theoretical probability is also returned.
- **status** (Optional[Tensor]) – external randomness, with shape [index], defaults to None

**Returns** The sample output and probability (optional) of the quantum line.

**Return type** Tuple[Tensor, Tensor]

```
measure_instruction(index: int) → None
```

add a measurement instruction flag, no effect on numerical simulation

**Parameters** **index** (int) – the corresponding qubit

```
measure_jit(*index: int, with_prob: bool = False, status: Optional[Any] = None) → Tuple[Any, Any]
Take measurement to the given quantum lines. This method is jittable is and about 100 times faster than unjit version!
```

**Parameters**

- **index** (int) – Measure on which quantum line.
- **with\_prob** (bool, optional) – If true, theoretical probability is also returned.
- **status** (Optional[Tensor]) – external randomness, with shape [index], defaults to None

**Returns** The sample output and probability (optional) of the quantum line.

**Return type** Tuple[Tensor, Tensor]

```
mpogates = ['multicontrol', 'mpo']
```

```
perfect_sampling(status: Optional[Any] = None) → Tuple[str, float]
```

Sampling bistrings from the circuit output based on quantum amplitudes. Reference: arXiv:1201.3974.

**Parameters** **status** (Optional[Tensor]) – external randomness, with shape [nqubits], defaults to None

**Returns** Sampled bit string and the corresponding theoretical probability.

**Return type** Tuple[str, float]

```
prepend(c: tensorcircuit.abstractcircuit.AbstractCircuit) → tensorcircuit.abstractcircuit.AbstractCircuit
prepend circuit c before
```

**Parameters** **c** (BaseCircuit) – The other circuit to be prepended

**Returns** The composed circuit

**Return type** BaseCircuit

```
probability() → Any
```

get the  $2^n$  length probability vector over computational basis

**Returns** probability vector

**Return type** Tensor

**quvector()** → *tensorcircuit.quantum.QuVector*

Get the representation of the output state in the form of QuVector while maintaining the circuit uncomputed

**Returns** QuVector representation of the output state from the circuit

**Return type** *QuVector*

**readouterror\_bs(*readout\_error*: *Optional[Sequence[Any]]* = None, *p*: *Optional[Any]* = None) → Any**

Apply readout error to original probabilities of bit string and return the noisy probabilities.

**Example**

```
>>> readout_error = []
>>> readout_error.append([0.9, 0.75]) # readout error for qubit 0, [p0/0, p1/1]
>>> readout_error.append([0.4, 0.7]) # readout error for qubit 1, [p0/0, p1/1]
```

**Parameters**

- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – list of readout error for each qubits.
- **p** (*Optional[Any]*) – probabilities of bit string

**Return type** Tensor

**replace\_inputs(*inputs*: Any) → None**

Replace the input state with the circuit structure unchanged.

**Parameters** **inputs** (*Tensor*) – Input wavefunction.

**reset\_instruction(*index*: int) → None**

add a reset instruction flag, no effect on numerical simulation

**Parameters** **index** (*int*) – the corresponding qubit

**sample(*batch*: *Optional[int]* = None, *allow\_state*: *bool* = False, *readout\_error*: *Optional[Sequence[Any]]* = None, *format*: *Optional[str]* = None, *random\_generator*: *Optional[Any]* = None, *status*: *Optional[Any]* = None) → Any**

batched sampling from state or circuit tensor network directly

**Parameters**

- **batch** (*Optional[int]*, *optional*) – number of samples, defaults to None
- **allow\_state** (*bool*, *optional*) – if true, we sample from the final state if memory allows, True is preferred, defaults to False
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to None
- **format** (*Optional[str]*) – sample format, defaults to None as backward compatibility check the doc in [tensorcircuit.quantum.measurement\\_results\(\)](#)
- **format** – alias for the argument format
- **random\_generator** (*Optional[Any]*, *optional*) – random generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random\_generator

**Returns** List (if batch) of tuple (binary configuration tensor and corresponding probability) if the format is None, and consistent with format when given

**Return type** Any

```
sample_expectation_ps(x: Optional[Sequence[int]] = None, y: Optional[Sequence[int]] = None, z:
 Optional[Sequence[int]] = None, shots: Optional[int] = None,
 random_generator: Optional[Any] = None, status: Optional[Any] = None,
 readout_error: Optional[Sequence[Any]] = None, noise_conf: Optional[Any] =
 None, nmc: int = 1000, statusc: Optional[Any] = None, **kws: Any) → Any
```

Compute the expectation with given Pauli string with measurement shots numbers

**Example**

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rx(1, theta=np.pi/2)
>>> c.sample_expectation_ps(x=[0], y=[1])
-0.99999976
>>> readout_error = []
>>> readout_error.append([0.9, 0.75])
>>> readout_error.append([0.4, 0.7])
>>> c.sample_expectation_ps(x=[0], y=[1], readout_error = readout_error)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> readout_error = [[0.9, 0.75], [0.4, 0.7]]
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> noise_conf.add_noise("readout", readout_error)
>>> c.sample_expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
0.44766843
```

**Parameters**

- **x** (*Optional[Sequence[int]]*, *optional*) – index for Pauli X, defaults to None
- **y** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Y, defaults to None
- **z** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Z, defaults to None
- **shots** (*Optional[int]*, *optional*) – number of measurement shots, defaults to None, indicating analytical result
- **random\_generator** (*Optional[Any]*) – random\_generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random\_generator
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to None. Overrided if noise\_conf is provided.
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None

- **nmc** (*int, optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **statusc** (*Optional[Tensor], optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** [description]

**Return type** Tensor

**select\_gate**(*which: Any, kraus: Sequence[tensorcircuit.gates.Gate], \*index: int*) → None

Apply which-th gate from kraus list, i.e. apply kraus[which]

**Parameters**

- **which** (*Tensor*) – Tensor of shape [] and dtype int
- **kraus** (*Sequence[Gate]*) – A list of gate in the form of `tc.gate` or *Tensor*
- **index** (*int*) – the qubit lines the gate applied on

**sexpss**(*x: Optional[Sequence[int]] = None, y: Optional[Sequence[int]] = None, z: Optional[Sequence[int]]*

*= None, shots: Optional[int] = None, random\_generator: Optional[Any] = None, status:*

*Optional[Any] = None, readout\_error: Optional[Sequence[Any]] = None, noise\_conf:*

*Optional[Any] = None, nmc: int = 1000, statusc: Optional[Any] = None, \*\*kws: Any*) → Any

Compute the expectation with given Pauli string with measurement shots numbers

**Example**

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rx(1, theta=np.pi/2)
>>> c.sample_expectation_ps(x=[0], y=[1])
-0.99999976
>>> readout_error = []
>>> readout_error.append([0.9, 0.75])
>>> readout_error.append([0.4, 0.7])
>>> c.sample_expectation_ps(x=[0], y=[1], readout_error = readout_error)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> readout_error = [[0.9, 0.75], [0.4, 0.7]]
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> noise_conf.add_noise("readout", readout_error)
>>> c.sample_expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
0.44766843
```

**Parameters**

- **x** (*Optional[Sequence[int]], optional*) – index for Pauli X, defaults to None

- **y** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Y, defaults to None
- **z** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Z, defaults to None
- **shots** (*Optional[int]*, *optional*) – number of measurement shots, defaults to None, indicating analytical result
- **random\_generator** (*Optional[Any]*) – random\_generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random\_generator
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to None. Overrided if noise\_conf is provided.
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **nmc** (*int*, *optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **statusuc** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** [description]

**Return type** Tensor

**sgates** = ['i', 'x', 'y', 'z', 'h', 't', 's', 'td', 'sd', 'wroot', 'cnot', 'cz', 'swap', 'cy', 'ox', 'oy', 'oz', 'toffoli', 'fredkin']

**split**: *Optional[Dict[str, Any]]*

**static standardize\_gate**(*name: str*) → str

standardize the gate name to tc common gate sets

**Parameters** **name** (*str*) – non-standard gate name

**Returns** the standard gate name

**Return type** str

**tex**(*\*\*kws: Any*) → str

Generate latex string based on quantikz latex package

**Returns** Latex string that can be directly compiled via, e.g. latexit

**Return type** str

**to\_cirq**(*enable\_instruction: bool = False*) → Any

Translate `tc.Circuit` to a cirq circuit object.

**Parameters** **enable\_instruction** (*bool, defaults to False*) – whether also export measurement and reset instructions

**Returns** A cirq circuit of this circuit.

**to\_graphviz**(*graph: Optional[graphviz.graphs.Graph] = None, include\_all\_names: bool = False, engine: str = 'neato'*) → graphviz.graphs.Graph

Not an ideal visualization for quantum circuit, but reserve here as a general approach to show the tensor-network [Deprecated, use `Circuit.vis_tex` or `Circuit.draw` instead]

**to\_json**(*file: Optional[str] = None, simplified: bool = False*) → Any

circuit dumps to json

**Parameters**

- **file** (*Optional[str]*, *optional*) – file str to dump the json to, defaults to None, return the json str
- **simplified** (*bool*) – If False, keep all info for each gate, defaults to be False. If True, suitable for IO since less information is required

**Returns** None if dumps to file otherwise the json str

**Return type** Any

**to\_openqasm**(\*\**kws*: Any) → str

transform circuit to openqasm via qiskit circuit, see <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.qasm.html> for usage on possible options for *kws*

**Returns** circuit representation in openqasm format

**Return type** str

**to\_qir**() → List[Dict[str, Any]]

Return the quantum intermediate representation of the circuit.

**Example**

```
>>> c = tc.Circuit(2)
>>> c.CNOT(0, 1)
>>> c.to_qir()
[{'gatef': cnot, 'gate': Gate(
 name: 'cnot',
 tensor:
 array([[[[1.+0.j, 0.+0.j],
 [0.+0.j, 0.+0.j]],
 [[0.+0.j, 1.+0.j],
 [0.+0.j, 0.+0.j]]],

 [[[0.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]],
 [[0.+0.j, 0.+0.j]]]], dtype=complex64),
 edges: [
 Edge(Dangling Edge)[0],
 Edge(Dangling Edge)[1],
 Edge('cnot'[2] -> 'qb-1'[0]),
 Edge('cnot'[3] -> 'qb-2'[0])
], 'index': (0, 1), 'name': 'cnot', 'split': None, 'mpo': False}]
```

**Returns** The quantum intermediate representation of the circuit.

**Return type** List[Dict[str, Any]]

**to\_qiskit**(enable\_instruction: bool = False) → Any

Translate `tc.Circuit` to a qiskit `QuantumCircuit` object.

**Parameters** `enable_instruction` (*bool*, *defaults to False*) – whether also export measurement and reset instructions

**Returns** A qiskit object of this circuit.

```
vgates = ['r', 'cr', 'u', 'cu', 'rx', 'ry', 'rz', 'phase', 'rxx', 'ryy', 'rzz',
'cphase', 'crx', 'cry', 'crz', 'orx', 'ory', 'orz', 'iswap', 'any', 'exp', 'expl']
```

```
vis_tex(**kws: Any) → str
```

Generate latex string based on quantikz latex package

**Returns** Latex string that can be directly compiled via, e.g. latexit

**Return type** str

## 4.1.5 tensorcircuit.channels

Some common noise quantum channels.

```
class tensorcircuit.channels.KrausList(iterable, name, is_unitary)
```

Bases: list

```
__init__(iterable, name, is_unitary)
```

```
append(object, /)
```

Append object to the end of the list.

```
clear()
```

Remove all items from list.

```
copy()
```

Return a shallow copy of the list.

```
count(value, /)
```

Return number of occurrences of value.

```
extend(iterable, /)
```

Extend list by appending elements from the iterable.

```
index(value, start=0, stop=9223372036854775807, /)
```

Return first index of value.

Raises ValueError if the value is not present.

```
insert(index, object, /)
```

Insert object before index.

```
pop(index=-1, /)
```

Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

```
remove(value, /)
```

Remove first occurrence of value.

Raises ValueError if the value is not present.

```
reverse()
```

Reverse *IN PLACE*.

```
sort(*, key=None, reverse=False)
```

Stable sort *IN PLACE*.

```
tensorcircuit.channels.amplitudedampingchannel(gamma: float, p: float) →
Sequence[tensorcircuit.gates.Gate]
```

Return an amplitude damping channel. Notice: Amplitude damping corresponds to p = 1.

$$\sqrt{p} \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{bmatrix} \quad \sqrt{p} \begin{bmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{bmatrix} \quad \sqrt{1-p} \begin{bmatrix} \sqrt{1-\gamma} & 0 \\ 0 & 1 \end{bmatrix} \quad \sqrt{1-p} \begin{bmatrix} 0 & 0 \\ \sqrt{\gamma} & 0 \end{bmatrix}$$

**Example**

```
>>> cs = amplitudedampingchannel(0.25, 0.3)
>>> tc.channels.single_qubit_kraus_identity_check(cs)
```

**Parameters**

- **gamma** (*float*) – the damping parameter of amplitude ( $\gamma$ )
- **p** (*float*) –  $p$

**Returns** An amplitude damping channel with given  $\gamma$  and  $p$ **Return type** Sequence[*Gate*]

`tensorcircuit.channels.check_rep_transformation(kraus: Sequence[tensorcircuit.gates.Gate], density_matrix: Any, verbose: bool = False)`

Check the convertation between those representations.

**Parameters**

- **kraus** (*Sequence[Gate]*) – A sequence of Gate
- **density\_matrix** (*Matrix*) – Initial density matrix
- **verbose** (*bool, optional*) – Whether print Kraus and new Kraus operators, defaults to False

`tensorcircuit.channels.choi_to_kraus(choi: Any, truncation_rules: Optional[Dict[str, Any]] = None) → Any`

Convert the Choi matrix representation to Kraus operator representation.

This can be done by firstly geting eigen-decomposition of Choi-matrix

$$\Lambda = \sum_k \gamma_k |\phi_k\rangle\langle\phi_k|$$

Then define Kraus operators

$$K_k = \sqrt{\gamma_k} V_k$$

where  $\gamma_k \geq 0$  and  $\phi_k$  is the col-val vectorization of  $V_k$ .**Examples**

```
>>> kraus = tc.channels.phasedampingchannel(0.2)
>>> superop = tc.channels.kraus_to_choi(kraus)
>>> kraus_new = tc.channels.choi_to_kraus(superop, truncation_rules={"max_singular_
↪values":3})
```

**Parameters**

- **choi** (*Matrix*) – Choi matrix
- **truncation\_rules** (*Dictionary*) – A dictionary to restrict the calculation of kraus matrices. The restriction can be the number of kraus terms, which is jitable. It can also be the truncattion error, which is not jitable.

**Returns** A list of Kraus operators**Return type** Sequence[*Matrix*]

`tensorcircuit.channels.choi_to_super(choi: Any) → Any`

Convert from Choi representation to Superoperator representation.

**Parameters** `choi (Matrix)` – Choi matrix

**Returns** Superoperator

**Return type** Matrix

`tensorcircuit.channels.composedkraus(kraus1: tensorcircuit.channels.KrausList, kraus2: tensorcircuit.channels.KrausList) → tensorcircuit.channels.KrausList`

Compose the noise channels

**Parameters**

- `kraus1 (KrausList)` – One noise channel
- `kraus2 (KrausList)` – Another noise channel

**Returns** Composed nosie channel

**Return type** `KrausList`

`tensorcircuit.channels.depolarizingchannel(px: float, py: float, pz: float) → Sequence[tensorcircuit.gates.Gate]`

Return a Depolarizing Channel

$$\sqrt{1 - p_x - p_y - p_z} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \sqrt{p_x} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \sqrt{p_y} \begin{bmatrix} 0 & -1j \\ 1j & 0 \end{bmatrix} \quad \sqrt{p_z} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

**Example**

```
>>> cs = depolarizingchannel(0.1, 0.15, 0.2)
>>> tc.channels.single_qubit_kraus_identity_check(cs)
```

**Parameters**

- `px (float)` –  $p_x$
- `py (float)` –  $p_y$
- `pz (float)` –  $p_z$

**Returns** Sequences of Gates

**Return type** Sequence[`Gate`]

`tensorcircuit.channels.evol_kraus(density_matrix: Any, kraus_list: Sequence[Any]) → Any`

The dynamic evolution according to Kraus operators.

$$\rho' = \sum_k K_k \rho K_k^\dagger$$

**Examples**

```
>>> density_matrix = np.array([[0.5, 0.5], [0.5, 0.5]])
>>> kraus = tc.channels.phasedampingchannel(0.2)
>>> evol_kraus(density_matrix, kraus)
```

**Parameters**

- `density_matrix (Matrix)` – Initial density matrix

- **kraus\_list** (*Sequence[Matrix]*) – A list of Kraus operator

**Returns** Final density matrix

**Return type** Matrix

`tensorcircuit.channels.evol_superop(density_matrix: Any, superop: Any) → Any`

The dynamic evolution according to Superoperator.

**Examples**

```
>>> density_matrix = np.array([[0.5,0.5],[0.5,0.5]])
>>> kraus = tc.channels.phasedampingchannel(0.2)
>>> superop = kraus_to_super(kraus)
>>> evol_superop(density_matrix,superop)
```

**Parameters**

- **density\_matrix** (*Matrix*) – Initial density matrix
- **superop** (*Sequence[Matrix]*) – Superoperator

**Returns** Final density matrix

**Return type** Matrix

`tensorcircuit.channels.generaldepolarizingchannel(p: Union[float, Sequence[Any]], num_qubits: int = 1) → Sequence[tensorcircuit.gates.Gate]`

Return a Depolarizing Channel for 1 qubit or 2 qubits

**Example**

```
>>> cs = tc.channels.generaldepolarizingchannel([0.1,0.1,0.1],1)
>>> tc.channels.kraus_identity_check(cs)
>>> cs = tc.channels.generaldepolarizingchannel(0.02,2)
>>> tc.channels.kraus_identity_check(cs)
```

**Parameters**

- **p** (*Union[float, Sequence]*) – parameter for each Pauli channel
- **num\_qubits** (*int, optional*) – number of qubits, 1 and 2 are available, defaults 1

**Returns** Sequences of Gates

**Return type** Sequence[*Gate*]

`tensorcircuit.channels.is_hermitian_matrix(mat: Any, rtol: float = 1e-08, atol: float = 1e-05)`

Test if an array is a Hermitian matrix

**Parameters**

- **mat** (*Matrix*) – Matrix
- **rtol** (*float, optional*) – \_description\_, defaults to 1e-8
- **atol** (*float, optional*) – \_description\_, defaults to 1e-5

**Returns** \_description\_

**Return type** \_type\_

`tensorcircuit.channels.kraus_identity_check(kraus: Sequence[tensorcircuit.gates.Gate]) → None`  
Check identity of Kraus operators.

$$\sum_k K_k^\dagger K_k = I$$

### Examples

```
>>> cs = resetchannel()
>>> tc.channels.kraus_identity_check(cs)
```

**Parameters** `kraus` (`Sequence[Gate]`) – List of Kraus operators.

`tensorcircuit.channels.kraus_to_choi(kraus_list: Sequence[Any]) → Any`  
Convert from Kraus representation to Choi representation.

**Parameters** `kraus_list` (`Sequence[Matrix]`) – A list Kraus operators

**Returns** Choi matrix

**Return type** Matrix

`tensorcircuit.channels.kraus_to_super(kraus_list: Sequence[Any]) → Any`  
Convert Kraus operator representation to Louivile-Superoperator representation.

In the col-vec basis, the evolution of a state  $\rho$  in terms of tensor components of superoperator  $\varepsilon$  can be expressed as

$$\rho'_{mn} = \sum_{\mu\nu} \varepsilon_{nm,\nu\mu} \rho_{\mu\nu}$$

The superoperator  $\varepsilon$  must make the dynamic map from  $\rho$  to  $\rho'$  to satisfy hermitian-preserving (HP), trace-preserving (TP), and completely positive (CP).

We can construct the superoperator from Kraus operators by

$$\varepsilon = \sum_k K_k^* \otimes K_k$$

### Examples

```
>>> kraus = resetchannel()
>>> tc.channels.kraus_to_super(kraus)
```

**Parameters** `kraus_list` (`Sequence[Gate]`) – A sequence of Gate

**Returns** The corresponding Tensor of Superoperator

**Return type** Matrix

`tensorcircuit.channels.kraus_to_super_gate(kraus_list: Sequence[tensorcircuit.gates.Gate]) → Any`  
Convert Kraus operators to one Tensor (as one Super Gate).

$$\sum_k K_k \otimes K_k^*$$

**Parameters** `kraus_list` (`Sequence[Gate]`) – A sequence of Gate

**Returns** The corresponding Tensor of the list of Kraus operators

**Return type** Tensor

`tensorcircuit.channels.krausgate_to_krausmatrix(kraus_list: Sequence[tensorcircuit.gates.Gate]) → Sequence[Any]`

Convert Kraus of Gate form to Matrix form.

**Parameters** `kraus_list (Sequence[Gate])` – A list of Kraus

**Returns** A list of Kraus operators

**Return type** Sequence[Matrix]

`tensorcircuit.channels.krausmatrix_to_krausgate(kraus_list: Sequence[Any]) → Sequence[tensorcircuit.gates.Gate]`

Convert Kraus of Matrix form to Gate form.

**Parameters** `kraus_list (Sequence[Matrix])` – A list of Kraus

**Returns** A list of Kraus operators

**Return type** Sequence[Gate]

`tensorcircuit.channels.phasedampingchannel(gamma: float) → Sequence[tensorcircuit.gates.Gate]`

Return a phase damping channel with given  $\gamma$

$$\begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ 0 & \sqrt{\gamma} \end{bmatrix}$$

**Example**

```
>>> cs = phasedampingchannel(0.6)
>>> tc.channels.single_qubit_kraus_identity_check(cs)
```

**Parameters** `gamma (float)` – The damping parameter of phase ( $\gamma$ )

**Returns** A phase damping channel with given  $\gamma$

**Return type** Sequence[Gate]

`tensorcircuit.channels.resetchannel() → Sequence[tensorcircuit.gates.Gate]`

Reset channel

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

**Example**

```
>>> cs = resetchannel()
>>> tc.channels.single_qubit_kraus_identity_check(cs)
```

**Returns** Reset channel

**Return type** Sequence[Gate]

`tensorcircuit.channels.reshuffle(op: Any, order: Sequence[int]) → Any`

Reshuffle the dimension index of a matrix.

**Parameters**

- `op (Matrix)` – Input matrix
- `order (Tuple)` – required order

**Returns** Reshuffled matrix

**Return type** Matrix

```
tensorcircuit.channels.single_qubit_kraus_identity_check(kraus:
 Sequence[tensorcircuit.gates.Gate]) →
 None
```

Check identity of Kraus operators.

$$\sum_k K_k^\dagger K_k = I$$

### Examples

```
>>> cs = resetchannel()
>>> tc.channels.kraus_identity_check(cs)
```

**Parameters** **kraus** (Sequence[[Gate](#)]) – List of Kraus operators.

```
tensorcircuit.channels.super_to_choi(superop: Any) → Any
```

Convert Louivile-Superoperator representation to Choi representation.

In the col-vec basis, the evolution of a state  $\rho$  in terms of Choi matrix  $\Lambda$  can be expressed as

$$\rho'_{mn} = \sum_{\mu,\nu} \Lambda_{\mu m, \nu n} \rho_{\mu\nu}$$

The Choi matrix  $\Lambda$  must make the dynamic map from  $\rho$  to  $\rho'$  to satisfy hermitian-preserving (HP), trace-preserving (TP), and completely positive (CP).

In terms of tensor components we have the relationship of Louivile-Superoperator representation and Choi representation

$$\Lambda_{mn,\mu\nu} = \varepsilon_{\nu n, \mu m}$$

### Examples

```
>>> kraus = resetchannel()
>>> superop = tc.channels.kraus_to_super(kraus)
>>> tc.channels.super_to_choi(superop)
```

**Parameters** **superop** ([Matrix](#)) – Superoperator

**Returns** Choi matrix

**Return type** Matrix

```
tensorcircuit.channels.super_to_kraus(superop: Any) → Any
```

Convert from Superoperator representation to Kraus representation.

**Parameters** **superop** ([Matrix](#)) – Superoperator

**Returns** A list of Kraus operator

**Return type** Matrix

```
tensorcircuit.channels.thermalrelaxationchannel(t1: float, t2: float, time: float, method: str = 'ByChoi',
 excitedstatepopulation: float = 0.0) →
 Sequence[tensorcircuit.gates.Gate]
```

Return a thermal\_relaxation\_channel

**Example**

```
>>> cs = thermalrelaxationchannel(100,200,100,"AUTO",0.1)
>>> tc.channels.single_qubit_kraus_identity_check(cs)
```

**Parameters**

- **t1** (*float*) – the T1 relaxation time.
- **t2** (*float*) – the T2 dephasing time.
- **time** (*str*) – gate time
- **method** – method to express error (default: “ByChoi”). When  $T1 > T2$ , choose method “ByKraus” or “ByChoi” for jit. When  $T1 < T2$ , choose method “ByChoi” for jit. Users can also set method as “AUTO” and never mind the relative magnitude of  $T1, T2$ , which is not jitable.
- **excitedstatepopulation** – the population of state  $|1\rangle$  at equilibrium (default: 0)

**Returns** A thermal\_relaxation\_channel**Return type** Sequence[*Gate*]

## 4.1.6 tensorcircuit.circuit

Quantum circuit: the state simulator

```
class tensorcircuit.circuit.Circuit(nqubits: int, inputs: Optional[Any] = None, mps_inputs:
 Optional[tensorcircuit.quantum.QuOperator] = None, split:
 Optional[Dict[str, Any]] = None)
```

Bases: *tensorcircuit.basecircuit.BaseCircuit*

Circuit class. Simple usage demo below.

```
c = tc.Circuit(3)
c.H(1)
c.CNOT(0, 1)
c.RX(2, theta=tc.num_to_tensor(1.))
c.expectation([tc.gates.z(), (2,)]) # 0.54
```

**ANY**(\*index: *int*, \*\*vars: *Any*) → NoneApply ANY gate with parameters on the circuit. See [tensorcircuit.gates.any\\_gate\(\)](#).**Parameters**

- **index** (*int*) – Qubit number that the gate applies on.
- **vars** (*float*) – Parameters for the gate.

**CNOT**(\*index: *int*, split: *Optional[Dict[str, Any]]* = None, name: *Optional[str]* = None) → NoneApply CNOT gate on the circuit. See [tensorcircuit.gates.cnot\\_gate\(\)](#).**Parameters** **index** (*int*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

**CPhase**(\*index: int, \*\*vars: Any) → None

Apply **CPhase** gate with parameters on the circuit. See [tensorcircuit.gates.cphase\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**CR**(\*index: int, \*\*vars: Any) → None

Apply **CR** gate with parameters on the circuit. See [tensorcircuit.gates.cr\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**CRX**(\*index: int, \*\*vars: Any) → None

Apply **CRX** gate with parameters on the circuit. See [tensorcircuit.gates.crx\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**CRY**(\*index: int, \*\*vars: Any) → None

Apply **CRY** gate with parameters on the circuit. See [tensorcircuit.gates.cry\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**CRZ**(\*index: int, \*\*vars: Any) → None

Apply **CRZ** gate with parameters on the circuit. See [tensorcircuit.gates.crz\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**CU**(\*index: int, \*\*vars: Any) → None

Apply **CU** gate with parameters on the circuit. See [tensorcircuit.gates.cu\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**CY**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **CY** gate on the circuit. See [tensorcircuit.gates.cy\\_gate\(\)](#).

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. - 1.j \\ 0. + 0.j & 0. + 0.j & 0. + 1.j & 0. + 0.j \end{bmatrix}$$

**CZ**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **CZ** gate on the circuit. See [tensorcircuit.gates.cz\\_gate\(\)](#).

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & -1. + 0.j \end{bmatrix}$$

`EXP(*index: int, **vars: Any) → None`

Apply `EXP` gate with parameters on the circuit. See `tensorcircuit.gates.exp_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`EXP1(*index: int, **vars: Any) → None`

Apply `EXP1` gate with parameters on the circuit. See `tensorcircuit.gates.exp1_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`FREDKIN(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `FREDKIN` gate on the circuit. See `tensorcircuit.gates.fredkin_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j \end{bmatrix}$$

`H(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `H` gate on the circuit. See `tensorcircuit.gates.h_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & 0.70710677 + 0.j \\ 0.70710677 + 0.j & -0.70710677 + 0.j \end{bmatrix}$$

`I(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `I` gate on the circuit. See `tensorcircuit.gates.i_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`ISWAP(*index: int, **vars: Any) → None`

Apply `ISWAP` gate with parameters on the circuit. See `tensorcircuit.gates.iswap_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**MPO**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply mpo gate in MPO format on the circuit. See [tensorcircuit.gates.mpo\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**MULTICONTROL**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply multicontrol gate in MPO format on the circuit. See [tensorcircuit.gates.multicontrol\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**ORX**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply ORX gate with parameters on the circuit. See [tensorcircuit.gates.orx\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**ORY**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply ORY gate with parameters on the circuit. See [tensorcircuit.gates.ory\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**ORZ**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply ORZ gate with parameters on the circuit. See [tensorcircuit.gates.orz\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**OX**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply OX gate on the circuit. See [tensorcircuit.gates.ox\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**OY**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply OY gate on the circuit. See [tensorcircuit.gates.oy\\_gate\(\)](#).

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j & 0. + 0.j & 0. + 0.j \\ 0. + 1.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`OZ(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **OZ** gate on the circuit. See `tensorcircuit.gates.oz_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`PHASE(*index: int, **vars: Any) → None`

Apply **PHASE** gate with parameters on the circuit. See `tensorcircuit.gates.phase_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`R(*index: int, **vars: Any) → None`

Apply **R** gate with parameters on the circuit. See `tensorcircuit.gates.r_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`RX(*index: int, **vars: Any) → None`

Apply **RX** gate with parameters on the circuit. See `tensorcircuit.gates.rx_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`RXX(*index: int, **vars: Any) → None`

Apply **RXX** gate with parameters on the circuit. See `tensorcircuit.gates.rxx_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`RY(*index: int, **vars: Any) → None`

Apply **RY** gate with parameters on the circuit. See `tensorcircuit.gates.ry_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`RYY(*index: int, **vars: Any) → None`

Apply **RYY** gate with parameters on the circuit. See `tensorcircuit.gates.ryy_gate()`.

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RZ**(\**index: int, \*\*vars: Any*) → None

Apply **RZ** gate with parameters on the circuit. See [tensorcircuit.gates.rz\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RZZ**(\**index: int, \*\*vars: Any*) → None

Apply **RZZ** gate with parameters on the circuit. See [tensorcircuit.gates.rzz\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**S**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply **S** gate on the circuit. See [tensorcircuit.gates.s\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 1.j \end{bmatrix}$$

**SD**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply **SD** gate on the circuit. See [tensorcircuit.gates.sd\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

**SWAP**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply **SWAP** gate on the circuit. See [tensorcircuit.gates.swap\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**T**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply **T** gate on the circuit. See [tensorcircuit.gates.t\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 + 0.70710677j & \end{bmatrix}$$

**TD**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply **TD** gate on the circuit. See [tensorcircuit.gates.td\\_gate\(\)](#).

**Parameters** `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

`TOFFOLI(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

**Parameters** `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`U(*index: int, **vars: Any) → None`

Apply **U** gate with parameters on the circuit. See `tensorcircuit.gates.u_gate()`.

#### Parameters

- `index` (`int.`) – Qubit number that the gate applies on.
- `vars` (`float.`) – Parameters for the gate.

`WROOT(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **WROOT** gate on the circuit. See `tensorcircuit.gates.wroot_gate()`.

**Parameters** `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & -0.5 & -0.5j \\ 0.5 & -0.5j & 0.70710677 + 0.j \end{bmatrix}$$

`X(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **X** gate on the circuit. See `tensorcircuit.gates.x_gate()`.

**Parameters** `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j \\ 1. + 0.j & 0. + 0.j \end{bmatrix}$$

`Y(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **Y** gate on the circuit. See `tensorcircuit.gates.y_gate()`.

**Parameters** `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j \\ 0. + 1.j & 0. + 0.j \end{bmatrix}$$

`Z(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **Z** gate on the circuit. See `tensorcircuit.gates.z_gate()`.

**Parameters** `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j \end{bmatrix}$$

**\_\_init\_\_(nqubits: int, inputs: Optional[Any] = None, mps\_inputs: Optional[tensorcircuit.quantum.QuOperator] = None, split: Optional[Dict[str, Any]] = None) → None**

Circuit object based on state simulator.

#### Parameters

- **nqubits (int)** – The number of qubits in the circuit.
- **inputs (Optional[Tensor], optional)** – If not None, the initial state of the circuit is taken as inputs instead of  $|0\rangle^n$  qubits, defaults to None.
- **mps\_inputs (Optional[QuOperator])** – QuVector for a MPS like initial wavefunction.
- **split (Optional[Dict[str, Any]])** – dict if two qubit gate is ready for split, including parameters for at least one of `max_singular_values` and `max_truncation_err`.

**static all\_zero\_nodes(n: int, d: int = 2, prefix: str = 'qb-') → List[tensornetwork.network\_components.Node]**

**amplitude(l: Union[str, Any]) → Any**

Returns the amplitude of the circuit given the bitstring l. For state simulator, it computes  $\langle l|\psi\rangle$ , for density matrix simulator, it computes  $Tr(\rho|l\rangle\langle 1|)$  Note how these two are different up to a square operation.

#### Example

```
>>> c = tc.Circuit(2)
>>> c.X(0)
>>> c.amplitude("10")
array(1.+0.j, dtype=complex64)
>>> c.CNOT(0, 1)
>>> c.amplitude("11")
array(1.+0.j, dtype=complex64)
```

**Parameters 1 (Union[str, Tensor])** – The bitstring of 0 and 1s.

**Returns** The amplitude of the circuit.

**Return type** tn.Node.tensor

**amplitudedamping(\*index: int, status: Optional[float] = None, name: Optional[str] = None, \*\*vars: float) → None**

Apply amplitudedamping quantum channel on the circuit. See `tensorcircuit.channels.amplitudedampingchannel()`

#### Parameters

- **index (int.)** – Qubit number that the gate applies on.
- **status (Tensor)** – uniform external random number between 0 and 1
- **vars (float.)** – Parameters for the channel.

**any(\*index: int, \*\*vars: Any) → None**

Apply ANY gate with parameters on the circuit. See `tensorcircuit.gates.any_gate()`.

#### Parameters

- **index (int.)** – Qubit number that the gate applies on.
- **vars (float.)** – Parameters for the gate.

**append**(*c*: tensorcircuit.abstractcircuit.AbstractCircuit, *indices*: *Optional[List[int]]* = *None*) → *tensorcircuit.abstractcircuit.AbstractCircuit*  
 append circuit *c* before

#### Example

```
>>> c1 = tc.Circuit(2)
>>> c1.H(0)
>>> c1.H(1)
>>> c2 = tc.Circuit(2)
>>> c2.cnot(0, 1)
>>> c1.append(c2)
<tensorcircuit.circuit.Circuit object at 0x7f8402968970>
>>> c1.draw()

q_0: H ──
 |
q_1: H ┌ X ┌
```

#### Parameters

- **c** (*BaseCircuit*) – The other circuit to be appended
- **indices** (*Optional[List[int]]*, *optional*) – the qubit indices to which *c* is appended on. Defaults to *None*, which means plain concatenation.

**Returns** The composed circuit

**Return type** *BaseCircuit*

**append\_from\_qir**(*qir*: *List[Dict[str, Any]]*) → *None*

Apply the ciurict in form of quantum intermediate representation after the current cirucit.

#### Example

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.to_qir()
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None,
 → 'mpo': False}]
>>> c2 = tc.Circuit(3)
>>> c2.CNOT(0, 1)
>>> c2.to_qir()
[{'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None,
 → 'mpo': False}]
>>> c.append_from_qir(c2.to_qir())
>>> c.to_qir()
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None,
 → 'mpo': False},
 {'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None,
 → 'mpo': False}]
```

**Parameters** *qir* (*List[Dict[str, Any]]*) – The quantum intermediate representation.

```
apply(gate: Union[tensorcircuit.gates.Gate, tensorcircuit.quantum.QuOperator], *index: int, name: Optional[str] = None, split: Optional[Dict[str, Any]] = None, mpo: bool = False, ir_dict: Optional[Dict[str, Any]] = None) → None
```

An implementation of this method should also append gate directionary to self.\_cir

```
apply_general_gate(gate: Union[tensorcircuit.gates.Gate, tensorcircuit.quantum.QuOperator], *index: int, name: Optional[str] = None, split: Optional[Dict[str, Any]] = None, mpo: bool = False, ir_dict: Optional[Dict[str, Any]] = None) → None
```

An implementation of this method should also append gate directionary to self.\_cir

```
static apply_general_gate_delayed(gatef: Callable[[], tensorcircuit.gates.Gate], name: Optional[str] = None, mpo: bool = False) → Callable[[], None]
```

```
apply_general_kraus(kraus: Sequence[tensorcircuit.gates.Gate], *index: int, status: Optional[float] = None, name: Optional[str] = None) → Any
```

Monte Carlo trajectory simulation of general Kraus channel whose Kraus operators cannot be amplified to unitary operators. For unitary operators composed Kraus channel, [unitary\\_kraus\(\)](#) is much faster.

This function is jittable in theory. But only jax+GPU combination is recommended for jit since the graph building time is too long for other backend options; though the running time of the function is very fast for every case.

#### Parameters

- **kraus** (`Sequence[Gate]`) – A list of `tn.Node` for Kraus operators.
- **index** (`int`) – The qubits index that Kraus channel is applied on.
- **status** (`Optional[float]`, `optional`) – Random tensor uniformly between 0 or 1, defaults to be None, when the random number will be generated automatically

```
static apply_general_kraus_delayed(krausf: Callable[[], Sequence[tensorcircuit.gates.Gate]], is_unitary: bool = False) → Callable[[], None]
```

```
static apply_general_variable_gate_delayed(gatef: Callable[[], tensorcircuit.gates.Gate], name: Optional[str] = None, mpo: bool = False) → Callable[[], None]
```

```
barrier_instruction(*index: List[int]) → None
```

add a barrier instruction flag, no effect on numerical simulation

#### Parameters **index** (`List[int]`) – the corresponding qubits

```
ccnot(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None
```

Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

#### Parameters **index** (`int`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

```
ccx(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None
```

Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

**Parameters** `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`circuit_param: Dict[str, Any]`

`cnot(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply CNOT gate on the circuit. See `tensorcircuit.gates.cnot_gate()`.

**Parameters** `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

`static coloring_copied_nodes(nodes: Sequence[tensornetwork.network_components.Node], nodes0: Sequence[tensornetwork.network_components.Node], is_dagger: bool = True, flag: str = 'inputs') → None`

`static coloring_nodes(nodes: Sequence[tensornetwork.network_components.Node], is_dagger: bool = False, flag: str = 'inputs') → None`

`cond_measurement(index: int, status: Optional[float] = None) → Any`

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a int Tensor and thus maintained a jittable pipeline.

### Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
two possible outputs: (1, 1) or (-1, -1)
```

---

**Note:** In terms of DMCircuit, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measurement results

---

**Parameters** `index` (`int`) – the qubit for the z-basis measurement

**Returns** 0 or 1 for z measurement on up and down freedom

**Return type** Tensor

`cond_measurement(index: int, status: Optional[float] = None) → Any`

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a int Tensor and thus maintained a jittable pipeline.

**Example**

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
two possible outputs: (1, 1) or (-1, -1)
```

---

**Note:** In terms of DMCircuit, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measurement results

---

**Parameters** `index (int)` – the qubit for the z-basis measurement

**Returns** 0 or 1 for z measurement on up and down freedom

**Return type** Tensor

**conditional\_gate**(`which: Any, kraus: Sequence[tensorcircuit.gates.Gate], *index: int`) → None

Apply which-th gate from kraus list, i.e. apply kraus[which]

**Parameters**

- `which (Tensor)` – Tensor of shape [] and dtype int
- `kraus (Sequence[Gate])` – A list of gate in the form of `tc.gate` or Tensor
- `index (int)` – the qubit lines the gate applied on

**static copy**(`nodes: Sequence[tensornetwork.network_components.Node], dangling:`

`Optional[Sequence[tensornetwork.network_components.Edge]] = None, conj: Optional[bool] = False`) → Tuple[List[tensornetwork.network\_components.Node], List[tensornetwork.network\_components.Edge]]

copy all nodes and dangling edges correspondingly

**Returns**

**cphase**(\*`index: int, **vars: Any`) → None

Apply CPHASE gate with parameters on the circuit. See `tensorcircuit.gates.cphase_gate()`.

**Parameters**

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

**cr**(\*`index: int, **vars: Any`) → None

Apply CR gate with parameters on the circuit. See `tensorcircuit.gates.cr_gate()`.

**Parameters**

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

**crx**(\*`index: int, **vars: Any`) → None

Apply CRX gate with parameters on the circuit. See `tensorcircuit.gates.crx_gate()`.

**Parameters**

- `index (int.)` – Qubit number that the gate applies on.

- **vars** (*float.*) – Parameters for the gate.

**cry**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **CRY** gate with parameters on the circuit. See `tensorcircuit.gates.cry_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**crz**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **CRZ** gate with parameters on the circuit. See `tensorcircuit.gates.crz_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**cswap**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **FREDKIN** gate on the circuit. See `tensorcircuit.gates.fredkin_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**cu**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **CU** gate with parameters on the circuit. See `tensorcircuit.gates.cu_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**cx**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **CNOT** gate on the circuit. See `tensorcircuit.gates.cnot_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

**cy**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **CY** gate on the circuit. See `tensorcircuit.gates.cy_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. - 1.j \\ 0. + 0.j & 0. + 0.j & 0. + 1.j & 0. + 0.j \end{bmatrix}$$

**cz**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply CZ gate on the circuit. See `tensorcircuit.gates.cz_gate()`.

**Parameters** `index` (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & -1. + 0.j \end{bmatrix}$$

**depolarizing**(\*index: int, status: Optional[float] = None, name: Optional[str] = None, \*\*vars: float) → None

Apply depolarizing quantum channel on the circuit. See `tensorcircuit.channels.depolarizingchannel()`

**Parameters**

- `index` (int.) – Qubit number that the gate applies on.
- `status` (Tensor) – uniform external random number between 0 and 1
- `vars` (float.) – Parameters for the channel.

**depolarizing2**(index: int, \*, px: float, py: float, pz: float, status: Optional[float] = None) → float

**depolarizing\_reference**(index: int, \*, px: float, py: float, pz: float, status: Optional[float] = None) → Any

Apply depolarizing channel in a Monte Carlo way, i.e. for each call of this method, one of gates from X, Y, Z, I are applied on the circuit based on the probability indicated by px, py, pz.

**Parameters**

- `index` (int) – The qubit that depolarizing channel is on
- `px` (float) – probability for X noise
- `py` (float) – probability for Y noise
- `pz` (float) – probability for Z noise
- `status` (Optional[float], optional) – random seed uniformly from 0 to 1, defaults to None (generated implicitly)

**Returns** int Tensor, the element lookup: [0: x, 1: y, 2: z, 3: I]

**Return type** Tensor

**draw**(\*\*kws: Any) → Any

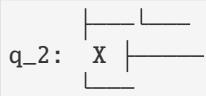
Visualise the circuit. This method receives the keywords as same as qiskit.circuit.QuantumCircuit.draw. More details can be found here: <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.draw.html>.

**Example**

```
>>> c = tc.Circuit(3)
>>> c.H(1)
>>> c.X(2)
>>> c.CNOT(0, 1)
>>> c.draw(output='text')
q_0: _____
 |
q_1: H ┌ X ┌
```

(continues on next page)

(continued from previous page)

**exp**(\*index: int, \*\*vars: Any) → NoneApply EXP gate with parameters on the circuit. See `tensorcircuit.gates.exp_gate()`.**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**exp1**(\*index: int, \*\*vars: Any) → NoneApply EXP1 gate with parameters on the circuit. See `tensorcircuit.gates.exp1_gate()`.**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**expectation**(\*ops: Tuple[tensornetwork.network\_components.Node, List[int]], reuse: bool = True, enable\_lightcone: bool = False, noise\_conf: Optional[Any] = None, nmc: int = 1000, status: Optional[Any] = None, \*\*kws: Any) → Any

Compute the expectation of corresponding operators.

**Example**

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.expectation((tc.gates.z(), [0]))
array(0.+0.j, dtype=complex64)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> c.expectation((tc.gates.x(), [0]), noise_conf=noise_conf, nmc=10000)
(0.46274087-3.764033e-09j)
```

**Parameters**

- **ops** (Tuple[tn.Node, List[int]]) – Operator and its position on the circuit, eg. `(tc.gates.z(), [1, ])`, `(tc.gates.x(), [2, ])` is for operator  $Z_1X_2$ .
- **reuse** (bool, optional) – If True, then the wavefunction tensor is cached for further expectation evaluation, defaults to true.
- **enable\_lightcone** (bool, optional) – whether enable light cone simplification, defaults to False

- **noise\_conf** (*Optional[NoiseConf], optional*) – Noise Configuration, defaults to None
- **nmc** (*int, optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **status** (*Optional[Tensor], optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Raises ValueError** – “Cannot measure two operators in one index”

**Returns** Tensor with one element

**Return type** Tensor

**expectation\_before**(\*ops: Tuple[tensornetwork.network\_components.Node, List[int]], reuse: bool = True, \*\*kws: Any) → List[tensornetwork.network\_components.Node]

Get the tensor network in the form of a list of nodes for the expectation calculation before the real contraction

**Parameters reuse** (*bool, optional*) – \_description\_, defaults to True

**Raises ValueError** – \_description\_

**Returns** \_description\_

**Return type** List[tn.Node]

**expectation\_ps**(x: *Optional[Sequence[int]]* = None, y: *Optional[Sequence[int]]* = None, z: *Optional[Sequence[int]]* = None, reuse: *bool* = True, noise\_conf: *Optional[Any]* = None, nmc: *int* = 1000, status: *Optional[Any]* = None, \*\*kws: Any) → Any

Shortcut for Pauli string expectation. x, y, z list are for X, Y, Z positions

### Example

```
>>> c = tc.Circuit(2)
>>> c.X(0)
>>> c.H(1)
>>> c.expectation_ps(x=[1], z=[0])
array(-0.99999994+0.j, dtype=complex64)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> c.expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
(0.46274087-3.764033e-09j)
```

### Parameters

- **x** (*Optional[Sequence[int]], optional*) – sites to apply X gate, defaults to None

- **y** (*Optional[Sequence[int]]*, *optional*) – sites to apply Y gate, defaults to None
- **z** (*Optional[Sequence[int]]*, *optional*) – sites to apply Z gate, defaults to None
- **reuse** (*bool*, *optional*) – whether to cache and reuse the wavefunction, defaults to True
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **nmc** (*int*, *optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **status** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** Expectation value

**Return type** Tensor

**fredkin**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply FREDKIN gate on the circuit. See `tensorcircuit.gates.fredkin_gate()`.

**Parameters** **index** (*int*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**classmethod from\_json**(*jsonstr: str, circuit\_params: Optional[Dict[str, Any]] = None*) →

`tensorcircuit.abstractcircuit.AbstractCircuit`

load json str as a Circuit

**Parameters**

- **jsonstr** (*str*) – \_description\_
- **circuit\_params** (*Optional[Dict[str, Any]]*, *optional*) – Extra circuit parameters in the format of `__init__`, defaults to None

**Returns** \_description\_

**Return type** `AbstractCircuit`

**classmethod from\_json\_file**(*file: str, circuit\_params: Optional[Dict[str, Any]] = None*) →

`tensorcircuit.abstractcircuit.AbstractCircuit`

load json file and convert it to a circuit

**Parameters**

- **file** (*str*) – filename
- **circuit\_params** (*Optional[Dict[str, Any]]*, *optional*) – \_description\_, defaults to None

**Returns** \_description\_

**Return type** *AbstractCircuit*

```
classmethod from_openqasm(qasmstr: str, circuit_params: Optional[Dict[str, Any]] = None,
 keep_measure_order: bool = False) →
 tensorcircuit.abstractcircuit.AbstractCircuit

classmethod from_openqasm_file(file: str, circuit_params: Optional[Dict[str, Any]] = None,
 keep_measure_order: bool = False) →
 tensorcircuit.abstractcircuit.AbstractCircuit

classmethod from_qir(qir: List[Dict[str, Any]], circuit_params: Optional[Dict[str, Any]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
```

Restore the circuit from the quantum intermediate representation.

#### Example

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.rx(1, theta=tc.array_to_tensor(0.7))
>>> c.expl(0, 1, unitary=tc.gates._zz_matrix, theta=tc.array_to_tensor(-0.2),
 →split=split)
>>> len(c)
7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)
>>> qirs = c.to_qir()
>>>
>>> c = tc.Circuit.from_qir(qirs, circuit_params={"nqubits": 3})
>>> len(c._nodes)
7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)
```

#### Parameters

- **qir** (*List[Dict[str, Any]]*) – The quantum intermediate representation of a circuit.
- **circuit\_params** (*Optional[Dict[str, Any]]*) – Extra circuit parameters.

**Returns** The circuit have same gates in the qir.

**Return type** *Circuit*

```
classmethod from_qiskit(qc: Any, n: Optional[int] = None, inputs: Optional[List[float]] = None,
 circuit_params: Optional[Dict[str, Any]] = None, binding_params:
 Optional[Union[Sequence[float], Dict[Any, float]]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
```

Import Qiskit QuantumCircuit object as a `tc.Circuit` object.

#### Example

```
>>> from qiskit import QuantumCircuit
>>> qisc = QuantumCircuit(3)
>>> qisc.h(2)
>>> qisc.cswap(1, 2, 0)
>>> qisc.swap(0, 1)
>>> c = tc.Circuit.from_qiskit(qisc)
```

**Parameters**

- **qc** (*QuantumCircuit in Qiskit*) – Qiskit Circuit object
- **n** (*int*) – The number of qubits for the circuit
- **inputs** (*Optional[List[float]]*, *optional*) – possible input wavefunction for `tc.Circuit`, defaults to None
- **circuit\_params** (*Optional[Dict[str, Any]]*) – kwargs given in `Circuit.__init__` construction function, default to None.
- **binding\_params** (*Optional[Union[Sequence[float], Dict[Any, float]]]*) – (variational) parameters for the circuit. Could be either a sequence or dictionary depending on the type of parameters in the Qiskit circuit. For `ParameterVectorElement` use sequence. For `Parameter` use dictionary

**Returns** The same circuit but as tensorcircuit object

**Return type** `Circuit`

```
classmethod from_qsim_file(file: str, circuit_params: Optional[Dict[str, Any]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
```

```
static front_from_nodes(nodes: List[tensornetwork.network_components.Node]) →
 List[tensornetwork.network_components.Edge]
```

```
gate_aliases = [['cnot', 'cx'], ['fredkin', 'cswap'], ['toffoli', 'ccnot'],
['toffoli', 'ccx'], ['any', 'unitary'], ['sd', 'sdg'], ['td', 'tdg']]
```

```
gate_count(gate_list: Optional[Sequence[str]] = None) → int
count the gate number of the circuit
```

**Example**

```
>>> c = tc.Circuit(3)
>>> c.h(0)
>>> c.multicontrol(0, 1, 2, ctrl=[0, 1], unitary=tc.gates._x_matrix)
>>> c.toffoli(1, 2, 0)
>>> c.gate_count()
3
>>> c.gate_count(["multicontrol", "toffoli"])
2
```

**Parameters** `gate_list` (*Optional[Sequence[str]]*, *optional*) – gate name list to be counted, defaults to None (counting all gates)

**Returns** the total number of all gates or gates in the `gate_list`

**Return type** `int`

```
gate_summary() → Dict[str, int]
```

return the summary dictionary on gate type - gate count pair

**Returns** the gate count dict by gate type

**Return type** `Dict[str, int]`

```
general_kraus(kraus: Sequence[tensorcircuit.gates.Gate], *index: int, status: Optional[float] = None,
name: Optional[str] = None) → Any
```

Monte Carlo trajectory simulation of general Kraus channel whose Kraus operators cannot be amplified to unitary operators. For unitary operators composed Kraus channel, `unitary_kraus()` is much faster.

This function is jittable in theory. But only jax+GPU combination is recommended for jit since the graph building time is too long for other backend options; though the running time of the function is very fast for every case.

**Parameters**

- **kraus** (*Sequence[Gate]*) – A list of `tn.Node` for Kraus operators.
- **index** (*int*) – The qubits index that Kraus channel is applied on.
- **status** (*Optional[float]*, *optional*) – Random tensor uniformly between 0 or 1, defaults to be None, when the random number will be generated automatically

**generaldepolarizing**(\**index: int, status: Optional[float] = None, name: Optional[str] = None, \*\*vars: float*) → *None*

Apply generaldepolarizing quantum channel on the circuit. See `tensorcircuit.channels.generaldepolarizingchannel()`

**Parameters**

- **index** (*int*) – Qubit number that the gate applies on.
- **status** (*Tensor*) – uniform external random number between 0 and 1
- **vars** (*float*) – Parameters for the channel.

**get\_circuit\_as\_quoperator()** → `tensorcircuit.quantum.QuOperator`

Get the QuOperator MPO like representation of the circuit unitary without contraction.

**Returns** QuOperator object for the circuit unitary (open indices for the input state)

**Return type** `QuOperator`

**get\_positional\_logical\_mapping()** → `Dict[int, int]`

Get positional logical mapping dict based on measure instruction. This function is useful when we only measure part of the qubits in the circuit, to process the count result from partial measurement, we must be aware of the mapping, i.e. for each position in the count bitstring, what is the corresponding qubits (logical) defined on the circuit

**Returns** `positional_logical_mapping`

**Return type** `Dict[int, int]`

**get\_quoperator()** → `tensorcircuit.quantum.QuOperator`

Get the QuOperator MPO like representation of the circuit unitary without contraction.

**Returns** QuOperator object for the circuit unitary (open indices for the input state)

**Return type** `QuOperator`

**get\_quvector()** → `tensorcircuit.quantum.QuVector`

Get the representation of the output state in the form of QuVector while maintaining the circuit uncomputed

**Returns** QuVector representation of the output state from the circuit

**Return type** `QuVector`

**get\_state\_as\_quvector()** → `tensorcircuit.quantum.QuVector`

Get the representation of the output state in the form of QuVector while maintaining the circuit uncomputed

**Returns** QuVector representation of the output state from the circuit

**Return type** `QuVector`

**h(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None**  
 Apply **H** gate on the circuit. See `tensorcircuit.gates.h_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & 0.70710677 + 0.j \\ 0.70710677 + 0.j & -0.70710677 + 0.j \end{bmatrix}$$

**i(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None**  
 Apply **I** gate on the circuit. See `tensorcircuit.gates.i_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**initial\_mapping(logical\_physical\_mapping: Dict[int, int], n: Optional[int] = None, circuit\_params: Optional[Dict[str, Any]] = None) → tensorcircuit.abstractcircuit.AbstractCircuit**  
 generate a new circuit with the qubit mapping given by `logical_physical_mapping`

**Parameters**

- `logical_physical_mapping (Dict[int, int])` – how to map logical qubits to the physical qubits on the new circuit
- `n (Optional[int], optional)` – number of qubit of the new circuit, can be different from the original one, defaults to None
- `circuit_params (Optional[Dict[str, Any]], optional)` – `_description_`, defaults to None

**Returns** `_description_`

**Return type** `AbstractCircuit`

**inputs:** `Any`

**inverse(circuit\_params: Optional[Dict[str, Any]] = None) → tensorcircuit.abstractcircuit.AbstractCircuit**  
 inverse the circuit, return a new inversed circuit

#### EXAMPLE

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rzz(1, 2, theta=0.8)
>>> c1 = c.inverse()
```

**Parameters** `circuit_params (Optional[Dict[str, Any]], optional)` – keywords dict for initialization the new circuit, defaults to None

**Returns** the inversed circuit

**Return type** `Circuit`

**is\_dm: bool = False**

**is\_mps: bool = False**

**is\_valid() → bool**

[WIP], check whether the circuit is legal.

**Returns** The bool indicating whether the circuit is legal

**Return type** bool

**iswap**(\*index: int, \*\*vars: Any) → None

Apply ISWAP gate with parameters on the circuit. See [`tensorcircuit.gates.iswap\\_gate\(\)`](#).

**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**matrix()** → Any

Get the unitary matrix for the circuit irrespective with the circuit input state.

**Returns** The circuit unitary matrix

**Return type** Tensor

**measure**(\*index: int, with\_prob: bool = False, status: Optional[Any] = None) → Tuple[Any, Any]

Take measurement to the given quantum lines. This method is jittable is and about 100 times faster than unjit version!

**Parameters**

- **index** (int) – Measure on which quantum line.
- **with\_prob** (bool, optional) – If true, theoretical probability is also returned.
- **status** (Optional [Tensor]) – external randomness, with shape [index], defaults to None

**Returns** The sample output and probability (optional) of the quantum line.

**Return type** Tuple[Tensor, Tensor]

**measure\_instruction**(index: int) → None

add a measurement instruction flag, no effect on numerical simulation

**Parameters** **index** (int) – the corresponding qubit

**measure\_jit**(\*index: int, with\_prob: bool = False, status: Optional[Any] = None) → Tuple[Any, Any]

Take measurement to the given quantum lines. This method is jittable is and about 100 times faster than unjit version!

**Parameters**

- **index** (int) – Measure on which quantum line.
- **with\_prob** (bool, optional) – If true, theoretical probability is also returned.
- **status** (Optional [Tensor]) – external randomness, with shape [index], defaults to None

**Returns** The sample output and probability (optional) of the quantum line.

**Return type** Tuple[Tensor, Tensor]

**measure\_reference**(\*index: int, with\_prob: bool = False) → Tuple[str, float]

Take measurement on the given quantum lines by **index**.

**Example**

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.h(1)
>>> c.toffoli(0, 1, 2)
```

(continues on next page)

(continued from previous page)

```
>>> c.measure(2)
('1', -1.0)
>>> # Another possible output: ('0', -1.0)
>>> c.measure(2, with_prob=True)
('1', (0.25000011920928955+0j))
>>> # Another possible output: ('0', (0.7499998807907104+0j))
```

### Parameters

- **index** – Measure on which quantum line.
- **with\_prob** – If true, theoretical probability is also returned.

**Returns** The sample output and probability (optional) of the quantum line.

**Return type** Tuple[str, float]

**mid\_measure**(*index: int, keep: int = 0*) → Any

Middle measurement in z-basis on the circuit, note the wavefunction output is not normalized with `mid_measurement` involved, one should normalize the state manually if needed. This is a post-selection method as `keep` is provided as a prior.

### Parameters

- **index (int)** – The index of qubit that the Z direction postselection applied on.
- **keep (int, optional)** – 0 for spin up, 1 for spin down, defaults to be 0.

**mid\_measurement**(*index: int, keep: int = 0*) → Any

Middle measurement in z-basis on the circuit, note the wavefunction output is not normalized with `mid_measurement` involved, one should normalize the state manually if needed. This is a post-selection method as `keep` is provided as a prior.

### Parameters

- **index (int)** – The index of qubit that the Z direction postselection applied on.
- **keep (int, optional)** – 0 for spin up, 1 for spin down, defaults to be 0.

**mpo**(\**index: int, \*\*vars: Any*) → None

Apply mpo gate in MPO format on the circuit. See `tensorcircuit.gates.mpo_gate()`.

### Parameters

- **index (int.)** – Qubit number that the gate applies on.
- **vars (float.)** – Parameters for the gate.

**mpogates** = ['multicontrol', 'mpo']

**multicontrol**(\**index: int, \*\*vars: Any*) → None

Apply multicontrol gate in MPO format on the circuit. See `tensorcircuit.gates.multicontrol_gate()`.

### Parameters

- **index (int.)** – Qubit number that the gate applies on.
- **vars (float.)** – Parameters for the gate.

**orx**(\**index: int, \*\*vars: Any*) → None

Apply ORX gate with parameters on the circuit. See `tensorcircuit.gates.orx_gate()`.

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**ory**(\**index*: *int*, \*\**vars*: *Any*) → *None*Apply **ORY** gate with parameters on the circuit. See `tensorcircuit.gates.ory_gate()`.**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**orz**(\**index*: *int*, \*\**vars*: *Any*) → *None*Apply **ORZ** gate with parameters on the circuit. See `tensorcircuit.gates.orz_gate()`.**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**ox**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*Apply **OX** gate on the circuit. See `tensorcircuit.gates.ox_gate()`.**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**oy**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*Apply **OY** gate on the circuit. See `tensorcircuit.gates.oy_gate()`.**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j & 0. + 0.j & 0. + 0.j \\ 0. + 1.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**oz**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*Apply **OZ** gate on the circuit. See `tensorcircuit.gates.oz_gate()`.**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**perfect\_sampling**(*status*: *Optional[Any]* = *None*) → *Tuple[str, float]*

Sampling bistrings from the circuit output based on quantum amplitudes. Reference: arXiv:1201.3974.

**Parameters** **status** (*Optional[Tensor]*) – external randomness, with shape [nqubits], default to None**Returns** Sampled bit string and the corresponding theoretical probability.

**Return type** Tuple[str, float]

**phase**(\*index: int, \*\*vars: Any) → None

Apply PHASE gate with parameters on the circuit. See [tensorcircuit.gates.phase\\_gate\(\)](#).

**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**phasedamping**(\*index: int, status: Optional[float] = None, name: Optional[str] = None, \*\*vars: float) → None

Apply phasedamping quantum channel on the circuit. See [tensorcircuit.channels.phasedampingchannel\(\)](#)

**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **status** (Tensor) – uniform external random number between 0 and 1
- **vars** (float.) – Parameters for the channel.

**post\_select**(index: int, keep: int = 0) → Any

Middle measurement in z-basis on the circuit, note the wavefunction output is not normalized with mid\_measurement involved, one should normalize the state manually if needed. This is a post-selection method as keep is provided as a prior.

**Parameters**

- **index** (int) – The index of qubit that the Z direction postselection applied on.
- **keep** (int, optional) – 0 for spin up, 1 for spin down, defaults to be 0.

**post\_selection**(index: int, keep: int = 0) → Any

Middle measurement in z-basis on the circuit, note the wavefunction output is not normalized with mid\_measurement involved, one should normalize the state manually if needed. This is a post-selection method as keep is provided as a prior.

**Parameters**

- **index** (int) – The index of qubit that the Z direction postselection applied on.
- **keep** (int, optional) – 0 for spin up, 1 for spin down, defaults to be 0.

**prepend**(c: [tensorcircuit.abstractcircuit.AbstractCircuit](#)) → [tensorcircuit.abstractcircuit.AbstractCircuit](#)

prepend circuit c before

**Parameters** **c** ([BaseCircuit](#)) – The other circuit to be prepended

**Returns** The composed circuit

**Return type** [BaseCircuit](#)

**probability**() → Any

get the  $2^n$  length probability vector over computational basis

**Returns** probability vector

**Return type** Tensor

**quoperator**() → [tensorcircuit.quantum.QuOperator](#)

Get the QuOperator MPO like representation of the circuit unitary without contraction.

**Returns** QuOperator object for the circuit unitary (open indices for the input state)

**Return type** *QuOperator***quvector()** → *tensorcircuit.quantum.QuVector*

Get the representation of the output state in the form of QuVector while maintaining the circuit uncomputed

**Returns** QuVector representation of the output state from the circuit

**Return type** *QuVector***r(\*index: int, \*\*vars: Any)** → None

Apply R gate with parameters on the circuit. See [tensorcircuit.gates.r\\_gate\(\)](#).

**Parameters**

- **index (int.)** – Qubit number that the gate applies on.
- **vars (float.)** – Parameters for the gate.

**readouterror\_bs(readout\_error: Optional[Sequence[Any]] = None, p: Optional[Any] = None)** → Any

Apply readout error to original probabilities of bit string and return the noisy probabilities.

**Example**

```
>>> readout_error = []
>>> readout_error.append([0.9, 0.75]) # readout error for qubit 0, [p0/0, p1/1]
>>> readout_error.append([0.4, 0.7]) # readout error for qubit 1, [p0/0, p1/1]
```

**Parameters**

- **readout\_error (Optional[Sequence[Any]] Tensor, List, Tuple)** – list of readout error for each qubits.
- **p (Optional[Any])** – probabilities of bit string

**Return type** Tensor**replace\_inputs(inputs: Any)** → None

Replace the input state with the circuit structure unchanged.

**Parameters** **inputs (Tensor)** – Input wavefunction.

**replace\_mps\_inputs(mps\_inputs: tensorcircuit.quantum.QuOperator)** → None

Replace the input state in MPS representation while keep the circuit structure unchanged.

**Example**

```
>>> c = tc.Circuit(2)
>>> c.X(0)
>>>
>>> c2 = tc.Circuit(2, mps_inputs=c.quvector())
>>> c2.X(0)
>>> c2.wavefunction()
array([1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j], dtype=complex64)
>>>
>>> c3 = tc.Circuit(2)
>>> c3.X(0)
>>> c3.replace_mps_inputs(c.quvector())
>>> c3.wavefunction()
array([1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j], dtype=complex64)
```

**Parameters** `mps_inputs` (`Tuple[Sequence[Gate], Sequence[Edge]]`) – (Nodes, dangling Edges) for a MPS like initial wavefunction.

**reset**(\*`index: int, status: Optional[float] = None, name: Optional[str] = None, **vars: float`) → None  
Apply reset quantum channel on the circuit. See `tensorcircuit.channels.resetchannel()`

#### Parameters

- **index** (`int.`) – Qubit number that the gate applies on.
- **status** (`Tensor`) – uniform external random number between 0 and 1
- **vars** (`float.`) – Parameters for the channel.

**reset\_instruction**(`index: int`) → None  
add a reset instruction flag, no effect on numerical simulation

#### Parameters `index` (`int`) – the corresponding qubit

**rx**(\*`index: int, **vars: Any`) → None  
Apply **RX** gate with parameters on the circuit. See `tensorcircuit.gates.rx_gate()`.

#### Parameters

- **index** (`int.`) – Qubit number that the gate applies on.
- **vars** (`float.`) – Parameters for the gate.

**rxx**(\*`index: int, **vars: Any`) → None  
Apply **RXX** gate with parameters on the circuit. See `tensorcircuit.gates.rxx_gate()`.

#### Parameters

- **index** (`int.`) – Qubit number that the gate applies on.
- **vars** (`float.`) – Parameters for the gate.

**ry**(\*`index: int, **vars: Any`) → None  
Apply **RY** gate with parameters on the circuit. See `tensorcircuit.gates.ry_gate()`.

#### Parameters

- **index** (`int.`) – Qubit number that the gate applies on.
- **vars** (`float.`) – Parameters for the gate.

**ryy**(\*`index: int, **vars: Any`) → None  
Apply **RYY** gate with parameters on the circuit. See `tensorcircuit.gates.ryy_gate()`.

#### Parameters

- **index** (`int.`) – Qubit number that the gate applies on.
- **vars** (`float.`) – Parameters for the gate.

**rz**(\*`index: int, **vars: Any`) → None  
Apply **RZ** gate with parameters on the circuit. See `tensorcircuit.gates.rz_gate()`.

#### Parameters

- **index** (`int.`) – Qubit number that the gate applies on.
- **vars** (`float.`) – Parameters for the gate.

**rzz**(\*`index: int, **vars: Any`) → None  
Apply **RZZ** gate with parameters on the circuit. See `tensorcircuit.gates.rzz_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**s**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None  
Apply S gate on the circuit. See `tensorcircuit.gates.s_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 1.j \end{bmatrix}$$

**sample**(*batch: Optional[int] = None, allow\_state: bool = False, readout\_error: Optional[Sequence[Any]] = None, format: Optional[str] = None, random\_generator: Optional[Any] = None, status: Optional[Any] = None*) → Any  
batched sampling from state or circuit tensor network directly

**Parameters**

- **batch** (*Optional[int]*, *optional*) – number of samples, defaults to None
- **allow\_state** (*bool*, *optional*) – if true, we sample from the final state if memory allows, True is preferred, defaults to False
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to None
- **format** (*Optional[str]*) – sample format, defaults to None as backward compatibility check the doc in `tensorcircuit.quantum.measurement_results()`
- **format** – alias for the argument `format`
- **random\_generator** (*Optional[Any]*, *optional*) – random generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random\_generator

**Returns** List (if batch) of tuple (binary configuration tensor and corresponding probability) if the format is None, and consistent with format when given

**Return type** Any

**sample\_expectation\_ps**(*x: Optional[Sequence[int]] = None, y: Optional[Sequence[int]] = None, z: Optional[Sequence[int]] = None, shots: Optional[int] = None, random\_generator: Optional[Any] = None, status: Optional[Any] = None, readout\_error: Optional[Sequence[Any]] = None, noise\_conf: Optional[Any] = None, nmc: int = 1000, statusc: Optional[Any] = None, \*\*kws: Any*) → Any  
Compute the expectation with given Pauli string with measurement shots numbers

**Example**

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rx(1, theta=np.pi/2)
>>> c.sample_expectation_ps(x=[0], y=[1])
-0.99999976
>>> readout_error = []
>>> readout_error.append([0.9, 0.75])
>>> readout_error.append([0.4, 0.7])
>>> c.sample_expectation_ps(x=[0], y=[1], readout_error = readout_error)
```

```

>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> readout_error = [[0.9, 0.75], [0.4, 0.7]]
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> noise_conf.add_noise("readout", readout_error)
>>> c.sample_expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
0.44766843

```

**Parameters**

- **x** (*Optional[Sequence[int]]*, *optional*) – index for Pauli X, defaults to None
- **y** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Y, defaults to None
- **z** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Z, defaults to None
- **shots** (*Optional[int]*, *optional*) – number of measurement shots, defaults to None, indicating analytical result
- **random\_generator** (*Optional[Any]*) – random\_generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random\_generator
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to None. Overrided if noise\_conf is provided.
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **nmc** (*int, optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **statusc** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** [description]**Return type** Tensor**sd**(\*index: int, split: *Optional[Dict[str, Any]]* = None, name: *Optional[str]* = None) → NoneApply **SD** gate on the circuit. See `tensorcircuit.gates.sd_gate()`.**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

**sdg**(\*index: int, split: *Optional[Dict[str, Any]]* = None, name: *Optional[str]* = None) → NoneApply **SD** gate on the circuit. See `tensorcircuit.gates.sd_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

`select_gate(which: Any, kraus: Sequence[tensorcircuit.gates.Gate], *index: int) → None`

Apply which-th gate from kraus list, i.e. apply kraus[which]

#### Parameters

- `which (Tensor)` – Tensor of shape [] and dtype int
- `kraus (Sequence[Gate])` – A list of gate in the form of `tc.gate` or `Tensor`
- `index (int)` – the qubit lines the gate applied on

`sexps(x: Optional[Sequence[int]] = None, y: Optional[Sequence[int]] = None, z: Optional[Sequence[int]] = None, shots: Optional[int] = None, random_generator: Optional[Any] = None, status: Optional[Any] = None, readout_error: Optional[Sequence[Any]] = None, noise_conf: Optional[Any] = None, nmc: int = 1000, statusc: Optional[Any] = None, **kws: Any) → Any`

Compute the expectation with given Pauli string with measurement shots numbers

#### Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rx(1, theta=np.pi/2)
>>> c.sample_expectation_ps(x=[0], y=[1])
-0.99999976
>>> readout_error = []
>>> readout_error.append([0.9, 0.75])
>>> readout_error.append([0.4, 0.7])
>>> c.sample_expectation_ps(x=[0], y=[1], readout_error = readout_error)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> readout_error = [[0.9, 0.75], [0.4, 0.7]]
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> noise_conf.add_noise("readout", readout_error)
>>> c.sample_expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
0.44766843
```

#### Parameters

- `x (Optional[Sequence[int]], optional)` – index for Pauli X, defaults to None
- `y (Optional[Sequence[int]], optional)` – index for Pauli Y, defaults to None
- `z (Optional[Sequence[int]], optional)` – index for Pauli Z, defaults to None

- **shots** (*Optional[int]*, *optional*) – number of measurement shots, defaults to None, indicating analytical result
- **random\_generator** (*Optional[Any]*) – random\_generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random\_generator
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to None. Overrided if noise\_conf is provided.
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **nmc** (*int*, *optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **statussc** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** [description]

**Return type** Tensor

```
sgates = ['i', 'x', 'y', 'z', 'h', 't', 's', 'td', 'sd', 'wroot', 'cnot', 'cz',
'swap', 'cy', 'ox', 'oy', 'oz', 'toffoli', 'fredkin']
```

**split: Optional[Dict[str, Any]]**

**static standardize\_gate(name: str) → str**

standardize the gate name to common gate sets

**Parameters name (str)** – non-standard gate name

**Returns** the standard gate name

**Return type** str

**state(form: str = 'default')** → <property object at 0x7f72990543b0>

Compute the output wavefunction from the circuit.

**Parameters form(str, optional)** – The str indicating the form of the output wavefunction.

“default”: [-1], “ket”: [-1, 1], “bra”: [1, -1]

**Returns** Tensor with the corresponding shape.

**Return type** Tensor

**swap(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None)** → None

Apply SWAP gate on the circuit. See `tensorcircuit.gates.swap_gate()`.

**Parameters index (int.)** – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**t(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None)** → None

Apply T gate on the circuit. See `tensorcircuit.gates.t_gate()`.

**Parameters index (int.)** – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 + 0.70710677j & \end{bmatrix}$$

**td**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None  
Apply TD gate on the circuit. See [tensorcircuit.gates.td\\_gate\(\)](#).

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

**tdg**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None  
Apply TD gate on the circuit. See [tensorcircuit.gates.td\\_gate\(\)](#).

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

**tex**(\*\*kws: Any) → str  
Generate latex string based on quantikz latex package

**Returns** Latex string that can be directly compiled via, e.g. latexit

**Return type** str

**thermalrelaxation**(\*index: int, status: Optional[float] = None, name: Optional[str] = None, \*\*vars: float) → None  
Apply thermalrelaxation quantum channel on the circuit. See [tensorcircuit.channels.thermalrelaxationchannel\(\)](#).

**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **status** (Tensor) – uniform external random number between 0 and 1
- **vars** (float.) – Parameters for the channel.

**to\_cirq**(enable\_instruction: bool = False) → Any  
Translate tc.Circuit to a cirq circuit object.

**Parameters** **enable\_instruction** (bool, defaults to False) – whether also export measurement and reset instructions

**Returns** A cirq circuit of this circuit.

**to\_graphviz**(graph: Optional[graphviz.Graphs.Graph] = None, include\_all\_names: bool = False, engine: str = 'neato') → graphviz.Graph  
Not an ideal visualization for quantum circuit, but reserve here as a general approach to show the tensor-network [Deprecated, use Circuit.vis\_tex or Circuit.draw instead]

**to\_json**(file: Optional[str] = None, simplified: bool = False) → Any  
circuit dumps to json

**Parameters**

- **file** (Optional[str], optional) – file str to dump the json to, defaults to None, return the json str
- **simplified** (bool) – If False, keep all info for each gate, defaults to be False. If True, suitable for IO since less information is required

**Returns** None if dumps to file otherwise the json str

**Return type** Any

**to\_openqasm**(*\*\*kws: Any*) → str

transform circuit to openqasm via qiskit circuit, see <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.qasm.html> for usage on possible options for *kws*

**Returns** circuit representation in openqasm format

**Return type** str

**to\_qir**() → List[Dict[str, Any]]

Return the quantum intermediate representation of the circuit.

**Example**

```
>>> c = tc.Circuit(2)
>>> c.CNOT(0, 1)
>>> c.to_qir()
[{'gatef': cnot, 'gate': Gate(
 name: 'cnot',
 tensor:
 array([[[[1.+0.j, 0.+0.j],
 [0.+0.j, 0.+0.j]],
 [[0.+0.j, 1.+0.j],
 [0.+0.j, 0.+0.j]]],
 [[[0.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]],
 [[0.+0.j, 0.+0.j]]], dtype=complex64),
 edges: [
 Edge(Dangling Edge)[0],
 Edge(Dangling Edge)[1],
 Edge('cnot'[2] -> 'qb-1'[0]),
 Edge('cnot'[3] -> 'qb-2'[0])
], 'index': (0, 1), 'name': 'cnot', 'split': None, 'mpo': False}]
```

**Returns** The quantum intermediate representation of the circuit.

**Return type** List[Dict[str, Any]]

**to\_qiskit**(*enable\_instruction: bool = False*) → Any

Translate `tc.Circuit` to a qiskit `QuantumCircuit` object.

**Parameters** `enable_instruction (bool, defaults to False)` – whether also export measurement and reset instructions

**Returns** A qiskit object of this circuit.

**toffoli**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`u(*index: int, **vars: Any) → None`

Apply U gate with parameters on the circuit. See [tensorcircuit.gates.u\\_gate\(\)](#).

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`unitary(*index: int, **vars: Any) → None`

Apply ANY gate with parameters on the circuit. See [tensorcircuit.gates.any\\_gate\(\)](#).

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`unitary_kraus(kraus: Sequence[tensorcircuit.gates.Gate], *index: int, prob: Optional[Sequence[float]] = None, status: Optional[float] = None, name: Optional[str] = None) → Any`

Apply unitary gates in kraus randomly based on corresponding prob. If prob is None, this is reduced to kraus channel language.

#### Parameters

- `kraus (Sequence[Gate])` – List of `tc.gates.Gate` or just Tensors
- `prob (Optional[Sequence[float]], optional)` – prob list with the same size as kraus, defaults to None
- `status (Optional[float], optional)` – random seed between 0 to 1, defaults to None

**Returns** shape [] int dtype tensor indicates which kraus gate is actually applied

**Return type** Tensor

`unitary_kraus2(kraus: Sequence[tensorcircuit.gates.Gate], *index: int, prob: Optional[Sequence[float]] = None, status: Optional[float] = None, name: Optional[str] = None) → Any`

`vgates = ['r', 'cr', 'u', 'cu', 'rx', 'ry', 'rz', 'phase', 'rxx', 'ryy', 'rzz', 'cphase', 'crx', 'cry', 'crz', 'orx', 'ory', 'orz', 'iswap', 'any', 'exp', 'exp1']`

`vis_tex(**kws: Any) → str`

Generate latex string based on quantikz latex package

**Returns** Latex string that can be directly compiled via, e.g. latexit

**Return type** str

`wavefunction(form: str = 'default') → <property object at 0x7f72990543b0>`

Compute the output wavefunction from the circuit.

**Parameters** `form(str, optional)` – The str indicating the form of the output wavefunction.  
“default”: [-1], “ket”: [-1, 1], “bra”: [1, -1]

**Returns** Tensor with the corresponding shape.

**Return type** Tensor

**wroot**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None  
Apply **WROOT** gate on the circuit. See `tensorcircuit.gates.wroot_gate()`.

**Parameters** `index(int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & -0.5 & -0.5j \\ 0.5 & -0.5j & 0.70710677 + 0.j \end{bmatrix}$$

**x**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None  
Apply **X** gate on the circuit. See `tensorcircuit.gates.x_gate()`.

**Parameters** `index(int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j \\ 1. + 0.j & 0. + 0.j \end{bmatrix}$$

**y**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None  
Apply **Y** gate on the circuit. See `tensorcircuit.gates.y_gate()`.

**Parameters** `index(int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j \\ 0. + 1.j & 0. + 0.j \end{bmatrix}$$

**z**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None  
Apply **Z** gate on the circuit. See `tensorcircuit.gates.z_gate()`.

**Parameters** `index(int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j \end{bmatrix}$$

`tensorcircuit.circuit.expectation(*ops: Tuple[tensornetwork.network_components.Node, List[int]], ket: Any, bra: Optional[Any] = None, conj: bool = True, normalization: bool = False) → Any`

Compute  $\langle bra | ops | ket \rangle$ .

Example 1 (*bra* is same as *ket*)

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.ry(1, theta=tc.num_to_tensor(0.8 + 0.7j))
>>> c.cnot(1, 2)
>>> state = c.wavefunction() # the state of this circuit
>>> x1z2 = [(tc.gates.x(), [0]), (tc.gates.z(), [1])] # input qubits
>>>
>>> # Expectation of this circuit / <state|*x1z2|state>
>>> c.expectation(*x1z2)
array(0.69670665+0.j, dtype=complex64)
>>> tc.expectation(*x1z2, ket=state)
```

(continues on next page)

(continued from previous page)

```
(0.6967066526412964+0j)
>>>
>>> # Normalize(expectation of Circuit) / Normalize(<state|*x1z2/state>)
>>> c.expectation(*x1z2) / tc.backend.norm(state) ** 2
(0.5550700389340034+0j)
>>> tc.expectation(*x1z2, ket=state, normalization=True)
(0.55507004+0j)
```

Example 2 (*bra* is different from *ket*)

```
>>> c = tc.Circuit(2)
>>> c.X(1)
>>> s1 = c.state()
>>> c2 = tc.Circuit(2)
>>> c2.X(0)
>>> s2 = c2.state()
>>> c3 = tc.Circuit(2)
>>> c3.H(1)
>>> s3 = c3.state()
>>> x1x2 = [(tc.gates.x(), [0]), (tc.gates.x(), [1])]
>>>
>>> tc.expectation(*x1x2, ket=s1, bra=s2)
(1+0j)
>>> tc.expectation(*x1x2, ket=s3, bra=s2)
(0.7071067690849304+0j) # 1/sqrt(2)
```

### Parameters

- **ket** (*Tensor*) – *ket*. The state in tensor or QuVector format
- **bra** (*Optional[Tensor]*, *optional*) – *bra*, defaults to None, which is the same as *ket*.
- **conj** (*bool*, *optional*) – *bra* changes to the adjoint matrix of *bra*, defaults to True.
- **normalization** (*bool*, *optional*) – Normalize the *ket* and *bra*, defaults to False.

**Raises** `ValueError` – “Cannot measure two operators in one index”

**Returns** The result of  $\langle \text{bra} | \text{ops} | \text{ket} \rangle$ .

**Return type** Tensor

## 4.1.7 tensorcircuit.compiler

### tensorcircuit.compiler.qiskit\_compiler

compiler interface via qiskit

```
tensorcircuit.compiler.qiskit_compiler(circuit: Any, info: Optional[Dict[str, Any]] = None, output: str = 'tc', compiled_options: Optional[Dict[str, Any]] = None) → Any
```

## 4.1.8 tensorcircuit.cons

Constants and setups

```
tensorcircuit.cons.contraction_info_decorator(algorithm: Callable[..., Any]) → Callable[..., Any]
tensorcircuit.cons.contractor(nodes: List[Any], *, optimizer: Any = <function greedy>, memory_limit: Optional[int] = None, output_edge_order: Optional[List[Any]] = None, ignore_edge_order: bool = False, **kws: Any) → Any
tensorcircuit.cons.custom(nodes: List[Any], optimizer: Any, memory_limit: Optional[int] = None, output_edge_order: Optional[List[Any]] = None, ignore_edge_order: bool = False, **kws: Any) → Any
tensorcircuit.cons.custom_stateful(nodes: List[Any], optimizer: Any, memory_limit: Optional[int] = None, opt_conf: Optional[Dict[str, Any]] = None, output_edge_order: Optional[List[Any]] = None, ignore_edge_order: bool = False, **kws: Any) → Any
tensorcircuit.cons.d2s(n: int, dl: List[Any]) → List[Any]
tensorcircuit.cons.experimental_contractor(nodes: List[Any], output_edge_order: Optional[List[Any]] = None, ignore_edge_order: bool = False, local_steps: int = 2) → Any
tensorcircuit.cons.get_contractor(method: Optional[str] = None, optimizer: Optional[Any] = None, memory_limit: Optional[int] = None, opt_conf: Optional[Dict[str, Any]] = None, *, set_global: bool = False, contraction_info: bool = False, debug_level: int = 0, **kws: Any) → Callable[..., Any]
```

To set runtime contractor of the tensornetwork for a better contraction path. For more information on the usage of contractor, please refer to independent tutorial.

### Parameters

- **method** (*Optional[str]*, *optional*) – “auto”, “greedy”, “branch”, “plain”, “tng”, “custom”, “custom\_stateful”. defaults to None (“auto”)
- **optimizer** (*Optional[Any]*, *optional*) – Valid for “custom” or “custom\_stateful” as method, defaults to None
- **memory\_limit** (*Optional[int]*, *optional*) – It is not very useful, as `memory_limit` leads to branch contraction instead of `greedy` which is rather slow, defaults to None

### Raises

- **Exception** – Tensornetwork version is too low to support some of the contractors.
- **ValueError** – Unknown method options.

**Returns** The new tensornetwork with its contractor set.

**Return type** tn.Node

```
tensorcircuit.cons.get_dtype(dtype: Optional[str] = None, *, set_global: bool = False) → Tuple[str, str]
Set the global runtime numerical dtype of tensors.
```

**Parameters** `dtype` (*Optional[str]*, *optional*) – “complex64”/“float32” or “complex128”/“float64”, defaults to None, which is equivalent to “complex64”.

**Returns** complex dtype str and the corresponding real dtype str

**Return type** Tuple[str, str]

```
tensorcircuit.cons.nodes_to_adj(ns: List[Any]) → Any
```

`tensorcircuit.cons.plain_contractor(nodes: List[Any], output_edge_order: Optional[List[Any]] = None, ignore_edge_order: bool = False) → Any`

The naive state-vector simulator contraction path.

**Parameters**

- **nodes** (`List[Any]`) – The list of `tn.Node`.
- **output\_edge\_order** (`Optional[List[Any]]`, `optional`) – The list of dangling node edges, defaults to be `None`.

**Returns** The `tn.Node` after contraction**Return type** `tn.Node`

`tensorcircuit.cons.runtime_backend(backend: Optional[str] = None) → Iterator[Any]`

Context manager to set with-level runtime backend

**Parameters** `backend` (`Optional[str]`, `optional`) – “numpy”, “tensorflow”, “jax”, “pytorch”, defaults to `None`

**Yield** the backend object**Return type** `Iterator[Any]`

`tensorcircuit.cons.runtime_contractor(*confargs: Any, **confkws: Any) → Iterator[Any]`

Context manager to change with-levek contractor

**Yield** \_description\_**Return type** `Iterator[Any]`

`tensorcircuit.cons.runtime_dtype(dtype: Optional[str] = None) → Iterator[Tuple[str, str]]`

Context manager to set with-level runtime dtype

**Parameters** `dtype` (`Optional[str]`, `optional`) – “complex64” or “complex128”, defaults to `None` (“complex64”)

**Yield** complex dtype str and real dtype str**Return type** `Iterator[Tuple[str, str]]`

`tensorcircuit.cons.set_backend(backend: Optional[str] = None, set_global: bool = True) → Any`

To set the runtime backend of tensorcircuit.

Note: `tc.set_backend` and `tc.cons.set_tensornetwork_backend` are the same.

**Example**

```
>>> tc.set_backend("numpy")
numpy_backend
>>> tc.gates.num_to_tensor(0.1)
array(0.1+0.j, dtype=complex64)
>>>
>>> tc.set_backend("tensorflow")
tensorflow_backend
>>> tc.gates.num_to_tensor(0.1)
<tf.Tensor: shape=(), dtype=complex64, numpy=(0.1+0j)>
>>>
>>> tc.set_backend("pytorch")
pytorch_backend
>>> tc.gates.num_to_tensor(0.1)
tensor(0.1000+0.j)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> tc.set_backend("jax")
jax_backend
>>> tc.gates.num_to_tensor(0.1)
DeviceArray(0.1+0.j, dtype=complex64)
```

**Parameters**

- **backend** (*Optional[str]*, *optional*) – “numpy”, “tensorflow”, “jax”, “pytorch”. defaults to None, which gives the same behavior as `tensorcircuit.backend_contextmanager.get_default_backend()`.
- **set\_global** (*bool*) – Whether the object should be set as global.

**Returns** The `tc.backend` object that with all registered universal functions.**Return type** backend object

`tensorcircuit.cons.set_contractor`(*method: Optional[str] = None*, *optimizer: Optional[Any] = None*,  
*memory\_limit: Optional[int] = None*, *opt\_conf: Optional[Dict[str, Any]] = None*, *set\_global: bool = True*, *contraction\_info: bool = False*,  
*debug\_level: int = 0*, *\*\*kws: Any*) → Callable[[...], Any]

To set runtime contractor of the tensornetwork for a better contraction path. For more information on the usage of contractor, please refer to independent tutorial.

**Parameters**

- **method** (*Optional[str]*, *optional*) – “auto”, “greedy”, “branch”, “plain”, “tng”, “custom”, “custom\_stateful”. defaults to None (“auto”)
- **optimizer** (*Optional[Any]*, *optional*) – Valid for “custom” or “custom\_stateful” as method, defaults to None
- **memory\_limit** (*Optional[int]*, *optional*) – It is not very useful, as `memory_limit` leads to branch contraction instead of greedy which is rather slow, defaults to None

**Raises**

- **Exception** – Tensor network version is too low to support some of the contractors.
- **ValueError** – Unknown method options.

**Returns** The new tensor network with its contractor set.**Return type** tn.Node

`tensorcircuit.cons.set_dtype`(*dtype: Optional[str] = None*, *set\_global: bool = True*) → Tuple[str, str]  
Set the global runtime numerical dtype of tensors.

**Parameters** **dtype** (*Optional[str]*, *optional*) – “complex64”/“float32” or “complex128”/“float64”, defaults to None, which is equivalent to “complex64”.

**Returns** complex dtype str and the corresponding real dtype str**Return type** Tuple[str, str]

`tensorcircuit.cons.set_function_backend`(*backend: Optional[str] = None*) → Callable[[...], Any]  
Function decorator to set function-level runtime backend

**Parameters** **backend** (*Optional[str]*, *optional*) – “numpy”, “tensorflow”, “jax”, “pytorch”, defaults to None

**Returns** Decorated function

**Return type** Callable[..., Any]

`tensorcircuit.cons.set_function_contractor(*confargs: Any, **confkws: Any) → Callable[..., Any]`  
Function decorate to change function-level contractor

**Returns** \_description\_

**Return type** Callable[..., Any]

`tensorcircuit.cons.set_function_dtype(dtype: Optional[str] = None) → Callable[..., Any]`  
Function decorator to set function-level numerical dtype

**Parameters** `dtype (Optional[str], optional)` – “complex64” or “complex128”, defaults to None

**Returns** The decorated function

**Return type** Callable[..., Any]

`tensorcircuit.cons.set_tensornetwork_backend(backend: Optional[str] = None, set_global: bool = True) → Any`

To set the runtime backend of tensorcircuit.

Note: `tc.set_backend` and `tc.cons.set_tensornetwork_backend` are the same.

### Example

```
>>> tc.set_backend("numpy")
numpy_backend
>>> tc.gates.num_to_tensor(0.1)
array(0.1+0.j, dtype=complex64)
>>>
>>> tc.set_backend("tensorflow")
tensorflow_backend
>>> tc.gates.num_to_tensor(0.1)
<tf.Tensor: shape=(), dtype=complex64, numpy=(0.1+0j)>
>>>
>>> tc.set_backend("pytorch")
pytorch_backend
>>> tc.gates.num_to_tensor(0.1)
tensor(0.1000+0.j)
>>>
>>> tc.set_backend("jax")
jax_backend
>>> tc.gates.num_to_tensor(0.1)
DeviceArray(0.1+0.j, dtype=complex64)
```

### Parameters

- `backend (Optional[str], optional)` – “numpy”, “tensorflow”, “jax”, “pytorch”. defaults to None, which gives the same behavior as `tensornetwork.backend_contextmanager.get_default_backend()`.
- `set_global (bool)` – Whether the object should be set as global.

**Returns** The `tc.backend` object that with all registered universal functions.

**Return type** backend object

`tensorcircuit.cons.split_rules(max_singular_values: Optional[int] = None, max_truncation_err: Optional[float] = None, relative: bool = False) → Any`

Obtain the direcionary of truncation rules

#### Parameters

- `max_singular_values (int, optional)` – The maximum number of singular values to keep.
- `max_truncation_err (float, optional)` – The maximum allowed truncation error.
- `relative (bool, optional)` – Multiply `max_truncation_err` with the largest singular value.

`tensorcircuit.cons.tn_greedy_contractor(nodes: List[Any], output_edge_order: Optional[List[Any]] = None, ignore_edge_order: bool = False, max_branch: int = 1) → Any`

## 4.1.9 `tensorcircuit.densitymatrix`

Quantum circuit class but with density matrix simulator

`class tensorcircuit.densitymatrix.DMCircuit(nqubits: int, empty: bool = False, inputs: Optional[Any] = None, mps_inputs: Optional[tensorcircuit.quantum.QuOperator] = None, dminputs: Optional[Any] = None, mpo_dminputs: Optional[tensorcircuit.quantum.QuOperator] = None, split: Optional[Dict[str, Any]] = None)`

Bases: `tensorcircuit.basecircuit.BaseCircuit`

`ANY(*index: int, **vars: Any) → None`

Apply `ANY` gate with parameters on the circuit. See `tensorcircuit.gates.any_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`CNOT(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `CNOT` gate on the circuit. See `tensorcircuit.gates.cnot_gate()`.

`Parameters index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

`CPhase(*index: int, **vars: Any) → None`

Apply `CPhase` gate with parameters on the circuit. See `tensorcircuit.gates.cphase_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`CR(*index: int, **vars: Any) → None`

Apply `CR` gate with parameters on the circuit. See `tensorcircuit.gates.cr_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**CRX**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **CRX** gate with parameters on the circuit. See `tensorcircuit.gates.crx_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**CRY**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **CRY** gate with parameters on the circuit. See `tensorcircuit.gates.cry_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**CRZ**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **CRZ** gate with parameters on the circuit. See `tensorcircuit.gates.crz_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**CU**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **CU** gate with parameters on the circuit. See `tensorcircuit.gates.cu_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**CY**(\**index*: *int*, *split*: *Optional[Dict[str, Any]] = None*, *name*: *Optional[str] = None*) → None

Apply **CY** gate on the circuit. See `tensorcircuit.gates.cy_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. - 1.j \\ 0. + 0.j & 0. + 0.j & 0. + 1.j & 0. + 0.j \end{bmatrix}$$

**CZ**(\**index*: *int*, *split*: *Optional[Dict[str, Any]] = None*, *name*: *Optional[str] = None*) → None

Apply **CZ** gate on the circuit. See `tensorcircuit.gates.cz_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & -1. + 0.j \end{bmatrix}$$

**EXP**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **EXP** gate with parameters on the circuit. See `tensorcircuit.gates.exp_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**EXP1**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **EXP1** gate with parameters on the circuit. See `tensorcircuit.gates.exp1_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**FREDKIN**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **FREDKIN** gate on the circuit. See `tensorcircuit.gates.fredkin_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**H**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **H** gate on the circuit. See `tensorcircuit.gates.h_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & 0.70710677 + 0.j \\ 0.70710677 + 0.j & -0.70710677 + 0.j \end{bmatrix}$$

**I**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **I** gate on the circuit. See `tensorcircuit.gates.i_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**ISWAP**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **ISWAP** gate with parameters on the circuit. See `tensorcircuit.gates.iswap_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**MPO**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply mpo gate in MPO format on the circuit. See `tensorcircuit.gates.mpo_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**MULTICONTROL**(\*index: int, \*\*vars: Any) → None

Apply multicontrol gate in MPO format on the circuit. See [tensorcircuit.gates.multicontrol\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**ORX**(\*index: int, \*\*vars: Any) → None

Apply **ORX** gate with parameters on the circuit. See [tensorcircuit.gates.orx\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**ORY**(\*index: int, \*\*vars: Any) → None

Apply **ORY** gate with parameters on the circuit. See [tensorcircuit.gates.ory\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**ORZ**(\*index: int, \*\*vars: Any) → None

Apply **ORZ** gate with parameters on the circuit. See [tensorcircuit.gates.orz\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**OX**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **OX** gate on the circuit. See [tensorcircuit.gates.ox\\_gate\(\)](#).

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**OY**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **OY** gate on the circuit. See [tensorcircuit.gates.oy\\_gate\(\)](#).

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j & 0. + 0.j & 0. + 0.j \\ 0. + 1.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**OZ**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **OZ** gate on the circuit. See [tensorcircuit.gates.oz\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**PHASE**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **PHASE** gate with parameters on the circuit. See [tensorcircuit.gates.phase\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**R**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **R** gate with parameters on the circuit. See [tensorcircuit.gates.r\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RX**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **RX** gate with parameters on the circuit. See [tensorcircuit.gates.rx\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RXX**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **RXX** gate with parameters on the circuit. See [tensorcircuit.gates.rxx\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RY**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **RY** gate with parameters on the circuit. See [tensorcircuit.gates.ry\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RYY**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **RYY** gate with parameters on the circuit. See [tensorcircuit.gates.ryy\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RZ**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **RZ** gate with parameters on the circuit. See [tensorcircuit.gates.rz\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.

- **vars** (*float.*) – Parameters for the gate.

**RZZ**(\**index*: *int*, \*\**vars*: *Any*) → *None*

Apply **RZZ** gate with parameters on the circuit. See [tensorcircuit.gates.rzz\\_gate\(\)](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**S**(\**index*: *int*, *split*: *Optional[Dict[str, Any]] = None*, *name*: *Optional[str] = None*) → *None*

Apply **S** gate on the circuit. See [tensorcircuit.gates.s\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 1.j \end{bmatrix}$$

**SD**(\**index*: *int*, *split*: *Optional[Dict[str, Any]] = None*, *name*: *Optional[str] = None*) → *None*

Apply **SD** gate on the circuit. See [tensorcircuit.gates.sd\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

**SWAP**(\**index*: *int*, *split*: *Optional[Dict[str, Any]] = None*, *name*: *Optional[str] = None*) → *None*

Apply **SWAP** gate on the circuit. See [tensorcircuit.gates.swap\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**T**(\**index*: *int*, *split*: *Optional[Dict[str, Any]] = None*, *name*: *Optional[str] = None*) → *None*

Apply **T** gate on the circuit. See [tensorcircuit.gates.t\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 + 0.70710677j & \end{bmatrix}$$

**TD**(\**index*: *int*, *split*: *Optional[Dict[str, Any]] = None*, *name*: *Optional[str] = None*) → *None*

Apply **TD** gate on the circuit. See [tensorcircuit.gates.td\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

**TOFFOLI**(\**index*: *int*, *split*: *Optional[Dict[str, Any]] = None*, *name*: *Optional[str] = None*) → *None*

Apply **TOFFOLI** gate on the circuit. See [tensorcircuit.gates.toffoli\\_gate\(\)](#).

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`U(*index: int, **vars: Any) → None`

Apply `U` gate with parameters on the circuit. See `tensorcircuit.gates.u_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`WROOT(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `WROOT` gate on the circuit. See `tensorcircuit.gates.wroot_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & -0.5 & -0.5j \\ 0.5 & -0.5j & 0.70710677 + 0.j \end{bmatrix}$$

`X(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `X` gate on the circuit. See `tensorcircuit.gates.x_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j \\ 1. + 0.j & 0. + 0.j \end{bmatrix}$$

`Y(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `Y` gate on the circuit. See `tensorcircuit.gates.y_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j \\ 0. + 1.j & 0. + 0.j \end{bmatrix}$$

`Z(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `Z` gate on the circuit. See `tensorcircuit.gates.z_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j \end{bmatrix}$$

`__init__(nqubits: int, empty: bool = False, inputs: Optional[Any] = None, mps_inputs:`

`Optional[tensorcircuit.quantum.QuOperator] = None, dmininputs: Optional[Any] = None,`  
`mpo_dmininputs: Optional[tensorcircuit.quantum.QuOperator] = None, split: Optional[Dict[str, Any]] = None) → None`

The density matrix simulator based on tensornetwork engine.

## Parameters

- **nqubits** (*int*) – Number of qubits
- **empty** (*bool, optional*) – if True, nothing initialized, only for internal use, defaults to False
- **inputs** (*Optional[Tensor], optional*) – the state input for the circuit, defaults to None
- **mps\_inputs** (*Optional[QuVector]*) – QuVector for a MPS like initial pure state.
- **dminputs** (*Optional[Tensor], optional*) – the density matrix input for the circuit, defaults to None
- **mpo\_dminputs** (*Optional[QuOperator]*) – QuOperator for a MPO like initial density matrix.
- **split** (*Optional[Dict[str, Any]]*) – dict if two qubit gate is ready for split, including parameters for at least one of `max_singular_values` and `max_truncation_err`.

**static all\_zero\_nodes**(*n: int, d: int = 2, prefix: str = 'qb-'*) → List[tensornetwork.network\_components.Node]

**amplitude**(*l: Union[str, Any]*) → Any

Returns the amplitude of the circuit given the bitstring l. For state simulator, it computes  $\langle l|\psi \rangle$ , for density matrix simulator, it computes  $Tr(\rho|l\rangle\langle 1|)$  Note how these two are different up to a square operation.

### Example

```
>>> c = tc.Circuit(2)
>>> c.X(0)
>>> c.amplitude("10")
array(1.+0.j, dtype=complex64)
>>> c.CNOT(0, 1)
>>> c.amplitude("11")
array(1.+0.j, dtype=complex64)
```

**Parameters 1** (*Union[str, Tensor]*) – The bitstring of 0 and 1s.

**Returns** The amplitude of the circuit.

**Return type** tn.Node.tensor

**amplitudedamping**(\**index: int, \*\*vars: float*) → None

Apply amplitudedamping quantum channel on the circuit. See [tensorcircuit.channels.amplitudedampingchannel\(\)](#)

### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the channel.

**any**(\**index: int, \*\*vars: Any*) → None

Apply ANY gate with parameters on the circuit. See [tensorcircuit.gates.any\\_gate\(\)](#).

### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**append**(*c*: tensorcircuit.abstractcircuit.AbstractCircuit, *indices*: *Optional[List[int]]* = *None*) → *tensorcircuit.abstractcircuit.AbstractCircuit*  
 append circuit *c* before

#### Example

```
>>> c1 = tc.Circuit(2)
>>> c1.H(0)
>>> c1.H(1)
>>> c2 = tc.Circuit(2)
>>> c2.cnot(0, 1)
>>> c1.append(c2)
<tensorcircuit.circuit.Circuit object at 0x7f8402968970>
>>> c1.draw()

q_0: H ──
 |
 └──
q_1: H ┌ X ┌
 └── └
```

#### Parameters

- **c** (*BaseCircuit*) – The other circuit to be appended
- **indices** (*Optional[List[int]]*, *optional*) – the qubit indices to which *c* is appended on. Defaults to *None*, which means plain concatenation.

**Returns** The composed circuit

**Return type** *BaseCircuit*

**append\_from\_qir**(*qir*: *List[Dict[str, Any]]*) → *None*

Apply the ciurict in form of quantum intermediate representation after the current cirucit.

#### Example

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.to_qir()
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None,
 → 'mpo': False}]
>>> c2 = tc.Circuit(3)
>>> c2.CNOT(0, 1)
>>> c2.to_qir()
[{'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None,
 → 'mpo': False}]
>>> c.append_from_qir(c2.to_qir())
>>> c.to_qir()
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None,
 → 'mpo': False},
 {'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None,
 → 'mpo': False}]
```

**Parameters** *qir* (*List[Dict[str, Any]]*) – The quantum intermediate representation.

---

**apply**(*gate*: Union[tensorcircuit.gates.Gate, tensorcircuit.quantum.QuOperator], \**index*: int, *name*: Optional[str] = None, *split*: Optional[Dict[str, Any]] = None, *mpo*: bool = False, *ir\_dict*: Optional[Dict[str, Any]] = None) → None  
An implementation of this method should also append gate directionary to self.\_cir

**apply\_general\_gate**(*gate*: Union[tensorcircuit.gates.Gate, tensorcircuit.quantum.QuOperator], \**index*: int, *name*: Optional[str] = None, *split*: Optional[Dict[str, Any]] = None, *mpo*: bool = False, *ir\_dict*: Optional[Dict[str, Any]] = None) → None  
An implementation of this method should also append gate directionary to self.\_cir

**static apply\_general\_gate\_delayed**(*gatef*: Callable[], tensorcircuit.gates.Gate], *name*: Optional[str] = None, *mpo*: bool = False) → Callable[[], None]

**apply\_general\_kraus**(*kraus*: Sequence[tensorcircuit.gates.Gate], *index*: Sequence[Tuple[int, ...]], \*\**kws*: Any) → None

**static apply\_general\_kraus\_delayed**(*krausf*: Callable[[], Sequence[tensorcircuit.gates.Gate]]) → Callable[[], None]

**static apply\_general\_variable\_gate\_delayed**(*gatef*: Callable[[], tensorcircuit.gates.Gate], *name*: Optional[str] = None, *mpo*: bool = False) → Callable[[], None]

**barrier\_instruction**(\**index*: List[int]) → None  
add a barrier instruction flag, no effect on numerical simulation

**Parameters** **index** (List[int]) – the corresponding qubits

**ccnot**(\**index*: int, *split*: Optional[Dict[str, Any]] = None, *name*: Optional[str] = None) → None  
Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**ccx**(\**index*: int, *split*: Optional[Dict[str, Any]] = None, *name*: Optional[str] = None) → None  
Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**static check\_density\_matrix**(*dm*: Any) → None

**static check\_kraus**(*kraus*: Sequence[tensorcircuit.gates.Gate]) → bool

**circuit\_param: Dict[str, Any]**

**cnot(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None**  
Apply CNOT gate on the circuit. See `tensorcircuit.gates.cnot_gate()`.

**Parameters index (int.)** – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

**static coloring\_copied\_nodes(nodes: Sequence[tensornetwork.network\_components.Node], nodes0: Sequence[tensornetwork.network\_components.Node], is\_dagger: bool = True, flag: str = 'inputs') → None**

**static coloring\_nodes(nodes: Sequence[tensornetwork.network\_components.Node], is\_dagger: bool = False, flag: str = 'inputs') → None**

**cond\_measure(index: int, status: Optional[float] = None) → Any**

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a int Tensor and thus maintained a jittable pipeline.

#### Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
two possible outputs: (1, 1) or (-1, -1)
```

---

**Note:** In terms of DMCircuit, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measuremet results

---

**Parameters index (int)** – the qubit for the z-basis measurement

**Returns** 0 or 1 for z measurement on up and down freedom

**Return type** Tensor

**cond\_measurement(index: int, status: Optional[float] = None) → Any**

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a int Tensor and thus maintained a jittable pipeline.

#### Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
two possible outputs: (1, 1) or (-1, -1)
```

---

**Note:** In terms of DMCircuit, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measuremet resuls

---

**Parameters** `index (int)` – the qubit for the z-basis measurement

**Returns** 0 or 1 for z measurement on up and down freedom

**Return type** Tensor

**conditional\_gate**(`which: Any, kraus: Sequence[tensorcircuit.gates.Gate], *index: int`) → None  
Apply which-th gate from kraus list, i.e. apply kraus[which]

**Parameters**

- `which (Tensor)` – Tensor of shape [] and dtype int
- `kraus (Sequence[Gate])` – A list of gate in the form of `tc.gate` or Tensor
- `index (int)` – the qubit lines the gate applied on

**static copy**(`nodes: Sequence[tornetwork.network_components.Node], dangling: Optional[Sequence[tornetwork.network_components.Edge]] = None, conj: Optional[bool] = False`) → Tuple[List[tornetwork.network\_components.Node], List[tornetwork.network\_components.Edge]]  
copy all nodes and dangling edges correspondingly

**Returns**

**cphase**(\*`index: int, **vars: Any`) → None  
Apply CPHASE gate with parameters on the circuit. See `tensorcircuit.gates.cphase_gate()`.

**Parameters**

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

**cr**(\*`index: int, **vars: Any`) → None  
Apply CR gate with parameters on the circuit. See `tensorcircuit.gates.cr_gate()`.

**Parameters**

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

**crx**(\*`index: int, **vars: Any`) → None  
Apply CRX gate with parameters on the circuit. See `tensorcircuit.gates.crx_gate()`.

**Parameters**

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

**cry**(\*`index: int, **vars: Any`) → None  
Apply CRY gate with parameters on the circuit. See `tensorcircuit.gates.cry_gate()`.

**Parameters**

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

**crz**(\*index: int, \*\*vars: Any) → None

Apply **CRZ** gate with parameters on the circuit. See `tensorcircuit.gates.crz_gate()`.

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**cswap**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **FREDKIN** gate on the circuit. See `tensorcircuit.gates.fredkin_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**cu**(\*index: int, \*\*vars: Any) → None

Apply **CU** gate with parameters on the circuit. See `tensorcircuit.gates.cu_gate()`.

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**cx**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **CNOT** gate on the circuit. See `tensorcircuit.gates.cnot_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

**cy**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **CY** gate on the circuit. See `tensorcircuit.gates.cy_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. - 1.j \\ 0. + 0.j & 0. + 0.j & 0. + 1.j & 0. + 0.j \end{bmatrix}$$

**cz**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **CZ** gate on the circuit. See `tensorcircuit.gates.cz_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & -1. + 0.j \end{bmatrix}$$

**densitymatrix**(*check: bool = False, reuse: bool = True*) → Any

Return the output density matrix of the circuit.

#### Parameters

- **check (bool, optional)** – check whether the final return is a legal density matrix, defaults to False
- **reuse (bool, optional)** – whether to reuse previous results, defaults to True

**Returns** The output densitymatrix in 2D shape tensor form

**Return type** Tensor

**depolarizing**(\**index: int, \*\*vars: float*) → None

Apply depolarizing quantum channel on the circuit. See [tensorcircuit.channels.depolarizingchannel\(\)](#).

#### Parameters

- **index (int.)** – Qubit number that the gate applies on.
- **vars (float.)** – Parameters for the channel.

**draw(\*\*kws: Any)** → Any

Visualise the circuit. This method receives the keywords as same as qiskit.circuit.QuantumCircuit.draw. More details can be found here: <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.draw.html>.

#### Example

```
>>> c = tc.Circuit(3)
>>> c.H(1)
>>> c.X(2)
>>> c.CNOT(0, 1)
>>> c.draw(output='text')
q_0: _____
 |
q_1: H | X |
 | |
 | |
q_2: X | _____
```

**exp(\**index: int, \*\*vars: Any*) → None**

Apply EXP gate with parameters on the circuit. See [tensorcircuit.gates.exp\\_gate\(\)](#).

#### Parameters

- **index (int.)** – Qubit number that the gate applies on.
- **vars (float.)** – Parameters for the gate.

**exp1(\**index: int, \*\*vars: Any*) → None**

Apply EXP1 gate with parameters on the circuit. See [tensorcircuit.gates.exp1\\_gate\(\)](#).

#### Parameters

- **index (int.)** – Qubit number that the gate applies on.
- **vars (float.)** – Parameters for the gate.

**expectation**(\*ops: *Tuple[tensornetwork.network\_components.Node, List[int]]*, reuse: *bool* = True, noise\_conf: *Optional[Any]* = None, status: *Optional[Any]* = None, \*\*kws: *Any*) → <property object at 0x7f72990543b0>

Compute the expectation of corresponding operators.

#### Parameters

- **ops** (*Tuple[tn.Node, List[int]]*) – Operator and its position on the circuit, eg. (*tc.gates.z()*, [1, ]), (*tc.gates.x()*, [2, ]) is for operator  $Z_1X_2$ .
- **reuse** (*bool*) – whether contract the density matrix in advance, defaults to True
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **status** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** Tensor with one element

**Return type** Tensor

**expectation\_before**(\*ops: *Tuple[tensornetwork.network\_components.Node, List[int]]*, reuse: *bool* = True, \*\*kws: *Any*) → *List[tensornetwork.network\_components.Node]*

Get the tensor network in the form of a list of nodes for the expectation calculation before the real contraction

**Parameters** **reuse** (*bool*, *optional*) – \_description\_, defaults to True

**Raises** **ValueError** – \_description\_

**Returns** \_description\_

**Return type** *List[tn.Node]*

**expectation\_ps**(*x: Optional[Sequence[int]*] = None, *y: Optional[Sequence[int]*] = None, *z: Optional[Sequence[int]*] = None, *reuse: bool* = True, *noise\_conf: Optional[Any]* = None, *nmc: int* = 1000, *status: Optional[Any]* = None, \*\*kws: *Any*) → *Any*

Shortcut for Pauli string expectation. x, y, z list are for X, Y, Z positions

#### Example

```
>>> c = tc.Circuit(2)
>>> c.X(0)
>>> c.H(1)
>>> c.expectation_ps(x=[1], z=[0])
array(-0.99999994+0.j, dtype=complex64)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
```

(continues on next page)

(continued from previous page)

```
>>> c.expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
(0.46274087-3.764033e-09j)
```

**Parameters**

- **x** (*Optional[Sequence[int]]*, *optional*) – sites to apply X gate, defaults to None
- **y** (*Optional[Sequence[int]]*, *optional*) – sites to apply Y gate, defaults to None
- **z** (*Optional[Sequence[int]]*, *optional*) – sites to apply Z gate, defaults to None
- **reuse** (*bool*, *optional*) – whether to cache and reuse the wavefunction, defaults to True
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **nmc** (*int*, *optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **status** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** Expectation value**Return type** Tensor

**fredkin**(\*index: *int*, split: *Optional[Dict[str, Any]]* = None, name: *Optional[str]* = None) → None  
 Apply FREDKIN gate on the circuit. See `tensorcircuit.gates.fredkin_gate()`.

**Parameters** **index** (*int*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**classmethod** **from\_json**(jsonstr: *str*, circuit\_params: *Optional[Dict[str, Any]]* = None) → *tensorcircuit.abstractcircuit.AbstractCircuit*

load json str as a Circuit

**Parameters**

- **jsonstr** (*str*) – \_description\_
- **circuit\_params** (*Optional[Dict[str, Any]]*, *optional*) – Extra circuit parameters in the format of `__init__`, defaults to None

**Returns** \_description\_**Return type** *AbstractCircuit*

---

```
classmethod from_json_file(file: str, circuit_params: Optional[Dict[str, Any]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
load json file and convert it to a circuit

Parameters

- file (str) – filename
- circuit_params (Optional[Dict[str, Any]], optional) – _description_,
defaults to None

Returns _description_
Return type AbstractCircuit
```

```
classmethod from_openqasm(qasmstr: str, circuit_params: Optional[Dict[str, Any]] = None,
 keep_measure_order: bool = False) →
 tensorcircuit.abstractcircuit.AbstractCircuit

classmethod from_openqasm_file(file: str, circuit_params: Optional[Dict[str, Any]] = None,
 keep_measure_order: bool = False) →
 tensorcircuit.abstractcircuit.AbstractCircuit

classmethod from_qir(qir: List[Dict[str, Any]], circuit_params: Optional[Dict[str, Any]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
```

Restore the circuit from the quantum intermediate representation.

### Example

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.rx(1, theta=tc.array_to_tensor(0.7))
>>> c.exp1(0, 1, unitary=tc.gates._zz_matrix, theta=tc.array_to_tensor(-0.2),
→split=split)
>>> len(c)
7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)
>>> qirs = c.to_qir()
>>>
>>> c = tc.Circuit.from_qir(qirs, circuit_params={"nqubits": 3})
>>> len(c._nodes)
7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)
```

### Parameters

- **qir** (List[Dict[str, Any]]) – The quantum intermediate representation of a circuit.
- **circuit\_params** (Optional[Dict[str, Any]]) – Extra circuit parameters.

**Returns** The circuit have same gates in the qir.

**Return type** *Circuit*

```
classmethod from_qiskit(qc: Any, n: Optional[int] = None, inputs: Optional[List[float]] = None,
 circuit_params: Optional[Dict[str, Any]] = None, binding_params:
 Optional[Union[Sequence[float], Dict[Any, float]]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
```

Import Qiskit QuantumCircuit object as a tc.Circuit object.

#### Example

```
>>> from qiskit import QuantumCircuit
>>> qisc = QuantumCircuit(3)
>>> qisc.h(2)
>>> qisc.cswap(1, 2, 0)
>>> qisc.swap(0, 1)
>>> c = tc.Circuit.from_qiskit(qisc)
```

#### Parameters

- **qc** (*QuantumCircuit in Qiskit*) – Qiskit Circuit object
- **n** (*int*) – The number of qubits for the circuit
- **inputs** (*Optional[List[float]]*, *optional*) – possible input wavefunction for tc.Circuit, defaults to None
- **circuit\_params** (*Optional[Dict[str, Any]]*) – kwargs given in Circuit.*\_\_init\_\_* construction function, default to None.
- **binding\_params** (*Optional[Union[Sequence[float], Dict[Any, float]]]*) – (variational) parameters for the circuit. Could be either a sequence or dictionary depending on the type of parameters in the Qiskit circuit. For ParameterVectorElement use sequence. For Parameter use dictionary

**Returns** The same circuit but as tensorcircuit object

**Return type** *Circuit*

```
classmethod from_qsim_file(file: str, circuit_params: Optional[Dict[str, Any]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
```

```
static front_from_nodes(nodes: List[tensornetwork.network_components.Node]) →
 List[tensornetwork.network_components.Edge]
```

```
gate_aliases = [['cnot', 'cx'], ['fredkin', 'cswap'], ['toffoli', 'ccnot'],
['toffoli', 'ccx'], ['any', 'unitary'], ['sd', 'sgd'], ['td', 'tdg']]
```

```
gate_count(gate_list: Optional[Sequence[str]] = None) → int
count the gate number of the circuit
```

#### Example

```
>>> c = tc.Circuit(3)
>>> c.h(0)
>>> c.multicontrol(0, 1, 2, ctrl=[0, 1], unitary=tc.gates._x_matrix)
>>> c.toffoli(1, 2, 0)
>>> c.gate_count()
3
>>> c.gate_count(["multicontrol", "toffoli"])
2
```

**Parameters** `gate_list` (*Optional[Sequence[str]]*, *optional*) – gate name list to be counted, defaults to None (counting all gates)

**Returns** the total number of all gates or gates in the `gate_list`

**Return type** int

`gate_summary()` → Dict[str, int]

return the summary dictionary on gate type - gate count pair

**Returns** the gate count dict by gate type

**Return type** Dict[str, int]

`general_kraus(kraus: Sequence[tensorcircuit.gates.Gate], index: Sequence[Tuple[int, ...]], **kws: Any)` → None

`generaldepolarizing(*index: int, **vars: float)` → None

Apply generaldepolarizing quantum channel on the circuit. See `tensorcircuit.channels.generaldepolarizingchannel()`

**Parameters**

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the channel.

`get_dm_as_quoperator()` → `tensorcircuit.quantum.QuOperator`

Get the representation of the output state in the form of QuOperator while maintaining the circuit uncomputed

**Returns** QuOperator representation of the output state from the circuit

**Return type** QuOperator

`get_dm_as_quvector()` → `tensorcircuit.quantum.QuVector`

Get the representation of the output state in the form of QuVector while maintaining the circuit uncomputed

**Returns** QuVector representation of the output state from the circuit

**Return type** QuVector

`get_positional_logical_mapping()` → Dict[int, int]

Get positional logical mapping dict based on measure instruction. This function is useful when we only measure part of the qubits in the circuit, to process the count result from partial measurement, we must be aware of the mapping, i.e. for each position in the count bitstring, what is the corresponding qubits (logical) defined on the circuit

**Returns** positional\_logical\_mapping

**Return type** Dict[int, int]

`get_quvector()` → `tensorcircuit.quantum.QuVector`

Get the representation of the output state in the form of QuVector while maintaining the circuit uncomputed

**Returns** QuVector representation of the output state from the circuit

**Return type** QuVector

`h(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None)` → None

Apply H gate on the circuit. See `tensorcircuit.gates.h_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & 0.70710677 + 0.j \\ 0.70710677 + 0.j & -0.70710677 + 0.j \end{bmatrix}$$

`i(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **I** gate on the circuit. See `tensorcircuit.gates.i_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`initial_mapping(logical_physical_mapping: Dict[int, int], n: Optional[int] = None, circuit_params:`

`Optional[Dict[str, Any]] = None) → tensorcircuit.abstractcircuit.AbstractCircuit`

generate a new circuit with the qubit mapping given by `logical_physical_mapping`

#### Parameters

- `logical_physical_mapping (Dict[int, int])` – how to map logical qubits to the physical qubits on the new circuit
- `n (Optional[int], optional)` – number of qubit of the new circuit, can be different from the original one, defaults to None
- `circuit_params (Optional[Dict[str, Any]], optional)` – `_description_`, defaults to None

**Returns** `_description_`

**Return type** `AbstractCircuit`

**inputs:** `Any`

`inverse(circuit_params: Optional[Dict[str, Any]] = None) → tensorcircuit.abstractcircuit.AbstractCircuit`

inverse the circuit, return a new inversed circuit

#### EXAMPLE

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rzz(1, 2, theta=0.8)
>>> c1 = c.inverse()
```

**Parameters** `circuit_params (Optional[Dict[str, Any]], optional)` – keywords dict for initialization the new circuit, defaults to None

**Returns** the inversed circuit

**Return type** `Circuit`

`is_dm: bool = True`

`is_mps: bool = False`

`iswap(*index: int, **vars: Any) → None`

Apply **ISWAP** gate with parameters on the circuit. See `tensorcircuit.gates.iswap_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.

- **vars** (*float.*) – Parameters for the gate.

**measure**(\**index*: int, *with\_prob*: bool = False, *status*: Optional[Any] = None) → Tuple[Any, Any]  
 Take measurement to the given quantum lines. This method is jittable is and about 100 times faster than unjit version!

#### Parameters

- **index** (int) – Measure on which quantum line.
- **with\_prob** (bool, optional) – If true, theoretical probability is also returned.
- **status** (Optional[Tensor]) – external randomness, with shape [index], defaults to None

**Returns** The sample output and probability (optional) of the quantum line.

**Return type** Tuple[Tensor, Tensor]

**measure\_instruction**(*index*: int) → None  
 add a measurement instruction flag, no effect on numerical simulation

#### Parameters **index** (int) – the corresponding qubit

**measure\_jit**(\**index*: int, *with\_prob*: bool = False, *status*: Optional[Any] = None) → Tuple[Any, Any]  
 Take measurement to the given quantum lines. This method is jittable is and about 100 times faster than unjit version!

#### Parameters

- **index** (int) – Measure on which quantum line.
- **with\_prob** (bool, optional) – If true, theoretical probability is also returned.
- **status** (Optional[Tensor]) – external randomness, with shape [index], defaults to None

**Returns** The sample output and probability (optional) of the quantum line.

**Return type** Tuple[Tensor, Tensor]

**mpo**(\**index*: int, \*\**vars*: Any) → None  
 Apply mpo gate in MPO format on the circuit. See [tensorcircuit.gates.mpo\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**mpogates** = ['multicontrol', 'mpo']

**multicontrol**(\**index*: int, \*\**vars*: Any) → None  
 Apply multicontrol gate in MPO format on the circuit. See [tensorcircuit.gates.multicontrol\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**orx**(\**index*: int, \*\**vars*: Any) → None

Apply ORX gate with parameters on the circuit. See [tensorcircuit.gates.orx\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.

- **vars** (*float.*) – Parameters for the gate.

**ory**(\**index*: *int*, \*\**vars*: *Any*) → *None*

Apply **ORY** gate with parameters on the circuit. See `tensorcircuit.gates.ory_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**orz**(\**index*: *int*, \*\**vars*: *Any*) → *None*

Apply **ORZ** gate with parameters on the circuit. See `tensorcircuit.gates.orz_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**ox**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*

Apply **OX** gate on the circuit. See `tensorcircuit.gates.ox_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**oy**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*

Apply **OY** gate on the circuit. See `tensorcircuit.gates.oy_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j & 0. + 0.j & 0. + 0.j \\ 0. + 1.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**oz**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*

Apply **OZ** gate on the circuit. See `tensorcircuit.gates.oz_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**perfect\_sampling**(*status*: *Optional[Any]* = *None*) → *Tuple[str, float]*

Sampling bistrings from the circuit output based on quantum amplitudes. Reference: arXiv:1201.3974.

**Parameters** **status** (*Optional[Tensor]*) – external randomness, with shape [nqubits], defaults to None

**Returns** Sampled bit string and the corresponding theoretical probability.

**Return type** *Tuple[str, float]*

**phase**(\**index*: *int*, \*\**vars*: *Any*) → *None*

Apply **PHASE** gate with parameters on the circuit. See `tensorcircuit.gates.phase_gate()`.

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**phasedamping**(\**index: int*, \*\**vars: float*) → NoneApply phasedamping quantum channel on the circuit. See [tensorcircuit.channels.phasedampingchannel\(\)](#).**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the channel.

**prepend**(*c: tensorcircuit.abstractcircuit.AbstractCircuit*) → *tensorcircuit.abstractcircuit.AbstractCircuit*prepend circuit *c* before**Parameters** *c* ([BaseCircuit](#)) – The other circuit to be prepended**Returns** The composed circuit**Return type** [BaseCircuit](#)**probability()** → Anyget the  $2^n$  length probability vector over computational basis**Returns** probability vector**Return type** Tensor**quvector()** → [tensorcircuit.quantum.QuVector](#)

Get the representation of the output state in the form of QuVector while maintaining the circuit uncomputed

**Returns** QuVector representation of the output state from the circuit**Return type** [QuVector](#)**r(\*index: int, \*\*vars: Any)** → NoneApply R gate with parameters on the circuit. See [tensorcircuit.gates.r\\_gate\(\)](#).**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**readouterror\_bs**(*readout\_error: Optional[Sequence[Any]] = None, p: Optional[Any] = None*) → Any

Apply readout error to original probabilities of bit string and return the noisy probabilities.

**Example**

```
>>> readout_error = []
>>> readout_error.append([0.9, 0.75]) # readout error for qubit 0, [p0/0, p1/1]
>>> readout_error.append([0.4, 0.7]) # readout error for qubit 1, [p0/0, p1/1]
```

**Parameters**

- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – list of readout error for each qubits.
- **p** (*Optional[Any]*) – probabilities of bit string

**Return type** Tensor

**replace\_inputs**(*inputs*: Any) → None

Replace the input state with the circuit structure unchanged.

**Parameters** **inputs** (Tensor) – Input wavefunction.**reset**(\**index*: int, \*\**vars*: float) → NoneApply reset quantum channel on the circuit. See [tensorcircuit.channels.resetchannel\(\)](#)**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the channel.

**reset\_instruction**(*index*: int) → None

add a reset instruction flag, no effect on numerical simulation

**Parameters** **index** (int) – the corresponding qubit**rx**(\**index*: int, \*\**vars*: Any) → NoneApply RX gate with parameters on the circuit. See [tensorcircuit.gates.rx\\_gate\(\)](#).**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**rxx**(\**index*: int, \*\**vars*: Any) → NoneApply RXX gate with parameters on the circuit. See [tensorcircuit.gates.rxx\\_gate\(\)](#).**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**ry**(\**index*: int, \*\**vars*: Any) → NoneApply RY gate with parameters on the circuit. See [tensorcircuit.gates.ry\\_gate\(\)](#).**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**ryy**(\**index*: int, \*\**vars*: Any) → NoneApply RYY gate with parameters on the circuit. See [tensorcircuit.gates.ryy\\_gate\(\)](#).**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**rz**(\**index*: int, \*\**vars*: Any) → NoneApply RZ gate with parameters on the circuit. See [tensorcircuit.gates.rz\\_gate\(\)](#).**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**rzz**(\**index*: int, \*\**vars*: Any) → NoneApply RZZ gate with parameters on the circuit. See [tensorcircuit.gates.rzz\\_gate\(\)](#).**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.

- **vars** (*float.*) – Parameters for the gate.

**s**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None  
Apply S gate on the circuit. See `tensorcircuit.gates.s_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 1.j \end{bmatrix}$$

**sample**(*batch: Optional[int] = None, allow\_state: bool = False, readout\_error: Optional[Sequence[Any]] = None, format: Optional[str] = None, random\_generator: Optional[Any] = None, status: Optional[Any] = None*) → Any  
batched sampling from state or circuit tensor network directly

**Parameters**

- **batch** (*Optional[int]*, *optional*) – number of samples, defaults to None
- **allow\_state** (*bool*, *optional*) – if true, we sample from the final state if memory allows, True is preferred, defaults to False
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to None
- **format** (*Optional[str]*) – sample format, defaults to None as backward compatibility check the doc in `tensorcircuit.quantum.measurement_results()`
- **format** – alias for the argument `format`
- **random\_generator** (*Optional[Any]*, *optional*) – random generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random\_generator

**Returns** List (if batch) of tuple (binary configuration tensor and corresponding probability) if the format is None, and consistent with format when given

**Return type** Any

**sample\_expectation\_ps**(*x: Optional[Sequence[int]] = None, y: Optional[Sequence[int]] = None, z: Optional[Sequence[int]] = None, shots: Optional[int] = None, random\_generator: Optional[Any] = None, status: Optional[Any] = None, readout\_error: Optional[Sequence[Any]] = None, noise\_conf: Optional[Any] = None, nmc: int = 1000, statusc: Optional[Any] = None, \*\*kws: Any*) → Any  
Compute the expectation with given Pauli string with measurement shots numbers

**Example**

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rx(1, theta=np.pi/2)
>>> c.sample_expectation_ps(x=[0], y=[1])
-0.99999976
>>> readout_error = []
>>> readout_error.append([0.9, 0.75])
>>> readout_error.append([0.4, 0.7])
>>> c.sample_expectation_ps(x=[0], y=[1], readout_error = readout_error)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> readout_error = [[0.9, 0.75], [0.4, 0.7]]
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> noise_conf.add_noise("readout", readout_error)
>>> c.sample_expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
0.44766843
```

**Parameters**

- **x** (*Optional[Sequence[int]]*, *optional*) – index for Pauli X, defaults to None
- **y** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Y, defaults to None
- **z** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Z, defaults to None
- **shots** (*Optional[int]*, *optional*) – number of measurement shots, defaults to None, indicating analytical result
- **random\_generator** (*Optional[Any]*) – random\_generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random\_generator
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to None. Overrided if noise\_conf is provided.
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **nmc** (*int, optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **statusc** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** [description]**Return type** Tensor**sd**(\*index: int, split: *Optional[Dict[str, Any]]* = None, name: *Optional[str]* = None) → NoneApply **SD** gate on the circuit. See `tensorcircuit.gates.sd_gate()`.**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

**sdg**(\*index: int, split: *Optional[Dict[str, Any]]* = None, name: *Optional[str]* = None) → NoneApply **SD** gate on the circuit. See `tensorcircuit.gates.sd_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

`select_gate(which: Any, kraus: Sequence[tensorcircuit.gates.Gate], *index: int) → None`

Apply which-th gate from kraus list, i.e. apply kraus[which]

#### Parameters

- `which (Tensor)` – Tensor of shape [] and dtype int
- `kraus (Sequence[Gate])` – A list of gate in the form of `tc.gate` or `Tensor`
- `index (int)` – the qubit lines the gate applied on

`sexps(x: Optional[Sequence[int]] = None, y: Optional[Sequence[int]] = None, z: Optional[Sequence[int]] = None, shots: Optional[int] = None, random_generator: Optional[Any] = None, status: Optional[Any] = None, readout_error: Optional[Sequence[Any]] = None, noise_conf: Optional[Any] = None, nmc: int = 1000, statusc: Optional[Any] = None, **kws: Any) → Any`

Compute the expectation with given Pauli string with measurement shots numbers

#### Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rx(1, theta=np.pi/2)
>>> c.sample_expectation_ps(x=[0], y=[1])
-0.99999976
>>> readout_error = []
>>> readout_error.append([0.9, 0.75])
>>> readout_error.append([0.4, 0.7])
>>> c.sample_expectation_ps(x=[0], y=[1], readout_error = readout_error)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> readout_error = [[0.9, 0.75], [0.4, 0.7]]
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> noise_conf.add_noise("readout", readout_error)
>>> c.sample_expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
0.44766843
```

#### Parameters

- `x (Optional[Sequence[int]], optional)` – index for Pauli X, defaults to None
- `y (Optional[Sequence[int]], optional)` – index for Pauli Y, defaults to None
- `z (Optional[Sequence[int]], optional)` – index for Pauli Z, defaults to None

- **shots** (*Optional[int]*, *optional*) – number of measurement shots, defaults to None, indicating analytical result
- **random\_generator** (*Optional[Any]*) – random\_generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random\_generator
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to None. Overrided if noise\_conf is provided.
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **nmc** (*int*, *optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **statussc** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** [description]

**Return type** Tensor

```
sgates = ['i', 'x', 'y', 'z', 'h', 't', 's', 'td', 'sd', 'wroot', 'cnot', 'cz',
'swap', 'cy', 'ox', 'oy', 'oz', 'toffoli', 'fredkin']
```

```
split: Optional[Dict[str, Any]]
```

```
static standardize_gate(name: str) → str
```

standardize the gate name to common gate sets

**Parameters** **name** (*str*) – non-standard gate name

**Returns** the standard gate name

**Return type** str

```
state(check: bool = False, reuse: bool = True) → Any
```

Return the output density matrix of the circuit.

**Parameters**

- **check** (*bool*, *optional*) – check whether the final return is a legal density matrix, defaults to False
- **reuse** (*bool*, *optional*) – whether to reuse previous results, defaults to True

**Returns** The output densitymatrix in 2D shape tensor form

**Return type** Tensor

```
swap(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None
```

Apply SWAP gate on the circuit. See `tensorcircuit.gates.swap_gate()`.

**Parameters** **index** (*int*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

```
t(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None
```

Apply T gate on the circuit. See `tensorcircuit.gates.t_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 + 0.70710677j & \end{bmatrix}$$

`td(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **TD** gate on the circuit. See `tensorcircuit.gates.td_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

`tdg(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **TD** gate on the circuit. See `tensorcircuit.gates.td_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

`tex(**kws: Any) → str`

Generate latex string based on quantikz latex package

**Returns** Latex string that can be directly compiled via, e.g. `lataexit`

**Return type** `str`

`thermalrelaxation(*index: int, **vars: float) → None`

Apply thermalrelaxation quantum channel on the circuit. See `tensorcircuit.channels.thermalrelaxationchannel()`

**Parameters**

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the channel.

`to_circuit(circuit_params: Optional[Dict[str, Any]] = None) → tensorcircuit.circuit.Circuit`

convert into state simulator (current implementation ignores all noise channels)

**Parameters** `circuit_params (Optional[Dict[str, Any]], optional)` – kws to initialize circuit object, defaults to None

**Returns** Circuit with no noise

**Return type** `Circuit`

`to_cirq(enable_instruction: bool = False) → Any`

Translate `tc.Circuit` to a cirq circuit object.

**Parameters** `enable_instruction (bool, defaults to False)` – whether also export measurement and reset instructions

**Returns** A cirq circuit of this circuit.

`to_graphviz(graph: Optional[graphviz.graphs.Graph] = None, include_all_names: bool = False, engine: str = 'neato') → graphviz.graphs.Graph`

Not an ideal visualization for quantum circuit, but reserve here as a general approach to show the tensor-network [Deprecated, use `Circuit.vis_tex` or `Circuit.draw` instead]

`to_json(file: Optional[str] = None, simplified: bool = False) → Any`

circuit dumps to json

**Parameters**

- **file** (*Optional[str], optional*) – file str to dump the json to, defaults to None, return the json str
- **simplified** (*bool*) – If False, keep all info for each gate, defaults to be False. If True, suitable for IO since less information is required

**Returns** None if dumps to file otherwise the json str

**Return type** Any

**to\_openqasm**(*\*\*kws: Any*) → str

transform circuit to openqasm via qiskit circuit, see <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.qasm.html> for usage on possible options for **kws**

**Returns** circuit representation in openqasm format

**Return type** str

**to\_qir()** → List[Dict[str, Any]]

Return the quantum intermediate representation of the circuit.

**Example**

```
>>> c = tc.Circuit(2)
>>> c.CNOT(0, 1)
>>> c.to_qir()
[{'gatef': cnot, 'gate': Gate(
 name: 'cnot',
 tensor:
 array([[[[1.+0.j, 0.+0.j],
 [0.+0.j, 0.+0.j]],
 [[[0.+0.j, 1.+0.j],
 [0.+0.j, 0.+0.j]],
 [[[0.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]],
 [[[0.+0.j, 0.+0.j],
 [1.+0.j, 0.+0.j]]]], dtype=complex64),
 edges: [
 Edge(Dangling Edge)[0],
 Edge(Dangling Edge)[1],
 Edge('cnot'[2] -> 'qb-1'[0]),
 Edge('cnot'[3] -> 'qb-2'[0])
], 'index': (0, 1), 'name': 'cnot', 'split': None, 'mpo': False}]}
```

**Returns** The quantum intermediate representation of the circuit.

**Return type** List[Dict[str, Any]]

**to\_qiskit**(*enable\_instruction: bool = False*) → Any

Translate `tc.Circuit` to a qiskit QuantumCircuit object.

**Parameters** **enable\_instruction** (*bool, defaults to False*) – whether also export measurement and reset instructions

**Returns** A qiskit object of this circuit.

**toffoli**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None  
Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

**u**(\*index: int, \*\*vars: Any) → None  
Apply **U** gate with parameters on the circuit. See `tensorcircuit.gates.u_gate()`.

**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**unitary**(\*index: int, \*\*vars: Any) → None  
Apply **ANY** gate with parameters on the circuit. See `tensorcircuit.gates.any_gate()`.

**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**vgates** = ['r', 'cr', 'u', 'cu', 'rx', 'ry', 'rz', 'phase', 'rxx', 'ryy', 'rzz', 'cphase', 'crx', 'cry', 'crz', 'orx', 'ory', 'orz', 'iswap', 'any', 'exp', 'exp1']

**vis\_tex**(\*\*kws: Any) → str  
Generate latex string based on quantikz latex package

**Returns** Latex string that can be directly compiled via, e.g. `latticeit`

**Return type** str

**wavefunction**() → Any

get the wavefunction of outputs, raise error if the final state is not purified [Experimental: the phase factor is not fixed for different backend]

**Returns** wavefunction vector

**Return type** Tensor

**wroot**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None  
Apply **WROOT** gate on the circuit. See `tensorcircuit.gates.wroot_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & -0.5 & -0.5j \\ 0.5 & -0.5j & 0.70710677 + 0.j \end{bmatrix}$$

**x**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None  
Apply **X** gate on the circuit. See `tensorcircuit.gates.x_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j \\ 1. + 0.j & 0. + 0.j \end{bmatrix}$$

`y(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **Y** gate on the circuit. See `tensorcircuit.gates.y_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j \\ 0. + 1.j & 0. + 0.j \end{bmatrix}$$

`z(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **Z** gate on the circuit. See `tensorcircuit.gates.z_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j \end{bmatrix}$$

`class tensorcircuit.densitymatrix.DMCircuit2(nqubits: int, empty: bool = False, inputs: Optional[Any] = None, mps_inputs: Optional[tensorcircuit.quantum.QuOperator] = None, dminputs: Optional[Any] = None, mpo_dminputs: Optional[tensorcircuit.quantum.QuOperator] = None, split: Optional[Dict[str, Any]] = None)`

Bases: `tensorcircuit.densitymatrix.DMCircuit`

`ANY(*index: int, **vars: Any) → None`

Apply **ANY** gate with parameters on the circuit. See `tensorcircuit.gates.any_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`CNOT(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **CNOT** gate on the circuit. See `tensorcircuit.gates.cnot_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

`CPHASE(*index: int, **vars: Any) → None`

Apply **CPHASE** gate with parameters on the circuit. See `tensorcircuit.gates.cphase_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`CR(*index: int, **vars: Any) → None`

Apply **CR** gate with parameters on the circuit. See `tensorcircuit.gates.cr_gate()`.

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**CRX**(\**index*: *int*, \*\**vars*: *Any*) → NoneApply **CRX** gate with parameters on the circuit. See `tensorcircuit.gates.crx_gate()`.**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**CRY**(\**index*: *int*, \*\**vars*: *Any*) → NoneApply **CRY** gate with parameters on the circuit. See `tensorcircuit.gates.cry_gate()`.**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**CRZ**(\**index*: *int*, \*\**vars*: *Any*) → NoneApply **CRZ** gate with parameters on the circuit. See `tensorcircuit.gates.crz_gate()`.**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**CU**(\**index*: *int*, \*\**vars*: *Any*) → NoneApply **CU** gate with parameters on the circuit. See `tensorcircuit.gates.cu_gate()`.**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**CY**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → NoneApply **CY** gate on the circuit. See `tensorcircuit.gates.cy_gate()`.**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. - 1.j \\ 0. + 0.j & 0. + 0.j & 0. + 1.j & 0. + 0.j \end{bmatrix}$$

**CZ**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → NoneApply **CZ** gate on the circuit. See `tensorcircuit.gates.cz_gate()`.**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & -1. + 0.j \end{bmatrix}$$

**EXP**(\**index*: *int*, \*\**vars*: *Any*) → NoneApply **EXP** gate with parameters on the circuit. See `tensorcircuit.gates.exp_gate()`.

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**EXP1**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **EXP1** gate with parameters on the circuit. See [tensorcircuit.gates.exp1\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**FREDKIN**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **FREDKIN** gate on the circuit. See [tensorcircuit.gates.fredkin\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**H**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **H** gate on the circuit. See [tensorcircuit.gates.h\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & 0.70710677 + 0.j \\ 0.70710677 + 0.j & -0.70710677 + 0.j \end{bmatrix}$$

**I**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **I** gate on the circuit. See [tensorcircuit.gates.i\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**ISWAP**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **ISWAP** gate with parameters on the circuit. See [tensorcircuit.gates.iswap\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**MPO**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply mpo gate in MPO format on the circuit. See [tensorcircuit.gates.mpo\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**MULTICONTROL**(\*index: int, \*\*vars: Any) → None

Apply multicontrol gate in MPO format on the circuit. See [tensorcircuit.gates.multicontrol\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**ORX**(\*index: int, \*\*vars: Any) → None

Apply **ORX** gate with parameters on the circuit. See [tensorcircuit.gates.orx\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**ORY**(\*index: int, \*\*vars: Any) → None

Apply **ORY** gate with parameters on the circuit. See [tensorcircuit.gates.ory\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**ORZ**(\*index: int, \*\*vars: Any) → None

Apply **ORZ** gate with parameters on the circuit. See [tensorcircuit.gates.orz\\_gate\(\)](#).

#### Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**OX**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **OX** gate on the circuit. See [tensorcircuit.gates.ox\\_gate\(\)](#).

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**OY**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **OY** gate on the circuit. See [tensorcircuit.gates.oy\\_gate\(\)](#).

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j & 0. + 0.j & 0. + 0.j \\ 0. + 1.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**OZ**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **OZ** gate on the circuit. See [tensorcircuit.gates.oz\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**PHASE**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **PHASE** gate with parameters on the circuit. See [\*tensorcircuit.gates.phase\\_gate\(\)\*](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**R**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **R** gate with parameters on the circuit. See [\*tensorcircuit.gates.r\\_gate\(\)\*](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RX**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **RX** gate with parameters on the circuit. See [\*tensorcircuit.gates.rx\\_gate\(\)\*](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RXX**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **RXX** gate with parameters on the circuit. See [\*tensorcircuit.gates.rxx\\_gate\(\)\*](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RY**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **RY** gate with parameters on the circuit. See [\*tensorcircuit.gates.ry\\_gate\(\)\*](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RYY**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **RYY** gate with parameters on the circuit. See [\*tensorcircuit.gates.ryy\\_gate\(\)\*](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**RZ**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **RZ** gate with parameters on the circuit. See [\*tensorcircuit.gates.rz\\_gate\(\)\*](#).

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.

- **vars** (*float.*) – Parameters for the gate.

**RZZ**(\**index*: *int*, \*\**vars*: *Any*) → None

Apply **RZZ** gate with parameters on the circuit. See `tensorcircuit.gates.rzz_gate()`.

#### Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**S**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **S** gate on the circuit. See `tensorcircuit.gates.s_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 1.j \end{bmatrix}$$

**SD**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **SD** gate on the circuit. See `tensorcircuit.gates.sd_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

**SWAP**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **SWAP** gate on the circuit. See `tensorcircuit.gates.swap_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**T**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **T** gate on the circuit. See `tensorcircuit.gates.t_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 + 0.70710677j & \end{bmatrix}$$

**TD**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **TD** gate on the circuit. See `tensorcircuit.gates.td_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

**TOFFOLI**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`U(*index: int, **vars: Any) → None`

Apply `U` gate with parameters on the circuit. See `tensorcircuit.gates.u_gate()`.

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`WROOT(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `WROOT` gate on the circuit. See `tensorcircuit.gates.wroot_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & -0.5 & -0.5j \\ 0.5 & -0.5j & 0.70710677 + 0.j \end{bmatrix}$$

`X(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `X` gate on the circuit. See `tensorcircuit.gates.x_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j \\ 1. + 0.j & 0. + 0.j \end{bmatrix}$$

`Y(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `Y` gate on the circuit. See `tensorcircuit.gates.y_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j \\ 0. + 1.j & 0. + 0.j \end{bmatrix}$$

`Z(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply `Z` gate on the circuit. See `tensorcircuit.gates.z_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j \end{bmatrix}$$

`__init__(nqubits: int, empty: bool = False, inputs: Optional[Any] = None, mps_inputs:`

`Optional[tensorcircuit.quantum.QuOperator] = None, dmininputs: Optional[Any] = None,`  
`mpo_dmininputs: Optional[tensorcircuit.quantum.QuOperator] = None, split: Optional[Dict[str, Any]] = None) → None`

The density matrix simulator based on tensornetwork engine.

**Parameters**

- **nqubits** (*int*) – Number of qubits
- **empty** (*bool, optional*) – if True, nothing initialized, only for internal use, defaults to False
- **inputs** (*Optional[Tensor], optional*) – the state input for the circuit, defaults to None
- **mps\_inputs** (*Optional[QuVector]*) – QuVector for a MPS like initial pure state.
- **dminputs** (*Optional[Tensor], optional*) – the density matrix input for the circuit, defaults to None
- **mpo\_dminputs** (*Optional[QuOperator]*) – QuOperator for a MPO like initial density matrix.
- **split** (*Optional[Dict[str, Any]]*) – dict if two qubit gate is ready for split, including parameters for at least one of `max_singular_values` and `max_truncation_err`.

**static all\_zero\_nodes**(*n: int, d: int = 2, prefix: str = 'qb-'*) → List[tensornetwork.network\_components.Node]

**amplitude**(*l: Union[str, Any]*) → Any

Returns the amplitude of the circuit given the bitstring l. For state simulator, it computes  $\langle l|\psi \rangle$ , for density matrix simulator, it computes  $Tr(\rho|l\rangle\langle l|)$  Note how these two are different up to a square operation.

**Example**

```
>>> c = tc.Circuit(2)
>>> c.X(0)
>>> c.amplitude("10")
array(1.+0.j, dtype=complex64)
>>> c.CNOT(0, 1)
>>> c.amplitude("11")
array(1.+0.j, dtype=complex64)
```

**Parameters 1** (*Union[str, Tensor]*) – The bitstring of 0 and 1s.

**Returns** The amplitude of the circuit.

**Return type** tn.Node.tensor

**amplitudedamping**(\**index: int, \*\*vars: float*) → None

Apply amplitudedamping quantum channel on the circuit. See [tensorcircuit.channels.amplitudedampingchannel\(\)](#)

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the channel.

**any**(\**index: int, \*\*vars: Any*) → None

Apply ANY gate with parameters on the circuit. See [tensorcircuit.gates.any\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**append**(*c*: tensorcircuit.abstractcircuit.AbstractCircuit, *indices*: *Optional[List[int]]* = *None*) → *tensorcircuit.abstractcircuit.AbstractCircuit*  
 append circuit *c* before

#### Example

```
>>> c1 = tc.Circuit(2)
>>> c1.H(0)
>>> c1.H(1)
>>> c2 = tc.Circuit(2)
>>> c2.cnot(0, 1)
>>> c1.append(c2)
<tensorcircuit.circuit.Circuit object at 0x7f8402968970>
>>> c1.draw()

q_0: H ┌─┐
 └─┘
q_1: H ┌ X ┌
 └─┘ └─┘
```

#### Parameters

- **c** (*BaseCircuit*) – The other circuit to be appended
- **indices** (*Optional[List[int]]*, *optional*) – the qubit indices to which *c* is appended on. Defaults to *None*, which means plain concatenation.

**Returns** The composed circuit

**Return type** *BaseCircuit*

**append\_from\_qir**(*qir*: *List[Dict[str, Any]]*) → *None*

Apply the ciurict in form of quantum intermediate representation after the current cirucit.

#### Example

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.to_qir()
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None,
 → 'mpo': False}]
>>> c2 = tc.Circuit(3)
>>> c2.CNOT(0, 1)
>>> c2.to_qir()
[{'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None,
 → 'mpo': False}]
>>> c.append_from_qir(c2.to_qir())
>>> c.to_qir()
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None,
 → 'mpo': False},
 {'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None,
 → 'mpo': False}]
```

**Parameters** *qir* (*List[Dict[str, Any]]*) – The quantum intermediate representation.

---

**apply**(*gate*: Union[tensorcircuit.gates.Gate, tensorcircuit.quantum.QuOperator], \**index*: int, *name*: Optional[str] = None, *split*: Optional[Dict[str, Any]] = None, *mpo*: bool = False, *ir\_dict*: Optional[Dict[str, Any]] = None) → None  
An implementation of this method should also append gate directionary to self.\_cir

**apply\_general\_gate**(*gate*: Union[tensorcircuit.gates.Gate, tensorcircuit.quantum.QuOperator], \**index*: int, *name*: Optional[str] = None, *split*: Optional[Dict[str, Any]] = None, *mpo*: bool = False, *ir\_dict*: Optional[Dict[str, Any]] = None) → None  
An implementation of this method should also append gate directionary to self.\_cir

**static apply\_general\_gate\_delayed**(*gatef*: Callable[], tensorcircuit.gates.Gate], *name*: Optional[str] = None, *mpo*: bool = False) → Callable[[], None]

**apply\_general\_kraus**(*kraus*: Sequence[tensorcircuit.gates.Gate], \**index*: int, \*\**kws*: Any) → None

**static apply\_general\_kraus\_delayed**(*krausf*: Callable[[], Sequence[tensorcircuit.gates.Gate]]) → Callable[[], None]

**static apply\_general\_variable\_gate\_delayed**(*gatef*: Callable[[], tensorcircuit.gates.Gate], *name*: Optional[str] = None, *mpo*: bool = False) → Callable[[], None]

**barrier\_instruction**(\**index*: List[int]) → None  
add a barrier instruction flag, no effect on numerical simulation

**Parameters** **index** (List[int]) – the corresponding qubits

**ccnot**(\**index*: int, *split*: Optional[Dict[str, Any]] = None, *name*: Optional[str] = None) → None  
Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**ccx**(\**index*: int, *split*: Optional[Dict[str, Any]] = None, *name*: Optional[str] = None) → None  
Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**static check\_density\_matrix**(*dm*: Any) → None

**static check\_kraus**(*kraus*: Sequence[tensorcircuit.gates.Gate]) → bool

**circuit\_param**: Dict[str, Any]

---

**cnot**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply CNOT gate on the circuit. See `tensorcircuit.gates.cnot_gate()`.

**Parameters** `index` (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

**static coloring\_copied\_nodes**(nodes: Sequence[tensornetwork.network\_components.Node], nodes0: Sequence[tensornetwork.network\_components.Node], is\_dagger: bool = True, flag: str = 'inputs') → None

**static coloring\_nodes**(nodes: Sequence[tensornetwork.network\_components.Node], is\_dagger: bool = False, flag: str = 'inputs') → None

**cond\_measure**(index: int, status: Optional[float] = None) → Any

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a int Tensor and thus maintained a jittable pipeline.

#### Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
two possible outputs: (1, 1) or (-1, -1)
```

---

**Note:** In terms of DMCircuit, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measuremet results

**Parameters** `index` (int) – the qubit for the z-basis measurement

**Returns** 0 or 1 for z measurement on up and down freedom

**Return type** Tensor

**cond\_measurement**(index: int, status: Optional[float] = None) → Any

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a int Tensor and thus maintained a jittable pipeline.

#### Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
two possible outputs: (1, 1) or (-1, -1)
```

---

**Note:** In terms of DMCircuit, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measuremet results

**Parameters** `index` (`int`) – the qubit for the z-basis measurement

**Returns** 0 or 1 for z measurement on up and down freedom

**Return type** Tensor

**conditional\_gate**(`which: Any, kraus: Sequence[tensorcircuit.gates.Gate], *index: int`) → None

Apply which-th gate from kraus list, i.e. apply `kraus[which]`

**Parameters**

- `which` (`Tensor`) – Tensor of shape [] and dtype int
- `kraus` (`Sequence[Gate]`) – A list of gate in the form of `tc.gate` or `Tensor`
- `index` (`int`) – the qubit lines the gate applied on

**static copy**(`nodes: Sequence[tensornetwork.network_components.Node], dangling:`

`Optional[Sequence[tensornetwork.network_components.Edge]] = None, conj: Optional[bool] = False`) → `Tuple[List[tensornetwork.network_components.Node], List[tensornetwork.network_components.Edge]]`

copy all nodes and dangling edges correspondingly

**Returns**

**cphase**(\*`index: int, **vars: Any`) → None

Apply CPHASE gate with parameters on the circuit. See `tensorcircuit.gates.cphase_gate()`.

**Parameters**

- `index` (`int.`) – Qubit number that the gate applies on.
- `vars` (`float.`) – Parameters for the gate.

**cr**(\*`index: int, **vars: Any`) → None

Apply CR gate with parameters on the circuit. See `tensorcircuit.gates.cr_gate()`.

**Parameters**

- `index` (`int.`) – Qubit number that the gate applies on.
- `vars` (`float.`) – Parameters for the gate.

**crx**(\*`index: int, **vars: Any`) → None

Apply CRX gate with parameters on the circuit. See `tensorcircuit.gates.crx_gate()`.

**Parameters**

- `index` (`int.`) – Qubit number that the gate applies on.
- `vars` (`float.`) – Parameters for the gate.

**cry**(\*`index: int, **vars: Any`) → None

Apply CRY gate with parameters on the circuit. See `tensorcircuit.gates.cry_gate()`.

**Parameters**

- `index` (`int.`) – Qubit number that the gate applies on.
- `vars` (`float.`) – Parameters for the gate.

**crz**(\*`index: int, **vars: Any`) → None

Apply CRZ gate with parameters on the circuit. See `tensorcircuit.gates.crz_gate()`.

**Parameters**

- `index` (`int.`) – Qubit number that the gate applies on.

- **vars** (*float.*) – Parameters for the gate.

**cswap**(\**index*: int, *split*: Optional[Dict[str, Any]] = None, *name*: Optional[str] = None) → None  
Apply **FREDKIN** gate on the circuit. See `tensorcircuit.gates.fredkin_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**cu**(\**index*: int, \*\**vars*: Any) → None  
Apply **CU** gate with parameters on the circuit. See `tensorcircuit.gates.cu_gate()`.

**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**cx**(\**index*: int, *split*: Optional[Dict[str, Any]] = None, *name*: Optional[str] = None) → None  
Apply **CNOT** gate on the circuit. See `tensorcircuit.gates.cnot_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

**cy**(\**index*: int, *split*: Optional[Dict[str, Any]] = None, *name*: Optional[str] = None) → None  
Apply **CY** gate on the circuit. See `tensorcircuit.gates.cy_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. - 1.j \\ 0. + 0.j & 0. + 0.j & 0. + 1.j & 0. + 0.j \end{bmatrix}$$

**cz**(\**index*: int, *split*: Optional[Dict[str, Any]] = None, *name*: Optional[str] = None) → None  
Apply **CZ** gate on the circuit. See `tensorcircuit.gates.cz_gate()`.

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & -1. + 0.j \end{bmatrix}$$

**densitymatrix**(*check*: bool = False, *reuse*: bool = True) → Any  
Return the output density matrix of the circuit.

**Parameters**

- **check** (*bool, optional*) – check whether the final return is a legal density matrix, defaults to False
- **reuse** (*bool, optional*) – whether to reuse previous results, defaults to True

**Returns** The output densitymatrix in 2D shape tensor form

**Return type** Tensor

**depolarizing**(\**index: int*, \*\**vars: float*) → None  
Apply depolarizing quantum channel on the circuit. See [tensorcircuit.channels.depolarizingchannel\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the channel.

**draw**(\*\**kws: Any*) → Any

Visualise the circuit. This method receives the keywords as same as qiskit.circuit.QuantumCircuit.draw. More details can be found here: <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.draw.html>.

**Example**

```
>>> c = tc.Circuit(3)
>>> c.H(1)
>>> c.X(2)
>>> c.CNOT(0, 1)
>>> c.draw(output='text')
q_0: _____
q_1: H X |
 | |
q_2: X _____
```

**exp**(\**index: int*, \*\**vars: Any*) → None

Apply EXP gate with parameters on the circuit. See [tensorcircuit.gates.exp\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**exp1**(\**index: int*, \*\**vars: Any*) → None

Apply EXP1 gate with parameters on the circuit. See [tensorcircuit.gates.exp1\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**expectation**(\**ops: Tuple[tensornetwork.network\_components.Node, List[int]]*, *reuse: bool = True*, *noise\_conf: Optional[Any] = None*, *status: Optional[Any] = None*, \*\**kws: Any*) → <property object at 0x7f72990543b0>

Compute the expectation of corresponding operators.

**Parameters**

- **ops** (*Tuple[tn.Node, List[int]]*) – Operator and its position on the circuit, eg. `(tc.gates.z(), [1, ])`, `(tc.gates.x(), [2, ])` is for operator  $Z_1X_2$ .
- **reuse** (*bool*) – whether contract the density matrix in advance, defaults to True
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **status** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** Tensor with one element

**Return type** Tensor

**expectation\_before**(\**ops*: *Tuple[tensornetwork.network\_components.Node, List[int]]*, *reuse*: *bool* = True, \*\**kws*: *Any*) → *List[tensornetwork.network\_components.Node]*

Get the tensor network in the form of a list of nodes for the expectation calculation before the real contraction

**Parameters** **reuse** (*bool*, *optional*) – \_description\_, defaults to True

**Raises** **ValueError** – \_description\_

**Returns** \_description\_

**Return type** *List[tn.Node]*

**expectation\_ps**(*x*: *Optional[Sequence[int]]* = None, *y*: *Optional[Sequence[int]]* = None, *z*: *Optional[Sequence[int]]* = None, *reuse*: *bool* = True, *noise\_conf*: *Optional[Any]* = None, *nmc*: *int* = 1000, *status*: *Optional[Any]* = None, \*\**kws*: *Any*) → *Any*

Shortcut for Pauli string expectation. *x*, *y*, *z* list are for X, Y, Z positions

### Example

```
>>> c = tc.Circuit(2)
>>> c.X(0)
>>> c.H(1)
>>> c.expectation_ps(x=[1], z=[0])
array(-0.99999994+0.j, dtype=complex64)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> c.expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
(0.46274087-3.764033e-09j)
```

### Parameters

- **x** (*Optional[Sequence[int]]*, *optional*) – sites to apply X gate, defaults to None

- **y** (*Optional[Sequence[int]]*, *optional*) – sites to apply Y gate, defaults to None
- **z** (*Optional[Sequence[int]]*, *optional*) – sites to apply Z gate, defaults to None
- **reuse** (*bool*, *optional*) – whether to cache and reuse the wavefunction, defaults to True
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **nmc** (*int*, *optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **status** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** Expectation value

**Return type** Tensor

**fredkin**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply FREDKIN gate on the circuit. See `tensorcircuit.gates.fredkin_gate()`.

**Parameters** **index** (*int*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**classmethod from\_json**(*jsonstr: str, circuit\_params: Optional[Dict[str, Any]] = None*) →

`tensorcircuit.abstractcircuit.AbstractCircuit`

load json str as a Circuit

**Parameters**

- **jsonstr** (*str*) – \_description\_
- **circuit\_params** (*Optional[Dict[str, Any]]*, *optional*) – Extra circuit parameters in the format of `__init__`, defaults to None

**Returns** \_description\_

**Return type** `AbstractCircuit`

**classmethod from\_json\_file**(*file: str, circuit\_params: Optional[Dict[str, Any]] = None*) →

`tensorcircuit.abstractcircuit.AbstractCircuit`

load json file and convert it to a circuit

**Parameters**

- **file** (*str*) – filename
- **circuit\_params** (*Optional[Dict[str, Any]]*, *optional*) – \_description\_, defaults to None

**Returns** \_description\_

**Return type** *AbstractCircuit*

```
classmethod from_openqasm(qasmstr: str, circuit_params: Optional[Dict[str, Any]] = None,
 keep_measure_order: bool = False) →
 tensorcircuit.abstractcircuit.AbstractCircuit

classmethod from_openqasm_file(file: str, circuit_params: Optional[Dict[str, Any]] = None,
 keep_measure_order: bool = False) →
 tensorcircuit.abstractcircuit.AbstractCircuit

classmethod from_qir(qir: List[Dict[str, Any]], circuit_params: Optional[Dict[str, Any]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
```

Restore the circuit from the quantum intermediate representation.

#### Example

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.rx(1, theta=tc.array_to_tensor(0.7))
>>> c.expl(0, 1, unitary=tc.gates._zz_matrix, theta=tc.array_to_tensor(-0.2),
 →split=split)
>>> len(c)
7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)
>>> qirs = c.to_qir()
>>>
>>> c = tc.Circuit.from_qir(qirs, circuit_params={"nqubits": 3})
>>> len(c._nodes)
7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)
```

#### Parameters

- **qir** (*List[Dict[str, Any]]*) – The quantum intermediate representation of a circuit.
- **circuit\_params** (*Optional[Dict[str, Any]]*) – Extra circuit parameters.

**Returns** The circuit have same gates in the qir.

**Return type** *Circuit*

```
classmethod from_qiskit(qc: Any, n: Optional[int] = None, inputs: Optional[List[float]] = None,
 circuit_params: Optional[Dict[str, Any]] = None, binding_params:
 Optional[Union[Sequence[float], Dict[Any, float]]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
```

Import Qiskit QuantumCircuit object as a `tc.Circuit` object.

#### Example

```
>>> from qiskit import QuantumCircuit
>>> qisc = QuantumCircuit(3)
>>> qisc.h(2)
>>> qisc.cswap(1, 2, 0)
>>> qisc.swap(0, 1)
>>> c = tc.Circuit.from_qiskit(qisc)
```

**Parameters**

- **qc** (*QuantumCircuit in Qiskit*) – Qiskit Circuit object
- **n** (*int*) – The number of qubits for the circuit
- **inputs** (*Optional[List[float]]*, *optional*) – possible input wavefunction for `tc.Circuit`, defaults to None
- **circuit\_params** (*Optional[Dict[str, Any]]*) – kwargs given in `Circuit.__init__` construction function, default to None.
- **binding\_params** (*Optional[Union[Sequence[float], Dict[Any, float]]]*) – (variational) parameters for the circuit. Could be either a sequence or dictionary depending on the type of parameters in the Qiskit circuit. For `ParameterVectorElement` use sequence. For `Parameter` use dictionary

**Returns** The same circuit but as tensorcircuit object**Return type** `Circuit`

```
classmethod from_qsim_file(file: str, circuit_params: Optional[Dict[str, Any]] = None) →
 tensorcircuit.abstractcircuit.AbstractCircuit
```

```
static front_from_nodes(nodes: List[tensornetwork.network_components.Node]) →
 List[tensornetwork.network_components.Edge]
```

```
gate_aliases = [['cnot', 'cx'], ['fredkin', 'cswap'], ['toffoli', 'ccnot'],
['toffoli', 'ccx'], ['any', 'unitary'], ['sd', 'sdg'], ['td', 'tdg']]
```

```
gate_count(gate_list: Optional[Sequence[str]] = None) → int
count the gate number of the circuit
```

**Example**

```
>>> c = tc.Circuit(3)
>>> c.h(0)
>>> c.multicontrol(0, 1, 2, ctrl=[0, 1], unitary=tc.gates._x_matrix)
>>> c.toffoli(1, 2, 0)
>>> c.gate_count()
3
>>> c.gate_count(["multicontrol", "toffoli"])
2
```

**Parameters** `gate_list` (*Optional[Sequence[str]]*, *optional*) – gate name list to be counted, defaults to None (counting all gates)**Returns** the total number of all gates or gates in the `gate_list`**Return type** `int`

```
gate_summary() → Dict[str, int]
```

return the summary dictionary on gate type - gate count pair

**Returns** the gate count dict by gate type**Return type** `Dict[str, int]`

```
general_kraus(kraus: Sequence[tensorcircuit.gates.Gate], *index: int, **kws: Any) → None
```

**generaldepolarizing**(\*index: int, \*\*vars: float) → None  
Apply generaldepolarizing quantum channel on the circuit. See [tensorcircuit.channels](#).  
[generaldepolarizingchannel\(\)](#)

**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the channel.

**get\_dm\_as\_quoperator()** → [tensorcircuit.quantum.QuOperator](#)  
Get the representation of the output state in the form of QuOperator while maintaining the circuit uncomputed

**Returns** QuOperator representation of the output state from the circuit

**Return type** [QuOperator](#)

**get\_dm\_as\_quvector()** → [tensorcircuit.quantum.QuVector](#)  
Get the representation of the output state in the form of QuVector while maintaining the circuit uncomputed

**Returns** QuVector representation of the output state from the circuit

**Return type** [QuVector](#)

**get\_positional\_logical\_mapping()** → Dict[int, int]  
Get positional logical mapping dict based on measure instruction. This function is useful when we only measure part of the qubits in the circuit, to process the count result from partial measurement, we must be aware of the mapping, i.e. for each position in the count bitstring, what is the corresponding qubits (logical) defined on the circuit

**Returns** positional\_logical\_mapping

**Return type** Dict[int, int]

**get\_quvector()** → [tensorcircuit.quantum.QuVector](#)  
Get the representation of the output state in the form of QuVector while maintaining the circuit uncomputed

**Returns** QuVector representation of the output state from the circuit

**Return type** [QuVector](#)

**h**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None  
Apply H gate on the circuit. See [tensorcircuit.gates.h\\_gate\(\)](#).

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & 0.70710677 + 0.j \\ 0.70710677 + 0.j & -0.70710677 + 0.j \end{bmatrix}$$

**i**(\*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None  
Apply I gate on the circuit. See [tensorcircuit.gates.i\\_gate\(\)](#).

**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**initial\_mapping**(logical\_physical\_mapping: Dict[int, int], n: Optional[int] = None, circuit\_params: Optional[Dict[str, Any]] = None) → [tensorcircuit.abstractcircuit.AbstractCircuit](#)  
generate a new circuit with the qubit mapping given by logical\_physical\_mapping

**Parameters**

- **logical\_physical\_mapping** (*Dict[int, int]*) – how to map logical qubits to the physical qubits on the new circuit
- **n** (*Optional[int]*, *optional*) – number of qubit of the new circuit, can be different from the original one, defaults to None
- **circuit\_params** (*Optional[Dict[str, Any]]*, *optional*) – `_description_`, defaults to None

**Returns** `_description_`**Return type** *AbstractCircuit***inputs:** *Any***inverse**(*circuit\_params: Optional[Dict[str, Any]] = None*) → *tensorcircuit.abstractcircuit.AbstractCircuit*  
inverse the circuit, return a new inversed circuit**EXAMPLE**

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rzz(1, 2, theta=0.8)
>>> c1 = c.inverse()
```

**Parameters** **circuit\_params** (*Optional[Dict[str, Any]]*, *optional*) – keywords dict for initialization the new circuit, defaults to None**Returns** the inversed circuit**Return type** *Circuit***is\_dm:** *bool* = True**is\_mps:** *bool* = False**iswap**(\**index: int*, \*\**vars: Any*) → NoneApply ISWAP gate with parameters on the circuit. See *tensorcircuit.gates.iswap\_gate()*.**Parameters**

- **index** (*int*) – Qubit number that the gate applies on.
- **vars** (*float*) – Parameters for the gate.

**measure**(\**index: int*, **with\_prob**: *bool* = False, **status**: *Optional[Any] = None*) → Tuple[*Any*, *Any*]

Take measurement to the given quantum lines. This method is jittable is and about 100 times faster than unjit version!

**Parameters**

- **index** (*int*) – Measure on which quantum line.
- **with\_prob** (*bool*, *optional*) – If true, theoretical probability is also returned.
- **status** (*Optional[Tensor]*) – external randomness, with shape [index], defaults to None

**Returns** The sample output and probability (optional) of the quantum line.**Return type** Tuple[*Tensor*, *Tensor*]

**measure\_instruction**(*index: int*) → None  
add a measurement instruction flag, no effect on numerical simulation

**Parameters** **index** (*int*) – the corresponding qubit

**measure\_jit**(\**index: int, with\_prob: bool = False, status: Optional[Any] = None*) → Tuple[*Any, Any*]  
Take measurement to the given quantum lines. This method is jittable is and about 100 times faster than unjit version!

**Parameters**

- **index** (*int*) – Measure on which quantum line.
- **with\_prob** (*bool, optional*) – If true, theoretical probability is also returned.
- **status** (*Optional [Tensor]*) – external randomness, with shape [index], defaults to None

**Returns** The sample output and probability (optional) of the quantum line.

**Return type** Tuple[*Tensor, Tensor*]

**mpo**(\**index: int, \*\*vars: Any*) → None  
Apply mpo gate in MPO format on the circuit. See [tensorcircuit.gates.mpo\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**mpogates** = ['multicontrol', 'mpo']

**multicontrol**(\**index: int, \*\*vars: Any*) → None  
Apply multicontrol gate in MPO format on the circuit. See [tensorcircuit.gates.multicontrol\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**orx**(\**index: int, \*\*vars: Any*) → None  
Apply ORX gate with parameters on the circuit. See [tensorcircuit.gates.orx\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**ory**(\**index: int, \*\*vars: Any*) → None  
Apply ORY gate with parameters on the circuit. See [tensorcircuit.gates.ory\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**orz**(\**index: int, \*\*vars: Any*) → None  
Apply ORZ gate with parameters on the circuit. See [tensorcircuit.gates.orz\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.

- **vars** (*float.*) – Parameters for the gate.

**ox**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*  
Apply **OX** gate on the circuit. See [tensorcircuit.gates.ox\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**oy**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*  
Apply **OY** gate on the circuit. See [tensorcircuit.gates.oy\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j & 0. + 0.j & 0. + 0.j \\ 0. + 1.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**oz**(\**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*  
Apply **OZ** gate on the circuit. See [tensorcircuit.gates.oz\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**perfect\_sampling**(*status*: *Optional[Any]* = *None*) → *Tuple[str, float]*

Sampling bistrings from the circuit output based on quantum amplitudes. Reference: arXiv:1201.3974.

**Parameters** **status** (*Optional[Tensor]*) – external randomness, with shape [nqubits], defaults to None

**Returns** Sampled bit string and the corresponding theoretical probability.

**Return type** *Tuple[str, float]*

**phase**(\**index*: *int*, \*\**vars*: *Any*) → *None*

Apply **PHASE** gate with parameters on the circuit. See [tensorcircuit.gates.phase\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**phasedamping**(\**index*: *int*, \*\**vars*: *float*) → *None*

Apply phasedamping quantum channel on the circuit. See [tensorcircuit.channels.phasedampingchannel\(\)](#)

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the channel.

**prepend**(*c*: `tensorcircuit.abstractcircuit.AbstractCircuit`) → `tensorcircuit.abstractcircuit.AbstractCircuit`  
prepend circuit *c* before

**Parameters** *c* (`BaseCircuit`) – The other circuit to be prepended

**Returns** The composed circuit

**Return type** `BaseCircuit`

**probability**() → Any

get the  $2^n$  length probability vector over computational basis

**Returns** probability vector

**Return type** Tensor

**quvector**() → `tensorcircuit.quantum.QuVector`

Get the representation of the output state in the form of QuVector while maintaining the circuit uncomputed

**Returns** QuVector representation of the output state from the circuit

**Return type** `QuVector`

**r**(\**index*: int, \*\**vars*: Any) → None

Apply R gate with parameters on the circuit. See `tensorcircuit.gates.r_gate()`.

**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

**readouterror\_bs**(*readout\_error*: `Optional[Sequence[Any]]` = `None`, *p*: `Optional[Any]` = `None`) → Any

Apply readout error to original probabilities of bit string and return the noisy probabilities.

**Example**

```
>>> readout_error = []
>>> readout_error.append([0.9, 0.75]) # readout error for qubit 0, [p0/0, p1/1]
>>> readout_error.append([0.4, 0.7]) # readout error for qubit 1, [p0/0, p1/1]
```

**Parameters**

- **readout\_error** (`Optional[Sequence[Any]]`) – list of readout error for each qubits.
- **p** (`Optional[Any]`) – probabilities of bit string

**Return type** Tensor

**replace\_inputs**(*inputs*: Any) → None

Replace the input state with the circuit structure unchanged.

**Parameters** *inputs* (`Tensor`) – Input wavefunction.

**reset**(\**index*: int, \*\**vars*: float) → None

Apply reset quantum channel on the circuit. See `tensorcircuit.channels.resetchannel()`

**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the channel.

**reset\_instruction**(*index: int*) → None

add a reset instruction flag, no effect on numerical simulation

**Parameters** **index** (*int*) – the corresponding qubit

**rx**(\**index: int, \*\*vars: Any*) → None

Apply **RX** gate with parameters on the circuit. See [tensorcircuit.gates.rx\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**rxx**(\**index: int, \*\*vars: Any*) → None

Apply **RXX** gate with parameters on the circuit. See [tensorcircuit.gates.rxx\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**ry**(\**index: int, \*\*vars: Any*) → None

Apply **RY** gate with parameters on the circuit. See [tensorcircuit.gates.ry\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**ryy**(\**index: int, \*\*vars: Any*) → None

Apply **RYY** gate with parameters on the circuit. See [tensorcircuit.gates.ryy\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**rz**(\**index: int, \*\*vars: Any*) → None

Apply **RZ** gate with parameters on the circuit. See [tensorcircuit.gates.rz\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**rzz**(\**index: int, \*\*vars: Any*) → None

Apply **RZZ** gate with parameters on the circuit. See [tensorcircuit.gates.rzz\\_gate\(\)](#).

**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

**s**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply **S** gate on the circuit. See [tensorcircuit.gates.s\\_gate\(\)](#).

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 1.j \end{bmatrix}$$

**sample**(batch: *Optional[int]* = *None*, allow\_state: *bool* = *False*, readout\_error: *Optional[Sequence[Any]]* = *None*, format: *Optional[str]* = *None*, random\_generator: *Optional[Any]* = *None*, status: *Optional[Any]* = *None*) → *Any*  
 batched sampling from state or circuit tensor network directly

**Parameters**

- **batch** (*Optional[int]*, *optional*) – number of samples, defaults to *None*
- **allow\_state** (*bool*, *optional*) – if true, we sample from the final state if memory allows, *True* is preferred, defaults to *False*
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to *None*
- **format** (*Optional[str]*) – sample format, defaults to *None* as backward compatibility check the doc in [tensorcircuit.quantum.measurement\\_results\(\)](#)
- **format** – alias for the argument **format**
- **random\_generator** (*Optional[Any]*, *optional*) – random generator, defaults to *None*
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite **random\_generator**

**Returns** List (if batch) of tuple (binary configuration tensor and corresponding probability) if the format is *None*, and consistent with format when given

**Return type** *Any*

**sample\_expectation\_ps**(x: *Optional[Sequence[int]]* = *None*, y: *Optional[Sequence[int]]* = *None*, z: *Optional[Sequence[int]]* = *None*, shots: *Optional[int]* = *None*, random\_generator: *Optional[Any]* = *None*, status: *Optional[Any]* = *None*, readout\_error: *Optional[Sequence[Any]]* = *None*, noise\_conf: *Optional[Any]* = *None*, nmc: *int* = 1000, statusc: *Optional[Any]* = *None*, \*\*kws: *Any*) → *Any*

Compute the expectation with given Pauli string with measurement shots numbers

**Example**

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rx(1, theta=np.pi/2)
>>> c.sample_expectation_ps(x=[0], y=[1])
-0.99999976
>>> readout_error = []
>>> readout_error.append([0.9, 0.75])
>>> readout_error.append([0.4, 0.7])
>>> c.sample_expectation_ps(x=[0], y=[1], readout_error = readout_error)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> readout_error = [[0.9, 0.75], [0.4, 0.7]]
```

(continues on next page)

(continued from previous page)

```
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> noise_conf.add_noise("readout", readout_error)
>>> c.sample_expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
0.44766843
```

**Parameters**

- **x** (*Optional[Sequence[int]]*, *optional*) – index for Pauli X, defaults to None
- **y** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Y, defaults to None
- **z** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Z, defaults to None
- **shots** (*Optional[int]*, *optional*) – number of measurement shots, defaults to None, indicating analytical result
- **random\_generator** (*Optional[Any]*) – random\_generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random\_generator
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to None. Overrided if noise\_conf is provided.
- **noise\_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to None
- **nmc** (*int, optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **statusc** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** [description]**Return type** Tensor

**sd**(\*index: int, split: *Optional[Dict[str, Any]]* = None, name: *Optional[str]* = None) → None  
 Apply **SD** gate on the circuit. See `tensorcircuit.gates.sd_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

**sdg**(\*index: int, split: *Optional[Dict[str, Any]]* = None, name: *Optional[str]* = None) → None  
 Apply **SD** gate on the circuit. See `tensorcircuit.gates.sd_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

**select\_gate**(which: Any, kraus: *Sequence[tensorcircuit.gates.Gate]*, \*index: int) → None  
 Apply which-th gate from kraus list, i.e. apply kraus[which]

**Parameters**

- **which** (*Tensor*) – Tensor of shape [] and dtype int
- **kraus** (*Sequence[Gate]*) – A list of gate in the form of `tc.gate` or *Tensor*
- **index** (*int*) – the qubit lines the gate applied on

**sexpss**(*x*: *Optional[Sequence[int]]* = *None*, *y*: *Optional[Sequence[int]]* = *None*, *z*: *Optional[Sequence[int]]* = *None*, *shots*: *Optional[int]* = *None*, *random\_generator*: *Optional[Any]* = *None*, *status*: *Optional[Any]* = *None*, *readout\_error*: *Optional[Sequence[Any]]* = *None*, *noise\_conf*: *Optional[Any]* = *None*, *nmc*: *int* = 1000, *statusc*: *Optional[Any]* = *None*, *\*\*kws*: *Any*) → *Any*

Compute the expectation with given Pauli string with measurement shots numbers

#### Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rx(1, theta=np.pi/2)
>>> c.sample_expectation_ps(x=[0], y=[1])
-0.99999976
>>> readout_error = []
>>> readout_error.append([0.9, 0.75])
>>> readout_error.append([0.4, 0.7])
>>> c.sample_expectation_ps(x=[0], y=[1], readout_error = readout_error)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> readout_error = [[0.9, 0.75], [0.4, 0.7]]
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> noise_conf.add_noise("readout", readout_error)
>>> c.sample_expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
0.44766843
```

#### Parameters

- **x** (*Optional[Sequence[int]]*, *optional*) – index for Pauli X, defaults to *None*
- **y** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Y, defaults to *None*
- **z** (*Optional[Sequence[int]]*, *optional*) – index for Pauli Z, defaults to *None*
- **shots** (*Optional[int]*, *optional*) – number of measurement shots, defaults to *None*, indicating analytical result
- **random\_generator** (*Optional[Any]*) – random\_generator, defaults to *None*
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random\_generator
- **readout\_error** (*Optional[Sequence[Any]] Tensor, List, Tuple*) – readout\_error, defaults to *None*. Overrided if noise\_conf is provided.

- **noise\_conf** (*Optional[NoiseConf], optional*) – Noise Configuration, defaults to None
- **nmc** (*int, optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **statusc** (*Optional[Tensor], optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

**Returns** [description]

**Return type** Tensor

**sgates** = ['i', 'x', 'y', 'z', 'h', 't', 's', 'td', 'sd', 'wroot', 'cnot', 'cz', 'swap', 'cy', 'ox', 'oy', 'oz', 'toffoli', 'fredkin']

**split**: *Optional[Dict[str, Any]]*

**static standardize\_gate**(*name: str*) → str

standardize the gate name to tc common gate sets

**Parameters** **name** (*str*) – non-standard gate name

**Returns** the standard gate name

**Return type** str

**state**(*check: bool = False, reuse: bool = True*) → Any

Return the output density matrix of the circuit.

**Parameters**

- **check** (*bool, optional*) – check whether the final return is a legal density matrix, defaults to False
- **reuse** (*bool, optional*) – whether to reuse previous results, defaults to True

**Returns** The output densitymatrix in 2D shape tensor form

**Return type** Tensor

**swap**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply SWAP gate on the circuit. See `tensorcircuit.gates.swap_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**t**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply T gate on the circuit. See `tensorcircuit.gates.t_gate()`.

**Parameters** **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 + 0.70710677j & 0. \end{bmatrix}$$

**td**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply TD gate on the circuit. See `tensorcircuit.gates.td_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

**tdg**(\*`index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None`) → None

Apply **TD** gate on the circuit. See `tensorcircuit.gates.td_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

**tex**(\*\*`kws: Any`) → str

Generate latex string based on quantikz latex package

**Returns** Latex string that can be directly compiled via, e.g. latexit

**Return type** str

**thermalrelaxation**(\*`index: int, **vars: float`) → None

Apply thermalrelaxation quantum channel on the circuit. See `tensorcircuit.channels.thermalrelaxationchannel()`

**Parameters**

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the channel.

**to\_circuit**(`circuit_params: Optional[Dict[str, Any]] = None`) → `tensorcircuit.circuit.Circuit`

convert into state simulator (current implementation ignores all noise channels)

**Parameters** `circuit_params (Optional[Dict[str, Any]], optional)` – kws to initialize circuit object, defaults to None

**Returns** Circuit with no noise

**Return type** `Circuit`

**to\_cirq**(`enable_instruction: bool = False`) → Any

Translate `tc.Circuit` to a cirq circuit object.

**Parameters** `enable_instruction (bool, defaults to False)` – whether also export measurement and reset instructions

**Returns** A cirq circuit of this circuit.

**to\_graphviz**(`graph: Optional[graphviz.graphs.Graph] = None, include_all_names: bool = False, engine: str = 'neato'`) → graphviz.graphs.Graph

Not an ideal visualization for quantum circuit, but reserve here as a general approach to show the tensor-network [Deprecated, use `Circuit.vis_tex` or `Circuit.draw` instead]

**to\_json**(`file: Optional[str] = None, simplified: bool = False`) → Any

circuit dumps to json

**Parameters**

- `file (Optional[str], optional)` – file str to dump the json to, defaults to None, return the json str
- `simplified (bool)` – If False, keep all info for each gate, defaults to be False. If True, suitable for IO since less information is required

**Returns** None if dumps to file otherwise the json str

**Return type** Any

**to\_openqasm**(*\*\*kws: Any*) → str

transform circuit to openqasm via qiskit circuit, see <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.qasm.html> for usage on possible options for kws

**Returns** circuit representation in openqasm format

**Return type** str

**to\_qir()** → List[Dict[str, Any]]

Return the quantum intermediate representation of the circuit.

**Example**

```
>>> c = tc.Circuit(2)
>>> c.CNOT(0, 1)
>>> c.to_qir()
[{'gatef': cnot, 'gate': Gate(
 name: 'cnot',
 tensor:
 array([[[[1.+0.j, 0.+0.j],
 [0.+0.j, 0.+0.j]],
 [[[0.+0.j, 1.+0.j],
 [0.+0.j, 0.+0.j]],
 [[[0.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]],
 [[[0.+0.j, 0.+0.j],
 [1.+0.j, 0.+0.j]]]], dtype=complex64),
 edges: [
 Edge(Dangling Edge)[0],
 Edge(Dangling Edge)[1],
 Edge('cnot'[2] -> 'qb-1'[0]),
 Edge('cnot'[3] -> 'qb-2'[0])
],
 'index': (0, 1),
 'name': 'cnot',
 'split': None,
 'mpo': False}]]
```

**Returns** The quantum intermediate representation of the circuit.

**Return type** List[Dict[str, Any]]

**to\_qiskit**(*enable\_instruction: bool = False*) → Any

Translate `tc.Circuit` to a qiskit `QuantumCircuit` object.

**Parameters** `enable_instruction (bool, defaults to False)` – whether also export measurement and reset instructions

**Returns** A qiskit object of this circuit.

**toffoli**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None

Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`u(*index: int, **vars: Any) → None`

Apply **U** gate with parameters on the circuit. See [tensorcircuit.gates.u\\_gate\(\)](#).

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`unitary(*index: int, **vars: Any) → None`

Apply ANY gate with parameters on the circuit. See [tensorcircuit.gates.any\\_gate\(\)](#).

#### Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`vgates = ['r', 'cr', 'u', 'cu', 'rx', 'ry', 'rz', 'phase', 'rxx', 'ryy', 'rzz', 'cphase', 'crx', 'cry', 'crz', 'orx', 'ory', 'orz', 'iswap', 'any', 'exp', 'expl']`

`vis_tex(**kws: Any) → str`

Generate latex string based on quantikz latex package

**Returns** Latex string that can be directly compiled via, e.g. latexit

#### Return type

`wavefunction() → Any`

get the wavefunction of outputs, raise error if the final state is not purified [Experimental: the phase factor is not fixed for different backend]

**Returns** wavefunction vector

#### Return type

`wroot(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **WROOT** gate on the circuit. See [tensorcircuit.gates.wroot\\_gate\(\)](#).

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & -0.5 & -0.5j \\ 0.5 & -0.5j & 0.70710677 + 0.j \end{bmatrix}$$

`x(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **X** gate on the circuit. See [tensorcircuit.gates.x\\_gate\(\)](#).

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j \\ 1. + 0.j & 0. + 0.j \end{bmatrix}$$

**y**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None  
Apply Y gate on the circuit. See `tensorcircuit.gates.y_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j \\ 0. + 1.j & 0. + 0.j \end{bmatrix}$$

**z**(\**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None  
Apply Z gate on the circuit. See `tensorcircuit.gates.z_gate()`.

**Parameters** `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j \end{bmatrix}$$

## 4.1.10 tensorcircuit.experimental

Experimental features

`tensorcircuit.experimental.adaptive_vmap(f: Callable[..., Any], vectorized_argnums: Union[int, Sequence[int]] = 0, chunk_size: Optional[int] = None) → Callable[..., Any]`

`tensorcircuit.experimental.dynamics_matrix(f: Callable[..., Any], *, kernel: str = 'dynamics', postprocess: Optional[str] = None, mode: str = 'fwd') → Callable[..., Any]`

`tensorcircuit.experimental.dynamics_rhs(f: Callable[..., Any], h: Any) → Callable[..., Any]`

`tensorcircuit.experimental.hamiltonian_evol(tlist: Any, h: Any, psi0: Any, callback: Optional[Callable[..., Any]] = None) → Any`

Fast implementation of static full Hamiltonian evolution (default as imaginary time)

### Parameters

- `tlist (Tensor) – _description_`
- `h (Tensor) – _description_`
- `psi0 (Tensor) – _description_`
- `callback (Optional[Callable[..., Any]], optional) – _description_, defaults to None`

### Returns

Tensor  
Return type result dynamics on tlist

`tensorcircuit.experimental.parameter_shift_grad(f: Callable[..., Any], argnums: Union[int, Sequence[int]] = 0, jit: bool = False, shifts: Tuple[float, float] = (1.5707963267948966, 2)) → Callable[..., Any]`

similar to `grad` function but using parameter shift internally instead of AD, vmap is utilized for evaluation, so the speed is still ok

### Parameters

- `f (Callable[..., Tensor]) – quantum function with weights in and expectation out`
- `argnums (Union[int, Sequence[int]], optional) – label which args should be differentiated, defaults to 0`

- **jit** (*bool, optional*) – whether jit the original function  $f$  at the beginning, defaults to False
- **shifts** (*Tuple[float, float]*) – two floats for the delta shift on the numerator and dominator, defaults to (pi/2, 2) for parameter shift

**Returns** the grad function

**Return type** Callable[..., Tensor]

```
tensorcircuit.experimental.parameter_shift_grad_v2(f: Callable[..., Any], argnums: Union[int, Sequence[int]] = 0, jit: bool = False, random_argnums: Optional[Sequence[int]] = None, shifts: Tuple[float, float] = (1.5707963267948966, 2)) → Callable[[], Any]
```

similar to *grad* function but using parameter shift internally instead of AD, vmap is utilized for evaluation, v2 also supports random generator for finite measurement shot, only jax backend is supported, since no vmap randomness is available in tensorflow

#### Parameters

- **f** (*Callable[..., Tensor]*) – quantum function with weights in and expectation out
- **argnums** (*Union[int, Sequence[int]], optional*) – label which args should be differentiated, defaults to 0
- **jit** (*bool, optional*) – whether jit the original function  $f$  at the beginning, defaults to False
- **shifts** (*Tuple[float, float]*) – two floats for the delta shift on the numerator and dominator, defaults to (pi/2, 2) for parameter shift

**Returns** the grad function

**Return type** Callable[..., Tensor]

```
tensorcircuit.experimental.qng(f: Callable[[], Any], kernel: str = 'qng', postprocess: Optional[str] = 'qng', mode: str = 'fwd') → Callable[[], Any]
```

```
tensorcircuit.experimental.qng2(f: Callable[[], Any], kernel: str = 'qng', postprocess: Optional[str] = 'qng', mode: str = 'rev') → Callable[[], Any]
```

### 4.1.11 tensorcircuit.gates

Declarations of single-qubit and two-qubit gates and their corresponding matrix.

```
class tensorcircuit.gates.Gate(tensor: Union[Any, tensornetwork.network_components.AbstractNode], name: Optional[str] = None, axis_names: Optional[List[str]] = None, backend: Optional[Union[str, tensornetwork.backends.abstract_backend.AbstractBackend]] = None)
```

Bases: `tensornetwork.network_components.Node`

Wrapper of tn.Node, quantum gate

```
__init__(tensor: Union[Any, tensornetwork.network_components.AbstractNode], name: Optional[str] = None, axis_names: Optional[List[str]] = None, backend: Optional[Union[str, tensornetwork.backends.abstract_backend.AbstractBackend]] = None) → None
```

Create a node.

#### Parameters

- **tensor** – The concrete that is represented by this node, or a *AbstractNode* object. If a tensor is passed, it can be either a numpy array or the tensor-type of the used backend. If a *AbstractNode* is passed, the passed node has to have the same backend as given by *backend*.
- **name** – Name of the node. Used primarily for debugging.
- **axis\_names** – List of names for each of the tensor’s axes.
- **backend** – The name of the backend or an instance of a *AbstractBackend*.

**Raises ValueError** – If there is a repeated name in *axis\_names* or if the length doesn’t match the shape of the tensor.

**add\_axis\_names**(*axis\_names*: *List[str]*) → None

Add axis names to a Node.

**Parameters** **axis\_names** – List of names for each of the tensor’s axes.

**Raises ValueError** – If there is a repeated name in *axis\_names* or if the length doesn’t match the shape of the tensor.

**add\_edge**(*edge*: *tensoRnetwork.network\_components.Edge*, *axis*: *Union[int, str]*, *override*: *bool = False*) → None

Add an edge to the node on the given axis.

**Parameters**

- **edge** – The edge to add.
- **axis** – The axis the edge points to.
- **override** – If true, replace the existing edge with the new one.

**Raises ValueError** – If the edge on axis is not dangling.

**property** **axis\_names**: *List[str]*

**copy**(*conjugate*: *bool = False*) → *tensorcircuit.gates.Gate*

**disable**() → None

**property** **dtype**

**property** **edges**: *List[tensoRnetwork.network\_components.Edge]*

**fresh\_edges**(*axis\_names*: *Optional[List[str]] = None*) → None

**classmethod** **from\_serial\_dict**(*serial\_dict*) → *tensoRnetwork.network\_components.Node*

Return a node given a serialized dict representing it.

**Parameters** **serial\_dict** – A python dict representing a serialized node.

**Returns** A node.

**get\_all\_dangling**() → *List[tensoRnetwork.network\_components.Edge]*

Return the set of dangling edges connected to this node.

**get\_all\_edges**() → *List[tensoRnetwork.network\_components.Edge]*

**get\_all\_nondangling**() → *Set[tensoRnetwork.network\_components.Edge]*

Return the set of nondangling edges connected to this node.

**get\_axis\_number**(*axis*: *Union[str, int]*) → *int*

Get the axis number for a given axis name or value.

**get\_dimension**(axis: Union[str, int]) → Optional[int]  
Get the dimension of the given axis.

**Parameters** **axis** – The axis of the underlying tensor.

**Returns** The dimension of the given axis.

**Raises** **ValueError** – if axis isn't an int or if axis is too large or small.

**get\_edge**(axis: Union[int, str]) → tensornetwork.network\_components.Edge

**get\_rank**() → int  
Return rank of tensor represented by self.

**get\_tensor**() → Any

**has\_dangling\_edge**() → bool

**has\_nondangling\_edge**() → bool

**property name: str**

**op\_protection**(other: Union[int, float, complex, tensornetwork.network\_components.Node]) → Any

**reorder\_axes**(perm: List[int]) → tensornetwork.network\_components.AbstractNode  
Reorder axes of the node's tensor.  
This will also update all of the node's edges.

**Parameters** **perm** – Permutation of the dimensions of the node's tensor.

**Returns** This node post reordering.

**Raises** **AttributeError** – If the Node has no tensor.

**reorder\_edges**(edge\_order: List[tensornetwork.network\_components.Edge]) → tensornetwork.network\_components.AbstractNode  
Reorder the edges for this given Node.  
This will reorder the node's edges and transpose the underlying tensor accordingly.

**Parameters** **edge\_order** – List of edges. The order in the list determines the new edge ordering.

**Returns** This node post reordering.

**Raises**

- **ValueError** – If either the list of edges is not the same as expected or if you try to reorder with a trace edge.
- **AttributeError** – If the Node has no tensor.

**set\_name**(name) → None

**set\_tensor**(tensor) → None

**property shape: Tuple[Optional[int], ...]**

**property sparse\_shape: Any**

**property tensor: Any**

**tensor\_from\_edge\_order**(perm: List[tensornetwork.network\_components.Edge]) → tensornetwork.network\_components.AbstractNode

**to\_serial\_dict()** → Dict

Return a serializable dict representing the node.

Returns: A dict object.

**class tensorcircuit.gates.GateF(m: Any, n: Optional[str] = None, ctrl: Optional[List[int]] = None)**

Bases: object

**\_\_init\_\_(m: Any, n: Optional[str] = None, ctrl: Optional[List[int]] = None)**

**adjoint()** → *tensorcircuit.gates.GateF*

**controlled()** → *tensorcircuit.gates.GateF*

**ided(before: bool = True)** → *tensorcircuit.gates.GateF*

**ocontrolled()** → *tensorcircuit.gates.GateF*

**class tensorcircuit.gates.GateVF(f: Callable[[], tensorcircuit.gates.Gate], n: Optional[str] = None, ctrl: Optional[List[int]] = None)**

Bases: *tensorcircuit.gates.GateF*

**\_\_init\_\_(f: Callable[[], tensorcircuit.gates.Gate], n: Optional[str] = None, ctrl: Optional[List[int]] = None)**

**adjoint()** → *tensorcircuit.gates.GateVF*

**controlled()** → *tensorcircuit.gates.GateF*

**ided(before: bool = True)** → *tensorcircuit.gates.GateF*

**ocontrolled()** → *tensorcircuit.gates.GateF*

**tensorcircuit.gates.any\_gate(unitary: Any, name: str = 'any')** → *tensorcircuit.gates.Gate*

Note one should provide the gate with properly reshaped.

#### Parameters

- **unitary** (*Tensor*) – corresponding gate
- **name** (*str*) – The name of the gate.

**Returns** the resulted gate

**Return type** *Gate*

**tensorcircuit.gates.array\_to\_tensor(\*num: Union[float, Any], dtype: Optional[str] = None)** → Any

Convert the inputs to Tensor with specified dtype.

#### Example

```
>>> from tensorcircuit.gates import num_to_tensor
>>> # OR
>>> from tensorcircuit.gates import array_to_tensor
>>>
>>> x, y, z = 0, 0.1, np.array([1])
>>>
>>> tc.set_backend('numpy')
numpy_backend
>>> num_to_tensor(x, y, z)
[array(0.+0.j, dtype=complex64), array(0.1+0.j, dtype=complex64), array([1.+0.j],
˓→dtype=complex64)]
>>>
>>> tc.set_backend('tensorflow')
```

(continues on next page)

(continued from previous page)

```

tensorflow_backend
>>> num_to_tensor(x, y, z)
[<tf.Tensor: shape=(), dtype=complex64, numpy=0j>,
 <tf.Tensor: shape=(), dtype=complex64, numpy=(0.1+0j)>,
 <tf.Tensor: shape=(1,), dtype=complex64, numpy=array([1.+0.j], dtype=complex64)>]
>>>
>>> tc.set_backend('pytorch')
pytorch_backend
>>> num_to_tensor(x, y, z)
[tensor(0.+0.j), tensor(0.1000+0.j), tensor([1.+0.j])]
>>>
>>> tc.set_backend('jax')
jax_backend
>>> num_to_tensor(x, y, z)
[DeviceArray(0.+0.j, dtype=complex64),
 DeviceArray(0.1+0.j, dtype=complex64),
 DeviceArray([1.+0.j], dtype=complex64)]

```

**Parameters**

- **num** (*Union[float, Tensor]*) – inputs
- **dtype** (*str, optional*) – dtype of the output Tensors

**Returns** List of Tensors**Return type** List[Tensor]**tensorcircuit.gates.bmatrix**(*a: Any*) → strReturns a *TEX* bmatrix.**Example**

```

>>> gate = tc.gates.r_gate()
>>> array = tc.gates.matrix_for_gate(gate)
>>> array
array([[1.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]], dtype=complex64)
>>> print(tc.gates.bmatrix(array))
\begin{bmatrix} 1.+0.j & 0.+0.j \\ 0.+0.j & 1.+0.j \end{bmatrix}

```

Formatted Display:

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

**Parameters** *a* (*np.array*) – 2D numpy array**Raises** **ValueError** – ValueError(“bmatrix can at most display two dimensions”)**Returns** *TEX*-formatted string for bmatrix of the array *a***Return type** str**tensorcircuit.gates.cr\_gate**(*theta: float = 0, alpha: float = 0, phi: float = 0*) → *tensorcircuit.gates.Gate*Controlled rotation gate. When the control qubit is 1, *rgate* is applied to the target qubit.**Parameters**

- **theta** (*float, optional*) – angle in radians
- **alpha** (*float, optional*) – angle in radians
- **phi** (*float, optional*) – angle in radians

**Returns** CR Gate**Return type** *Gate*

`tensorcircuit.gates.exp1_gate(unitary: Any, theta: float, half: bool = False, name: str = 'none') → tensorcircuit.gates.Gate`

Faster exponential gate directly implemented based on RHS. Only works when  $U^2 = I$  is an identity matrix.

$$\begin{aligned}\exp(U) &= e^{-j\theta U} \\ &= \cos(\theta)I - j \sin(\theta)U\end{aligned}$$

**Parameters**

- **unitary** (*Tensor*) – input unitary  $U$
- **hermitian** (*Tensor*) – alias for the argument **unitary**
- **hamiltonian** (*Tensor*) – alias for the argument **unitary**
- **theta** (*float*) – angle in radians
- **half** (*bool*) – if True, the angel theta is mutiplied by 1/2, defaults to False
- **name** (*str, optional*) – suffix of Gate name

**Returns** Exponential Gate**Return type** *Gate*

`tensorcircuit.gates.exp_gate(unitary: Any, theta: float, name: str = 'none') → tensorcircuit.gates.Gate`

Exponential gate.

$$\exp(U) = e^{-j\theta U}$$

**Parameters**

- **unitary** (*Tensor*) – input unitary  $U$
- **hermitian** (*Tensor*) – alias for the argument **unitary**
- **hamiltonian** (*Tensor*) – alias for the argument **unitary**
- **theta** (*float*) – angle in radians
- **name** – suffix of Gate name

**Returns** Exponential Gate**Return type** *Gate*

`tensorcircuit.gates.exponential_gate(unitary: Any, theta: float, name: str = 'none') → tensorcircuit.gates.Gate`

Exponential gate.

$$\exp(U) = e^{-j\theta U}$$

**Parameters**

- **unitary** (*Tensor*) – input unitary  $U$
- **hermitian** (*Tensor*) – alias for the argument **unitary**

- **hamiltonian** (*Tensor*) – alias for the argument **unitary**
- **theta** (*float*) – angle in radians
- **name** – suffix of Gate name

**Returns** Exponential Gate

**Return type** *Gate*

`tensorcircuit.gates.exponential_gate_unity(unitary: Any, theta: float, half: bool = False, name: str = 'none') → tensorcircuit.gates.Gate`

Faster exponential gate directly implemented based on RHS. Only works when  $U^2 = I$  is an identity matrix.

$$\begin{aligned}\exp(U) &= e^{-j\theta U} \\ &= \cos(\theta)I - j \sin(\theta)U\end{aligned}$$

**Parameters**

- **unitary** (*Tensor*) – input unitary  $U$
- **hermitian** (*Tensor*) – alias for the argument **unitary**
- **hamiltonian** (*Tensor*) – alias for the argument **unitary**
- **theta** (*float*) – angle in radians
- **half** (*bool*) – if True, the angel theta is mutiplied by 1/2, defaults to False
- **name** (*str, optional*) – suffix of Gate name

**Returns** Exponential Gate

**Return type** *Gate*

`tensorcircuit.gates.gate_wrapper(m: Any, n: Optional[str] = None) → tensorcircuit.gates.Gate`

`tensorcircuit.gates.get_u_parameter(m: Any) → Tuple[float, float, float]`

From the single qubit unitary to infer three angles of IBMUgate,

**Parameters** **m** (*Tensor*) – numpy array, no backend agnostic version for now

**Returns** theta, phi, lbd

**Return type** *Tuple[Tensor, Tensor, Tensor]*

`tensorcircuit.gates.iswap_gate(theta: float = 1.0) → tensorcircuit.gates.Gate`

iSwap gate.

$$\text{iSwap}(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\frac{\pi}{2}\theta) & j \sin(\frac{\pi}{2}\theta) & 0 \\ 0 & j \sin(\frac{\pi}{2}\theta) & \cos(\frac{\pi}{2}\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

**Parameters** **theta** (*float*) – angle in radians

**Returns** iSwap Gate

**Return type** *Gate*

`tensorcircuit.gates.matrix_for_gate(gate: tensorcircuit.gates.Gate, tol: float = 1e-06) → Any`

Convert Gate to numpy array.

**Example**

```
>>> gate = tc.gates.r_gate()
>>> tc.gates.matrix_for_gate(gate)
array([[1.+0.j, 0.+0.j],
 [0.+0.j, 1.+0.j]], dtype=complex64)
```

**Parameters** `gate` (`Gate`) – input Gate

**Returns** Corresponding Tensor

**Return type** Tensor

`tensorcircuit.gates.meta_gate()` → None

Inner helper function to generate gate functions, such as `z()` from `_z_matrix`

`tensorcircuit.gates.meta_vgate()` → None

`tensorcircuit.gates.mpo_gate(mpo: Any, name: str = 'mpo')` → Any

`tensorcircuit.gates.multicontrol_gate(unitary: Any, ctrl: Union[int, Sequence[int]] = I)` → Any

Multicontrol gate. If the control qubits equal to `ctrl`,  $U$  is applied to the target qubits.

E.g., `multicontrol_gate(tc.gates._zz_matrix, [1, 0, 1])` returns a gate of 5 qubits, where the last 2 qubits are applied  $ZZ$  gate, if the first 3 qubits are 101.

**Parameters**

- `unitary` (`Tensor`) – input unitary  $U$
- `ctrl` (`Union[int, Sequence[int]]`) – control bit sequence

**Returns** Multicontrol Gate

**Return type** Operator

`tensorcircuit.gates.num_to_tensor(*num: Union[float, Any], dtype: Optional[str] = None)` → Any

Convert the inputs to Tensor with specified `dtype`.

**Example**

```
>>> from tensorcircuit.gates import num_to_tensor
>>> # OR
>>> from tensorcircuit.gates import array_to_tensor
>>>
>>> x, y, z = 0, 0.1, np.array([1])
>>>
>>> tc.set_backend('numpy')
numpy_backend
>>> num_to_tensor(x, y, z)
[array(0.+0.j, dtype=complex64), array(0.1+0.j, dtype=complex64), array([1.+0.j],
˓→dtype=complex64)]
>>>
>>> tc.set_backend('tensorflow')
tensorflow_backend
>>> num_to_tensor(x, y, z)
[<tf.Tensor: shape=(), dtype=complex64, numpy=0j>,
 <tf.Tensor: shape=(), dtype=complex64, numpy=(0.1+0j)>,
 <tf.Tensor: shape=(1,), dtype=complex64, numpy=array([1.+0.j], dtype=complex64)>]
>>>
```

(continues on next page)

(continued from previous page)

```
>>> tc.set_backend('pytorch')
pytorch_backend
>>> num_to_tensor(x, y, z)
[tensor(0.+0.j), tensor(0.1000+0.j), tensor([1.+0.j])]
>>>
>>> tc.set_backend('jax')
jax_backend
>>> num_to_tensor(x, y, z)
[DeviceArray(0.+0.j, dtype=complex64),
 DeviceArray(0.1+0.j, dtype=complex64),
 DeviceArray([1.+0.j], dtype=complex64)]
```

**Parameters**

- **num** (*Union[float, Tensor]*) – inputs
- **dtype** (*str, optional*) – dtype of the output Tensors

**Returns** List of Tensors**Return type** List[Tensor]**tensorcircuit.gates.phase\_gate**(*theta: float = 0*) → *tensorcircuit.gates.Gate*

The phase gate

$$\text{phase}(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$$

**Parameters** **theta** (*float, optional*) – angle in radians, defaults to 0**Returns** phase gate**Return type** *Gate***tensorcircuit.gates.r\_gate**(*theta: float = 0, alpha: float = 0, phi: float = 0*) → *tensorcircuit.gates.Gate*

General single qubit rotation gate

$$R(\theta, \alpha, \phi) = j \cos(\theta)I - j \cos(\phi) \sin(\alpha) \sin(\theta)X - j \sin(\phi) \sin(\alpha) \sin(\theta)Y - j \sin(\theta) \cos(\alpha)Z$$

**Parameters**

- **theta** (*float, optional*) – angle in radians
- **alpha** (*float, optional*) – angle in radians
- **phi** (*float, optional*) – angle in radians

**Returns** R Gate**Return type** *Gate***tensorcircuit.gates.random\_single\_qubit\_gate**() → *tensorcircuit.gates.Gate*Random single qubit gate described in <https://arxiv.org/abs/2002.07730>.**Returns** A random single-qubit gate**Return type** *Gate***tensorcircuit.gates.random\_two\_qubit\_gate**() → *tensorcircuit.gates.Gate*

Returns a random two-qubit gate.

**Returns** A random two-qubit gate

**Return type** *Gate*

```
tensorcircuit.gates.rgate_theoretical(theta: float = 0, alpha: float = 0, phi: float = 0) →
 tensorcircuit.gates.Gate
```

Rotation gate implemented by matrix exponential. The output is the same as *rgate*.

$$R(\theta, \alpha, \phi) = e^{-j\theta[\sin(\alpha)\cos(\phi)X + \sin(\alpha)\sin(\phi)Y + \cos(\alpha)Z]}$$

**Parameters**

- **theta** (*float, optional*) – angle in radians
- **alpha** (*float, optional*) – angle in radians
- **phi** (*float, optional*) – angle in radians

**Returns** Rotation Gate**Return type** *Gate*

```
tensorcircuit.gates.rx_gate(theta: float = 0) → tensorcircuit.gates.Gate
```

Rotation gate along *x* axis.

$$RX(\theta) = e^{-j\frac{\theta}{2}X}$$

**Parameters** **theta** (*float, optional*) – angle in radians**Returns** RX Gate**Return type** *Gate*

```
tensorcircuit.gates.rxx_gate(*, unitary: Any = array([[0.0, 0.0, 0.0, 1.0], [0.0, 0.0, 1.0, 0.0], [0.0, 1.0, 0.0,
 0.0], [1.0, 0.0, 0.0, 0.0]]), theta: float, half: bool = True, name: str = 'none') →
 tensorcircuit.gates.Gate
```

Faster exponential gate directly implemented based on RHS. Only works when  $U^2 = I$  is an identity matrix.

$$\begin{aligned}\exp(U) &= e^{-j\theta U} \\ &= \cos(\theta)I - j \sin(\theta)U\end{aligned}$$

**Parameters**

- **unitary** (*Tensor*) – input unitary *U*
- **hermitian** (*Tensor*) – alias for the argument *unitary*
- **hamiltonian** (*Tensor*) – alias for the argument *unitary*
- **theta** (*float*) – angle in radians
- **half** (*bool*) – if True, the angel theta is mutiplied by 1/2, defaults to False
- **name** (*str, optional*) – suffix of Gate name

**Returns** Exponential Gate**Return type** *Gate*

```
tensorcircuit.gates.ry_gate(theta: float = 0) → tensorcircuit.gates.Gate
```

Rotation gate along *y* axis.

$$RY(\theta) = e^{-j\frac{\theta}{2}Y}$$

**Parameters** **theta** (*float, optional*) – angle in radians**Returns** RY Gate

**Return type** *Gate*

```
tensorcircuit.gates.ryy_gate(*, unitary: Any = array([[0.0 + 0j, 0.0 - 0j, 0.0 - 0j, - 1.0 + 0j], [0.0 + 0j, 0.0 + 0j, 1.0 - 0j, 0.0 - 0j], [0.0 + 0j, 1.0 - 0j, 0.0 + 0j, 0.0 - 0j], [- 1.0 + 0j, 0.0 + 0j, 0.0 + 0j, 0.0 + 0j]]), theta: float, half: bool = True, name: str = 'none') → tensorcircuit.gates.Gate
```

Faster exponential gate directly implemented based on RHS. Only works when  $U^2 = I$  is an identity matrix.

$$\begin{aligned}\exp(U) &= e^{-j\theta U} \\ &= \cos(\theta)I - j \sin(\theta)U\end{aligned}$$

**Parameters**

- **unitary** (*Tensor*) – input unitary  $U$
- **hermitian** (*Tensor*) – alias for the argument **unitary**
- **hamiltonian** (*Tensor*) – alias for the argument **unitary**
- **theta** (*float*) – angle in radians
- **half** (*bool*) – if True, the angel theta is mutiplied by 1/2, defaults to False
- **name** (*str, optional*) – suffix of Gate name

**Returns** Exponential Gate**Return type** *Gate*

```
tensorcircuit.gates.rz_gate(theta: float = 0) → tensorcircuit.gates.Gate
```

Rotation gate along  $z$  axis.

$$RZ(\theta) = e^{-j\frac{\theta}{2}Z}$$

**Parameters** **theta** (*float, optional*) – angle in radians**Returns** RZ Gate**Return type** *Gate*

```
tensorcircuit.gates.rzz_gate(*, unitary: Any = array([[1.0, 0.0, 0.0, 0.0], [0.0, - 1.0, 0.0, - 0.0], [0.0, 0.0, - 1.0, - 0.0], [0.0, - 0.0, - 0.0, 1.0]]), theta: float, half: bool = True, name: str = 'none') → tensorcircuit.gates.Gate
```

Faster exponential gate directly implemented based on RHS. Only works when  $U^2 = I$  is an identity matrix.

$$\begin{aligned}\exp(U) &= e^{-j\theta U} \\ &= \cos(\theta)I - j \sin(\theta)U\end{aligned}$$

**Parameters**

- **unitary** (*Tensor*) – input unitary  $U$
- **hermitian** (*Tensor*) – alias for the argument **unitary**
- **hamiltonian** (*Tensor*) – alias for the argument **unitary**
- **theta** (*float*) – angle in radians
- **half** (*bool*) – if True, the angel theta is mutiplied by 1/2, defaults to False
- **name** (*str, optional*) – suffix of Gate name

**Returns** Exponential Gate**Return type** *Gate*

`tensorcircuit.gates.u_gate(theta: float = 0, phi: float = 0, lbd: float = 0) → tensorcircuit.gates.Gate`  
 IBMQ U gate following the converntion of OpenQASM3.0. See [OpenQASM doc](#)

$$U(\theta, \phi, \lambda) := \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda} \sin(\theta/2) \\ e^{i\phi} \sin(\theta/2) & e^{i(\phi+\lambda)} \cos(\theta/2) \end{pmatrix}.$$

**Parameters**

- `theta` (`float, optional`) – `_description_`, defaults to 0
- `phi` (`float, optional`) – `_description_`, defaults to 0
- `lbd` (`float, optional`) – `_description_`, defaults to 0

**Returns** `_description_`**Return type** `Gate`

## 4.1.12 tensorcircuit.interfaces

### tensorcircuit.interfaces.numpy

Interface wraps quantum function as a numpy function

`tensorcircuit.interfaces.numpy.numpy_interface(fun: Callable[[], Any], jit: bool = True) → Callable[[], Any]`

Convert `fun` on ML backend into a numpy function

**Example**

```
K = tc.set_backend("tensorflow")

def f(params, n):
 c = tc.Circuit(n)
 for i in range(n):
 c.rx(i, theta=params[i])
 for i in range(n-1):
 c.cnot(i, i+1)
 r = K.real(c.expectation_ps(z=[n-1]))
 return r

n = 3
f_np = tc.interfaces.numpy_interface(f, jit=True)
f_np(np.ones([n]), n) # 0.1577285
```

**Parameters**

- `fun` (`Callable[..., Any]`) – The quantum function
- `jit` (`bool, optional`) – whether to jit `fun`, defaults to True

**Returns** The numpy interface compatible version of `fun`**Return type** `Callable[..., Any]`

`tensorcircuit.interfaces.numpy.numpy_interface(fun: Callable[[], Any], jit: bool = True) → Callable[[], Any]`

Convert `fun` on ML backend into a numpy function

### Example

```
K = tc.set_backend("tensorflow")

def f(params, n):
 c = tc.Circuit(n)
 for i in range(n):
 c.rx(i, theta=params[i])
 for i in range(n-1):
 c.cnot(i, i+1)
 r = K.real(c.expectation_ps(z=[n-1]))
 return r

n = 3
f_np = tc.interfaces.numpy_interface(f, jit=True)
f_np(np.ones([n]), n) # 0.1577285
```

### Parameters

- **fun** (*Callable[..., Any]*) – The quantum function
- **jit** (*bool, optional*) – whether to jit *fun*, defaults to True

**Returns** The numpy interface compatible version of *fun*

**Return type** *Callable[..., Any]*

## tensorcircuit.interfaces.scipy

Interface wraps quantum function as a scipy function for optimization

```
tensorcircuit.interfaces.scipy.scipy_interface(fun: Callable[..., Any], shape: Optional[Tuple[int, ...]] = None, jit: bool = True, gradient: bool = True)
 → Callable[..., Any]
```

Convert *fun* into a scipy optimize interface compatible version

### Example

```
n = 3

def f(param):
 c = tc.Circuit(n)
 for i in range(n):
 c.rx(i, theta=param[0, i])
 c.rz(i, theta=param[1, i])
 loss = c.expectation(
 [
 tc.gates.y(),
 [
 0,
],
]
)
 return tc.backend.real(loss)
```

(continues on next page)

(continued from previous page)

```
A gradient-based optimization interface

f_scipy = tc.interfaces.scipy_optimize_interface(f, shape=[2, n])
r = optimize.minimize(f_scipy, np.zeros([2 * n]), method="L-BFGS-B", jac=True)

A gradient-free optimization interface

f_scipy = tc.interfaces.scipy_optimize_interface(f, shape=[2, n], gradient=False)
r = optimize.minimize(f_scipy, np.zeros([2 * n]), method="COBYLA")
```

**Parameters**

- **fun** (*Callable[..., Any]*) – The quantum function with scalar out that to be optimized
- **shape** (*Optional[Tuple[int, ...]]*, *optional*) – the shape of parameters that fun accepts, defaults to None
- **jit** (*bool*, *optional*) – whether to jit fun, defaults to True
- **gradient** (*bool*, *optional*) – whether using gradient-based or gradient free scipy optimize interface, defaults to True

**Returns** The scipy interface compatible version of fun**Return type** Callable[..., Any]

`tensorcircuit.interfaces.scipy.scipy_optimize_interface(fun: Callable[..., Any], shape: Optional[Tuple[int, ...]] = None, jit: bool = True, gradient: bool = True) → Callable[..., Any]`

Convert fun into a scipy optimize interface compatible version

**Example**

```
n = 3

def f(param):
 c = tc.Circuit(n)
 for i in range(n):
 c.rx(i, theta=param[0, i])
 c.rz(i, theta=param[1, i])
 loss = c.expectation(
 [
 tc.gates.y(),
 [
 0,
],
]
)
 return tc.backend.real(loss)

A gradient-based optimization interface
```

```
f_scipy = tc.interfaces.scipy_optimize_interface(f, shape=[2, n])
r = optimize.minimize(f_scipy, np.zeros([2 * n]), method="L-BFGS-B", jac=True)
```

(continues on next page)

(continued from previous page)

```
A gradient-free optimization interface

f_scipy = tc.interfaces.scipy_optimize_interface(f, shape=[2, n], gradient=False)
r = optimize.minimize(f_scipy, np.zeros([2 * n]), method="COBYLA")
```

**Parameters**

- **fun** (*Callable[..., Any]*) – The quantum function with scalar out that to be optimized
- **shape** (*Optional[Tuple[int, ...]]*, *optional*) – the shape of parameters that fun accepts, defaults to None
- **jit** (*bool*, *optional*) – whether to jit fun, defaults to True
- **gradient** (*bool*, *optional*) – whether using gradient-based or gradient free scipy optimize interface, defaults to True

**Returns** The scipy interface compatible version of **fun****Return type** *Callable[..., Any]***tensorcircuit.interfaces.tensorflow**

Interface wraps quantum function as a tensorflow function

```
tensorcircuit.interfaces.tensorflow.tensorflow_interface(fun: Callable[..., Any], ydtype: Any, jit:
 bool = False, enable_dlpick: bool =
 False) → Callable[..., Any]
```

Wrap a quantum function on different ML backend with a tensorflow interface.

**Example**

```
K = tc.set_backend("jax")

def f(params):
 c = tc.Circuit(1)
 c.rx(0, theta=params[0])
 c.ry(0, theta=params[1])
 return K.real(c.expectation([tc.gates.z(), [0]]))

f = tc.interfaces.tf_interface(f, ydtype=tf.float32, jit=True)

tfb = tc.get_backend("tensorflow")
grads = tfb.jit(tfb.grad(f))(tfb.ones([2]))
```

**Parameters**

- **fun** (*Callable[..., Any]*) – The quantum function with tensor in and tensor out
- **ydtype** (*Any*) – output tf dtype or in str
- **jit** (*bool*, *optional*) – whether to jit fun, defaults to False

- **enable\_dlpark** (bool, optional) – whether transform tensor backend via dlpark, defaults to False

**Returns** The same quantum function but now with torch tensor in and torch tensor out while AD is also supported

**Return type** Callable[..., Any]

```
tensorcircuit.interfaces.tensorflow.tf_dtype(dtype: str) → Any
```

```
tensorcircuit.interfaces.tensorflow.tf_interface(fun: Callable[..., Any], ydtype: Any, jit: bool = False, enable_dlpark: bool = False) → Callable[..., Any]
```

Wrap a quantum function on different ML backend with a tensorflow interface.

**Example**

```
K = tc.set_backend("jax")

def f(params):
 c = tc.Circuit(1)
 c.rx(0, theta=params[0])
 c.ry(0, theta=params[1])
 return K.real(c.expectation([tc.gates.z(), [0]]))

f = tc.interfaces.tf_interface(f, ydtype=tf.float32, jit=True)

tfb = tc.get_backend("tensorflow")
grads = tfb.jit(tfb.grad(f))(tfb.ones([2]))
```

### Parameters

- **fun** (Callable[..., Any]) – The quantum function with tensor in and tensor out
- **ydtype** (Any) – output tf dtype or in str
- **jit** (bool, optional) – whether to jit fun, defaults to False
- **enable\_dlpark** (bool, optional) – whether transform tensor backend via dlpark, defaults to False

**Returns** The same quantum function but now with torch tensor in and torch tensor out while AD is also supported

**Return type** Callable[..., Any]

```
tensorcircuit.interfaces.tensorflow.tf_wrapper(fun: Callable[..., Any], enable_dlpark: bool = False) → Callable[..., Any]
```

**tensorcircuit.interfaces.tensortrans**

general function for interfaces transformation

```
tensorcircuit.interfaces.tensortrans.args_to_tensor(f: Callable[..., Any], argnums: Union[int, Sequence[int]] = 0, tensor_as_matrix: bool = False, gate_to_tensor: bool = False, gate_as_matrix: bool = True, qop_to_tensor: bool = False, qop_as_matrix: bool = True, cast_dtype: bool = True) → Callable[..., Any]
```

Function decorator that automatically convert inputs to tensors on current backend

**Example**

```
tc.set_backend("jax")

@partial(
 tc.interfaces.args_to_tensor,
 argnums=[0, 1, 2],
 gate_to_tensor=True,
 qop_to_tensor=True,
)
def f(a, b, c, d):
 return a, b, c, d

f(
 [tc.Gate(np.ones([2, 2])), tc.Gate(np.ones([2, 2, 2, 2]))],
 tc.QuOperator.from_tensor(np.ones([2, 2, 2, 2, 2, 2])),
 np.ones([2, 2, 2, 2]),
 tf.zeros([1, 2]),
)

([DeviceArray([[1.+0.j, 1.+0.j],
[1.+0.j, 1.+0.j]], dtype=complex64),
DeviceArray([[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j]], dtype=complex64)],
DeviceArray([[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j],
[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j,
1.+0.j]], dtype=complex64),
```

(continues on next page)

(continued from previous page)

```
DeviceArray([[[[1.+0.j, 1.+0.j],
[1.+0.j, 1.+0.j]],
#
[[1.+0.j, 1.+0.j],
[1.+0.j, 1.+0.j]]],
#
[[[1.+0.j, 1.+0.j],
[1.+0.j, 1.+0.j]],
#
[[1.+0.j, 1.+0.j],
[1.+0.j, 1.+0.j]]]], dtype=complex64),
<tf.Tensor: shape=(1, 2), dtype=float32, numpy=array([[0., 0.]], dtype=float32)>
```

**Parameters**

- **f**(*Callable[..., Any]*) – the wrapped function whose arguments in **argnums** position are expected to be tensor format
- **argnums** (*Union[int, Sequence[int]]*, *optional*) – position of args under the auto conversion, defaults to 0
- **tensor\_as\_matrix** (*bool*, *optional*) – try reshape all input tensor as matrix with shape rank 2, defaults to False
- **gate\_to\_tensor** (*bool*, *optional*) – convert Gate to tensor, defaults to False
- **gate\_as\_matrix** (*bool*, *optional*) – reshape tensor from Gate input as matrix, defaults to True
- **qop\_to\_tensor** (*bool*, *optional*) – convert QuOperator to tensor, defaults to False
- **qop\_as\_matrix** (*bool*, *optional*) – reshape tensor from QuOperator input as matrix, defaults to True
- **cast\_dtype** (*bool*, *optional*) – whether cast to backend dtype, defaults to True

**Returns** The wrapped function**Return type** *Callable[..., Any]*

`tensorcircuit.interfaces.tensortrans.gate_to_matrix(t: tensorcircuit.gates.Gate, is_reshapem: bool = True) → Any`

`tensorcircuit.interfaces.tensortrans.general_args_to_backend(args: Any, dtype: Optional[Any] = None, target_backend: Optional[Any] = None, enable_dlpark: bool = True) → Any`

`tensorcircuit.interfaces.tensortrans.general_args_to_numpy(args: Any) → Any`

Given a pytree, get the corresponding numpy array pytree

**Parameters** **args** (*Any*) – pytree**Returns** the same format pytree with all tensor replaced by numpy array**Return type** *Any*

```
tensorcircuit.interfaces.tensortrans.numpy_args_to_backend(args: Any, dtype: Optional[Any] = None, target_backend: Optional[Any] = None) → Any
```

Given a pytree of numpy arrays, get the corresponding tensor pytree

**Parameters**

- **args** (*Any*) – pytree of numpy arrays
- **dtype** (*Any, optional*) – str or str of the same pytree shape as args, defaults to None
- **target\_backend** (*Any, optional*) – str or backend object, defaults to None, indicating the current default backend

**Returns** the same format pytree with all numpy array replaced by the tensors in the target backend

**Return type** Any

```
tensorcircuit.interfaces.tensortrans.numpy_to_tensor(t: Any, backend: Any) → Any
```

```
tensorcircuit.interfaces.tensortrans.qop_to_matrix(t: tensorcircuit.quantum.QuOperator, is_reshape: bool = True) → Any
```

```
tensorcircuit.interfaces.tensortrans.tensor_to_backend_jittable(t: Any) → Any
```

```
tensorcircuit.interfaces.tensortrans.tensor_to_dlpack(t: Any) → Any
```

```
tensorcircuit.interfaces.tensortrans.tensor_to_dtype(t: Any) → str
```

```
tensorcircuit.interfaces.tensortrans.tensor_to_numpy(t: Any) → Any
```

```
tensorcircuit.interfaces.tensortrans.which_backend(a: Any, return_backend: bool = True) → Any
```

Given a tensor a, return the corresponding backend

**Parameters**

- **a** (*Tensor*) – the tensor
- **return\_backend** (*bool, optional*) – if true, return backend object, if false, return backend str, defaults to True

**Returns** the backend object or backend str

**Return type** Any

## tensorcircuit.interfaces.torch

Interface wraps quantum function as a torch function

```
tensorcircuit.interfaces.torch.pytorch_interface(fun: Callable[..., Any], jit: bool = False, enable_dlpark: bool = False) → Callable[..., Any]
```

Wrap a quantum function on different ML backend with a pytorch interface.

**Example**

```
import torch

tc.set_backend("tensorflow")

def f(params):
 c = tc.Circuit(1)
 c.rx(0, theta=params[0])
```

(continues on next page)

(continued from previous page)

```

 c.ry(0, theta=params[1])
 return c.expectation([tc.gates.z(), [0]])

f_torch = tc.interfaces.torch_interface(f, jit=True)

a = torch.ones([2], requires_grad=True)
b = f_torch(a)
c = b ** 2
c.backward()

print(a.grad)

```

**Parameters**

- **fun** (*Callable[..., Any]*) – The quantum function with tensor in and tensor out
- **jit** (*bool, optional*) – whether to jit fun, defaults to False
- **enable\_dlpack** (*bool, optional*) – whether transform tensor backend via dlpack, defaults to False

**Returns** The same quantum function but now with torch tensor in and torch tensor out while AD is also supported

**Return type** Callable[..., Any]

`tensorcircuit.interfaces.torch.torch_interface(fun: Callable[..., Any], jit: bool = False, enable_dlpack: bool = False) → Callable[..., Any]`

Wrap a quantum function on different ML backend with a pytorch interface.

**Example**

```

import torch

tc.set_backend("tensorflow")

def f(params):
 c = tc.Circuit(1)
 c.rx(0, theta=params[0])
 c.ry(0, theta=params[1])
 return c.expectation([tc.gates.z(), [0]])

f_torch = tc.interfaces.torch_interface(f, jit=True)

a = torch.ones([2], requires_grad=True)
b = f_torch(a)
c = b ** 2
c.backward()

print(a.grad)

```

**Parameters**

- **fun** (*Callable[..., Any]*) – The quantum function with tensor in and tensor out
- **jit** (*bool, optional*) – whether to jit fun, defaults to False
- **enable\_dlpack** (*bool, optional*) – whether transform tensor backend via dlpack, defaults to False

**Returns** The same quantum function but now with torch tensor in and torch tensor out while AD is also supported

**Return type** Callable[..., Any]

#### 4.1.13 tensorcircuit.keras

Keras layer for tc quantum function

##### tensorcircuit.keras.KerasLayer

alias of *tensorcircuit.keras.QuantumLayer*

**class** tensorcircuit.keras.QuantumLayer(\*args, \*\*kwargs)

Bases: keras.engine.base\_layer.Layer

**\_\_init\_\_**(*f: Callable[[], Any], weights\_shape: Sequence[Tuple[int, ...]], initializer: Union[str, Sequence[str]] = 'glorot\_uniform', constraint: Optional[Union[str, Sequence[str]]] = None, \*\*kwargs: Any*) → None

*QuantumLayer* wraps the quantum function *f* as a *keras.Layer* so that tensorcircuit is better integrated with tensorflow. Note that the input of the layer can be tensors or even list/dict of tensors.

##### Parameters

- **f** (*Callable[..., Any]*) – Callabel function.
- **weights\_shape** (*Sequence[Tuple[int, ...]]*) – The shape of the weights.
- **initializer** (*Union[Text, Sequence[Text]]*, *optional*) – The initializer of the weights, defaults to “glorot\_uniform”
- **constraint** (*Optional[Union[Text, Sequence[Text]]]*, *optional*) – [description], defaults to None

##### property activity\_regularizer

Optional regularizer function for the output of this layer.

##### add\_loss(*losses, \*\*kwargs*)

Add loss tensor(s), potentially dependent on layer inputs.

Some losses (for instance, activity regularization losses) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs *a* and *b*, some entries in *layer.losses* may be dependent on *a* and some on *b*. This method automatically keeps track of dependencies.

This method can be used inside a subclassed layer or model’s *call* function, in which case *losses* should be a Tensor or list of Tensors.

Example:

```
```python class MyLayer(tf.keras.layers.Layer):  
    def call(self, inputs): self.add_loss(tf.abs(tf.reduce_mean(inputs))) return inputs  
```
```

This method can also be called directly on a Functional Model during construction. In this case, any loss Tensors passed to this Model must be symbolic and be able to be traced back to the model’s *Input*’s. These losses become part of the model’s topology and are tracked in *get\_config*.

Example:

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.
Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.
Model(inputs, outputs) # Activity regularization. model.add_loss(tf.abs(tf.
reduce_mean(x)))`
```

If this is not the case for your loss (if, for example, your loss references a *Variable* of one of the model's layers), you can wrap your loss in a zero-argument lambda. These losses are not tracked as part of the model's topology since they can't be serialized.

Example:

```
`python inputs = tf.keras.Input(shape=(10,)) d = tf.keras.layers.Dense(10) x =
d(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.Model(inputs,
outputs) # Weight regularization. model.add_loss(lambda: tf.reduce_mean(d.
kernel))`
```

#### Parameters

- **losses** – Loss tensor, or list/tuple of tensors. Rather than tensors, losses may also be zero-argument callables which create a loss tensor.
- **\*\*kwargs** – Used for backwards compatibility only.

**add\_metric**(value, name=None, \*\*kwargs)

Adds metric tensor to the layer.

This method can be used inside the *call()* method of a subclassed layer or model.

```
```python class MyMetricLayer(tf.keras.layers.Layer):
```

```
    def __init__(self): super(MyMetricLayer, self).__init__(name='my_metric_layer') self.mean  
        = tf.keras.metrics.Mean(name='metric_1')  
  
    def call(self, inputs): self.add_metric(self.mean(inputs)) self.add_metric(tf.reduce_sum(inputs),  
        name='metric_2') return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any tensor passed to this Model must be symbolic and be able to be traced back to the model's *Input*'s. *These metrics become part of the model's topology and are tracked when you save the model via 'save()'*.

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.  
Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.  
keras.Model(inputs, outputs) model.add_metric(math_ops.reduce_sum(x),  
name='metric_1')`
```

Note: Calling *add_metric()* with the result of a metric object on a Functional Model, as shown in the example below, is not supported. This is because we cannot trace the metric result tensor back to the model's inputs.

```
`python inputs = tf.keras.Input(shape=(10,)) x = tf.keras.layers.  
Dense(10)(inputs) outputs = tf.keras.layers.Dense(1)(x) model = tf.keras.  
Model(inputs, outputs) model.add_metric(tf.keras.metrics.Mean()(x),  
name='metric_1')`
```

Parameters

- **value** – Metric tensor.
- **name** – String metric name.

- ****kwargs** – Additional keyword arguments for backward compatibility. Accepted values: *aggregation* - When the *value* tensor provided is not the result of calling a *keras.Metric* instance, it will be aggregated by default using a *keras.Metric.Mean*.

add_update(updates)

Add update op(s), potentially dependent on layer inputs.

Weight updates (for instance, the updates of the moving mean and variance in a BatchNormalization layer) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs *a* and *b*, some entries in *layer.updates* may be dependent on *a* and some on *b*. This method automatically keeps track of dependencies.

This call is ignored when eager execution is enabled (in that case, variable updates are run on the fly and thus do not need to be tracked for later execution).

Parameters **updates** – Update op, or list/tuple of update ops, or zero-arg callable that returns an update op. A zero-arg callable should be passed in order to disable running the updates by setting *trainable=False* on this Layer, when executing in Eager mode.

add_variable(*args, **kwargs)

Deprecated, do NOT use! Alias for *add_weight*.

add_weight(name=None, shape=None, dtype=None, initializer=None, regularizer=None, trainable=None, constraint=None, use_resource=None, synchronization=VariableSynchronization.AUTO, aggregation=VariableAggregationV2.NONE, **kwargs)

Adds a new variable to the layer.

Parameters

- **name** – Variable name.
- **shape** – Variable shape. Defaults to scalar if unspecified.
- **dtype** – The type of the variable. Defaults to *self.dtype*.
- **initializer** – Initializer instance (callable).
- **regularizer** – Regularizer instance (callable).
- **trainable** – Boolean, whether the variable should be part of the layer’s “trainable_variables” (e.g. variables, biases) or “non_trainable_variables” (e.g. BatchNorm mean and variance). Note that *trainable* cannot be *True* if *synchronization* is set to *ON_READ*.
- **constraint** – Constraint instance (callable).
- **use_resource** – Whether to use a *ResourceVariable* or not. See [this guide](https://www.tensorflow.org/guide/migrate/tf1_vs_tf2#resourcevariables_instead_of_referencevariables) for more information.
- **synchronization** – Indicates when a distributed a variable will be aggregated. Accepted values are constants defined in the class *tf.VariableSynchronization*. By default the synchronization is set to *AUTO* and the current *DistributionStrategy* chooses when to synchronize. If *synchronization* is set to *ON_READ*, *trainable* must not be set to *True*.
- **aggregation** – Indicates how a distributed variable will be aggregated. Accepted values are constants defined in the class *tf.VariableAggregation*.
- ****kwargs** – Additional keyword arguments. Accepted values are *getter*, *collections*, *experimental_autocast* and *caching_device*.

Returns The variable created.

Raises ValueError – When giving unsupported dtype and no initializer or when trainable has been set to True with synchronization set as *ON_READ*.

build(*input_shape*: *Optional[List[int]]* = *None*) → *None*

Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters **input_shape** – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*inputs*: *tensorflow.python.framework.ops.Tensor*, *training*: *Optional[bool]* = *None*, *mask*: *Optional[tensorflow.python.framework.ops.Tensor]* = *None*, ***kwargs*: *Any*) → *tensorflow.python.framework.ops.Tensor*

property compute_dtype

The dtype of the layer’s computations.

This is equivalent to *Layer.dtype_policy.compute_dtype*. Unless mixed precision is used, this is the same as *Layer.dtype*, the dtype of the weights.

Layers automatically cast their inputs to the compute dtype, which causes computations and the output to be in the compute dtype as well. This is done by the base Layer class in *Layer.__call__*, so you do not have to insert these casts if implementing your own layer.

Layers often perform certain internal computations in higher precision when *compute_dtype* is float16 or bfloat16 for numeric stability. The output will still typically be float16 or bfloat16 in such cases.

Returns The layer’s compute dtype.

compute_mask(*inputs*, *mask*=*None*)

Computes an output mask tensor.

Parameters

- **inputs** – Tensor or list of tensors.
- **mask** – Tensor or list of tensors.

Returns

None or a tensor (or list of tensors, one per output tensor of the layer).

compute_output_shape(*input_shape*)

Computes the output shape of the layer.

This method will cause the layer’s state to be built, if that has not happened before. This requires that the layer will later be used with inputs that match the input shape provided here.

Parameters **input_shape** – Shape tuple (tuple of integers) or *tf.TensorShape*, or structure of shape tuples / *tf.TensorShape* instances (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns A *tf.TensorShape* instance or structure of *tf.TensorShape* instances.

compute_output_signature(*input_signature*)

Compute the output tensor signature of the layer based on the inputs.

Unlike a TensorShape object, a TensorSpec object contains both shape and dtype information for a tensor. This method allows layers to provide output dtype information if it is different from the input dtype. For any layer that doesn't implement this function, the framework will fall back to use `compute_output_shape`, and will assume that the output dtype matches the input dtype.

Parameters `input_signature` – Single TensorSpec or nested structure of TensorSpec objects, describing a candidate input for the layer.

Returns

Single TensorSpec or nested structure of TensorSpec objects, describing how the layer would transform the provided input.

Raises `TypeError` – If `input_signature` contains a non-TensorSpec object.

count_params()

Count the total number of scalars composing the weights.

Returns An integer count.

Raises `ValueError` – if the layer isn't yet built (in which case its weights aren't yet defined).

property `dtype`

The dtype of the layer weights.

This is equivalent to `Layer.dtype_policy.variable_dtype`. Unless mixed precision is used, this is the same as `Layer.compute_dtype`, the dtype of the layer's computations.

property `dtype_policy`

The dtype policy associated with this layer.

This is an instance of a `tf.keras.mixed_precision.Policy`.

property `dynamic`

Whether the layer is dynamic (eager-only); set in the constructor.

finalize_state()

Finalizes the layers state after updating layer weights.

This function can be subclassed in a layer and will be called after updating a layer weights. It can be overridden to finalize any additional layer state after a weight update.

This function will be called after weights of a layer have been restored from a loaded model.

classmethod `from_config(config)`

Creates a layer from its config.

This method is the reverse of `get_config`, capable of instantiating the same layer from the config dictionary. It does not handle layer connectivity (handled by Network), nor weights (handled by `set_weights`).

Parameters `config` – A Python dictionary, typically the output of `get_config`.

Returns A layer instance.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by `Network` (one layer of abstraction above).

Note that `get_config()` does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns Python dictionary.

get_input_at(node_index)

Retrieves the input tensor(s) of a layer at a given node.

Parameters **node_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first input node of the layer.

Returns A tensor (or list of tensors if the layer has multiple inputs).

Raises **RuntimeError** – If called in Eager mode.

get_input_mask_at(node_index)

Retrieves the input mask tensor(s) of a layer at a given node.

Parameters **node_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A mask tensor (or list of tensors if the layer has multiple inputs).

get_input_shape_at(node_index)

Retrieves the input shape(s) of a layer at a given node.

Parameters **node_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A shape tuple (or list of shape tuples if the layer has multiple inputs).

Raises **RuntimeError** – If called in Eager mode.

get_output_at(node_index)

Retrieves the output tensor(s) of a layer at a given node.

Parameters **node_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first output node of the layer.

Returns A tensor (or list of tensors if the layer has multiple outputs).

Raises **RuntimeError** – If called in Eager mode.

get_output_mask_at(node_index)

Retrieves the output mask tensor(s) of a layer at a given node.

Parameters **node_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A mask tensor (or list of tensors if the layer has multiple outputs).

get_output_shape_at(node_index)

Retrieves the output shape(s) of a layer at a given node.

Parameters **node_index** – Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns A shape tuple (or list of shape tuples if the layer has multiple outputs).

Raises **RuntimeError** – If called in Eager mode.

get_weights()

Returns the current weights of the layer, as NumPy arrays.

The weights of a layer represent the state of the layer. This function returns both trainable and non-trainable weight values associated with this layer as a list of NumPy arrays, which can in turn be used to load state into similarly parameterized layers.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
       [2.],
       [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
```

Returns Weights values as a list of NumPy arrays.

property inbound_nodes

Return Functional API nodes upstream of this layer.

property input

Retrieves the input tensor(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer.

Returns Input tensor or list of input tensors.

Raises

- **RuntimeError** – If called in Eager mode.
- **AttributeError** – If no inbound nodes are found.

property input_mask

Retrieves the input mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Input mask tensor (potentially None) or list of input mask tensors.

Raises

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

property input_shape

Retrieves the input shape(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer, or if all inputs have the same shape.

Returns Input shape, as an integer shape tuple (or list of shape tuples, one tuple per input tensor).

Raises

- **AttributeError** – if the layer has no defined `input_shape`.
- **RuntimeError** – if called in Eager mode.

property `input_spec`

`InputSpec` instance(s) describing the input format for this layer.

When you create a layer subclass, you can set `self.input_spec` to enable the layer to run input compatibility checks when it is called. Consider a `Conv2D` layer: it can only be called on a single input tensor of rank 4. As such, you can set, in `__init__()`:

```
`python self.input_spec = tf.keras.layers.InputSpec(ndim=4)`
```

Now, if you try to call the layer on an input that isn't rank 4 (for instance, an input of shape `(2,)`, it will raise a nicely-formatted error:

```
`ValueError: Input 0 of layer conv2d is incompatible with the layer: expected ndim=4, found ndim=1. Full shape received: [2]`
```

Input checks that can be specified via `input_spec` include:

- Structure (e.g. a single input, a list of 2 inputs, etc)
- Shape - Rank (ndim)
- Dtype

For more information, see `tf.keras.layers.InputSpec`.

Returns A `tf.keras.layers.InputSpec` instance, or nested structure thereof.

property `losses`

List of losses added using the `add_loss()` API.

Variable regularization tensors are created when this property is accessed, so it is eager safe: accessing `losses` under a `tf.GradientTape` will propagate gradients back to the corresponding variables.

Examples:

```
>>> class MyLayer(tf.keras.layers.Layer):
...     def call(self, inputs):
...         self.add_loss(tf.abs(tf.reduce_mean(inputs)))
...         return inputs
>>> l = MyLayer()
>>> l(np.ones((10, 1)))
>>> l.losses
[1.0]
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> x = tf.keras.layers.Dense(10)(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Activity regularization.
>>> len(model.losses)
0
>>> model.add_loss(tf.abs(tf.reduce_mean(x)))
>>> len(model.losses)
1
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> d = tf.keras.layers.Dense(10, kernel_initializer='ones')
>>> x = d(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Weight regularization.
>>> model.add_loss(lambda: tf.reduce_mean(d.kernel))
>>> model.losses
[<tf.Tensor: shape=(), dtype=float32, numpy=1.0>]
```

Returns A list of tensors.

property metrics

List of metrics added using the `add_metric()` API.

Example:

```
>>> input = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2)
>>> output = d(input)
>>> d.add_metric(tf.reduce_max(output), name='max')
>>> d.add_metric(tf.reduce_min(output), name='min')
>>> [m.name for m in d.metrics]
['max', 'min']
```

Returns A list of `Metric` objects.

property name

Name of the layer (string), set in the constructor.

property name_scope

Returns a `tf.name_scope` instance for this class.

property non_trainable_variables

Sequence of non-trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don't expect the return value to change.

Returns A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

property non_trainable_weights

List of all non-trainable weights tracked by this layer.

Non-trainable weights are *not* updated during training. They are expected to be updated manually in `call()`.

Returns A list of non-trainable variables.

property outbound_nodes

Return Functional API nodes downstream of this layer.

property output

Retrieves the output tensor(s) of a layer.

Only applicable if the layer has exactly one output, i.e. if it is connected to one incoming layer.

Returns Output tensor or list of output tensors.

Raises

- **AttributeError** – if the layer is connected to more than one incoming layers.
- **RuntimeError** – if called in Eager mode.

property output_mask

Retrieves the output mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns Output mask tensor (potentially None) or list of output mask tensors.

Raises

- **AttributeError** – if the layer is connected to
- **more than one incoming layers.** –

property output_shape

Retrieves the output shape(s) of a layer.

Only applicable if the layer has one output, or if all outputs have the same shape.

Returns Output shape, as an integer shape tuple (or list of shape tuples, one tuple per output tensor).

Raises

- **AttributeError** – if the layer has no defined output shape.
- **RuntimeError** – if called in Eager mode.

set_weights(weights)

Sets the weights of the layer, from NumPy arrays.

The weights of a layer represent the state of the layer. This function sets the weight values from numpy arrays. The weight values should be passed in the order they are created by the layer. Note that the layer's weights must be instantiated before calling this function, by calling the layer.

For example, a *Dense* layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another *Dense* layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
...     kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
       [2.],
       [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
```

Parameters `weights` – a list of NumPy arrays. The number of arrays and their shape must match number of the dimensions of the weights of the layer (i.e. it should match the output of `get_weights`).

Raises `ValueError` – If the provided weights list does not match the layer’s specifications.

property stateful

property submodules

Sequence of all sub-modules.

Submodules are modules which are properties of this module, or found as properties of modules which are properties of this module (and so on).

```
>>> a = tf.Module()
>>> b = tf.Module()
>>> c = tf.Module()
>>> a.b = b
>>> b.c = c
>>> list(a.submodules) == [b, c]
True
>>> list(b.submodules) == [c]
True
>>> list(c.submodules) == []
True
```

Returns A sequence of all submodules.

property supports_masking

Whether this layer supports computing a mask using `compute_mask`.

property trainable

property trainable_variables

Sequence of trainable variables owned by this module and its submodules.

Note: this method uses reflection to find variables on the current instance and submodules. For performance reasons you may wish to cache the result of calling this method if you don’t expect the return value to change.

Returns A sequence of variables for the current module (sorted by attribute name) followed by variables from all submodules recursively (breadth first).

property trainable_weights

List of all trainable weights tracked by this layer.

Trainable weights are updated via gradient descent during training.

Returns A list of trainable variables.

property updates

property variable_dtype

Alias of `Layer.dtype`, the dtype of the weights.

property variables

Returns the list of all layer variables/weights.

Alias of `self.weights`.

Note: This will not track the weights of nested `tf.Modules` that are not themselves Keras layers.

Returns A list of variables.

property weights

Returns the list of all layer variables/weights.

Returns A list of variables.

classmethod with_name_scope(method)

Decorator to automatically enter the module name scope.

```
>>> class MyModule(tf.Module):
...     @tf.Module.with_name_scope
...     def __call__(self, x):
...         if not hasattr(self, 'w'):
...             self.w = tf.Variable(tf.random.normal([x.shape[1], 3]))
...         return tf.matmul(x, self.w)
```

Using the above module would produce `tf.Variable`'s and `tf.Tensor`'s whose names included the module name:

```
>>> mod = MyModule()
>>> mod(tf.ones([1, 2]))
<tf.Tensor: shape=(1, 3), dtype=float32, numpy=..., dtype=float32>
>>> mod.w
<tf.Variable 'my_module/Variable:0' shape=(2, 3) dtype=float32,
numpy=..., dtype=float32>
```

Parameters `method` – The method to wrap.

Returns The original method wrapped such that it enters the module's name scope.

`tensorcircuit.keras.load_func(*path: str, fallback: Optional[Callable[[], Any]] = None) → Callable[[], Any]`

Load function from the files in the `tf.savedmodel` format. We can load several functions at the same time, as they can be the same function of different input shapes.

Example

```
@tf.function
def test_circuit(weights):
    param = tf.cast(weights, tf.complex64)
    c = tc.Circuit(2)
    c.H(0)
    c.rx(0, theta=param[0])
    c.ry(1, theta=param[1])
    return tf.math.real(c.expectation((tc.gates.x(), [1])))
```

```
>>> test_circuit(weights=tf.ones([2]))
tf.Tensor(0.84147096, shape=(), dtype=float32)
>>> K.save_func(test_circuit, save_path)
>>> circuit_loaded = K.load_func(save_path)
>>> circuit_loaded(weights=tf.ones([2]))
tf.Tensor(0.84147096, shape=(), dtype=float32)
```

Parameters `fallback (Optional[Callable[[], Any]], optional)` – The fallback function when all functions loaded are failed, defaults to None

Raises ValueError – When there is not legal loaded function of the input shape and no fallback callable.

Returns A function that tries all loaded function against the input until the first success one.

Return type Callable[..., Any]

```
tensorcircuit.keras.output_asis_loss(y_true: tensorflow.python.framework.ops.Tensor, y_pred:  
                                     tensorflow.python.framework.ops.Tensor) →  
                                     tensorflow.python.framework.ops.Tensor
```

The keras loss function that directly taking the model output as the loss.

Parameters

- **y_true** (*tf.Tensor*) – Ignoring this parameter.
- **y_pred** (*tf.Tensor*) – Model output.

Returns Model output, which is y_pred.

Return type tf.Tensor

```
tensorcircuit.keras.save_func(f: Callable[..., Any], path: str) → None
```

Save tf function in the file (tf.savedmodel format).

Example

```
@tf.function  
def test_circuit(weights):  
    param = tf.cast(weights, tf.complex64)  
    c = tc.Circuit(2)  
    c.H(0)  
    c.rx(0, theta=param[0])  
    c.ry(1, theta=param[1])  
    return tf.math.real(c.expectation((tc.gates.x(), [1])))
```

```
>>> test_circuit(weights=tf.ones([2]))  
tf.Tensor(0.84147096, shape=(), dtype=float32)  
>>> K.save_func(test_circuit, save_path)  
>>> circuit_loaded = K.load_func(save_path)  
>>> circuit_loaded(weights=tf.ones([2]))  
tf.Tensor(0.84147096, shape=(), dtype=float32)  
>>> os.system(f"tree {save_path}")  
~/model  
|   saved_model.pb  
└── assets  
    └── variables  
        ├── variables.data-00000-of-00001  
        └── variables.index
```

Parameters

- **f** (*Callable[..., Any]*) – *tf.function* ed function with graph building
- **path** (*str*) – the dir path to save the function

4.1.14 tensorcircuit.mps_base

FiniteMPS from tensornetwork with bug fixed

```
class tensorcircuit.mps_base.FiniteMPS(tensors: List[Any], center_position: Optional[int] = None,
                                         canonicalize: Optional[bool] = True, backend:
                                         Optional[Union[str,
                                         tensornetwork.backends.abstract_backend.AbstractBackend]] =
                                         None)
Bases: tensornetwork.matrixproductstates.finite_mps.FiniteMPS

__init__(tensors: List[Any], center_position: Optional[int] = None, canonicalize: Optional[bool] = True,
         backend: Optional[Union[str, tensornetwork.backends.abstract_backend.AbstractBackend]] =
         None) → None
```

Initialize a *FiniteMPS*. If *canonicalize* is *True* the state is brought into canonical form, with *BaseMPS.center_position* at *center_position*. if *center_position* is *None* and *canonicalize* = *True*, *BaseMPS.center_position* is set to 0.

Parameters

- **tensors** – A list of *Tensor* objects.
- **center_position** – The initial position of the center site.
- **canonicalize** – If *True* the mps is canonicalized at initialization.
- **backend** – The name of the backend that should be used to perform contractions.
Available backends are currently ‘numpy’, ‘tensorflow’, ‘pytorch’, ‘jax’

apply_one_site_gate(gate: Any, site: int) → None

Apply a one-site gate to an MPS. This routine will in general destroy any canonical form of the state. If a canonical form is needed, the user can restore it using *FiniteMPS.position* :param gate: a one-body gate :param site: the site where the gate should be applied

apply_transfer_operator(site: int, direction: Union[str, int], matrix: Any) → Any

Compute the action of the MPS transfer-operator at site *site*.

Parameters

- **site** – A site of the MPS
- **direction** –
 - if *1*, ‘*l*’ or ‘*left*’: compute the left-action of the MPS transfer-operator at *site* on the input *matrix*.
 - if *-1*, ‘*r*’ or ‘*right*’: compute the right-action of the MPS transfer-operator at *site* on the input *matrix*
- **matrix** – A rank-2 tensor or matrix.

Returns The result of applying the MPS transfer-operator to *matrix*

Return type *Tensor*

apply_two_site_gate(gate: Any, site1: int, site2: int, max_singular_values: Optional[int] = None,
 max_truncation_err: Optional[float] = None, center_position: Optional[int] =
 None, relative: bool = False) → Any

Apply a two-site gate to an MPS. This routine will in general destroy any canonical form of the state. If a canonical form is needed, the user can restore it using *FiniteMPS.position*.

Parameters

- **gate** (*Tensor*) – A two-body gate.
- **site1** (*int*) – The first site where the gate acts.
- **site2** (*int*) – The second site where the gate acts.
- **max_singular_values** (*Optional[float], optional*) – The maximum number of singular values to keep.
- **max_truncation_err** (*Optional[float], optional*) – The maximum allowed truncation error.
- **center_position** (*Optional[int], optional*) – An optional value to choose the MPS tensor at *center_position* to be isometric after the application of the gate. Defaults to *site1*. If the MPS is canonical (i.e. `BaseMPS.center_position != None`), and if the orthogonality center coincides with either *site1* or *site2*, the orthogonality center will be shifted to *center_position* (*site1* by default). If the orthogonality center does not coincide with (*site1*, *site2*) then *MPS.center_position* is set to *None*.
- **relative** (*bool*) – Multiply *max_truncation_err* with the largest singular value.

Raises ValueError – “rank of gate is {} but has to be 4”, “site1 = {} is not between 0 <= site < N - 1 = {}”, “site2 = {} is not between 1 <= site < N = {}”, “Found site2 = {}, site1 = {}. Only nearest neighbor gates are currently supported”, “f center_position = {center_position} not in {{site1, site2}}”, or “center_position = {}, but gate is applied at sites {}, {}. Truncation should only be done if the gate is applied at the center position of the MPS.”

Returns A scalar tensor containing the truncated weight of the truncation.

Return type *Tensor*

bond_dimension(*bond*) → *List*

The bond dimension of *bond*

property bond_dimensions: List

A list of bond dimensions of *BaseMPS*

canonicalize(*normalize: bool = True*) → *numpy.number*

Bring the MPS into canonical form according to *center_position*. If *center_position* is *None*, the MPS is canonicalized with *center_position = 0*.

Parameters **normalize** – If *True*, normalize matrices when shifting the orthogonality center.

Returns The norm of the MPS.

Return type *Tensor*

center_position: Optional[int]

check_canonical() → *Any*

Check whether the MPS is in the expected canonical form.

Returns The L2 norm of the vector of local deviations.

check_orthonormality(*which: str, site: int*) → *Any*

Check orthonormality of tensor at site *site*.

Parameters

- **which** –
 - if ‘l’ or ‘left’: check left orthogonality
 - if ‘r’ or ‘right’: check right orthogonality

- **site** – The site of the tensor.

Returns The L2 norm of the deviation from identity.

Return type scalar *Tensor*

Raises ValueError – If which is different from ‘l’, ‘left’, ‘r’ or ‘right’.

conj() → *tensorcircuit.mps_base.FiniteMPS*

copy() → *tensorcircuit.mps_base.FiniteMPS*

property dtype: Type[*numpy.number*]

get_tensor(site: int) → Any

Returns the *Tensor* object at *site*.

If *site* == *len(self)* - 1 *BaseMPS.connector_matrix* is absorbed from the right-hand side into the returned *Tensor* object.

Parameters **site** – The site for which to return the *Node*.

Returns The tensor at *site*.

Return type *Tensor*

left_envs(sites: Sequence[int]) → Dict

Compute left reduced density matrices for site *sites*. This returns a dict *left_envs* mapping sites (int) to Tensors. *left_envs[site]* is the left-reduced density matrix to the left of site *site*.

Parameters **sites** (list of int) – A list of sites of the MPS.

Returns

The left-reduced density matrices at each site in *sites*.

Return type dict mapping int to *Tensor*

left_transfer_operator(A, l, Abar)

measure_local_operator(ops: List[Any], sites: Sequence[int]) → List[Any]

Measure the expectation value of local operators *ops* site *sites*.

Parameters

- **ops** (List[*Tensor*]) – A list Tensors of rank 2; the local operators to be measured.
- **sites** (Sequence[int]) – Sites where *ops* act.

Returns measurements $\langle \text{ops}[n] \rangle$ for n in *sites*

Return type List[*Tensor*]

measure_two_body_correlator(op1: Any, op2: Any, site1: int, sites2: Sequence[int]) → List[Any]

Compute the correlator $\langle \text{op1}[site1], \text{op2}[s] \rangle$ between *site1* and all sites *s* in *sites2*. If *s* == *site1*, *op2[s]* will be applied first.

Parameters

- **op1** (*Tensor*) – Tensor of rank 2; the local operator at *site1*.
- **op2** (*Tensor*) – Tensor of rank 2; the local operator at *sites2*.
- **site1** (int) – The site where *op1* acts
- **sites2** (Sequence[int]) – Sites where operator *op2* acts.

Returns Correlator $\langle \text{op1}[site1], \text{op2}[s] \rangle$ for *s* ∈ *sites2*.

Return type List[`Tensor`]

property physical_dimensions: List

A list of physical Hilbert-space dimensions of `BaseMPS`

position(site: `int`, normalize: `Optional[bool] = True`, D: `Optional[int] = None`, max_truncation_err:

`Optional[float] = None`) → `numpy.number`

Shift `center_position` to `site`.

Parameters

- **site** – The site to which `FiniteMPS.center_position` should be shifted
- **normalize** – If `True`, normalize matrices when shifting.
- **D** – If not `None`, truncate the MPS bond dimensions to `D`.
- **max_truncation_err** – if not `None`, truncate each bond dimension, but keeping the truncation error below `max_truncation_err`.

Returns The norm of the tensor at `FiniteMPS.center_position`

Return type `Tensor`

Raises `ValueError` – If `center_position` is `None`.

classmethod random(d: `List[int]`, D: `List[int]`, dtype: `Type[numpy.number]`, canonicalize: `bool = True`, backend: `Optional[Union[str, tensornetwork.backends.abstract_backend.AbstractBackend]] = None`) → `tensornetwork.matrixproductstates.finite_mps.FiniteMPS`

Initialize a random `FiniteMPS`. The resulting state is normalized. Its center-position is at 0.

Parameters

- **d** – A list of physical dimensions.
- **D** – A list of bond dimensions.
- **dtype** – A numpy dtype.
- **backend** – An optional backend.

Returns `FiniteMPS`

right_envs(sites: `Sequence[int]`) → `Dict`

Compute right reduced density matrices for site `sites`. This returns a dict `'right_envs'` mapping sites (`int`) to `Tensors`. `right_envs[site]` is the right-reduced density matrix to the right of site `site`.

Parameters `sites (list of int)` – A list of sites of the MPS.

Returns

The right-reduced density matrices at each site in `sites`.

Return type `dict` mapping `int` to `Tensor`

right_transfer_operator(B, r, Bbar)

save(path: `str`)

4.1.15 tensorcircuit.mpscircuit

Quantum circuit: MPS state simulator

```
class tensorcircuit.mpscircuit.MPSCircuit(nqubits: int, center_position: Optional[int] = None, tensors: Optional[Sequence[Any]] = None, wavefunction: Optional[Union[tensorcircuit.quantum.QuVector, Any]] = None, split: Optional[Dict[str, Any]] = None)
```

Bases: `tensorcircuit.abstractcircuit.AbstractCircuit`

MPSCircuit class. Simple usage demo below.

```
mps = tc.MPSCircuit(3)
mps.H(1)
mps.CNOT(0, 1)
mps.rx(2, theta=tc.num_to_tensor(1.))
mps.expectation((tc.gates.z(), 2))
```

ANY(*index: int, **vars: Any) → None

Apply ANY gate with parameters on the circuit. See `tensorcircuit.gates.any_gate()`.

Parameters

- **index** (`int`) – Qubit number that the gate applies on.
- **vars** (`float`) – Parameters for the gate.

CNOT(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply CNOT gate on the circuit. See `tensorcircuit.gates.cnot_gate()`.

Parameters **index** (`int`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

CPhase(*index: int, **vars: Any) → None

Apply CPhase gate with parameters on the circuit. See `tensorcircuit.gates.cphase_gate()`.

Parameters

- **index** (`int`) – Qubit number that the gate applies on.
- **vars** (`float`) – Parameters for the gate.

CR(*index: int, **vars: Any) → None

Apply CR gate with parameters on the circuit. See `tensorcircuit.gates.cr_gate()`.

Parameters

- **index** (`int`) – Qubit number that the gate applies on.
- **vars** (`float`) – Parameters for the gate.

CRX(*index: int, **vars: Any) → None

Apply CRX gate with parameters on the circuit. See `tensorcircuit.gates.crx_gate()`.

Parameters

- **index** (`int`) – Qubit number that the gate applies on.
- **vars** (`float`) – Parameters for the gate.

CRY(*index: int, **vars: Any) → None

Apply **CRY** gate with parameters on the circuit. See `tensorcircuit.gates.cry_gate()`.

Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

CRZ(*index: int, **vars: Any) → None

Apply **CRZ** gate with parameters on the circuit. See `tensorcircuit.gates.crz_gate()`.

Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

CU(*index: int, **vars: Any) → None

Apply **CU** gate with parameters on the circuit. See `tensorcircuit.gates.cu_gate()`.

Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

CY(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **CY** gate on the circuit. See `tensorcircuit.gates.cy_gate()`.

Parameters **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. - 1.j \\ 0. + 0.j & 0. + 0.j & 0. + 1.j & 0. + 0.j \end{bmatrix}$$

CZ(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **CZ** gate on the circuit. See `tensorcircuit.gates.cz_gate()`.

Parameters **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & -1. + 0.j \end{bmatrix}$$

EXP(*index: int, **vars: Any) → None

Apply **EXP** gate with parameters on the circuit. See `tensorcircuit.gates.exp_gate()`.

Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

EXP1(*index: int, **vars: Any) → None

Apply **EXP1** gate with parameters on the circuit. See `tensorcircuit.gates.exp1_gate()`.

Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

FREDKIN(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None
Apply **FREDKIN** gate on the circuit. See `tensorcircuit.gates.fredkin_gate()`.

Parameters `index` (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

H(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None
Apply **H** gate on the circuit. See `tensorcircuit.gates.h_gate()`.

Parameters `index` (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & 0.70710677 + 0.j \\ 0.70710677 + 0.j & -0.70710677 + 0.j \end{bmatrix}$$

I(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None
Apply **I** gate on the circuit. See `tensorcircuit.gates.i_gate()`.

Parameters `index` (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

ISWAP(*index: int, **vars: Any) → None
Apply **ISWAP** gate with parameters on the circuit. See `tensorcircuit.gates.iswap_gate()`.

Parameters

- `index` (int.) – Qubit number that the gate applies on.
- `vars` (float.) – Parameters for the gate.

MPO(*index: int, **vars: Any) → None
Apply mpo gate in MPO format on the circuit. See `tensorcircuit.gates.mpo_gate()`.

Parameters

- `index` (int.) – Qubit number that the gate applies on.
- `vars` (float.) – Parameters for the gate.

classmethod MPO_to_gate(tensors: Sequence[Any]) → `tensorcircuit.gates.Gate`
Convert MPO to gate

MULTICONTROL(*index: int, **vars: Any) → None
Apply multicontrol gate in MPO format on the circuit. See `tensorcircuit.gates.multicontrol_gate()`.

Parameters

- `index` (int.) – Qubit number that the gate applies on.
- `vars` (float.) – Parameters for the gate.

ORX(*index: int, **vars: Any) → NoneApply **ORX** gate with parameters on the circuit. See `tensorcircuit.gates.orx_gate()`.**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

ORY(*index: int, **vars: Any) → NoneApply **ORY** gate with parameters on the circuit. See `tensorcircuit.gates.ory_gate()`.**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

ORZ(*index: int, **vars: Any) → NoneApply **ORZ** gate with parameters on the circuit. See `tensorcircuit.gates.orz_gate()`.**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

OX(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → NoneApply **OX** gate on the circuit. See `tensorcircuit.gates.ox_gate()`.**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

OY(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → NoneApply **OY** gate on the circuit. See `tensorcircuit.gates.oy_gate()`.**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j & 0. + 0.j & 0. + 0.j \\ 0. + 1.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

OZ(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → NoneApply **OZ** gate on the circuit. See `tensorcircuit.gates.oz_gate()`.**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

PHASE(*index: int, **vars: Any) → NoneApply **PHASE** gate with parameters on the circuit. See `tensorcircuit.gates.phase_gate()`.**Parameters**

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

R(**index*: *int*, ***vars*: *Any*) → None

Apply **R** gate with parameters on the circuit. See [tensorcircuit.gates.r_gate\(\)](#).

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

RX(**index*: *int*, ***vars*: *Any*) → None

Apply **RX** gate with parameters on the circuit. See [tensorcircuit.gates.rx_gate\(\)](#).

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

RXX(**index*: *int*, ***vars*: *Any*) → None

Apply **RXX** gate with parameters on the circuit. See [tensorcircuit.gates.rxx_gate\(\)](#).

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

RY(**index*: *int*, ***vars*: *Any*) → None

Apply **RY** gate with parameters on the circuit. See [tensorcircuit.gates.ry_gate\(\)](#).

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

RYY(**index*: *int*, ***vars*: *Any*) → None

Apply **RYY** gate with parameters on the circuit. See [tensorcircuit.gates.ryy_gate\(\)](#).

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

RZ(**index*: *int*, ***vars*: *Any*) → None

Apply **RZ** gate with parameters on the circuit. See [tensorcircuit.gates.rz_gate\(\)](#).

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

RZZ(**index*: *int*, ***vars*: *Any*) → None

Apply **RZZ** gate with parameters on the circuit. See [tensorcircuit.gates.rzz_gate\(\)](#).

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

S(**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply **S** gate on the circuit. See [tensorcircuit.gates.s_gate\(\)](#).

Parameters `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 1.j \end{bmatrix}$$

`SD(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **SD** gate on the circuit. See `tensorcircuit.gates.sd_gate()`.

Parameters `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

`SWAP(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **SWAP** gate on the circuit. See `tensorcircuit.gates.swap_gate()`.

Parameters `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`T(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **T** gate on the circuit. See `tensorcircuit.gates.t_gate()`.

Parameters `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 + 0.70710677j & \end{bmatrix}$$

`TD(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **TD** gate on the circuit. See `tensorcircuit.gates.td_gate()`.

Parameters `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

`TOFFOLI(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

Parameters `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

`U(*index: int, **vars: Any) → None`

Apply **U** gate with parameters on the circuit. See `tensorcircuit.gates.u_gate()`.

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

WROOT(**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None
Apply **WROOT** gate on the circuit. See `tensorcircuit.gates.wroot_gate()`.

Parameters **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & -0.5 & -0.5j \\ 0.5 & -0.5j & 0.70710677 + 0.j \end{bmatrix}$$

X(**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None
Apply **X** gate on the circuit. See `tensorcircuit.gates.x_gate()`.

Parameters **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j \\ 1. + 0.j & 0. + 0.j \end{bmatrix}$$

Y(**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None
Apply **Y** gate on the circuit. See `tensorcircuit.gates.y_gate()`.

Parameters **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j \\ 0. + 1.j & 0. + 0.j \end{bmatrix}$$

Z(**index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*) → None
Apply **Z** gate on the circuit. See `tensorcircuit.gates.z_gate()`.

Parameters **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j \end{bmatrix}$$

__init__(**nqubits: int, center_position: Optional[int] = None, tensors: Optional[Sequence[Any]] = None, wavefunction: Optional[Union[tensorcircuit.quantum.QuVector, Any]] = None, split: Optional[Dict[str, Any]] = None*) → None
MPSCircuit object based on state simulator.

Parameters

- **nqubits** (*int*) – The number of qubits in the circuit.
- **center_position** (*int, optional*) – The center position of MPS, default to 0
- **tensors** (*Sequence[Tensor], optional*) – If not None, the initial state of the circuit is taken as **tensors** instead of $|0\rangle^n$ qubits, defaults to None. When **tensors** are specified, if **center_position** is None, then the tensors are canonicalized, otherwise it is assumed the tensors are already canonicalized at the **center_position**
- **wavefunction** (*Tensor*) – If not None, it is transformed to the MPS form according to the split rules
- **split** (*Any*) – Split rules

amplitude(*l: str*) → Any

any(**index*: int, ***vars*: Any) → None

Apply ANY gate with parameters on the circuit. See [tensorcircuit.gates.any_gate\(\)](#).

Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

append(*c*: tensorcircuit.abstractcircuit.AbstractCircuit, *indices*: Optional[List[int]] = None) →

tensorcircuit.abstractcircuit.AbstractCircuit

append circuit *c* before

Example

```
>>> c1 = tc.Circuit(2)
>>> c1.H(0)
>>> c1.H(1)
>>> c2 = tc.Circuit(2)
>>> c2.cnot(0, 1)
>>> c1.append(c2)
<tensorcircuit.circuit.Circuit object at 0x7f8402968970>
>>> c1.draw()
_____
q_0: H ┌─┐
      └─┘
q_1: H ┌── X ─┐
      └───┘
```

Parameters

- **c** (*BaseCircuit*) – The other circuit to be appended
- **indices** (Optional[List[int]], optional) – the qubit indices to which *c* is appended on. Defaults to None, which means plain concatenation.

Returns The composed circuit

Return type *BaseCircuit*

append_from_qir(*qir*: List[Dict[str, Any]]) → None

Apply the circuit in form of quantum intermediate representation after the current circuit.

Example

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.to_qir()
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None,
 ↪'mpo': False}]
>>> c2 = tc.Circuit(3)
>>> c2.CNOT(0, 1)
>>> c2.to_qir()
[{'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None,
 ↪'mpo': False}]
>>> c.append_from_qir(c2.to_qir())
>>> c.to_qir()
```

(continues on next page)

(continued from previous page)

```
[{'gatef': h, 'gate': Gate(...), 'index': (0,), 'name': 'h', 'split': None,
 → 'mpo': False},
 {'gatef': cnot, 'gate': Gate(...), 'index': (0, 1), 'name': 'cnot', 'split': None,
 → 'mpo': False}]
```

Parameters `qir` (`List[Dict[str, Any]]`) – The quantum intermediate representation.

apply(`gate: Union[tensorcircuit.gates.Gate, tensorcircuit.quantum.QuOperator]`, `*index: int, name: Optional[str] = None, split: Optional[Dict[str, Any]] = None, mpo: bool = False, ir_dict: Optional[Dict[str, Any]] = None`) → `None`
Apply a general qubit gate on MPS.

Parameters

- `gate` (`Gate`) – The Gate to be applied
- `index` (`int`) – Qubit indices of the gate

Raises `ValueError` – “MPS does not support application of gate on > 2 qubits.”

apply_MPO(`tensors: Sequence[Any], index_left: int, center_left: bool = True, split: Optional[Dict[str, Any]] = None`) → `None`
Apply a MPO to the MPS

apply_adjacent_double_gate(`gate: tensorcircuit.gates.Gate, index1: int, index2: int, center_position: Optional[int] = None, split: Optional[Dict[str, Any]] = None`) → `None`
Apply a double qubit gate on adjacent qubits of Matrix Product States (MPS).

Parameters

- `gate` (`Gate`) – The Gate to be applied
- `index1` (`int`) – The first qubit index of the gate
- `index2` (`int`) – The second qubit index of the gate
- `center_position` (`Optional[int]`) – Center position of MPS, default is None

apply_double_gate(`gate: tensorcircuit.gates.Gate, index1: int, index2: int, split: Optional[Dict[str, Any]] = None`) → `None`
Apply a double qubit gate on MPS.

Parameters

- `gate` (`Gate`) – The Gate to be applied
- `index1` (`int`) – The first qubit index of the gate
- `index2` (`int`) – The second qubit index of the gate

apply_general_gate(`gate: Union[tensorcircuit.gates.Gate, tensorcircuit.quantum.QuOperator]`, `*index: int, name: Optional[str] = None, split: Optional[Dict[str, Any]] = None, mpo: bool = False, ir_dict: Optional[Dict[str, Any]] = None`) → `None`
Apply a general qubit gate on MPS.

Parameters

- `gate` (`Gate`) – The Gate to be applied
- `index` (`int`) – Qubit indices of the gate

Raises `ValueError` – “MPS does not support application of gate on > 2 qubits.”

```
static apply_general_gate_delayed(gatef: Callable[[], tensorcircuit.gates.Gate], name: Optional[str] = None, mpo: bool = False) → Callable[..., None]
```

```
static apply_general_variable_gate_delayed(gatef: Callable[..., tensorcircuit.gates.Gate], name: Optional[str] = None, mpo: bool = False) → Callable[..., None]
```

apply_nqubit_gate(gate: tensorcircuit.gates.Gate, *index: int, split: Optional[Dict[str, Any]] = None) → None

Apply a n-qubit gate by transforming the gate to MPO

apply_single_gate(gate: tensorcircuit.gates.Gate, index: int) → None

Apply a single qubit gate on MPS; no truncation is needed.

Parameters

- **gate** ([Gate](#)) – gate to be applied
- **index** ([int](#)) – Qubit index of the gate

barrier_instruction(*index: List[int]) → None

add a barrier instruction flag, no effect on numerical simulation

Parameters **index** ([List\[int\]](#)) – the corresponding qubits

ccnot(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **TOFFOLI** gate on the circuit. See [tensorcircuit.gates.toffoli_gate\(\)](#).

Parameters **index** ([int](#)) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

ccx(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **TOFFOLI** gate on the circuit. See [tensorcircuit.gates.toffoli_gate\(\)](#).

Parameters **index** ([int](#)) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

circuit_param: Dict[str, Any]

cnot(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **CNOT** gate on the circuit. See [tensorcircuit.gates.cnot_gate\(\)](#).

Parameters `index` (`int`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

`cond_measure(index: int) → Any`

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a int Tensor and thus maintained a jittable pipeline.

Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
# two possible outputs: (1, 1) or (-1, -1)
```

Note: In terms of DMCircuit, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measuremet results

Parameters `index` (`int`) – the qubit for the z-basis measurement

Returns 0 or 1 for z measurement on up and down freedom

Return type Tensor

`cond_measurement(index: int) → Any`

Measurement on z basis at `index` qubit based on quantum amplitude (not post-selection). The highlight is that this method can return the measured result as a int Tensor and thus maintained a jittable pipeline.

Example

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> r = c.cond_measurement(0)
>>> c.conditional_gate(r, [tc.gates.i(), tc.gates.x()], 1)
>>> c.expectation([tc.gates.z(), [0]]), c.expectation([tc.gates.z(), [1]])
# two possible outputs: (1, 1) or (-1, -1)
```

Note: In terms of DMCircuit, this method returns nothing and the density matrix after this method is kept in mixed state without knowing the measuremet results

Parameters `index` (`int`) – the qubit for the z-basis measurement

Returns 0 or 1 for z measurement on up and down freedom

Return type Tensor

`conditional_gate(which: Any, kraus: Sequence[tensorcircuit.gates.Gate], *index: int) → None`

Apply which-th gate from `kraus` list, i.e. apply `kraus[which]`

Parameters

- **which** (*Tensor*) – Tensor of shape [] and dtype int
- **kraus** (*Sequence[Gate]*) – A list of gate in the form of `tc.gate` or *Tensor*
- **index** (*int*) – the qubit lines the gate applied on

conj() → *tensorcircuit.mpscircuit.MPSCircuit*

Compute the conjugate of the current MPS.

Returns The constructed MPS

Return type *MPSCircuit*

consecutive_swap(*index_from*: *int*, *index_to*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*) → *None*

copy() → *tensorcircuit.mpscircuit.MPSCircuit*

Copy the current MPS.

Returns The constructed MPS

Return type *MPSCircuit*

copy_without_tensor() → *tensorcircuit.mpscircuit.MPSCircuit*

Copy the current MPS without the tensors.

Returns The constructed MPS

Return type *MPSCircuit*

cphase(**index*: *int*, ***vars*: *Any*) → *None*

Apply CPHASE gate with parameters on the circuit. See `tensorcircuit.gates.cphase_gate()`.

Parameters

- **index** (*int*) – Qubit number that the gate applies on.
- **vars** (*float*) – Parameters for the gate.

cr(**index*: *int*, ***vars*: *Any*) → *None*

Apply CR gate with parameters on the circuit. See `tensorcircuit.gates.cr_gate()`.

Parameters

- **index** (*int*) – Qubit number that the gate applies on.
- **vars** (*float*) – Parameters for the gate.

crx(**index*: *int*, ***vars*: *Any*) → *None*

Apply CRX gate with parameters on the circuit. See `tensorcircuit.gates.crx_gate()`.

Parameters

- **index** (*int*) – Qubit number that the gate applies on.
- **vars** (*float*) – Parameters for the gate.

cry(**index*: *int*, ***vars*: *Any*) → *None*

Apply CRY gate with parameters on the circuit. See `tensorcircuit.gates.cry_gate()`.

Parameters

- **index** (*int*) – Qubit number that the gate applies on.
- **vars** (*float*) – Parameters for the gate.

crz(**index*: *int*, ***vars*: *Any*) → *None*

Apply CRZ gate with parameters on the circuit. See `tensorcircuit.gates.crz_gate()`.

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

cswap(**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*

Apply **FREDKIN** gate on the circuit. See `tensorcircuit.gates.fredkin_gate()`.

Parameters **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

cu(**index*: *int*, ***vars*: *Any*) → *None*

Apply **CU** gate with parameters on the circuit. See `tensorcircuit.gates.cu_gate()`.

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

cx(**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*

Apply **CNOT** gate on the circuit. See `tensorcircuit.gates.cnot_gate()`.

Parameters **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \end{bmatrix}$$

cy(**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*

Apply **CY** gate on the circuit. See `tensorcircuit.gates.cy_gate()`.

Parameters **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. - 1.j \\ 0. + 0.j & 0. + 0.j & 0. + 1.j & 0. + 0.j \end{bmatrix}$$

cz(**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*

Apply **CZ** gate on the circuit. See `tensorcircuit.gates.cz_gate()`.

Parameters **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & -1. + 0.j \end{bmatrix}$$

draw(kws: Any) → Any**

Visualise the circuit. This method receives the keywords as same as qiskit.circuit.QuantumCircuit.draw. More details can be found here: <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.draw.html>.

Example

```
>>> c = tc.Circuit(3)
>>> c.H(1)
>>> c.X(2)
>>> c.CNOT(0, 1)
>>> c.draw(output='text')
q_0: _____
      |
q_1: H   | X   |
      |   |   |
      |   |
q_2: X   | _____
```

exp(*index: int, **vars: Any) → None

Apply EXP gate with parameters on the circuit. See [tensorcircuit.gates.exp_gate\(\)](#).

Parameters

- **index (int.)** – Qubit number that the gate applies on.
- **vars (float.)** – Parameters for the gate.

exp1(*index: int, **vars: Any) → None

Apply EXP1 gate with parameters on the circuit. See [tensorcircuit.gates.exp1_gate\(\)](#).

Parameters

- **index (int.)** – Qubit number that the gate applies on.
- **vars (float.)** – Parameters for the gate.

expectation(*ops: Tuple[tensorcircuit.gates.Gate, List[int]], reuse: bool = True, other:

*Optional[tensorcircuit.mpscircuit.MPSCircuit] = None, conj: bool = True, normalize: bool = False, split: Optional[Dict[str, Any]] = None, **kws: Any] → Any*

Compute the expectation of corresponding operators in the form of tensor.

Parameters

- **ops (Tuple[tn.Node, List[int]])** – Operator and its position on the circuit, eg. (gates.Z(), [1]), (gates.X(), [2]) is for operator Z_1X_2
- **reuse (bool, optional)** – If True, then the wavefunction tensor is cached for further expectation evaluation, defaults to be true.
- **other (MPSCircuit, optional)** – If not None, will be used as bra
- **conj (bool, defaults to be True)** – Whether to conjugate the bra state
- **normalize (bool, defaults to be True)** – Whether to normalize the MPS
- **split (Any)** – Truncation split

Returns The expectation of corresponding operators

Return type Tensor

expectation_ps(*x*: *Optional[Sequence[int]]* = *None*, *y*: *Optional[Sequence[int]]* = *None*, *z*: *Optional[Sequence[int]]* = *None*, *reuse*: *bool* = *True*, *noise_conf*: *Optional[Any]* = *None*, *nmc*: *int* = *1000*, *status*: *Optional[Any]* = *None*, ***kws*: *Any*) → *Any*

Shortcut for Pauli string expectation. *x*, *y*, *z* list are for X, Y, Z positions

Example

```
>>> c = tc.Circuit(2)
>>> c.X(0)
>>> c.H(1)
>>> c.expectation_ps(x=[1], z=[0])
array(-0.99999994+0.j, dtype=complex64)
```

```
>>> c = tc.Circuit(2)
>>> c.cnot(0, 1)
>>> c.rx(0, theta=0.4)
>>> c.rx(1, theta=0.8)
>>> c.h(0)
>>> c.h(1)
>>> error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
>>> error2 = tc.channels.generaldepolarizingchannel(0.06, 2)
>>> noise_conf = NoiseConf()
>>> noise_conf.add_noise("rx", error1)
>>> noise_conf.add_noise("cnot", [error2], [[0, 1]])
>>> c.expectation_ps(x=[0], noise_conf=noise_conf, nmc=10000)
(0.46274087-3.764033e-09j)
```

Parameters

- **x** (*Optional[Sequence[int]]*, *optional*) – sites to apply X gate, defaults to *None*
- **y** (*Optional[Sequence[int]]*, *optional*) – sites to apply Y gate, defaults to *None*
- **z** (*Optional[Sequence[int]]*, *optional*) – sites to apply Z gate, defaults to *None*
- **reuse** (*bool*, *optional*) – whether to cache and reuse the wavefunction, defaults to *True*
- **noise_conf** (*Optional[NoiseConf]*, *optional*) – Noise Configuration, defaults to *None*
- **nmc** (*int*, *optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **status** (*Optional[Tensor]*, *optional*) – external randomness given by tensor uniformly from [0, 1], defaults to *None*, used for noisy circuit sampling

Returns Expectation value

Return type Tensor

fredkin(*index: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → *None*

Apply **FREDKIN** gate on the circuit. See `tensorcircuit.gates.fredkin_gate()`.

Parameters `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

classmethod `from_json(jsonstr: str, circuit_params: Optional[Dict[str, Any]] = None) → tensorcircuit.abstractcircuit.AbstractCircuit`

load json str as a Circuit

Parameters

- `jsonstr (str)` – _description_
- `circuit_params (Optional[Dict[str, Any]], optional)` – Extra circuit parameters in the format of `__init__`, defaults to None

Returns _description_

Return type `AbstractCircuit`

classmethod `from_json_file(file: str, circuit_params: Optional[Dict[str, Any]] = None) → tensorcircuit.abstractcircuit.AbstractCircuit`

load json file and convert it to a circuit

Parameters

- `file (str)` – filename
- `circuit_params (Optional[Dict[str, Any]], optional)` – _description_, defaults to None

Returns _description_

Return type `AbstractCircuit`

classmethod `from_openqasm(qasmstr: str, circuit_params: Optional[Dict[str, Any]] = None, keep_measure_order: bool = False) → tensorcircuit.abstractcircuit.AbstractCircuit`

classmethod `from_openqasm_file(file: str, circuit_params: Optional[Dict[str, Any]] = None, keep_measure_order: bool = False) → tensorcircuit.abstractcircuit.AbstractCircuit`

classmethod `from_qir(qir: List[Dict[str, Any]], circuit_params: Optional[Dict[str, Any]] = None) → tensorcircuit.abstractcircuit.AbstractCircuit`

Restore the circuit from the quantum intermediate representation.

Example

```
>>> c = tc.Circuit(3)
>>> c.H(0)
>>> c.rx(1, theta=tc.array_to_tensor(0.7))
>>> c.exp1(0, 1, unitary=tc.gates._zz_matrix, theta=tc.array_to_tensor(-0.2), u
→split=split)
>>> len(c)
```

(continues on next page)

(continued from previous page)

```

7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)
>>> qirs = c.to_qir()
>>>
>>> c = tc.Circuit.from_qir(qirs, circuit_params={"nqubits": 3})
>>> len(c._nodes)
7
>>> c.expectation((tc.gates.z(), [1]))
array(0.764842+0.j, dtype=complex64)

```

Parameters

- **qir** (*List[Dict[str, Any]]*) – The quantum intermediate representation of a circuit.
- **circuit_params** (*Optional[Dict[str, Any]]*) – Extra circuit parameters.

Returns The circuit have same gates in the qir.

Return type *Circuit*

```

classmethod from_qiskit(qc: Any, n: Optional[int] = None, inputs: Optional[List[float]] = None,
                           circuit_params: Optional[Dict[str, Any]] = None, binding_params:
                           Optional[Union[Sequence[float], Dict[Any, float]]] = None) →
                           tensorcircuit.abstractcircuit.AbstractCircuit

```

Import Qiskit QuantumCircuit object as a `tc.Circuit` object.

Example

```

>>> from qiskit import QuantumCircuit
>>> qisc = QuantumCircuit(3)
>>> qisc.h(2)
>>> qisc.cswap(1, 2, 0)
>>> qisc.swap(0, 1)
>>> c = tc.Circuit.from_qiskit(qisc)

```

Parameters

- **qc** (*QuantumCircuit* in Qiskit) – Qiskit Circuit object
- **n** (*int*) – The number of qubits for the circuit
- **inputs** (*Optional[List[float]]*, *optional*) – possible input wavefunction for `tc.Circuit`, defaults to None
- **circuit_params** (*Optional[Dict[str, Any]]*) – kwargs given in `Circuit.__init__` construction function, default to None.
- **binding_params** (*Optional[Union[Sequence[float], Dict[Any, float]]]*) – (variational) parameters for the circuit. Could be either a sequence or dictionary depending on the type of parameters in the Qiskit circuit. For `ParameterVectorElement` use sequence. For `Parameter` use dictionary

Returns The same circuit but as tensorcircuit object

Return type *Circuit*

```
classmethod from_qsim_file(file: str, circuit_params: Optional[Dict[str, Any]] = None) →  
    tensorcircuit.abstractcircuit.AbstractCircuit
```

```
gate_aliases = [['cnot', 'cx'], ['fredkin', 'cswap'], ['toffoli', 'ccnot'],  
    ['toffoli', 'ccx'], ['any', 'unitary'], ['sd', 'sdg'], ['td', 'tdg']]
```

```
gate_count(gate_list: Optional[Sequence[str]] = None) → int
```

count the gate number of the circuit

Example

```
>>> c = tc.Circuit(3)  
>>> c.h(0)  
>>> c.multicontrol(0, 1, 2, ctrl=[0, 1], unitary=tc.gates._x_matrix)  
>>> c.toffoli(1, 2, 0)  
>>> c.gate_count()  
3  
>>> c.gate_count(["multicontrol", "toffoli"])  
2
```

Parameters `gate_list` (*Optional[Sequence[str]]*, *optional*) – gate name list to be counted, defaults to None (counting all gates)

Returns the total number of all gates or gates in the `gate_list`

Return type int

```
gate_summary() → Dict[str, int]
```

return the summary dictionary on gate type - gate count pair

Returns the gate count dict by gate type

Return type Dict[str, int]

```
classmethod gate_to_MPO(gate: Union[tensorcircuit.gates.Gate, Any], *index: int) →  
    Tuple[Sequence[Any], int]
```

Convert gate to MPO form with identities at empty sites

```
get_bond_dimensions() → Any
```

Get the MPS bond dimensions

Returns MPS tensors

Return type Tensor

```
get_center_position() → Optional[int]
```

Get the center position of the MPS

Returns center position

Return type Optional[int]

```
get_norm() → Any
```

Get the normalized Center Position.

Returns Normalized Center Position.

Return type Tensor

```
get_positional_logical_mapping() → Dict[int, int]
```

Get positional logical mapping dict based on measure instruction. This function is useful when we only measure part of the qubits in the circuit, to process the count result from partial measurement, we must

be aware of the mapping, i.e. for each position in the count bitstring, what is the corresponding qubits (logical) defined on the circuit

Returns positional_logical_mapping

Return type Dict[int, int]

get_quvector() → *tensorcircuit.quantum.QuVector*

Get the representation of the output state in the form of QuVector has to be full contracted in MPS

Returns QuVector representation of the output state from the circuit

Return type *QuVector*

get_tensors() → List[Any]

Get the MPS tensors

Returns MPS tensors

Return type List[*Tensor*]

h(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **H** gate on the circuit. See `tensorcircuit.gates.h_gate()`.

Parameters **index** (*int*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & 0.70710677 + 0.j \\ 0.70710677 + 0.j & -0.70710677 + 0.j \end{bmatrix}$$

i(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply **I** gate on the circuit. See `tensorcircuit.gates.i_gate()`.

Parameters **index** (*int*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

initial_mapping(logical_physical_mapping: Dict[int, int], n: Optional[int] = None, circuit_params:

Optional[Dict[str, Any]] = None) → *tensorcircuit.abstractcircuit.AbstractCircuit*

generate a new circuit with the qubit mapping given by `logical_physical_mapping`

Parameters

- **logical_physical_mapping** (*Dict[int, int]*) – how to map logical qubits to the physical qubits on the new circuit
- **n** (*Optional[int]*, *optional*) – number of qubit of the new circuit, can be different from the original one, defaults to None
- **circuit_params** (*Optional[Dict[str, Any]]*, *optional*) – `_description_`, defaults to None

Returns `_description_`

Return type *AbstractCircuit*

inputs: Any

inverse(circuit_params: Optional[Dict[str, Any]] = None) → *tensorcircuit.abstractcircuit.AbstractCircuit*

inverse the circuit, return a new inversed circuit

EXAMPLE

```
>>> c = tc.Circuit(2)
>>> c.H(0)
>>> c.rzz(1, 2, theta=0.8)
>>> c1 = c.inverse()
```

Parameters `circuit_params` (*Optional[Dict[str, Any]]*, *optional*) – keywords dict for initialization the new circuit, defaults to None

Returns the inversed circuit

Return type `Circuit`

`is_mps: bool = True`

`is_valid() → bool`

Check whether the circuit is legal.

Returns Whether the circuit is legal.

Return type `bool`

`iswap(*index: int, **vars: Any) → None`

Apply ISWAP gate with parameters on the circuit. See `tensorcircuit.gates.iswap_gate()`.

Parameters

- `index (int.)` – Qubit number that the gate applies on.
- `vars (float.)` – Parameters for the gate.

`measure(*index: int, with_prob: bool = False, status: Optional[Any] = None) → Tuple[Any, Any]`

Take measurement to the given quantum lines.

Parameters

- `index (int)` – Measure on which quantum line.
- `with_prob (bool, optional)` – If true, theoretical probability is also returned.
- `status (Optional [Tensor])` – external randomness, with shape [index], defaults to None

Returns The sample output and probability (optional) of the quantum line.

Return type `Tuple[Tensor, Tensor]`

`measure_instruction(index: int) → None`

add a measurement instruction flag, no effect on numerical simulation

Parameters `index (int)` – the corresponding qubit

`mid_measurement(index: int, keep: int = 0) → None`

Middle measurement in the z-basis on the circuit, note the wavefunction output is not normalized with `mid_measurement` involved, one should normalized the state manually if needed.

Parameters

- `index (int)` – The index of qubit that the Z direction postselection applied on
- `keep (int, optional)` – 0 for spin up, 1 for spin down, defaults to 0

`mpo(*index: int, **vars: Any) → None`

Apply mpo gate in MPO format on the circuit. See `tensorcircuit.gates.mpo_gate()`.

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

mpogates = ['multicontrol', 'mpo']

multicontrol(**index*: *int*, ***vars*: *Any*) → None

Apply multicontrol gate in MPO format on the circuit. See [tensorcircuit.gates.multicontrol_gate\(\)](#).

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

normalize() → None

Normalize MPS Circuit according to the center position.

orx(**index*: *int*, ***vars*: *Any*) → None

Apply ORX gate with parameters on the circuit. See [tensorcircuit.gates.orx_gate\(\)](#).

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

ory(**index*: *int*, ***vars*: *Any*) → None

Apply ORY gate with parameters on the circuit. See [tensorcircuit.gates.ory_gate\(\)](#).

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

orz(**index*: *int*, ***vars*: *Any*) → None

Apply ORZ gate with parameters on the circuit. See [tensorcircuit.gates.orz_gate\(\)](#).

Parameters

- **index** (*int.*) – Qubit number that the gate applies on.
- **vars** (*float.*) – Parameters for the gate.

ox(**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply OX gate on the circuit. See [tensorcircuit.gates.ox_gate\(\)](#).

Parameters **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

oy(**index*: *int*, *split*: *Optional[Dict[str, Any]]* = *None*, *name*: *Optional[str]* = *None*) → None

Apply OY gate on the circuit. See [tensorcircuit.gates.oy_gate\(\)](#).

Parameters **index** (*int.*) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j & 0. + 0.j & 0. + 0.j \\ 0. + 1.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

oz(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply OZ gate on the circuit. See `tensorcircuit.gates.oz_gate()`.

Parameters `index` (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

phase(*index: int, **vars: Any) → None

Apply PHASE gate with parameters on the circuit. See `tensorcircuit.gates.phase_gate()`.

Parameters

- `index` (int.) – Qubit number that the gate applies on.
- `vars` (float.) – Parameters for the gate.

position(site: int) → None

Wrapper of `tn.FiniteMPS.position`. Set orthogonality center.

Parameters `site` (int) – The orthogonality center

prepend(c: `tensorcircuit.abstractcircuit.AbstractCircuit`) → `tensorcircuit.abstractcircuit.AbstractCircuit`
prepend circuit c before

Parameters `c` (`BaseCircuit`) – The other circuit to be prepended

Returns The composed circuit

Return type `BaseCircuit`

proj_with_mps(other: `tensorcircuit.mpscircuit.MPSCircuit`, conj: bool = True) → Any

Compute the projection between `other` as bra and `self` as ket.

Parameters `other` (`MPSCircuit`) – ket of the other MPS, which will be converted to bra automatically

Returns The projection in form of tensor

Return type Tensor

r(*index: int, **vars: Any) → None

Apply R gate with parameters on the circuit. See `tensorcircuit.gates.r_gate()`.

Parameters

- `index` (int.) – Qubit number that the gate applies on.
- `vars` (float.) – Parameters for the gate.

reduce_dimension(index_left: int, center_left: bool = True, split: Optional[Dict[str, Any]] = None) → None

Reduce the bond dimension between two adjacent sites by SVD

classmethod reduce_tensor_dimension(tensor_left: Any, tensor_right: Any, center_left: bool = True, split: Optional[Dict[str, Any]] = None) → Tuple[Any, Any]

Reduce the bond dimension between two general tensors by SVD

reset_instruction(index: int) → None

add a reset instruction flag, no effect on numerical simulation

Parameters `index` (int) – the corresponding qubit

rx(*index: int, **vars: Any) → NoneApply **RX** gate with parameters on the circuit. See [tensorcircuit.gates.rx_gate\(\)](#).**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

rxx(*index: int, **vars: Any) → NoneApply **RXX** gate with parameters on the circuit. See [tensorcircuit.gates.rxx_gate\(\)](#).**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

ry(*index: int, **vars: Any) → NoneApply **RY** gate with parameters on the circuit. See [tensorcircuit.gates.ry_gate\(\)](#).**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

ryy(*index: int, **vars: Any) → NoneApply **RYY** gate with parameters on the circuit. See [tensorcircuit.gates.ryy_gate\(\)](#).**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

rz(*index: int, **vars: Any) → NoneApply **RZ** gate with parameters on the circuit. See [tensorcircuit.gates.rz_gate\(\)](#).**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

rzz(*index: int, **vars: Any) → NoneApply **RZZ** gate with parameters on the circuit. See [tensorcircuit.gates.rzz_gate\(\)](#).**Parameters**

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

s(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → NoneApply **S** gate on the circuit. See [tensorcircuit.gates.s_gate\(\)](#).**Parameters** **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 1.j \end{bmatrix}$$

sd(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → NoneApply **SD** gate on the circuit. See [tensorcircuit.gates.sd_gate\(\)](#).

Parameters `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

`sdg(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **SD** gate on the circuit. See `tensorcircuit.gates.sd_gate()`.

Parameters `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. - 1.j \end{bmatrix}$$

`select_gate(which: Any, kraus: Sequence[tensorcircuit.gates.Gate], *index: int) → None`

Apply which-th gate from kraus list, i.e. apply `kraus[which]`

Parameters

- `which (Tensor)` – Tensor of shape [] and dtype int
- `kraus (Sequence[Gate])` – A list of gate in the form of `tc.gate` or `Tensor`
- `index (int)` – the qubit lines the gate applied on

`set_split_rules(split: Dict[str, Any]) → None`

Set truncation split when double qubit gates are applied. If nothing is specified, no truncation will take place and the bond dimension will keep growing. For more details, refer to `split_tensor`.

Parameters `split (Any)` – Truncation split

`sgates = ['i', 'x', 'y', 'z', 'h', 't', 's', 'td', 'sd', 'wroot', 'cnot', 'cz', 'swap', 'cy', 'ox', 'oy', 'oz', 'toffoli', 'fredkin']`

`slice(begin: int, end: int) → tensorcircuit.mpscircuit.MPSCircuit`

Get a slice of the MPS (only for internal use)

`static standardize_gate(name: str) → str`

standardize the gate name to tc common gate sets

Parameters `name (str)` – non-standard gate name

Returns the standard gate name

Return type str

`state(form: str = 'default') → Any`

Compute the output wavefunction from the circuit.

Parameters `form (str, optional)` – the str indicating the form of the output wavefunction

Returns Tensor with shape [1, -1]

Return type Tensor a b ab || |

i–A–B–j -> i–XX–j

`swap(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **SWAP** gate on the circuit. See `tensorcircuit.gates.swap_gate()`.

Parameters `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j \end{bmatrix}$$

t(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None
 Apply **T** gate on the circuit. See `tensorcircuit.gates.t_gate()`.

Parameters `index` (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 + 0.70710677j & \end{bmatrix}$$

td(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None
 Apply **TD** gate on the circuit. See `tensorcircuit.gates.td_gate()`.

Parameters `index` (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

tdg(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None
 Apply **TD** gate on the circuit. See `tensorcircuit.gates.td_gate()`.

Parameters `index` (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. & +0.j & 0. & +0.j \\ 0. & +0.j & 0.70710677 - 0.70710677j & \end{bmatrix}$$

tex(**kws: Any) → str
 Generate latex string based on quantikz latex package

Returns Latex string that can be directly compiled via, e.g. `latexit`

Return type str

to_cirq(enable_instruction: bool = False) → Any
 Translate `tc.Circuit` to a cirq circuit object.

Parameters `enable_instruction` (bool, defaults to False) – whether also export measurement and reset instructions

Returns A cirq circuit of this circuit.

to_json(file: Optional[str] = None, simplified: bool = False) → Any
 circuit dumps to json

Parameters

- `file` (Optional[str], optional) – file str to dump the json to, defaults to None, return the json str
- `simplified` (bool) – If False, keep all info for each gate, defaults to be False. If True, suitable for IO since less information is required

Returns None if dumps to file otherwise the json str

Return type Any

to_openqasm(**kws: Any) → str
 transform circuit to openqasm via qiskit circuit, see <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.qasm.html> for usage on possible options for kws

Returns circuit representation in openqasm format

Return type str

to_qir() → List[Dict[str, Any]]
 Return the quantum intermediate representation of the circuit.

Example

```
>>> c = tc.Circuit(2)
>>> c.CNOT(0, 1)
>>> c.to_qir()
[{'gatef': cnot, 'gate': Gate(
    name: 'cnot',
    tensor:
        array([[[[1.+0.j, 0.+0.j],
                 [0.+0.j, 0.+0.j]],
               [[0.+0.j, 1.+0.j],
                 [0.+0.j, 0.+0.j]],
               [[[0.+0.j, 0.+0.j],
                 [0.+0.j, 1.+0.j]],
              [[0.+0.j, 0.+0.j],
                [1.+0.j, 0.+0.j]]]], dtype=complex64),
    edges: [
        Edge(Dangling Edge)[0],
        Edge(Dangling Edge)[1],
        Edge('cnot'[2] -> 'qb-1'[0]),
        Edge('cnot'[3] -> 'qb-2'[0])
    ]),
    'index': (0, 1), 'name': 'cnot', 'split': None, 'mpo': False}]
```

Returns The quantum intermediate representation of the circuit.

Return type List[Dict[str, Any]]

to_qiskit(*enable_instruction: bool = False*) → Any

Translate `tc.Circuit` to a qiskit QuantumCircuit object.

Parameters `enable_instruction (bool, defaults to False)` – whether also export measurement and reset instructions

Returns A qiskit object of this circuit.

toffoli(index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None*)** → None

Apply **TOFFOLI** gate on the circuit. See `tensorcircuit.gates.toffoli_gate()`.

Parameters `index (int.)` – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j & 0. + 0.j \\ 0. + 0.j & 1. + 0.j \end{bmatrix}$$

u(*index: int, **vars: Any) → None

Apply U gate with parameters on the circuit. See `tensorcircuit.gates.u_gate()`.

Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

unitary(*index: int, **vars: Any) → None

Apply ANY gate with parameters on the circuit. See `tensorcircuit.gates.any_gate()`.

Parameters

- **index** (int.) – Qubit number that the gate applies on.
- **vars** (float.) – Parameters for the gate.

vgates = ['r', 'cr', 'u', 'cu', 'rx', 'ry', 'rz', 'phase', 'rxx', 'ryy', 'rzz', 'cphase', 'crx', 'cry', 'crz', 'orx', 'ory', 'orz', 'iswap', 'any', 'exp', 'exp1']

vis_tex(**kws: Any) → str

Generate latex string based on quantikz latex package

Returns Latex string that can be directly compiled via, e.g. latexit

Return type str

wavefunction(form: str = 'default') → Any

Compute the output wavefunction from the circuit.

Parameters **form** (str, optional) – the str indicating the form of the output wavefunction

Returns Tensor with shape [1, -1]

Return type Tensor a b ab || |

i-A-B-j -> i-XX-j

classmethod **wavefunction_to_tensors**(wavefunction: Any, dim_phys: int = 2, norm: bool = True, split: Optional[Dict[str, Any]] = None) → List[Any]

Construct the MPS tensors from a given wavefunction.

Parameters

- **wavefunction** (Tensor) – The given wavefunction (any shape is OK)
- **split** (Dict) – Truncation split
- **dim_phys** (int) – Physical dimension, 2 for MPS and 4 for MPO
- **norm** (bool) – Whether to normalize the wavefunction

Returns The tensors

Return type List[Tensor]

wroot(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply WROOT gate on the circuit. See `tensorcircuit.gates.wroot_gate()`.

Parameters **index** (int.) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0.70710677 + 0.j & -0.5 & -0.5j \\ 0.5 & -0.5j & 0.70710677 + 0.j \end{bmatrix}$$

x(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None

Apply X gate on the circuit. See `tensorcircuit.gates.x_gate()`.

Parameters `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 1. + 0.j \\ 1. + 0.j & 0. + 0.j \end{bmatrix}$$

`y(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **Y** gate on the circuit. See `tensorcircuit.gates.y_gate()`.

Parameters `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 0. + 0.j & 0. - 1.j \\ 0. + 1.j & 0. + 0.j \end{bmatrix}$$

`z(*index: int, split: Optional[Dict[str, Any]] = None, name: Optional[str] = None) → None`

Apply **Z** gate on the circuit. See `tensorcircuit.gates.z_gate()`.

Parameters `index` (`int.`) – Qubit number that the gate applies on. The matrix for the gate is

$$\begin{bmatrix} 1. + 0.j & 0. + 0.j \\ 0. + 0.j & -1. + 0.j \end{bmatrix}$$

`tensorcircuit.mpscircuit.split_tensor(tensor: Any, center_left: bool = True, split: Optional[Dict[str, Any]] = None) → Tuple[Any, Any]`

Split the tensor by SVD or QR depends on whether a truncation is required.

Parameters

- `tensor` (`Tensor`) – The input tensor to split.
- `center_left` (`bool, optional`) – Determine the orthogonal center is on the left tensor or the right tensor.

Returns Two tensors after splitting

Return type `Tuple[Tensor, Tensor]`

4.1.16 `tensorcircuit.noisemodel`

General Noise Model Construction.

`class tensorcircuit.noisemodel.NoiseConf`

Bases: `object`

Noise Configuration class.

```
error1 = tc.channels.generaldepolarizingchannel(0.1, 1)
error2 = tc.channels.thermalrelaxationchannel(300, 400, 100, "ByChoi", 0)
readout_error = [[0.9, 0.75], [0.4, 0.7]]

noise_conf = NoiseConf()
noise_conf.add_noise("x", error1)
noise_conf.add_noise("h", [error1, error2], [[0], [1]])
noise_conf.add_noise("readout", readout_error)
```

`__init__()` → `None`

Establish a noise configuration.

`add_noise(gate_name: str, kraus: Sequence[tensorcircuit.channels.KrausList], qubit:`

`Optional[Sequence[Any]] = None) → None`

Add noise channels on specific gates and specific qubits in form of Kraus operators.

Parameters

- `gate_name (str)` – noisy gate
- `kraus (Sequence[Gate])` – noise channel
- `qubit (Optional[Sequence[Any]], optional)` – the list of noisy qubit, defaults to None, indicating applying the noise channel on all qubits

`tensorcircuit.noisemodel.apply_qir_with_noise(c: Any, qir: List[Dict[str, Any]], noise_conf:`

`tensorcircuit.noisemodel.NoiseConf, status:`

`Optional[Any] = None) → Any`

Parameters

- `c (AbstractCircuit)` – A newly defined circuit
- `qir (List[Dict[str, Any]])` – The qir of the clean circuit
- `noise_conf (NoiseConf)` – Noise Configuration
- `status (1D Tensor, optional)` – The status for Monte Carlo sampling, defaults to None

Returns A newly constructed circuit with noise

Return type `AbstractCircuit`

`tensorcircuit.noisemodel.circuit_with_noise(c: tensorcircuit.abstractcircuit.AbstractCircuit, noise_conf:`

`tensorcircuit.noisemodel.NoiseConf, status: Optional[Any] = None) → Any`

Noisify a clean circuit.

Parameters

- `c (AbstractCircuit)` – A clean circuit
- `noise_conf (NoiseConf)` – Noise Configuration
- `status (1D Tensor, optional)` – The status for Monte Carlo sampling, defaults to None

Returns A newly constructed circuit with noise

Return type `AbstractCircuit`

`tensorcircuit.noisemodel.expectation_noisfy(c: Any, *ops:`

`Tuple[tornetwork.network_components.Node, List[int]], noise_conf: Optional[tensorcircuit.noisemodel.NoiseConf] = None, nmc: int = 1000, status: Optional[Any] = None, **kws: Any) → Any`

Calculate expectation value with noise configuration.

Parameters

- `c (Any)` – The clean circuit
- `noise_conf (Optional[NoiseConf], optional)` – Noise Configuration, defaults to None

- **nmc** (*int, optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **status** (*Optional[Tensor], optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling

Returns expectation value with noise

Return type Tensor

```
tensorcircuit.noisemodel.sample_expectation_ps_noisy(c: Any, x: Optional[Sequence[int]] = None, y: Optional[Sequence[int]] = None, z: Optional[Sequence[int]] = None, noise_conf: Optional[tensorcircuit.noisemodel.NoiseConf] = None, nmc: int = 1000, shots: Optional[int] = None, statusc: Optional[Any] = None, status: Optional[Any] = None, **kws: Any) → Any
```

Calculate sample_expectation_ps with noise configuration.

Parameters

- **c** (*Any*) – The clean circuit
- **x** (*Optional[Sequence[int]], optional*) – sites to apply X gate, defaults to None
- **y** (*Optional[Sequence[int]], optional*) – sites to apply Y gate, defaults to None
- **z** (*Optional[Sequence[int]], optional*) – sites to apply Z gate, defaults to None
- **noise_conf** (*Optional[NoiseConf], optional*) – Noise Configuration, defaults to None
- **nmc** (*int, optional*) – repetition time for Monte Carlo sampling for noisy calculation, defaults to 1000
- **shots** (*Optional[int], optional*) – number of measurement shots, defaults to None, indicating analytical result
- **statusc** (*Optional[Tensor], optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for noisy circuit sampling
- **status** (*Optional[Tensor], optional*) – external randomness given by tensor uniformly from [0, 1], defaults to None, used for measurement sampling

Returns sample expectation value with noise

Return type Tensor

4.1.17 tensorcircuit.quantum

Quantum state and operator class backend by tensornetwork

IMPORT

```
import tensorcircuit.quantum as qu
```

```
tensorcircuit.quantum.PauliString2COO(l: Sequence[int], weight: Optional[float] = None) → Any
```

Generate tensorflow sparse matrix from Pauli string sum

Parameters

- **l** (*Sequence[int]*) – 1D Tensor representing for a Pauli string, e.g. [1, 0, 0, 3, 2] is for $X_0Z_3Y_4$
- **weight** (*Optional[float], optional*) – the weight for the Pauli string defaults to None (all Pauli strings weight 1.0)

Returns the tensorflow sparse matrix

Return type Tensor

```
tensorcircuit.quantum.PauliStringSum2COO(ls: Sequence[Sequence[int]], weight:
                                         Optional[Sequence[float]] = None, numpy: bool = False) →
                                         Any
```

Generate sparse tensor from Pauli string sum. Currently requires tensorflow installed

Parameters

- **ls** (*Sequence[Sequence[int]]*) – 2D Tensor, each row is for a Pauli string, e.g. [1, 0, 0, 3, 2] is for $X_0Z_3Y_4$
- **weight** (*Optional[Sequence[float]], optional*) – 1D Tensor, each element corresponds the weight for each Pauli string defaults to None (all Pauli strings weight 1.0)
- **numpy** (*bool*) – default False. If True, return numpy coo else return backend compatible sparse tensor

Returns the scipy coo sparse matrix

Return type Tensor

```
tensorcircuit.quantum.PauliStringSum2COO_numpy(ls: Sequence[Sequence[int]], weight:
                                               Optional[Sequence[float]] = None, *, numpy: bool =
                                               True) → Any
```

Generate sparse tensor from Pauli string sum. Currently requires tensorflow installed

Parameters

- **ls** (*Sequence[Sequence[int]]*) – 2D Tensor, each row is for a Pauli string, e.g. [1, 0, 0, 3, 2] is for $X_0Z_3Y_4$
- **weight** (*Optional[Sequence[float]], optional*) – 1D Tensor, each element corresponds the weight for each Pauli string defaults to None (all Pauli strings weight 1.0)
- **numpy** (*bool*) – default False. If True, return numpy coo else return backend compatible sparse tensor

Returns the scipy coo sparse matrix

Return type Tensor

```
tensorcircuit.quantum.PauliStringSum2COO_tf(ls: Sequence[Sequence[int]], weight:
                                            Optional[Sequence[float]] = None) → Any
```

Generate tensorflow sparse matrix from Pauli string sum

Parameters

- **ls** (*Sequence[Sequence[int]]*) – 2D Tensor, each row is for a Pauli string, e.g. [1, 0, 0, 3, 2] is for $X_0Z_3Y_4$
- **weight** (*Optional[Sequence[float]], optional*) – 1D Tensor, each element corresponds the weight for each Pauli string defaults to None (all Pauli strings weight 1.0)

Returns the tensorflow coo sparse matrix

Return type Tensor

```
tensorcircuit.quantum.PauliStringSum2Dense(ls: Sequence[Sequence[int]], weight:  
    Optional[Sequence[float]] = None, numpy: bool = False)  
    → Any
```

Generate dense matrix from Pauli string sum. Currently requires tensorflow installed.

Parameters

- **ls** (`Sequence[Sequence[int]]`) – 2D Tensor, each row is for a Pauli string, e.g. [1, 0, 0, 3, 2] is for $X_0Z_3Y_4$
- **weight** (`Optional[Sequence[float]]`, `optional`) – 1D Tensor, each element corresponds the weight for each Pauli string defaults to None (all Pauli strings weight 1.0)
- **numpy** (`bool`) – default False. If True, return numpy coo else return backend compatible sparse tensor

Returns the tensorflow dense matrix

Return type Tensor

```
class tensorcircuit.quantum.QuAdjointVector(subsystem_edges:  
    Sequence[tensornetwork.network_components.Edge],  
    ref_nodes: Optional[Collection[tensornetwork.network_components.AbstractNode]]  
    = None, ignore_edges: Optional[Collection[tensornetwork.network_components.Edge]]  
    = None)
```

Bases: `tensorcircuit.quantum.QuOperator`

Represents an adjoint (row) vector via a tensor network.

```
__init__(subsystem_edges: Sequence[tensornetwork.network_components.Edge], ref_nodes:  
    Optional[Collection[tensornetwork.network_components.AbstractNode]] = None, ignore_edges:  
    Optional[Collection[tensornetwork.network_components.Edge]] = None) → None
```

Constructs a new `QuAdjointVector` from a tensor network. This encapsulates an existing tensor network, interpreting it as an adjoint vector (row vector).

Parameters

- **subsystem_edges** (`Sequence[Edge]`) – The edges of the network to be used as the input edges.
- **ref_nodes** (`Optional[Collection[AbstractNode]]`, `optional`) – Nodes used to refer to parts of the tensor network that are not connected to any input or output edges (for example: a scalar factor).
- **ignore_edges** (`Optional[Collection[Edge]]`, `optional`) – Optional collection of edges to ignore when performing consistency checks.

```
adjoint() → tensorcircuit.quantum.QuOperator
```

The adjoint of the operator. This creates a new `QuOperator` with complex-conjugate copies of all tensors in the network and with the input and output edges switched.

Returns The adjoint of the operator.

Return type `QuOperator`

```
check_network() → None
```

Check that the network has the expected dimensionality. This checks that all input and output edges are dangling and that there are no other dangling edges (except any specified in `ignore_edges`). If not, an exception is raised.

contract(*final_edge_order*: *Optional[Sequence[tensornetwork.network_components.Edge]]* = *None*) → *tensorcircuit.quantum.QuOperator*

Contract the tensor network in place. This modifies the tensor network representation of the operator (or vector, or scalar), reducing it to a single tensor, without changing the value.

Parameters **final_edge_order** (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor.

Returns The present object.

Return type *QuOperator*

copy() → *tensorcircuit.quantum.QuOperator*

The deep copy of the operator.

Returns The new copy of the operator.

Return type *QuOperator*

eval(*final_edge_order*: *Optional[Sequence[tensornetwork.network_components.Edge]]* = *None*) → Any

Contracts the tensor network in place and returns the final tensor. Note that this modifies the tensor network representing the operator. The default ordering for the axes of the final tensor is: **out_edges*, **in_edges*. If there are any “ignored” edges, their axes come first: **ignored_edges*, **out_edges*, **in_edges*.

Parameters **final_edge_order** (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor. The default ordering is determined by *out_edges* and *in_edges* (see above).

Raises **ValueError** – Node count ‘{}’ > 1 after contraction!

Returns The final tensor representing the operator.

Return type Tensor

eval_matrix(*final_edge_order*: *Optional[Sequence[tensornetwork.network_components.Edge]]* = *None*) → Any

Contracts the tensor network in place and returns the final tensor in two dimensional matrix. The default ordering for the axes of the final tensor is: (\prod dimension of *out_edges*, \prod dimension of *in_edges*)

Parameters **final_edge_order** (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor. The default ordering is determined by *out_edges* and *in_edges* (see above).

Raises **ValueError** – Node count ‘{}’ > 1 after contraction!

Returns The two-dimensional tensor representing the operator.

Return type Tensor

classmethod **from_local_tensor**(*tensor*: Any, *space*: Sequence[int], *loc*: Sequence[int], *out_axes*: *Optional[Sequence[int]]* = *None*, *in_axes*: *Optional[Sequence[int]]* = *None*) → *tensorcircuit.quantum.QuOperator*

classmethod **from_tensor**(*tensor*: Any, *subsystem_axes*: *Optional[Sequence[int]]* = *None*) → *tensorcircuit.quantum.QuAdjointVector*

Construct a *QuAdjointVector* directly from a single tensor. This first wraps the tensor in a *Node*, then constructs the *QuAdjointVector* from that *Node*.

Example

```
def show_attributes(op):
    print(f"op.is_scalar() \t\t-> {op.is_scalar()}")
    print(f"op.is_vector() \t\t-> {op.is_vector()}")
```

(continues on next page)

(continued from previous page)

```
print(f"op.is_adjoint_vector() \t-> {op.is_adjoint_vector()}")
print(f"op.eval() \n{op.eval()}"")
```

```
>>> psi_tensor = np.random.rand(2, 2)
>>> psi_tensor
array([[0.27260127, 0.91401091],
       [0.06490953, 0.38653646]])
>>> op = qu.QuAdjointVector.from_tensor(psi_tensor, [0, 1])
>>> show_attributes(op)
op.is_scalar()      -> False
op.is_vector()      -> False
op.is_adjoint_vector() -> True
op.eval()
[[0.27260127 0.91401091]
 [0.06490953 0.38653646]]
```

Parameters

- **tensor** (*Tensor*) – The tensor for constructing an QuAdjointVector.
- **subsystem_axes** (*Optional[Sequence[int]]*, *optional*) – Sequence of integer indices specifying the order in which to interpret the axes as subsystems (input edges). If not specified, the axes are taken in ascending order.

Returns The new constructed QuAdjointVector give from the given tensor.

Return type *QuAdjointVector*

property in_space: List[int]

is_adjoint_vector() → bool

Returns a bool indicating if QuOperator is an adjoint vector. Examples can be found in the *QuOperator.from_tensor*.

is_scalar() → bool

Returns a bool indicating if QuOperator is a scalar. Examples can be found in the *QuOperator.from_tensor*.

is_vector() → bool

Returns a bool indicating if QuOperator is a vector. Examples can be found in the *QuOperator.from_tensor*.

property nodes: Set[tensornetwork.network_components.AbstractNode]

All tensor-network nodes involved in the operator.

norm() → *tensorcircuit.quantum.QuOperator*

The norm of the operator. This is the 2-norm (also known as the Frobenius or Hilbert-Schmidt norm).

property out_space: List[int]

partial_trace(*subsystems_to_trace_out: Collection[int]*) → *tensorcircuit.quantum.QuOperator*

The partial trace of the operator. Subsystems to trace out are supplied as indices, so that dangling edges are connected to each other as: *out_edges[i] ^ in_edges[i]* for *i* in *subsystems_to_trace_out* This does not modify the original network. The original ordering of the remaining subsystems is maintained.

Parameters **subsystems_to_trace_out** (*Collection[int]*) – Indices of subsystems to trace out.

Returns A new QuOperator or QuScalar representing the result.

Return type *QuOperator*

projector() → *tensorcircuit.quantum.QuOperator*

The projector of the operator. The operator, as a linear operator, on the adjoint of the operator.

Set A is the operator in matrix form, then the projector of operator is defined as: $A^\dagger A$

Returns The projector of the operator.

Return type *QuOperator*

reduced_density(*subsystems_to_trace_out*: *Collection[int]*) → *tensorcircuit.quantum.QuOperator*

The reduced density of the operator.

Set A is the matrix of the operator, then the reduced density is defined as:

$$\text{Tr}_{\text{subsystems}}(A^\dagger A)$$

Firstly, take the projector of the operator, then trace out the subsystems to trace out are supplied as indices, so that dangling edges are connected to each other as: $\text{out_edges}[i] \wedge \text{in_edges}[i]$ for i in *subsystems_to_trace_out*. This does not modify the original network. The original ordering of the remaining subsystems is maintained.

Parameters **subsystems_to_trace_out** (*Collection[int]*) – Indices of subsystems to trace out.

Returns The QuOperator of the reduced density of the operator with given subsystems.

Return type *QuOperator*

property *space*: *List[int]***property** *subsystem_edges*: *List[tensornetwork.network_components.Edge]***tensor_product**(*other*: *tensorcircuit.quantum.QuOperator*) → *tensorcircuit.quantum.QuOperator*

Tensor product with another operator. Given two operators A and B , produces a new operator AB representing AB . The *out_edges* (*in_edges*) of AB is simply the concatenation of the *out_edges* (*in_edges*) of $A.\text{copy}()$ with that of $B.\text{copy}()$: $\text{new_out_edges} = [\ast \text{out_edges}_A.\text{copy}, \ast \text{out_edges}_B.\text{copy}]$ $\text{new_in_edges} = [\ast \text{in_edges}_A.\text{copy}, \ast \text{in_edges}_B.\text{copy}]$

Example

```
>>> psi = qu.QuVector.from_tensor(np.random.rand(2, 2))
>>> psi_psi = psi.tensor_product(psi)
>>> len(psi_psi.subsystem_edges)
4
>>> float(psi_psi.norm().eval())
2.9887872748523585
>>> psi.norm().eval() ** 2
2.9887872748523585
```

Parameters **other** (*QuOperator*) – The other operator (B).

Returns The result (AB).

Return type *QuOperator*

trace() → *tensorcircuit.quantum.QuOperator*

The trace of the operator.

```
class tensorcircuit.quantum.QuOperator(out_edges: Sequence[tensornetwork.network_components.Edge],
                                       in_edges: Sequence[tensornetwork.network_components.Edge],
                                       ref_nodes: Optional[Collection[tensornetwork.network_components.AbstractNode]] = None,
                                       ignore_edges: Optional[Collection[tensornetwork.network_components.Edge]] = None)
```

Bases: object

Represents a linear operator via a tensor network. To interpret a tensor network as a linear operator, some of the dangling edges must be designated as *out_edges* (output edges) and the rest as *in_edges* (input edges). Considered as a matrix, the *out_edges* represent the row index and the *in_edges* represent the column index. The (right) action of the operator on another then consists of connecting the *in_edges* of the first operator to the *out_edges* of the second. Can be used to do simple linear algebra with tensor networks.

```
__init__(out_edges: Sequence[tensornetwork.network_components.Edge], in_edges: Sequence[tensornetwork.network_components.Edge], ref_nodes: Optional[Collection[tensornetwork.network_components.AbstractNode]] = None, ignore_edges: Optional[Collection[tensornetwork.network_components.Edge]] = None) → None
```

Creates a new *QuOperator* from a tensor network. This encapsulates an existing tensor network, interpreting it as a linear operator. The network is checked for consistency: All dangling edges must either be in *out_edges*, *in_edges*, or *ignore_edges*.

Parameters

- **out_edges** (*Sequence[Edge]*) – The edges of the network to be used as the output edges.
- **in_edges** (*Sequence[Edge]*) – The edges of the network to be used as the input edges.
- **ref_nodes** (*Optional[Collection[AbstractNode]]*, *optional*) – Nodes used to refer to parts of the tensor network that are not connected to any input or output edges (for example: a scalar factor).
- **ignore_edges** (*Optional[Collection[Edge]]*, *optional*) – Optional collection of dangling edges to ignore when performing consistency checks.

Raises **ValueError** – At least one reference node is required to specify a scalar. None provided!

adjoint() → *tensorcircuit.quantum.QuOperator*

The adjoint of the operator. This creates a new *QuOperator* with complex-conjugate copies of all tensors in the network and with the input and output edges switched.

Returns The adjoint of the operator.

Return type *QuOperator*

check_network() → None

Check that the network has the expected dimensionality. This checks that all input and output edges are dangling and that there are no other dangling edges (except any specified in *ignore_edges*). If not, an exception is raised.

contract(final_edge_order: *Optional[Sequence[tensornetwork.network_components.Edge]]* = None) → *tensorcircuit.quantum.QuOperator*

Contract the tensor network in place. This modifies the tensor network representation of the operator (or vector, or scalar), reducing it to a single tensor, without changing the value.

Parameters **final_edge_order** (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor.

Returns The present object.

Return type *QuOperator*

copy() → *tensorcircuit.quantum.QuOperator*

The deep copy of the operator.

Returns The new copy of the operator.

Return type *QuOperator*

eval(*final_edge_order*: *Optional[Sequence[tensornetwork.network_components.Edge]]* = *None*) → *Any*

Contracts the tensor network in place and returns the final tensor. Note that this modifies the tensor network representing the operator. The default ordering for the axes of the final tensor is: **out_edges*, **in_edges*. If there are any “ignored” edges, their axes come first: **ignored_edges*, **out_edges*, **in_edges*.

Parameters *final_edge_order* (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor. The default ordering is determined by *out_edges* and *in_edges* (see above).

Raises **ValueError** – Node count ‘{}’ > 1 after contraction!

Returns The final tensor representing the operator.

Return type Tensor

eval_matrix(*final_edge_order*: *Optional[Sequence[tensornetwork.network_components.Edge]]* = *None*) → *Any*

Contracts the tensor network in place and returns the final tensor in two dimensional matrix. The default ordering for the axes of the final tensor is: (\prod dimension of *out_edges*, \prod dimension of *in_edges*)

Parameters *final_edge_order* (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor. The default ordering is determined by *out_edges* and *in_edges* (see above).

Raises **ValueError** – Node count ‘{}’ > 1 after contraction!

Returns The two-dimensional tensor representing the operator.

Return type Tensor

classmethod from_local_tensor(*tensor*: *Any*, *space*: *Sequence[int]*, *loc*: *Sequence[int]*, *out_axes*: *Optional[Sequence[int]]* = *None*, *in_axes*: *Optional[Sequence[int]]* = *None*) → *tensorcircuit.quantum.QuOperator*

classmethod from_tensor(*tensor*: *Any*, *out_axes*: *Optional[Sequence[int]]* = *None*, *in_axes*: *Optional[Sequence[int]]* = *None*) → *tensorcircuit.quantum.QuOperator*

Construct a *QuOperator* directly from a single tensor. This first wraps the tensor in a *Node*, then constructs the *QuOperator* from that *Node*.

Example

```
def show_attributes(op):
    print(f"op.is_scalar() \t\t-> {op.is_scalar()}")
    print(f"op.is_vector() \t\t-> {op.is_vector()}")
    print(f"op.is_adjoint_vector() \t-> {op.is_adjoint_vector()}")
    print(f"op.eval() \n{op.eval()}")
```

```
>>> psi_tensor = np.random.rand(2, 2)
>>> psi_tensor
array([[0.27260127, 0.91401091],
```

(continues on next page)

(continued from previous page)

```
[0.06490953, 0.38653646]])
>>> op = qu.QuOperator.from_tensor(psi_tensor, out_axes=[0], in_axes=[1])
>>> show_attributes(op)
op.is_scalar()      -> False
op.is_vector()      -> False
op.is_adjoint_vector() -> False
op.eval()
[[0.27260127 0.91401091]
 [0.06490953 0.38653646]]
```

Parameters

- **tensor** (*Tensor*) – The tensor.
- **out_axes** (*Optional[Sequence[int]]*, *optional*) – The axis indices of *tensor* to use as *out_edges*.
- **in_axes** (*Optional[Sequence[int]]*, *optional*) – The axis indices of *tensor* to use as *in_edges*.

Returns The new operator.

Return type *QuOperator*

property in_space: List[int]

is_adjoint_vector() → bool

Returns a bool indicating if QuOperator is an adjoint vector. Examples can be found in the *QuOperator.from_tensor*.

is_scalar() → bool

Returns a bool indicating if QuOperator is a scalar. Examples can be found in the *QuOperator.from_tensor*.

is_vector() → bool

Returns a bool indicating if QuOperator is a vector. Examples can be found in the *QuOperator.from_tensor*.

property nodes: Set[tensornetwork.network_components.AbstractNode]

All tensor-network nodes involved in the operator.

norm() → *tensorcircuit.quantum.QuOperator*

The norm of the operator. This is the 2-norm (also known as the Frobenius or Hilbert-Schmidt norm).

property out_space: List[int]

partial_trace(subsystems_to_trace_out: Collection[int]) → *tensorcircuit.quantum.QuOperator*

The partial trace of the operator. Subsystems to trace out are supplied as indices, so that dangling edges are connected to each other as: *out_edges[i] ^ in_edges[i]* for *i* in *subsystems_to_trace_out*. This does not modify the original network. The original ordering of the remaining subsystems is maintained.

Parameters **subsystems_to_trace_out** (*Collection[int]*) – Indices of subsystems to trace out.

Returns A new QuOperator or QuScalar representing the result.

Return type *QuOperator*

tensor_product(other: tensorcircuit.quantum.QuOperator) → *tensorcircuit.quantum.QuOperator*

Tensor product with another operator. Given two operators *A* and *B*, produces a new operator *AB* representing *AB*. The *out_edges* (*in_edges*) of *AB* is simply the concatenation of the *out_edges*

(*in_edges*) of *A.copy()* with that of *B.copy()*: *new_out_edges* = [**out_edges_A_copy*, **out_edges_B_copy*]
new_in_edges = [**in_edges_A_copy*, **in_edges_B_copy*]

Example

```
>>> psi = qu.QuVector.from_tensor(np.random.rand(2, 2))
>>> psi_psi = psi.tensor_product(psi)
>>> len(psi_psi.subsystem_edges)
4
>>> float(psi_psi.norm().eval())
2.9887872748523585
>>> psi.norm().eval() ** 2
2.9887872748523585
```

Parameters `other` (*QuOperator*) – The other operator (*B*).

Returns The result (*AB*).

Return type *QuOperator*

`trace()` → *tensorcircuit.quantum.QuOperator*

The trace of the operator.

```
class tensorcircuit.quantum.QuScalar(ref_nodes:
                                      Collection[tensornetwork.network_components.AbstractNode],
                                      ignore_edges:
                                      Optional[Collection[tensornetwork.network_components.Edge]] = None)
```

Bases: *tensorcircuit.quantum.QuOperator*

Represents a scalar via a tensor network.

```
__init__(ref_nodes: Collection[tensornetwork.network_components.AbstractNode], ignore_edges:
        Optional[Collection[tensornetwork.network_components.Edge]] = None) → None
```

Constructs a new *QuScalar* from a tensor network. This encapsulates an existing tensor network, interpreting it as a scalar.

Parameters

- `ref_nodes` (*Collection[AbstractNode]*) – Nodes used to refer to the tensor network (need not be exhaustive - one node from each disconnected subnetwork is sufficient).
- `ignore_edges` (*Optional[Collection[Edge]]*, *optional*) – Optional collection of edges to ignore when performing consistency checks.

`adjoint()` → *tensorcircuit.quantum.QuOperator*

The adjoint of the operator. This creates a new *QuOperator* with complex-conjugate copies of all tensors in the network and with the input and output edges switched.

Returns The adjoint of the operator.

Return type *QuOperator*

`check_network()` → None

Check that the network has the expected dimensionality. This checks that all input and output edges are dangling and that there are no other dangling edges (except any specified in *ignore_edges*). If not, an exception is raised.

contract(*final_edge_order*: *Optional[Sequence[tensornetwork.network_components.Edge]]* = *None*) → *tensorcircuit.quantum.QuOperator*

Contract the tensor network in place. This modifies the tensor network representation of the operator (or vector, or scalar), reducing it to a single tensor, without changing the value.

Parameters **final_edge_order** (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor.

Returns The present object.

Return type *QuOperator*

copy() → *tensorcircuit.quantum.QuOperator*

The deep copy of the operator.

Returns The new copy of the operator.

Return type *QuOperator*

eval(*final_edge_order*: *Optional[Sequence[tensornetwork.network_components.Edge]]* = *None*) → *Any*

Contracts the tensor network in place and returns the final tensor. Note that this modifies the tensor network representing the operator. The default ordering for the axes of the final tensor is: **out_edges*, **in_edges*. If there are any “ignored” edges, their axes come first: **ignored_edges*, **out_edges*, **in_edges*.

Parameters **final_edge_order** (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor. The default ordering is determined by *out_edges* and *in_edges* (see above).

Raises **ValueError** – Node count ‘{}’ > 1 after contraction!

Returns The final tensor representing the operator.

Return type Tensor

eval_matrix(*final_edge_order*: *Optional[Sequence[tensornetwork.network_components.Edge]]* = *None*) → *Any*

Contracts the tensor network in place and returns the final tensor in two dimensional matrix. The default ordering for the axes of the final tensor is: (\prod dimension of *out_edges*, \prod dimension of *in_edges*)

Parameters **final_edge_order** (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor. The default ordering is determined by *out_edges* and *in_edges* (see above).

Raises **ValueError** – Node count ‘{}’ > 1 after contraction!

Returns The two-dimensional tensor representing the operator.

Return type Tensor

classmethod **from_local_tensor**(*tensor*: Any, *space*: Sequence[int], *loc*: Sequence[int], *out_axes*: *Optional[Sequence[int]]* = *None*, *in_axes*: *Optional[Sequence[int]]* = *None*) → *tensorcircuit.quantum.QuOperator*

classmethod **from_tensor**(*tensor*: Any) → *tensorcircuit.quantum.QuScalar*

Construct a *QuScalar* directly from a single tensor. This first wraps the tensor in a *Node*, then constructs the *QuScalar* from that *Node*.

Example

```
def show_attributes(op):
    print(f"op.is_scalar() \t\t-> {op.is_scalar()}")
    print(f"op.is_vector() \t\t-> {op.is_vector()}")
```

(continues on next page)

(continued from previous page)

```
print(f"op.is_adjoint_vector() \t-> {op.is_adjoint_vector()}")
print(f"op.eval() \n{op.eval()}"")
```

```
>>> op = qu.QuScalar.from_tensor(1.0)
>>> show_attributes(op)
op.is_scalar()          -> True
op.is_vector()          -> False
op.is_adjoint_vector() -> False
op.eval()
1.0
```

Parameters `tensor` (`Tensor`) – The tensor for constructing a new QuScalar.

Returns The new constructed QuScalar from the given tensor.

Return type `QuScalar`

property `in_space: List[int]`

is_adjoint_vector() → bool

Returns a bool indicating if QuOperator is an adjoint vector. Examples can be found in the `QuOperator.from_tensor`.

is_scalar() → bool

Returns a bool indicating if QuOperator is a scalar. Examples can be found in the `QuOperator.from_tensor`.

is_vector() → bool

Returns a bool indicating if QuOperator is a vector. Examples can be found in the `QuOperator.from_tensor`.

property `nodes: Set[tensornetwork.network_components.AbstractNode]`

All tensor-network nodes involved in the operator.

norm() → `tensorcircuit.quantum.QuOperator`

The norm of the operator. This is the 2-norm (also known as the Frobenius or Hilbert-Schmidt norm).

property `out_space: List[int]`

partial_trace(`subsystems_to_trace_out: Collection[int]`) → `tensorcircuit.quantum.QuOperator`

The partial trace of the operator. Subsystems to trace out are supplied as indices, so that dangling edges are connected to each other as: `out_edges[i] ^ in_edges[i]` for i in `subsystems_to_trace_out`. This does not modify the original network. The original ordering of the remaining subsystems is maintained.

Parameters `subsystems_to_trace_out` (`Collection[int]`) – Indices of subsystems to trace out.

Returns A new QuOperator or QuScalar representing the result.

Return type `QuOperator`

tensor_product(`other: tensorcircuit.quantum.QuOperator`) → `tensorcircuit.quantum.QuOperator`

Tensor product with another operator. Given two operators A and B , produces a new operator AB representing AB . The `out_edges` (`in_edges`) of AB is simply the concatenation of the `out_edges` (`in_edges`) of A .`copy()` with that of B .`copy()`: `new_out_edges = [*out_edges_A_copy, *out_edges_B_copy]` `new_in_edges = [*in_edges_A_copy, *in_edges_B_copy]`

Example

```
>>> psi = qu.QuVector.from_tensor(np.random.rand(2, 2))
>>> psi_psi = psi.tensor_product(psi)
>>> len(psi_psi.subsystem_edges)
4
>>> float(psi_psi.norm().eval())
2.9887872748523585
>>> psi.norm().eval() ** 2
2.9887872748523585
```

Parameters `other` (`QuOperator`) – The other operator (B).

Returns The result (AB).

Return type `QuOperator`

`trace()` → `tensorcircuit.quantum.QuOperator`

The trace of the operator.

```
class tensorcircuit.quantum.QuVector(subsystem_edges:
    Sequence[tensornetwork.network_components.Edge], ref_nodes:
    Op-
    tional[Collection[tensornetwork.network_components.AbstractNode]] =
    None, ignore_edges:
    Optional[Collection[tensornetwork.network_components.Edge]] =
    None)
```

Bases: `tensorcircuit.quantum.QuOperator`

Represents a (column) vector via a tensor network.

```
__init__(subsystem_edges: Sequence[tensornetwork.network_components.Edge], ref_nodes:
    Optional[Collection[tensornetwork.network_components.AbstractNode]] = None, ignore_edges:
    Optional[Collection[tensornetwork.network_components.Edge]] = None) → None
```

Constructs a new `QuVector` from a tensor network. This encapsulates an existing tensor network, interpreting it as a (column) vector.

Parameters

- **subsystem_edges** (`Sequence[Edge]`) – The edges of the network to be used as the output edges.
- **ref_nodes** (`Optional[Collection[AbstractNode]], optional`) – Nodes used to refer to parts of the tensor network that are not connected to any input or output edges (for example: a scalar factor).
- **ignore_edges** (`Optional[Collection[Edge]], optional`) – Optional collection of edges to ignore when performing consistency checks.

`adjoint()` → `tensorcircuit.quantum.QuOperator`

The adjoint of the operator. This creates a new `QuOperator` with complex-conjugate copies of all tensors in the network and with the input and output edges switched.

Returns The adjoint of the operator.

Return type `QuOperator`

`check_network()` → `None`

Check that the network has the expected dimensionality. This checks that all input and output edges are dangling and that there are no other dangling edges (except any specified in `ignore_edges`). If not, an exception is raised.

contract(*final_edge_order*: *Optional[Sequence[tensornetwork.network_components.Edge]]* = *None*) → *tensorcircuit.quantum.QuOperator*

Contract the tensor network in place. This modifies the tensor network representation of the operator (or vector, or scalar), reducing it to a single tensor, without changing the value.

Parameters **final_edge_order** (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor.

Returns The present object.

Return type *QuOperator*

copy() → *tensorcircuit.quantum.QuOperator*

The deep copy of the operator.

Returns The new copy of the operator.

Return type *QuOperator*

eval(*final_edge_order*: *Optional[Sequence[tensornetwork.network_components.Edge]]* = *None*) → Any

Contracts the tensor network in place and returns the final tensor. Note that this modifies the tensor network representing the operator. The default ordering for the axes of the final tensor is: **out_edges*, **in_edges*. If there are any “ignored” edges, their axes come first: **ignored_edges*, **out_edges*, **in_edges*.

Parameters **final_edge_order** (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor. The default ordering is determined by *out_edges* and *in_edges* (see above).

Raises ValueError – Node count ‘{}’ > 1 after contraction!

Returns The final tensor representing the operator.

Return type Tensor

eval_matrix(*final_edge_order*: *Optional[Sequence[tensornetwork.network_components.Edge]]* = *None*) → Any

Contracts the tensor network in place and returns the final tensor in two dimensional matrix. The default ordering for the axes of the final tensor is: (\prod dimension of *out_edges*, \prod dimension of *in_edges*)

Parameters **final_edge_order** (*Optional[Sequence[Edge]]*, *optional*) – Manually specify the axis ordering of the final tensor. The default ordering is determined by *out_edges* and *in_edges* (see above).

Raises ValueError – Node count ‘{}’ > 1 after contraction!

Returns The two-dimensional tensor representing the operator.

Return type Tensor

classmethod from_local_tensor(*tensor*: Any, *space*: Sequence[int], *loc*: Sequence[int], *out_axes*: *Optional[Sequence[int]]* = *None*, *in_axes*: *Optional[Sequence[int]]* = *None*) → *tensorcircuit.quantum.QuOperator*

classmethod from_tensor(*tensor*: Any, *subsystem_axes*: *Optional[Sequence[int]]* = *None*) → *tensorcircuit.quantum.QuVector*

Construct a *QuVector* directly from a single tensor. This first wraps the tensor in a *Node*, then constructs the *QuVector* from that *Node*.

Example

```
def show_attributes(op):
    print(f"op.is_scalar() \t\t-> {op.is_scalar()}")
    print(f"op.is_vector() \t\t-> {op.is_vector()}")
```

(continues on next page)

(continued from previous page)

```
print(f"op.is_adjoint_vector() \t-> {op.is_adjoint_vector()}")
print(f"op.eval() \n{op.eval()}"")
```

```
>>> psi_tensor = np.random.rand(2, 2)
>>> psi_tensor
array([[0.27260127, 0.91401091],
       [0.06490953, 0.38653646]])
>>> op = qu.QuVector.from_tensor(psi_tensor, [0, 1])
>>> show_attributes(op)
op.is_scalar()      -> False
op.is_vector()      -> True
op.is_adjoint_vector() -> False
op.eval()
[[0.27260127 0.91401091]
 [0.06490953 0.38653646]]
```

Parameters

- **tensor** (*Tensor*) – The tensor for constructing a “QuVector”.
- **subsystem_axes** (*Optional[Sequence[int]]*, *optional*) – Sequence of integer indices specifying the order in which to interpret the axes as subsystems (output edges). If not specified, the axes are taken in ascending order.

Returns The new constructed QuVector from the given tensor.

Return type *QuVector*

property in_space: List[int]

is_adjoint_vector() → bool

Returns a bool indicating if QuOperator is an adjoint vector. Examples can be found in the *QuOperator.from_tensor*.

is_scalar() → bool

Returns a bool indicating if QuOperator is a scalar. Examples can be found in the *QuOperator.from_tensor*.

is_vector() → bool

Returns a bool indicating if QuOperator is a vector. Examples can be found in the *QuOperator.from_tensor*.

property nodes: Set[tensornetwork.network_components.AbstractNode]

All tensor-network nodes involved in the operator.

norm() → *tensorcircuit.quantum.QuOperator*

The norm of the operator. This is the 2-norm (also known as the Frobenius or Hilbert-Schmidt norm).

property out_space: List[int]

partial_trace(*subsystems_to_trace_out: Collection[int]*) → *tensorcircuit.quantum.QuOperator*

The partial trace of the operator. Subsystems to trace out are supplied as indices, so that dangling edges are connected to each other as: *out_edges[i] ^ in_edges[i]* for *i* in *subsystems_to_trace_out*. This does not modify the original network. The original ordering of the remaining subsystems is maintained.

Parameters **subsystems_to_trace_out** (*Collection[int]*) – Indices of subsystems to trace out.

Returns A new QuOperator or QuScalar representing the result.

Return type *QuOperator*

projector() → *tensorcircuit.quantum.QuOperator*

The projector of the operator. The operator, as a linear operator, on the adjoint of the operator.

Set A is the operator in matrix form, then the projector of operator is defined as: AA^\dagger

Returns The projector of the operator.

Return type *QuOperator*

reduced_density(*subsystems_to_trace_out*: *Collection[int]*) → *tensorcircuit.quantum.QuOperator*

The reduced density of the operator.

Set A is the matrix of the operator, then the reduced density is defined as:

$$\text{Tr}_{\text{subsystems}}(AA^\dagger)$$

Firstly, take the projector of the operator, then trace out the subsystems to trace out are supplied as indices, so that dangling edges are connected to each other as: $\text{out_edges}[i] \wedge \text{in_edges}[i]$ for i in *subsystems_to_trace_out*. This does not modify the original network. The original ordering of the remaining subsystems is maintained.

Parameters **subsystems_to_trace_out** (*Collection[int]*) – Indices of subsystems to trace out.

Returns The QuOperator of the reduced density of the operator with given subsystems.

Return type *QuOperator*

property *space*: *List[int]***property** *subsystem_edges*: *List[tensornetwork.network_components.Edge]***tensor_product**(*other*: *tensorcircuit.quantum.QuOperator*) → *tensorcircuit.quantum.QuOperator*

Tensor product with another operator. Given two operators A and B , produces a new operator AB representing AB . The *out_edges* (*in_edges*) of AB is simply the concatenation of the *out_edges* (*in_edges*) of *A.copy()* with that of *B.copy()*: *new_out_edges* = [**out_edges_A_copy*, **out_edges_B_copy*] *new_in_edges* = [**in_edges_A_copy*, **in_edges_B_copy*]

Example

```
>>> psi = qu.QuVector.from_tensor(np.random.rand(2, 2))
>>> psi_psi = psi.tensor_product(psi)
>>> len(psi_psi.subsystem_edges)
4
>>> float(psi_psi.norm().eval())
2.9887872748523585
>>> psi.norm().eval() ** 2
2.9887872748523585
```

Parameters **other** (*Quoperator*) – The other operator (B).

Returns The result (AB).

Return type *QuOperator*

trace() → *tensorcircuit.quantum.QuOperator*

The trace of the operator.

```
tensorcircuit.quantum.check_spaces(edges_1: Sequence[tensornetwork.network_components.Edge],  
                                  edges_2: Sequence[tensornetwork.network_components.Edge]) →  
                                  None
```

Check the vector spaces represented by two lists of edges are compatible. The number of edges must be the same and the dimensions of each pair of edges must match. Otherwise, an exception is raised.

Parameters

- **edges_1** (*Sequence[Edge]*) – List of edges representing a many-body Hilbert space.
- **edges_2** (*Sequence[Edge]*) – List of edges representing a many-body Hilbert space.

Raises ValueError – Hilbert-space mismatch: “Cannot connect {} subsystems with {} subsystems”, or “Input dimension {} != output dimension {}.”

```
tensorcircuit.quantum.correlation_from_counts(index: Sequence[int], results: Any) → Any
```

Compute $\prod_{i \in \text{index}} s_i$, where the probability for each bitstring is given as a vector **results**. Results is in the format of “count_vector”

Example

```
>>> prob = tc.array_to_tensor(np.array([0.6, 0.4, 0, 0]))  
>>> qu.correlation_from_counts([0, 1], prob)  
(0.20000002+0j)  
>>> qu.correlation_from_counts([1], prob)  
(0.20000002+0j)
```

Parameters

- **index** (*Sequence[int]*) – list of int, indicating the position in the bitstring
- **results** (*Tensor*) – probability vector of shape 2^n

Returns Correlation expectation from measurement shots.

Return type Tensor

```
tensorcircuit.quantum.correlation_from_samples(index: Sequence[int], results: Any, n: int) → Any
```

Compute $\prod_{i \in \text{index}} s_i (s = \pm 1)$, Results is in the format of “sample_int” or “sample_bin”

Parameters

- **index** (*Sequence[int]*) – list of int, indicating the position in the bitstring
- **results** (*Tensor*) – sample tensor
- **n** (*int*) – number of qubits

Returns Correlation expectation from measurement shots

Return type Tensor

```
tensorcircuit.quantum.count_d2s(drepr: Any, eps: float = 1e-07) → Tuple[Any, Any]
```

measurement shots results, dense representation to sparse tuple representation non-jittable due to the non fixed return shape count_tuple to count_vector

Example

```
>>> tc.quantum.counts_d2s(np.array([0.1, 0, -0.3, 0.2]))  
(array([0, 2, 3]), array([ 0.1, -0.3,  0.2]))
```

Parameters

- **drepr** (*Tensor*) – [description]
- **eps** (*float, optional*) – cutoff to determine nonzero elements, defaults to 1e-7

Returns [description]

Return type Tuple[*Tensor*, *Tensor*]

`tensorcircuit.quantum.count_s2d(srepr: Tuple[Any, Any], n: int) → Any`

measurement shots results, sparse tuple representation to dense representation count_vector to count_tuple

Parameters

- **srepr** (*Tuple[*Tensor*, *Tensor*]*) – [description]
- **n** (*int*) – number of qubits

Returns [description]

Return type *Tensor*

`tensorcircuit.quantum.count_t2v(drepr: Any, eps: float = 1e-07) → Tuple[Any, Any]`

measurement shots results, dense representation to sparse tuple representation non-jittable due to the non fixed return shape count_tuple to count_vector

Example

```
>>> tc.quantum.counts_d2s(np.array([0.1, 0, -0.3, 0.2]))
(array([0, 2, 3]), array([ 0.1, -0.3,  0.2]))
```

Parameters

- **drepr** (*Tensor*) – [description]
- **eps** (*float, optional*) – cutoff to determine nonzero elements, defaults to 1e-7

Returns [description]

Return type Tuple[*Tensor*, *Tensor*]

`tensorcircuit.quantum.count_tuple2dict(count: Tuple[Any, Any], n: int, key: str = 'bin') → Dict[Any, int]`

count_tuple to count_dict_bin or count_dict_int

Parameters

- **count** (*Tuple[*Tensor*, *Tensor*]*) – count_tuple format
- **n** (*int*) – number of qubits
- **key** (*str, optional*) – can be “int” or “bin”, defaults to “bin”

Returns count_dict

Return type _type_

`tensorcircuit.quantum.count_vector2dict(count: Any, n: int, key: str = 'bin') → Dict[Any, int]`

convert_vector to count_dict_bin or count_dict_int

Parameters

- **count** (*Tensor*) – tensor in shape [2**n]
- **n** (*int*) – number of qubits
- **key** (*str, optional*) – can be “int” or “bin”, defaults to “bin”

Returns _description_

Return type `_type_`

`tensorcircuit.quantum.counts_v2t(srepr: Tuple[Any, Any], n: int) → Any`
measurement shots results, sparse tuple representation to dense representation count_vector to count_tuple

Parameters

- `srepr (Tuple[Tensor, Tensor])` – [description]
- `n (int)` – number of qubits

Returns [description]**Return type** Tensor

`tensorcircuit.quantum.double_state(h: Any, beta: float = 1) → Any`
Compute the double state of the given Hamiltonian operator h.

Parameters

- `h (Tensor)` – Hamiltonian operator in form of Tensor.
- `beta (float, optional)` – Constant for the optimization, default is 1.

Returns The double state of h with the given beta.**Return type** Tensor

`tensorcircuit.quantum.eliminate_identities(nodes: Collection[tensornetwork.network_components.AbstractNode]) → Tuple[dict, dict]`

Eliminates any connected CopyNodes that are identity matrices. This will modify the network represented by `nodes`. Only identities that are connected to other nodes are eliminated.

Parameters `nodes (Collection[AbstractNode])` – Collection of nodes to search.

Returns The Dictionary mapping remaining Nodes to any replacements, Dictionary specifying all dangling-edge replacements.

Return type Dict[Union[CopyNode, AbstractNode], Union[Node, AbstractNode]], Dict[Edge, Edge]

`tensorcircuit.quantum.entropy(rho: Union[Any, tensorcircuit.quantum.QuOperator], eps: float = 1e-12) → Any`

Compute the entropy from the given density matrix rho.

Example

```
@partial(tc.backend.jit, jit_compile=False, static_argnums=(1, 2))
def entanglement1(param, n, nlayers):
    c = tc.Circuit(n)
    c = tc.templates.blocks.example_block(c, param, nlayers)
    w = c.wavefunction()
    rm = qu.reduced_density_matrix(w, int(n / 2))
    return qu.entropy(rm)

@partial(tc.backend.jit, jit_compile=False, static_argnums=(1, 2))
def entanglement2(param, n, nlayers):
    c = tc.Circuit(n)
    c = tc.templates.blocks.example_block(c, param, nlayers)
    w = c.get_quvector()
    rm = w.reduced_density([i for i in range(int(n / 2))])
    return qu.entropy(rm)
```

```
>>> param = tc.backend.ones([6, 6])
>>> tc.backend.trace(param)
>>> entanglement1(param, 6, 3)
1.3132654
>>> entanglement2(param, 6, 3)
1.3132653
```

Parameters

- **rho** (*Union[Tensor, QuOperator]*) – The density matrix in form of Tensor or QuOperator.
- **eps** (*float*) – Epsilon, default is 1e-12.

Returns Entropy on the given density matrix.**Return type** Tensor`tensorcircuit.quantum.fidelity(rho: Any, rho0: Any) → Any`

Return fidelity scalar between two states rho and rho0.

$$\text{Tr}(\sqrt{\sqrt{r}\rho\sqrt{r}\rho_0})$$

Parameters

- **rho** (*Tensor*) – The density matrix in form of Tensor.
- **rho0** (*Tensor*) – The density matrix in form of Tensor.

Returns The sqrtm of a Hermitian matrix a.**Return type** Tensor`tensorcircuit.quantum.free_energy(rho: Union[Any, tensorcircuit.quantum.QuOperator], h: Union[Any, tensorcircuit.quantum.QuOperator], beta: float = 1, eps: float = 1e-12) → Any`

Compute the free energy of the given density matrix.

Example

```
>>> rho = np.array([[1.0, 0], [0, 0]])
>>> h = np.array([[-1.0, 0], [0, 1]])
>>> qu.free_energy(rho, h, 0.5)
-0.9999999999979998
>>> hq = qu.QuOperator.from_tensor(h)
>>> qu.free_energy(rho, hq, 0.5)
array([-1.])
```

Parameters

- **rho** (*Union[Tensor, QuOperator]*) – The density matrix in form of Tensor or QuOperator.
- **h** (*Union[Tensor, QuOperator]*) – Hamiltonian operator in form of Tensor or QuOperator.
- **beta** (*float, optional*) – Constant for the optimization, default is 1.
- **eps** (*float, optional*) – Epsilon, default is 1e-12.

Returns The free energy of the given density matrix with the Hamiltonian operator.

Return type Tensor

`tensorcircuit.quantum.generate_local_hamiltonian(*hlist: Sequence[Any], matrix_form: bool = True)`
→ Union[tensorcircuit.quantum.QuOperator, Any]

Generate a local Hamiltonian operator based on the given sequence of Tensor. Note: further jit is recommended. For large Hilbert space, sparse Hamiltonian is recommended

Parameters

- **hlist** (`Sequence[Tensor]`) – A sequence of Tensor.
- **matrix_form** (`bool, optional`) – Return Hamiltonian operator in form of matrix, defaults to True.

Returns The Hamiltonian operator in form of QuOperator or matrix.

Return type Union[`QuOperator`, Tensor]

`tensorcircuit.quantum.gibbs_state(h: Any, beta: float = 1) → Any`

Compute the Gibbs state of the given Hamiltonian operator h.

Parameters

- **h** (`Tensor`) – Hamiltonian operator in form of Tensor.
- **beta** (`float, optional`) – Constant for the optimization, default is 1.

Returns The Gibbs state of h with the given beta.

Return type Tensor

`tensorcircuit.quantum.heisenberg_hamiltonian(g: Any, hzz: float = 1.0, hxx: float = 1.0, hyy: float = 1.0, hz: float = 0.0, hx: float = 0.0, hy: float = 0.0, sparse: bool = True, numpy: bool = False) → Any`

Generate Heisenberg Hamiltonian with possible external fields. Currently requires tensorflow installed

Example

```
>>> g = tc.templates.graphs.Line1D(6)
>>> h = qu.heisenberg_hamiltonian(g, sparse=False)
>>> tc.backend.eigh(h)[0][:10]
array([-11.2111025, -8.4721365, -8.472136 , -8.472136 , -6. ,
       -5.123106 , -5.123106 , -5.1231055, -5.1231055, -5.1231055],
      dtype=float32)
```

Parameters

- **g** (`Graph`) – input circuit graph
- **hzz** (`float`) – zz coupling, default is 1.0
- **hxx** (`float`) – xx coupling, default is 1.0
- **hyy** (`float`) – yy coupling, default is 1.0
- **hz** (`float`) – External field on z direction, default is 0.0
- **hx** (`float`) – External field on y direction, default is 0.0
- **hy** (`float`) – External field on x direction, default is 0.0
- **sparse** (`bool, defaults True`) – Whether to return sparse Hamiltonian operator, default is True.

- **numpy** (*bool*, *defaults False*,) – whether return the matrix in numpy or tensorflow form

Returns Hamiltonian measurements

Return type Tensor

```
tensorcircuit.quantum.identity(space: Sequence[int], dtype: Optional[Any] = None) →
    tensorcircuit.quantum.QuOperator
```

Construct a ‘QuOperator’ representing the identity on a given space. Internally, this is done by constructing ‘CopyNode’s for each edge, with dimension according to ‘space’.

Example

```
>>> E = qu.identity((2, 3, 4))
>>> float(E.trace().eval())
24.0

>>> tensor = np.random.rand(2, 2)
>>> psi = qu.QuVector.from_tensor(tensor)
>>> E = qu.identity((2, 2))
>>> psi.eval()
array([[0.03964233, 0.99298281],
       [0.38564989, 0.00950596]])
>>> (E @ psi).eval()
array([[0.03964233, 0.99298281],
       [0.38564989, 0.00950596]])
>>>
>>> (psi.adjoint() @ E @ psi).eval()
array(1.13640257)
>>> psi.norm().eval()
array(1.13640257)
```

Parameters

- **space** (*Sequence[int]*) – A sequence of integers for the dimensions of the tensor product factors of the space (the edges in the tensor network).
- **dtype** (*Any type*) – The data type by np.* (for conversion to dense). defaults None to tc dtype.

Returns The desired identity operator.

Return type *QuOperator*

```
tensorcircuit.quantum.measurement_counts(state: Any, counts: Optional[int] = 8192, format: str =
    'count_vector', is_prob: bool = False, random_generator:
    Optional[Any] = None, status: Optional[Any] = None, jittable:
    bool = False) → Any
```

Simulate the measuring of each qubit of p in the computational basis, thus producing output like that of qiskit.

Six formats of measurement counts results:

```
“sample_int”: # np.array([0, 0])
“sample_bin”: # [np.array([1, 0]), np.array([1, 0])]
“count_vector”: # np.array([2, 0, 0, 0])
```

```
“count_tuple”: # (np.array([0]), np.array([2]))  
“count_dict_bin”: # {"00": 2, "01": 0, "10": 0, "11": 0}  
“count_dict_int”: # {0: 2, 1: 0, 2: 0, 3: 0}
```

Example

```
>>> n = 4  
>>> w = tc.backend.ones([2**n])  
>>> tc.quantum.measurement_results(w, counts=3, format="sample_bin", jittable=True)  
array([[0, 0, 1, 0],  
       [0, 1, 1, 0],  
       [0, 1, 1, 1]])  
>>> tc.quantum.measurement_results(w, counts=3, format="sample_int", jittable=True)  
array([ 7, 15, 11])  
>>> tc.quantum.measurement_results(w, counts=3, format="count_vector",  
                                     jittable=True)  
array([0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1])  
>>> tc.quantum.measurement_results(w, counts=3, format="count_tuple")  
(array([1, 2, 8]), array([1, 1, 1]))  
>>> tc.quantum.measurement_results(w, counts=3, format="count_dict_bin")  
{'0001': 1, '0011': 1, '1101': 1}  
>>> tc.quantum.measurement_results(w, counts=3, format="count_dict_int")  
{3: 1, 6: 2}
```

Parameters

- **state** (*Tensor*) – The quantum state, assumed to be normalized, as either a ket or density operator.
- **counts** (*int*) – The number of counts to perform.
- **shots** (*int*) – alias for the argument **counts**
- **format** (*str*) – defaults to be “direct”, see supported format above
- **format** – alias for the argument **format**
- **is_prob** (*bool*) – if True, the *state* is directly regarded as a probability list, defaults to be False
- **random_generator** (*Optional[Any]*) – random_generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random_generator
- **jittable** (*bool*) – if True, jax backend try using a jittable count, defaults to False

Returns The counts for each bit string measured.

Return type Tuple[]

```
tensorcircuit.quantum.measurement_results(state: Any, counts: Optional[int] = 8192, format: str =  
                                         'count_vector', is_prob: bool = False, random_generator:  
                                         Optional[Any] = None, status: Optional[Any] = None,  
                                         jittable: bool = False) → Any
```

Simulate the measuring of each qubit of p in the computational basis, thus producing output like that of qiskit.

Six formats of measurement counts results:

```
“sample_int”: # np.array([0, 0])
```

```

“sample_bin”: # [np.array([1, 0]), np.array([1, 0])]

“count_vector”: # np.array([2, 0, 0, 0])

“count_tuple”: # (np.array([0]), np.array([2]))

“count_dict_bin”: # {"00": 2, "01": 0, "10": 0, "11": 0}

“count_dict_int”: # {0: 2, 1: 0, 2: 0, 3: 0}

```

Example

```

>>> n = 4
>>> w = tc.backend.ones([2**n])
>>> tc.quantum.measurement_results(w, counts=3, format="sample_bin", jittable=True)
array([[0, 0, 1, 0],
       [0, 1, 1, 0],
       [0, 1, 1, 1]])
>>> tc.quantum.measurement_results(w, counts=3, format="sample_int", jittable=True)
array([7, 15, 11])
>>> tc.quantum.measurement_results(w, counts=3, format="count_vector", ↴
    ↴jittable=True)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1])
>>> tc.quantum.measurement_results(w, counts=3, format="count_tuple")
(array([1, 2, 8]), array([1, 1, 1]))
>>> tc.quantum.measurement_results(w, counts=3, format="count_dict_bin")
{'0001': 1, '0011': 1, '1101': 1}
>>> tc.quantum.measurement_results(w, counts=3, format="count_dict_int")
{3: 1, 6: 2}

```

Parameters

- **state** (*Tensor*) – The quantum state, assumed to be normalized, as either a ket or density operator.
- **counts** (*int*) – The number of counts to perform.
- **shots** (*int*) – alias for the argument **counts**
- **format** (*str*) – defaults to be “direct”, see supported format above
- **format** – alias for the argument **format**
- **is_prob** (*bool*) – if True, the *state* is directly regarded as a probability list, defaults to be False
- **random_generator** (*Optional[Any]*) – random_generator, defaults to None
- **status** (*Optional[Tensor]*) – external randomness given by tensor uniformly from [0, 1], if set, can overwrite random_generator
- **jittable** (*bool*) – if True, jax backend try using a jittable count, defaults to False

Returns The counts for each bit string measured.

Return type Tuple[]

`tensorcircuit.quantum.mutual_information(s: Any, cut: Union[int, List[int]]) → Any`
Mutual information between AB subsystem described by cut.

Parameters

- **s** (*Tensor*) – The density matrix in form of Tensor.

- **cut** (*Union[int, List[int]]*) – The AB subsystem.

Returns The mutual information between AB subsystem described by `cut`.

Return type Tensor

```
tensorcircuit.quantum.op2tensor(fn: Callable[..., Any], op_argnums: Union[int, Sequence[int]] = 0) →  
Callable[[...], Any]  
  
tensorcircuit.quantum.ps2coo_core(idx_x: Any, idx_y: Any, idx_z: Any, weight: Any, nqubits: int) →  
Tuple[Any, Any]  
  
tensorcircuit.quantum.quantum_constructor(out_edges:  
                                      Sequence[tensornetwork.network_components.Edge],  
                                      in_edges:  
                                      Sequence[tensornetwork.network_components.Edge],  
                                      ref_nodes: Op-  
                                      tional[Collection[tensornetwork.network_components.AbstractNode]]  
                                      = None, ignore_edges: Op-  
                                      tional[Collection[tensornetwork.network_components.Edge]]  
                                      = None) → tensorcircuit.quantum.QuOperator
```

Constructs an appropriately specialized QuOperator. If there are no edges, creates a QuScalar. If there are only output (input) edges, creates a QuVector (QuAdjointVector). Otherwise creates a QuOperator.

Example

```
def show_attributes(op):  
    print(f"op.is_scalar()           -> {op.is_scalar()}"")  
    print(f"op.is_vector()           -> {op.is_vector()}"")  
    print(f"op.is_adjoint_vector()   -> {op.is_adjoint_vector()}"")  
    print(f"len(op.out_edges)        -> {len(op.out_edges)}")  
    print(f"len(op.in_edges)         -> {len(op.in_edges)}")
```

```
>>> psi_node = tn.Node(np.random.rand(2, 2))  
>>>  
>>> op = qu.quantum_constructor([psi_node[0]], [psi_node[1]])  
>>> show_attributes(op)  
op.is_scalar()           -> False  
op.is_vector()           -> False  
op.is_adjoint_vector()   -> False  
len(op.out_edges)        -> 1  
len(op.in_edges)         -> 1  
>>> # psi_node[0] -> op.out_edges[0]  
>>> # psi_node[1] -> op.in_edges[0]
```

```
>>> op = qu.quantum_constructor([psi_node[0], psi_node[1]], [])  
>>> show_attributes(op)  
op.is_scalar()           -> False  
op.is_vector()           -> True  
op.is_adjoint_vector()   -> False  
len(op.out_edges)        -> 2  
len(op.in_edges)         -> 0  
>>> # psi_node[0] -> op.out_edges[0]  
>>> # psi_node[1] -> op.out_edges[1]
```

```
>>> op = qu.quantum_constructor([], [psi_node[0], psi_node[1]])
>>> show_attributes(op)
op.is_scalar()      -> False
op.is_vector()      -> False
op.is_adjoint_vector() -> True
len(op.out_edges)   -> 0
len(op.in_edges)    -> 2
>>> # psi_node[0] -> op.in_edges[0]
>>> # psi_node[1] -> op.in_edges[1]
```

Parameters

- **out_edges** (*Sequence[Edge]*) – A list of output edges.
- **in_edges** (*Sequence[Edge]*) – A list of input edges.
- **ref_nodes** (*Optional[Collection[AbstractNode]]*, *optional*) – Reference nodes for the tensor network (needed if there is a scalar component).
- **ignore_edges** (*Optional[Collection[Edge]]*, *optional*) – Edges to ignore when checking the dimensionality of the tensor network.

Returns The new created QuOperator object.**Return type** *QuOperator*`tensorcircuit.quantum.quimb2qop(qb_mpo: Any) → tensorcircuit.quantum.QuOperator`

Convert MPO in Quimb package to QuOperator.

Parameters **tn_mpo** (*quimb.tensor.tensor_gen.**) – MPO in the form of Quimb package**Returns** MPO in the form of QuOperator**Return type** *QuOperator*`tensorcircuit.quantum.reduced_density_matrix(state: Union[Any, tensorcircuit.quantum.QuOperator], cut: Union[int, List[int]], p: Optional[Any] = None) → Union[Any, tensorcircuit.quantum.QuOperator]`Compute the reduced density matrix from the quantum state **state**.**Parameters**

- **state** (*Union[Tensor, QuOperator]*) – The quantum state in form of Tensor or QuOperator.
- **cut** (*Union[int, List[int]]*) – the index list that is traced out, if cut is a int, it indicates [0, cut] as the traced out region
- **p** (*Optional[Tensor]*) – probability decoration, default is None.

Returns The reduced density matrix.**Return type** *Union[Tensor, QuOperator]*`tensorcircuit.quantum.renyi_entropy(rho: Union[Any, tensorcircuit.quantum.QuOperator], k: int = 2) → Any`Compute the Rényi entropy of order **k** by given density matrix.**Parameters**

- **rho** (*Union[Tensor, QuOperator]*) – The density matrix in form of Tensor or QuOperator.

- **k** (*int, optional*) – The order of Rényi entropy, default is 2.

Returns The k th order of Rényi entropy.

Return type Tensor

```
tensorcircuit.quantum.renyi_free_energy(rho: Union[Any, tensorcircuit.quantum.QuOperator], h:  
Union[Any, tensorcircuit.quantum.QuOperator], beta: float = 1,  
k: int = 2) → Any
```

Compute the Rényi free energy of the corresponding density matrix and Hamiltonian.

Example

```
>>> rho = np.array([[1.0, 0], [0, 0]])  
>>> h = np.array([[-1.0, 0], [0, 1]])  
>>> qu.renyi_free_energy(rho, h, 0.5)  
-1.0  
>>> qu.free_energy(rho, h, 0.5)  
-0.999999999979998
```

Parameters

- **rho** (*Union[Tensor, QuOperator]*) – The density matrix in form of Tensor or QuOperator.
- **h** (*Union[Tensor, QuOperator]*) – Hamiltonian operator in form of Tensor or QuOperator.
- **beta** (*float, optional*) – Constant for the optimization, default is 1.
- **k** (*int, optional*) – The order of Rényi entropy, default is 2.

Returns The k th order of Rényi entropy.

Return type Tensor

```
tensorcircuit.quantum.sample2all(sample: Any, n: int, format: str = 'count_vector', jittable: bool = False)  
→ Any
```

transform sample_int or sample_bin form results to other forms specified by format

Parameters

- **sample** (*Tensor*) – measurement shots results in sample_int or sample_bin format
- **n** (*int*) – number of qubits
- **format** (*str, optional*) – see the doc in the doc in [tensorcircuit.quantum.measurement_results\(\)](#), defaults to “count_vector”
- **format** – alias for the argument format
- **jittable** (*bool, optional*) – only applicable to count transformation in jax backend, defaults to False

Returns measurement results specified as format

Return type Any

```
tensorcircuit.quantum.sample2count(sample: Any, n: int, jittable: bool = True) → Tuple[Any, Any]  
sample_int to count_tuple
```

Parameters

- **sample** (*Tensor*) – _description_

- **n** (*int*) – _description_
- **jittable** (*bool, optional*) – _description_, defaults to True

Returns _description_

Return type Tuple[Tensor, Tensor]

`tensorcircuit.quantum.sample_bin2int(sample: Any, n: int) → Any`
bin sample to int sample

Parameters

- **sample** (*Tensor*) – in shape [trials, n] of elements (0, 1)
- **n** (*int*) – number of qubits

Returns in shape [trials]

Return type Tensor

`tensorcircuit.quantum.sample_int2bin(sample: Any, n: int) → Any`
int sample to bin sample

Parameters

- **sample** (*Tensor*) – in shape [trials] of int elements in the range [0, 2**n)
- **n** (*int*) – number of qubits

Returns in shape [trials, n] of element (0, 1)

Return type Tensor

`tensorcircuit.quantum.spin_by_basis(n: int, m: int, elements: Tuple[int, int] = (1, -1)) → Any`
Generate all n-bitstrings as an array, each row is a bitstring basis. Return m-th col.

Example

```
>>> qu.spin_by_basis(2, 1)
array([ 1, -1,  1, -1])
```

Parameters

- **n** (*int*) – length of a bitstring
- **m** (*int*) – m<n,
- **elements** (*Tuple[int, int], optional*) – the binary elements to generate, default is (1, -1).

Returns The value for the m-th position in bitstring when going through all bitstring basis.

Return type Tensor

`tensorcircuit.quantum.taylorlnm(x: Any, k: int) → Any`
Taylor expansion of $\ln(x + 1)$.

Parameters

- **x** (*Tensor*) – The density matrix in form of Tensor.
- **k** (*int, optional*) – The k th order, default is 2.

Returns The k th order of Taylor expansion of $\ln(x + 1)$.

Return type Tensor

`tensorcircuit.quantum.tn2qop(tn_mpo: Any) → tensorcircuit.quantum.QuOperator`

Convert MPO in TensorNetwork package to QuOperator.

Parameters `tn_mpo` (`tn.matrixproductstates.mpo.*`) – MPO in the form of TensorNetwork package

Returns MPO in the form of QuOperator

Return type `QuOperator`

`tensorcircuit.quantum.trace_distance(rho: Any, rho0: Any, eps: float = 1e-12) → Any`

Compute the trace distance between two density matrix `rho` and `rho2`.

Parameters

- `rho` (`Tensor`) – The density matrix in form of Tensor.
- `rho0` (`Tensor`) – The density matrix in form of Tensor.
- `eps` (`float, optional`) – Epsilon, defaults to 1e-12

Returns The trace distance between two density matrix `rho` and `rho2`.

Return type Tensor

`tensorcircuit.quantum.trace_product(*o: Union[Any, tensorcircuit.quantum.QuOperator]) → Any`

Compute the trace of several inputs `o` as tensor or QuOperator.

$$\text{Tr}(\prod_i O_i)$$

Example

```
>>> o = np.ones([2, 2])
>>> h = np.eye(2)
>>> qu.trace_product(o, h)
2.0
>>> oq = qu.QuOperator.from_tensor(o)
>>> hq = qu.QuOperator.from_tensor(h)
>>> qu.trace_product(oq, hq)
array([[2.]])
>>> qu.trace_product(oq, h)
array([[2.]])
>>> qu.trace_product(o, hq)
array([[2.]])
```

Returns The trace of several inputs.

Return type Tensor

`tensorcircuit.quantum.truncated_free_energy(rho: Any, h: Any, beta: float = 1, k: int = 2) → Any`

Compute the truncated free energy from the given density matrix `rho`.

Parameters

- `rho` (`Tensor`) – The density matrix in form of Tensor.
- `h` (`Tensor`) – Hamiltonian operator in form of Tensor.
- `beta` (`float, optional`) – Constant for the optimization, default is 1.
- `k` (`int, optional`) – The `k` th order, defaults to 2

Returns The k th order of the truncated free energy.

Return type Tensor

4.1.18 tensorcircuit.results

tensorcircuit.results.counts

dict related functionalities

`tensorcircuit.results.counts.count2vec(count: Dict[str, int], normalization: bool = True) → Any`

`tensorcircuit.results.counts.expectation(count: Dict[str, int], z: Optional[Sequence[int]] = None, diagonal_op: Optional[Any] = None) → float`

compute diagonal operator expectation value from bit string count dictionary

Parameters

- **count** (*ct*) – count dict for bitstring histogram
- **z** (*Optional[Sequence[int]]*) – if defaults as None, then `diagonal_op` must be set a list of qubit that we measure Z op on
- **diagonal_op** (*Tensor*) – shape [n, 2], explicitly indicate the diagonal op on each qubit eg. [1, -1] for z [1, 1] for I, etc.

Returns the expectation value

Return type float

`tensorcircuit.results.counts.kl_divergence(c1: Dict[str, int], c2: Dict[str, int]) → float`

`tensorcircuit.results.counts.marginal_count(count: Dict[str, int], keep_list: Sequence[int]) → Dict[str, int]`

`tensorcircuit.results.counts.normalized_count(count: Dict[str, int]) → Dict[str, float]`

`tensorcircuit.results.counts.reverse_count(count: Dict[str, int]) → Dict[str, int]`

`tensorcircuit.results.counts.sort_count(count: Dict[str, int]) → Dict[str, int]`

`tensorcircuit.results.counts.vec2count(vec: Any, prune: bool = False) → Dict[str, int]`

tensorcircuit.results.readout_mitigation

readout error mitigation functionalities

`class tensorcircuit.results.readout_mitigation.ReadoutMit(execute: Callable[[], List[Dict[str, int]]], iter_threshold: int = 4096)`

Bases: object

`__init__(execute: Callable[[], List[Dict[str, int]]], iter_threshold: int = 4096)`

The Class for readout error mitigation

Parameters

- **execute** (*Callable[..., List[ct]]*) – execute function to run the circuit
- **iter_threshold** (*int, optional*) – iteration threshold, defaults to 4096

```
apply_correction(counts: Dict[str, int], qubits: Sequence[int], positional_logical_mapping:  
    Optional[Dict[int, int]] = None, logical_physical_mapping: Optional[Dict[int, int]] =  
    None, distance: Optional[int] = None, method: str = 'constrained_least_square',  
    max_iter: int = 25, tol: float = 1e-05, return_mitigation_overhead: bool = False,  
    details: bool = False) → Dict[str, int]
```

Main readout mitigation program for all methods.

Parameters

- **counts** (*ct*) – raw count
- **qubits** (*Sequence[int]*) – user-defined logical qubits to show final mitted results
- **positional_logical_mapping** (*Optional[Dict[int, int]]*, *optional*) – positional_logical_mapping, defaults to None.
- **logical_physical_mapping** (*Optional[Dict[int, int]]*, *optional*) – logical_physical_mapping, defaults to None
- **distance** (*int, optional*) – defaults to None
- **method** (*str, optional*) – mitigation method, defaults to “square”
- **max_iter** (*int, optional*) – defaults to 25
- **tol** (*float, optional*) – defaults to 1e-5

:param return_mitigation_overhead:defaults to False :type return_mitigation_overhead: bool, optional
:param details: defaults to False :type details: bool, optional :return: mitigated count :rtype: ct

```
apply_readout_mitigation(raw_count: Dict[str, int], method: str = 'inverse') → Dict[str, int]
```

Main readout mitigation program for method=“inverse” or “square”

Parameters

- **raw_count** (*ct*) – the raw count
- **method** (*str, optional*) – mitigation method, defaults to “inverse”

Returns mitigated count

Return type *ct*

```
cals_from_api(qubits: Union[int, List[int]], device: Optional[str] = None) → None
```

Get local calibriation matrix from cloud API from tc supported providers

Parameters

- **qubits** (*Union[int, List[int]]*) – list of physical qubits to be calibrated
- **device** (*Optional[str, optional]*) – the device str to qurey for the info, defaults to None

```
cals_from_system(qubits: Union[int, List[int]], shots: int = 8192, method: str = 'local', masks:  
    Optional[List[str]] = None) → None
```

Get calibration information from system.

Parameters

- **qubits** (*Sequence[Any]*) – calibration qubit list (physical qubits on device)
- **shots** (*int, optional*) – shots used for runing the circuit, defaults to 8192
- **method** (*str, optional*) – calibration method, defaults to “local”, it can also be “global”

expectation(*counts*: *Dict[str, int]*, *z*: *Optional[Sequence[int]]* = *None*, *diagonal_op*: *Optional[Any]* = *None*, *positional_logical_mapping*: *Optional[Dict[int, int]]* = *None*, *logical_physical_mapping*: *Optional[Dict[int, int]]* = *None*, *method*: *str* = 'constrained_least_square') → float

Calculate expectation value after readout error mitigation

Parameters

- **counts** (*ct*) – raw counts
- **z** (*Optional[Sequence[int]]*) – if defaults as None, then *diagonal_op* must be set a list of qubit that we measure Z op on
- **diagonal_op** (*Tensor*) – shape [n, 2], explicitly indicate the diagonal op on each qubit eg. [1, -1] for z [1, 1] for I, etc.
- **positional_logical_mapping** (*Optional[Dict[int, int]]*, *optional*) – positional_logical_mapping, defaults to None.
- **logical_physical_mapping** (*Optional[Dict[int, int]]*, *optional*) – logical_physical_mapping, defaults to None
- **method** (*str, optional*) – readout mitigation method, defaults to “constrained_least_square”

Returns expectation value after readout error mitigation

Return type float

get_matrix(*qubits*: *Optional[Sequence[Any]]* = *None*) → Any

Calculate cal_matrix according to use qubit list.

Parameters **qubits** (*Sequence[Any]*, *optional*) – used qubit list, defaults to None

Returns cal_matrix

Return type Tensor

global_miti_readout_circ() → List[*tensorcircuit.circuit.Circuit*]

Generate circuits for global calibration.

Returns circuit list

Return type List[*Circuit*]

local_miti_readout_circ() → List[*tensorcircuit.circuit.Circuit*]

Generate circuits for local calibration.

Returns circuit list

Return type List[*Circuit*]

local_miti_readout_circ_by_mask(*bsl*: *List[str]*) → List[*tensorcircuit.circuit.Circuit*]

mapping_preprocess(*counts*: *Dict[str, int]*, *qubits*: *Sequence[int]*, *positional_logical_mapping*: *Optional[Dict[int, int]]* = *None*, *logical_physical_mapping*: *Optional[Dict[int, int]]* = *None*) → *Dict[str, int]*

Preprocessing to deal with qubit mapping, including positional_logical_mapping and logical_physical_mapping. Return self.use_qubits(physical) and corresponding counts.

Parameters

- **counts** (*ct*) – raw_counts on positional_qubits

- **qubits** (*Sequence[int]*) – user-defined logical qubits to show final mitted results
- **positional_logical_mapping** (*Optional[Dict[int, int]]*, *optional*) – positional_logical_mapping, defaults to None.
- **logical_physical_mapping** (*Optional[Dict[int, int]]*, *optional*) – logical_physical_mapping, defaults to None

Returns counts on self.use_qubit(physical)

Return type ct

mitigate_probability(*probability_noise: Any, method: str = 'inverse'*) → Any

Get the mitigated probability.

Parameters

- **probability_noise** (*Tensor*) – probability of raw count
- **method** (*str, optional*) – mitigation methods, defaults to “inverse”, it can also be “square”

Returns mitigated probability

Return type Tensor

newrange(*m: int, qubits: Optional[Sequence[Any]]*) → int

Rerange the order according to used qubit list.

Parameters

- **m** (*int*) – index
- **qubits** (*Sequence[Any]*) – used qubit list

Returns new index

Return type int

reduced_cal_matrix(*counts, qubits, distance=None*)

ubs(*i: int, qubits: Optional[Sequence[Any]]*) → int

Help omit calibration results that not in used qubit list.

Parameters

- **i** (*int*) – index
- **qubits** (*Sequence[Any]*) – used qubit list

Returns omission related value

Return type int

4.1.19 tensorcircuit.simplify

Tensor network Simplification

tensorcircuit.simplify.infer_new_shape(*a: tensornetwork.network_components.Node, b: tensornetwork.network_components.Node, include_old: bool = True*) → Any

Get the new shape of two nodes, also supporting to return original shapes of two nodes.

Example

```
>>> a = tn.Node(np.ones([2, 3, 5]))
>>> b = tn.Node(np.ones([3, 5, 7]))
>>> a[1] ^ b[0]
>>> a[2] ^ b[1]
>>> tc.simplify.infer_new_shape(a, b)
>>> ((2, 7), (2, 3, 5), (3, 5, 7))
>>> # (shape of a, shape of b, new shape)
```

Parameters

- **a** (`tn.Node`) – node one
- **b** (`tn.Node`) – node two
- **include_old** (`bool`) – Whether to include original shape of two nodes, default is True.

Returns The new shape of the two nodes.**Return type** `Union[Tuple[int, ...], Tuple[Tuple[int, ...], Tuple[int, ...], Tuple[int, ...]]]`

`tensorcircuit.simplify.infer_new_size(a: tensornetwork.network_components.Node, b: tensornetwork.network_components.Node, include_old: bool = True) → Any`

`tensorcircuit.simplify.pseudo_contract_between(a: tensornetwork.network_components.Node, b: tensornetwork.network_components.Node, **kws: Any) → tensornetwork.network_components.Node`

Contract between Node a and b, with correct shape only and no calculation

Parameters

- **a** (`tn.Node`) – [description]
- **b** (`tn.Node`) – [description]

Returns [description]**Return type** `tn.Node`

4.1.20 tensorcircuit.templates

tensorcircuit.templates.blocks

Shortcuts for measurement patterns on circuit

`tensorcircuit.templates.blocks.Bell_pair_block(c: Any, links: Optional[Sequence[Tuple[int, int]]] = None) → Any`

For each pair in links, the input product state $|00\rangle$ is transformed as $(01\rangle - |10\rangle)$ **Parameters**

- **c** (`Circuit`) – Circuit in
- **links** (`Optional[Sequence[Tuple[int, int]]]`, `optional`) – pairs indices for Bell pairs, defaults to None, corresponds to neighbor links

Returns Circuit out**Return type** `Circuit`

```
tensorcircuit.templates.blocks.Grid2D_entangling(c: Any, coord:  
                                                tensorcircuit.templates.graphs.Grid2DCoord,  
                                                unitary: Any, params: Any, **kws: Any) → Any  
  
tensorcircuit.templates.blocks.QAOA_block(c: Any, g: Any, paramzz: Any, paramx: Any, **kws: Any) →  
                                                Any  
  
tensorcircuit.templates.blocks.example_block(c: Any, param: Any, nlayers: int = 2, is_split: bool =  
                                                False) → Any
```

The circuit ansatz is firstly one layer of Hadamard gates and then we have nlayers blocks of $e^{i\theta Z_i Z_{i+1}}$ two-qubit gate in ladder layout, following rx gate.

Parameters

- **c** (*Circuit*) – The circuit
- **param** (*Tensor*) – paramter tensor with $2^*nlayer*n$ elements
- **nlayers** (*int, optional*) – number of ZZ+RX blocks, defaults to 2
- **is_split** (*bool, optional*) – whether use SVD split to reduce ZZ gate bond dimension, defaults to False

Returns The circuit with example ansatz attached

Return type *Circuit*

```
tensorcircuit.templates.blocks.qft(c: Any, *index: int, do_swaps: bool = True, inverse: bool = False,  
                                 insert_barriers: bool = False) → Any
```

This function applies quantum fourier transformation (QFT) to the selected circuit lines

Parameters

- **c** (*Circuit*) – Circuit in
- ***index** – the indices of the circuit lines to apply QFT
- **do_swaps** (*bool*) – Whether to include the final swaps in the QFT
- **inverse** (*bool*) – If True, the inverse Fourier transform is constructed
- **insert_barriers** (*bool*) – If True, barriers are inserted as visualization improvement

Returns Circuit c

Return type *Circuit*

```
tensorcircuit.templates.blocks.state_centric(f: Callable[..., Any]) → Callable[..., Any]
```

Function decorator wraps the function with the first input and output in the format of circuit, the wrapped function has the first input and the output as the state tensor.

Parameters **f** (*Callable[..., Circuit]*) – Function with the fist input and the output as *Circuit* object.

Returns Wrapped function with the first input and the output as the state tensor correspondingly.

Return type *Callable[..., Tensor]*

tensorcircuit.templates.chems

Useful utilities for quantum chemistry related task

`tensorcircuit.templates.chems.get_ps(qo: Any, n: int) → Tuple[Any, Any]`

Get Pauli string array and weights array for a qubit Hamiltonian as a sum of Pauli strings defined in openfermion QubitOperator.

Parameters

- `qo` (openfermion.ops.operators.qubit_operator.QubitOperator) – openfermion.ops.operators.qubit_operator.QubitOperator
- `n (int)` – The number of qubits

Returns Pauli String array and weights array

Return type Tuple[Tensor, Tensor]

tensorcircuit.templates.dataset

Quantum machine learning related data preprocessing and embedding

`tensorcircuit.templates.dataset.amplitude_encoding(fig: Any, nqubits: int, index: Optional[Sequence[int]] = None) → Any`

`tensorcircuit.templates.dataset.mnist_pair_data(a: int, b: int, binarize: bool = False, threshold: float = 0.4, loader: Optional[Any] = None) → Any`

tensorcircuit.templates.graphs

Some common graphs and lattices

`tensorcircuit.templates.graphs.Even1D(n: int, s: int = 0) → Any`

`class tensorcircuit.templates.graphs.Grid2DCoord(n: int, m: int)`

Bases: object

Two-dimensional grid lattice

`__init__(n: int, m: int)`

Parameters

- `n (int)` – number of rows
- `m (int)` – number of cols

`all_cols(pbc: bool = False) → Sequence[Tuple[int, int]]`

return all col edge with 1d index encoding

Parameters `pbc (bool, optional)` – whether to include pbc edges (periodic boundary condition), defaults to False

Returns list of col edge

Return type Sequence[Tuple[int, int]]

`all_rows(pbc: bool = False) → Sequence[Tuple[int, int]]`

return all row edge with 1d index encoding

Parameters `pbc (bool, optional)` – whether to include pbc edges (periodic boundary condition), defaults to False

Returns list of row edge

Return type Sequence[Tuple[int, int]]

`lattice_graph(pbc: bool = True) → Any`

Get the 2D grid lattice in `nx.Graph` format

Parameters `pbc (bool, optional)` – whether to include pbc edges (periodic boundary condition), defaults to True

Returns _description_

Return type Graph

`one2two(i: int) → Tuple[int, int]`

`two2one(x: int, y: int) → int`

`tensorcircuit.templates.graphs.Line1D(n: int, node_weight: Optional[Sequence[float]] = None, edge_weight: Optional[Sequence[float]] = None, pbc: bool = True) → Any`

1D chain with n sites

Parameters

- `n (int)` – [description]
- `pbc (bool, optional)` – [description], defaults to True

Returns [description]

Return type Graph

`tensorcircuit.templates.graphs.Odd1D(n: int, *, s: int = 1) → Any`

tensorcircuit.templates.measurements

Shortcuts for measurement patterns on circuit

`tensorcircuit.templates.measurements.any_local_measurements(c: tensorcircuit.circuit.Circuit, structures: Any, onehot: bool = False, reuse: bool = True) → Any`

This measurements pattern is specifically suitable for vmap. Parameterize the local Pauli string to be measured.

Example

```
c = tc.Circuit(3)
c.X(0)
c.cnot(0, 1)
c.H(-1)
basis = tc.backend.convert_to_tensor(np.array([3, 3, 1]))
z0, z1, x2 = tc.templates.measurements.parameterized_local_measurements(
    c, structures=basis, onehot=True
)
# -1, -1, 1
```

Parameters

- `c (Circuit)` – The circuit to be measured

- **structures** (*Tensor*) – parameter tensors determines what Pauli string to be measured, shape is [nwires, 4] if onehot is False and [nwires] if onehot is True.
- **onehot** (*bool, optional*) – defaults to False. If set to be True, structures will first go through onehot procedure.
- **reuse** (*bool, optional*) – reuse the wavefunction when computing the expectations, defaults to be True

Returns The expectation value of given Pauli string by the tensor structures.

Return type Tensor

```
tensorcircuit.templates.measurements.any_measurements(c: tensorcircuit.circuit.Circuit, structures: Any, onehot: bool = False, reuse: bool = False) → Any
```

This measurements pattern is specifically suitable for vmap. Parameterize the Pauli string to be measured.

Example

```
c = tc.Circuit(3)
c.rx(0, theta=1.0)
c.cnot(0, 1)
c.cnot(1, 2)
c.ry(2, theta=-1.0)

z0x2 = c.expectation([tc.gates.z(), [0]], [tc.gates.x(), [2]])
z0x2p1 = tc.templates.measurements.parameterized_measurements(
    c, tc.array_to_tensor(np.array([3, 0, 1])), onehot=True
)
z0x2p2 = tc.templates.measurements.parameterized_measurements(
    c,
    tc.array_to_tensor(np.array([[0, 0, 0, 1], [1, 0, 0, 0], [0, 1, 0, 0]])), onehot=False,
)
np.testing.assert_allclose(z0x2, z0x2p1)
np.testing.assert_allclose(z0x2, z0x2p2)
```

Parameters

- **c** (*Circuit*) – The circuit to be measured
- **structures** (*Tensor*) – parameter tensors determines what Pauli string to be measured, shape is [nwires, 4] if onehot is False and [nwires] if onehot is True.
- **onehot** (*bool, optional*) – defaults to False. If set to be True, structures will first go through onehot procedure.
- **reuse** (*bool, optional*) – reuse the wavefunction when computing the expectations, defaults to be False

Returns The expectation value of given Pauli string by the tensor structures.

Return type Tensor

`tensorcircuit.templates.measurements.heisenberg_measurements(c: tensorcircuit.circuit.Circuit, g: Any, hzz: float = 1.0, hxx: float = 1.0, hyy: float = 1.0, hz: float = 0.0, hx: float = 0.0, hy: float = 0.0, reuse: bool = True) → Any`

Evaluate Heisenberg energy expectation, whose Hamiltonian is defined on the lattice graph `g` as follows: (`e` are edges in graph `g` where `e1` and `e2` are two nodes for edge `e` and `v` are nodes in graph `g`)

$$H = \sum_{e \in g} w_e (h_{xx} X_{e1} X_{e2} + h_{yy} Y_{e1} Y_{e2} + h_{zz} Z_{e1} Z_{e2}) + \sum_{v \in g} (h_x X_v + h_y Y_v + h_z Z_v)$$

Example

```
g = tc.templates.graphs.Line1D(n=5)
c = tc.Circuit(5)
c.X(0)
energy = tc.templates.measurements.heisenberg_measurements(c, g) # 1
```

Parameters

- `c (Circuit)` – Circuit to be measured
- `g (Graph)` – Lattice graph defining Heisenberg Hamiltonian
- `hzz (float, optional)` – [description], defaults to 1.0
- `hxx (float, optional)` – [description], defaults to 1.0
- `hyy (float, optional)` – [description], defaults to 1.0
- `hz (float, optional)` – [description], defaults to 0.0
- `hx (float, optional)` – [description], defaults to 0.0
- `hy (float, optional)` – [description], defaults to 0.0
- `reuse (bool, optional)` – [description], defaults to True

Returns Value of Heisenberg energy

Return type Tensor

`tensorcircuit.templates.measurements.mpo_expectation(c: tensorcircuit.circuit.Circuit, mpo: tensorcircuit.quantum.QuOperator) → Any`

Evaluate expectation of operator `mpo` defined in `QuOperator` MPO format with the output quantum state from circuit `c`.

Parameters

- `c (Circuit)` – The circuit for the output state
- `mpo (QuOperator)` – MPO operator

Returns a real and scalar tensor of shape [] as the expectation value

Return type Tensor

`tensorcircuit.templates.measurements.operator_expectation(c: tensorcircuit.circuit.Circuit, hamiltonian: Any) → Any`

Evaluate Hamiltonian expectation where `hamiltonian` can be dense matrix, sparse matrix or MPO.

Parameters

- `c (Circuit)` – The circuit whose output state is used to evaluate the expectation

- **hamiltonian** (*Tensor*) – Hamiltonian matrix in COO_sparse_matrix form

Returns a real and scalar tensor of shape [] as the expectation value

Return type Tensor

```
tensorcircuit.templates.measurements.parameterized_local_measurements(c: tensorcircuit.circuit.Circuit,
structures: Any, onehot: bool = False, reuse: bool = True) → Any
```

This measurements pattern is specifically suitable for vmap. Parameterize the local Pauli string to be measured.

Example

```
c = tc.Circuit(3)
c.X(0)
c.cnot(0, 1)
c.H(-1)
basis = tc.backend.convert_to_tensor(np.array([3, 3, 1]))
z0, z1, x2 = tc.templates.measurements.parameterized_local_measurements(
    c, structures=basis, onehot=True
)
# -1, -1, 1
```

Parameters

- **c** (*Circuit*) – The circuit to be measured
- **structures** (*Tensor*) – parameter tensors determines what Pauli string to be measured, shape is [nwires, 4] if onehot is False and [nwires] if onehot is True.
- **onehot** (*bool, optional*) – defaults to False. If set to be True, structures will first go through onehot procedure.
- **reuse** (*bool, optional*) – reuse the wavefunction when computing the expectations, defaults to be True

Returns The expectation value of given Pauli string by the tensor **structures**.

Return type Tensor

```
tensorcircuit.templates.measurements.parameterized_measurements(c: tensorcircuit.circuit.Circuit,
structures: Any, onehot: bool = False, reuse: bool = False) → Any
```

This measurements pattern is specifically suitable for vmap. Parameterize the Pauli string to be measured.

Example

```
c = tc.Circuit(3)
c.rx(0, theta=1.0)
c.cnot(0, 1)
c.cnot(1, 2)
c.ry(2, theta=-1.0)

z0x2 = c.expectation([tc.gates.z(), [0]], [tc.gates.x(), [2]])
z0x2p1 = tc.templates.measurements.parameterized_measurements(
```

(continues on next page)

(continued from previous page)

```

        c, tc.array_to_tensor(np.array([3, 0, 1])), onehot=True
    )
z0x2p2 = tc.templates.measurements.parameterized_measurements(
    c,
    tc.array_to_tensor(np.array([[0, 0, 0, 1], [1, 0, 0, 0], [0, 1, 0, 0]])),
    onehot=False,
)
np.testing.assert_allclose(z0x2, z0x2p1)
np.testing.assert_allclose(z0x2, z0x2p2)

```

Parameters

- **c** (*Circuit*) – The circuit to be measured
- **structures** (*Tensor*) – parameter tensors determines what Pauli string to be measured, shape is [nwires, 4] if onehot is False and [nwires] if onehot is True.
- **onehot** (*bool, optional*) – defaults to False. If set to be True, structures will first go through onehot procedure.
- **reuse** (*bool, optional*) – reuse the wavefunction when computing the expectations, defaults to be False

Returns The expectation value of given Pauli string by the tensor **structures**.**Return type** Tensor

`tensorcircuit.templates.measurements.sparse_expectation(c: tensorcircuit.circuit.Circuit,
hamiltonian: Any) → Any`

Evaluate Hamiltonian expectation where hamiltonian is kept in sparse matrix form to save space

Parameters

- **c** (*Circuit*) – The circuit whose output state is used to evaluate the expectation
- **hamiltonian** (*Tensor*) – Hamiltonian matrix in COO_sparse_matrix form

Returns a real and scalar tensor of shape [] as the expectation value**Return type** Tensor

`tensorcircuit.templates.measurements.spin_glass_measurements(c: tensorcircuit.circuit.Circuit, g:
Any, reuse: bool = True) → Any`

Compute spin glass energy defined on graph g expectation for output state of the circuit c. The Hamiltonian to be evaluated is defined as (first term is determined by node weights while the second term is determined by edge weights of the graph):

$$H = \sum_{v \in g} w_v Z_v + \sum_{e \in g} w_e Z_{e1} Z_{e2}$$

Example

```

import networkx as nx

# building the lattice graph for spin glass Hamiltonian
g = nx.Graph()
g.add_node(0, weight=1)
g.add_node(1, weight=-1)

```

(continues on next page)

(continued from previous page)

```

g.add_node(2, weight=1)
g.add_edge(0, 1, weight=-1)
g.add_edge(1, 2, weight=-1)

c = tc.Circuit(3)
c.X(1)
energy = tc.templates.measurements.spin_glass_measurements(c, g)
print(energy) # 5.0

```

Parameters

- **c** ([Circuit](#)) – The quantum circuit
- **g** ([Graph](#)) – The graph for spin glass Hamiltonian definition
- **reuse** (`bool, optional`) – Whether measure the circuit with reusing the wavefunction, defaults to True

Returns The spin glass energy expectation value**Return type** Tensor

4.1.21 tensorcircuit.torchnn

PyTorch nn Module wrapper for quantum function

```

class tensorcircuit.torchnn.QuantumNet(f: Callable[[], Any], weights_shape: Sequence[Tuple[int, ...]],  

                                         initializer: Optional[Union[Any, Sequence[Any]]] = None,  

                                         use_vmap: bool = True, vectorized_argnums: Union[int,  

                                         Sequence[int]] = 0, use_interface: bool = True, use_jit: bool =  

                                         True, enable_dlpack: bool = False)  

Bases: torch.nn.modules.module.Module  

T_destination  

alias of TypeVar('T_destination', bound=Dict[str, Any])  

__init__(f: Callable[[], Any], weights_shape: Sequence[Tuple[int, ...]], initializer: Optional[Union[Any,  

                                         Sequence[Any]]] = None, use_vmap: bool = True, vectorized_argnums: Union[int, Sequence[int]]  

                                         = 0, use_interface: bool = True, use_jit: bool = True, enable_dlpack: bool = False)  

PyTorch nn Module wrapper on quantum function f.

```

Example

```

K = tc.set_backend("tensorflow")

n = 6
nlayers = 2
batch = 2

def qpred(x, weights):
    c = tc.Circuit(n)
    for i in range(n):
        c.rx(i, theta=x[i])
    for j in range(nlayers):
        for i in range(n - 1):

```

(continues on next page)

(continued from previous page)

```

    c.cnot(i, i + 1)
    for i in range(n):
        c.rx(i, theta=weights[2 * j, i])
        c.ry(i, theta=weights[2 * j + 1, i])
    ypred = K.stack([c.expectation_ps(x=[i]) for i in range(n)])
    ypred = K.real(ypred)
    return ypred

ql = tc.torchnn.QuantumNet(qpred, weights_shape=[2*nlayers, n])

ql(torch.ones([batch, n]))

```

Parameters

- **f** (*Callable[..., Any]*) – Quantum function with tensor in (input and weights) and tensor out.
- **weights_shape** (*Sequence[Tuple[int, ...]]*) – list of shape tuple for different weights as the non-first parameters for f
- **initializer** (*Union[Any, Sequence[Any]]*, *optional*) – function that gives the shape tuple returns torch tensor, defaults to None
- **use_vmap** (*bool*, *optional*) – whether apply vmap (batch input) on f, defaults to True
- **vectorized_argnums** (*Union[int, Sequence[int]]*) – which position of input should be batched, need to be customized when multiple inputs for the torch model, defaults to be 0.
- **use_interface** (*bool*, *optional*) – whether transform f with torch interface, defaults to True
- **use_jit** (*bool*, *optional*) – whether jit f, defaults to True
- **enable_dlpark** (*bool*, *optional*) – whether enable dlpark in interfaces, defaults to False

add_module(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → *torch.nn.modules.module.T*

Applies fn recursively to every submodule (as returned by .children()) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (*Module -> None*) – function to be applied to each submodule**Returns** self**Return type** Module

Example:

```

>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
       [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
       [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)

```

bfloat16() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(reuse: bool = True) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Parameters **reuse** (bool) – if True, then yields buffers of this module and all submodules.

Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device*: *Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to double datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to float datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*inputs: Any) → Any

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

get_buffer(target: str) → torch.Tensor

Returns the buffer given by target if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify target.

Parameters `target` – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by target

Return type torch.Tensor

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any

Returns any extra state to include in the module's state_dict. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(target: str) → torch.nn.parameter.Parameter

Returns the parameter given by target if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify target.

Parameters `target` – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by target

Return type torch.nn.Parameter

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(  
    (net_b): Module(  
        (net_c): Module(  
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))  
        )  
        (linear): Linear(in_features=100, out_features=200, bias=True)  
    )  
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in *target*. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters **target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by *target*

Return type torch.nn.Module

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

half() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict*: *Mapping[str, Any]*, *strict*: *bool* = True)

Copies parameters and buffers from *state_dict* into this module and its descendants. If *strict* is True, then the keys of *state_dict* must exactly match the keys returned by this module's *state_dict()* function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's *state_dict()* function. Default: True

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type NamedTuple with *missing_keys* and *unexpected_keys* fields

Note: If a parameter or buffer is registered as None and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
    (0): Linear(in_features=2, out_features=2, bias=True)
    (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix*: *str* = "", *recurse*: *bool* = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields (*str, torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields (str, Module) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(memo: Optional[Set[torch.nn.modules.module.Module]] = None, prefix: str = "", remove_duplicate: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields (str, Module) – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(prefix: str = "", recurse: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix (str)** – prefix to prepend to all parameter names.

- **recurse (bool)** – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields (str, Parameter) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse (bool)** – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields Parameter – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[Tuple[torch.Tensor, ...], torch.Tensor]]*, *None*, *torch.Tensor*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type torch.utils.hooks.RemovableHandle

register_buffer(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Parameters

- **name (str)** – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor (Tensor or None)** – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module's `state_dict`.

- **persistent** (`bool`) – whether the buffer is part of this module’s `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(`hook: Callable[[], None]`) → `torch.utils.hooks.RemovableHandle`
Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(`hook: Callable[[], None]`) → `torch.utils.hooks.RemovableHandle`
Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(`hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]`) → `torch.utils.hooks.RemovableHandle`
Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module’s forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

`register_load_state_dict_post_hook(hook)`

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

`register_module(name: str, module: Optional[torch.nn.modules.module.Module]) -> None`

Alias for `add_module()`.

`register_parameter(name: str, param: Optional[torch.nn.parameter.Parameter]) -> None`

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (`str`) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (`Parameter` or `None`) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

`requires_grad_(requires_grad: bool = True) -> torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

Parameters `requires_grad` (`bool`) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns `self`

Return type `Module`

set_extra_state(state: Any)

This function is called from `load_state_dict()` to handle any extra state found within the `state_dict`. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its `state_dict`.

Parameters `state (dict)` – Extra state from the `state_dict`

share_memory() → torch.nn.modules.module.T

See `torch.Tensor.share_memory_()`

state_dict(*args, destination=None, prefix='', keep_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination (dict, optional)** – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix (str, optional)** – a prefix added to parameter and buffer names to compose the keys in `state_dict`. Default: ''.
- **keep_vars (bool, optional)** – by default the `Tensor`s returned in the `state_dict` are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(*args, **kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

`to(device=None, dtype=None, non_blocking=False)`

`to(dtype, non_blocking=False)`

```
to(tensor, non_blocking=False)
to(memory_format=torch.channels_last)
```

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
       [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
       [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
       [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
       [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
       [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
       [0.6122+0.j, 0.1150+0.j],
       [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

to_empty(**device*: Union[str, torch.device]) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode*: bool = True) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Parameters **mode** (bool) – whether to set training mode (True) or evaluation mode (False).
Default: True.

Returns self

Return type Module

training: bool**type**(*dst_type*: Union[torch.dtype, str]) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst_type*.

Note: This method modifies the module in-place.

Parameters **dst_type** (type or string) – the desired type

Returns self

Return type Module

xpu(*device*: Optional[Union[int, torch.device]] = None) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device (int, optional)` – if specified, all parameters will be copied to that device

Returns self

Return type Module

`zero_grad(set_to_none: bool = False) → None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters `set_to_none (bool)` – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

`tensorcircuit.torchnn.TorchLayer`

alias of `tensorcircuit.torchnn.QuantumNet`

4.1.22 tensorcircuit.translation

Circuit object translation in different packages

`tensorcircuit.translation.ctrl_str2ctrl_state(ctrl_str: str, nctrl: int) → List[int]`

`tensorcircuit.translation.eqasm2tc(eqasm: str, nqubits: Optional[int] = None, headers: Tuple[int, int] = (6, 1)) → tensorcircuit.circuit.Circuit`

Translation qexe/eqasm instruction to tensorcircuit Circuit object

Parameters

- `eqasm (str)` – _description_
- `nqubits (Optional[int], optional)` – _description_, defaults to None
- `headers (Tuple[int, int], optional)` – lines of ignored code at the head and the tail, defaults to (6, 1)

Returns _description_

Return type `Circuit`

`tensorcircuit.translation.json2qir(tcqasm: List[Dict[str, Any]]) → List[Dict[str, Any]]`

`tensorcircuit.translation.json_to_tensor(a: Any) → Any`

`tensorcircuit.translation.perm_matrix(n: int) → Any`

Generate a permutation matrix P. Due to the different convention or qubits' order in qiskit and tensorcircuit, the unitary represented by the same circuit is different. They are related by this permutation matrix P: P @ U_qiskit @ P = U_tc

Parameters `n (int)` – # of qubits

Returns The permutation matrix P

Return type Tensor

`tensorcircuit.translation.qir2cirq(qir: List[Dict[str, Any]], n: int, extra_qir: Optional[List[Dict[str, Any]]] = None) → Any`

Generate a cirq circuit using the quantum intermediate representation (qir) in tensorcircuit.

Example

```
>>> c = tc.Circuit(2)
>>> c.H(1)
>>> c.X(1)
>>> cisc = tc.translation.qir2cirq(c.to_qir(), 2)
>>> print(cisc)
1: —H—X—
```

Parameters

- **qir** (*List[Dict[str, Any]]*) – The quantum intermediate representation of a circuit.
- **n** (*int*) – # of qubits
- **extra_qir** (*Optional[List[Dict[str, Any]]]*) – The extra quantum IR of tc circuit including measure and reset on hardware, defaults to None

Returns qiskit cirq object

Return type Any

#TODO(@erertertet): add default theta to iswap gate add more cirq built-in gate instead of customized add unitary test with tolerance add support of cirq built-in ControlledGate for multiplecontroll support more element in qir, e.g. barrier, measure...

`tensorcircuit.translation.qir2json(qir: List[Dict[str, Any]], simplified: bool = False) → List[Dict[str, Any]]`

transform qir to json compatible list of dict where array is replaced by real and imaginary list

Parameters

- **qir** (*List[Dict[str, Any]]*) – _description_
- **simplified** (*bool*) – If False, keep all info for each gate, defaults to be False. If True, suitable for IO since less information is required

Returns _description_

Return type List[Dict[str, Any]]

`tensorcircuit.translation.qir2qiskit(qir: List[Dict[str, Any]], n: int, extra_qir: Optional[List[Dict[str, Any]]] = None) → Any`

Generate a qiskit quantum circuit using the quantum intermediate representation (qir) in tensorcircuit.

Example

```
>>> c = tc.Circuit(2)
>>> c.H(1)
>>> c.X(1)
>>> qisc = tc.translation.qir2qiskit(c.to_qir(), 2)
>>> qisc.data
[(Instruction(name='h', num_qubits=1, num_clbits=0, params=[]), [Qubit(QuantumRegister(2, 'q'), 1)], []),
 (Instruction(name='x', num_qubits=1, num_clbits=0, params=[]), [Qubit(QuantumRegister(2, 'q'), 1)])]
```

Parameters

- **qir** (*List[Dict[str, Any]]*) – The quantum intermediate representation of a circuit.

- **n** (*int*) – # of qubits
- **extra_qir** (*Optional[List[Dict[str, Any]]]*) – The extra quantum IR of tc circuit including measure and reset on hardware, defaults to None

Returns qiskit QuantumCircuit object

Return type Any

```
tensorcircuit.translation.qiskit2tc(qcdata: List[Any], n: int, inputs: Optional[List[float]] = None,
                                    is_dm: bool = False, circuit_constructor: Optional[Any] = None,
                                    circuit_params: Optional[Dict[str, Any]] = None, binding_params:
                                    Optional[Union[Sequence[float], Dict[Any, float]]] = None) → Any
```

Generate a tensorcircuit circuit using the quantum circuit data in qiskit.

Example

```
>>> qisc = QuantumCircuit(2)
>>> qisc.h(0)
>>> qisc.x(1)
>>> qc = tc.translation.qiskit2tc(qisc.data, 2)
>>> qc.to_qir()[0]['gatef']
h
```

Parameters

- **qcdata** (*List[CircuitInstruction]*) – Quantum circuit data from qiskit.
- **n** (*int*) – # of qubits
- **inputs** (*Optional[List[float]]*) – Input state of the circuit. Default is None.
- **circuit_constructor** – Circuit, DMCircuit or MPSCircuit
- **circuit_params** (*Optional[Dict[str, Any]]*) – kwargs given in Circuit.__init__ construction function, default to None.
- **binding_params** (*Optional[Union[Sequence[float], Dict[Any, float]]]*) – (variational) parameters for the circuit. Could be either a sequence or dictionary depending on the type of parameters in the Qiskit circuit. For ParameterVectorElement use sequence. For Parameter use dictionary

Returns A quantum circuit in tensorcircuit

Return type Any

```
tensorcircuit.translation.qiskit_from_qasm_str_ordered_measure(qasm_str: str) → Any
```

qiskit from_qasm_str method cannot keep the order of measure as the qasm file, we provide this alternative function in case the order of measure instruction matters

Parameters **qasm_str** (*str*) – open qasm str

Returns qiskit.circuit.QuantumCircuit

Return type Any

```
tensorcircuit.translation.tensor_to_json(a: Any) → Any
```

4.1.23 tensorcircuit.utils

Helper functions

`tensorcircuit.utils.append(f: Callable[...], Any], *op: Callable[...], Any]) → Any`

Functional programming paradigm to build function pipeline

Example

```
>>> f = tc.utils.append(lambda x: x**2, lambda x: x+1, tc.backend.mean)
>>> f(tc.backend.ones(2))
(2+0j)
```

Parameters

- `f (Callable[..., Any])` – The function which are attached with other functions
- `op (Callable[..., Any])` – Function to be attached

Returns The final results after function pipeline

Return type Any

`tensorcircuit.utils.arg_alias(f: Callable[...], Any], alias_dict: Dict[str, Union[str, Sequence[str]]], fix_doc: bool = True) → Callable[...], Any]`

function argument alias decorator with new docstring

Parameters

- `f (Callable[..., Any])` – _description_
- `alias_dict (Dict[str, Union[str, Sequence[str]]])` – _description_
- `fix_doc (bool)` – whether to add doc for these new alias arguments, defaults True

Returns the decorated function

Return type Callable[..., Any]

`tensorcircuit.utils.benchmark(f: Any, *args: Any, tries: int = 5, verbose: bool = True) → Tuple[Any, float, float]`

benchmark jittable function with staging time and running time

Parameters

- `f (Any)` – _description_
- `tries (int, optional)` – _description_, defaults to 5
- `verbose (bool, optional)` – _description_, defaults to True

Returns _description_

Return type Tuple[Any, float, float]

`tensorcircuit.utils.is_m1mac() → bool`

check whether the running platform is MAC with M1 chip

Returns True for MAC M1 platform

Return type bool

`tensorcircuit.utils.is_number(x: Any) → bool`

`tensorcircuit.utils.is_sequence(x: Any) → bool`

`tensorcircuit.utils.return_partial(f: Callable[..., Any], return_argnums: Union[int, Sequence[int]] = 0) → Callable[..., Any]`

Return a callable function for output ith parts of the original output along the first axis. Original output supports List and Tensor.

Example

```
>>> from tensorcircuit.utils import return_partial
>>> testin = np.array([[1,2],[3,4],[5,6],[7,8]])
>>> # Method 1:
>>> return_partial(lambda x: x, [1, 3])(testin)
(array([3, 4]), array([7, 8]))
>>> # Method 2:
>>> from functools import partial
>>> @partial(return_partial, return_argnums=(0,2))
... def f(inp):
...     return inp
...
>>> f(testin)
(array([1, 2]), array([5, 6]))
```

Parameters

- `f(Callable[..., Any])` – The function to be applied this method
- `return_partial(Union[int, Sequence[int]])` – The ith parts of original output along the first axis (axis=0 or dim=0)

Returns The modified callable function

Return type Callable[..., Any]

4.1.24 tensorcircuit.vis

Visualization on circuits

`tensorcircuit.vis.gate_name_trans(gate_name: str) → Tuple[int, str]`

Translating from the gate name to gate information including the number of control qubits and the reduced gate name.

Example

```
>>> string = r'ccnot'
>>> tc.vis.gate_name_trans(string)
2 'not'
```

Parameters `gate_name(str)` – String of gate name

Returns # of control qubits, reduced gate name

Return type Tuple[int, str]

`tensorcircuit.vis.qir2tex(qir: List[Dict[str, Any]], n: int, init: Optional[List[str]] = None, measure: Optional[List[str]] = None, rcompress: bool = False, lcompress: bool = False, standalone: bool = False, return_string_table: bool = False) → Union[str, Tuple[str, List[List[str]]]]`

Generate Tex code from ‘qir’ string to illustrate the circuit structure. This visualization is based on quantikz

package.

Example

```
>>> qir=[{'index': [0], 'name': 'h'}, {'index': [1], 'name': 'phase'}]
>>> tc.vis.qir2tex(qir,2)
'\\begin{quantikz}\n\\ ... \n\\end{quantikz}'
```

Parameters

- **qir** (*List[Dict[str, Any]]*) – The quantum intermediate representation of a circuit in tensorcircuit.
- **n** (*int*) – # of qubits
- **init** (*Optional[List[str]]*) – Initial state, default is an all zero state ‘000...000’.
- **measure** (*Optional[List[str]]*) – Measurement Basis, default is None which means no measurement in the end of the circuit.
- **rcompress** – If true, a right compression of the circuit will be conducted. A right compression means we will try to shift gates from right to left if possible.

Default is false. :type rcompress: bool :param lcompress: If true, a left compression of the circuit will be conducted.

A left compression means we will try to shift gates from left to right if possible. Default is false.

Parameters

- **standalone** (*bool*) – If true, the tex code will be designed to generate a standalone document. Default is false which means the generated tex code is just a quantikz code block.
- **return_string_table** (*bool*) – If true, a string table of tex code will also be returned. Default is false.

Returns Tex code of circuit visualization based on quantikz package. If return_string_table is true, a string table of tex code will also be returned.

Return type Union[str, Tuple[str, List[List[str]]]]

```
tensorcircuit.vis.render_pdf(texcode: str, filename: Optional[str] = None, latex: Optional[str] = None,
                             filepath: Optional[str] = None, notebook: bool = False) → Any
```

Generate the PDF file with given latex string and filename. Latex command and file path can be specified. When notebook is True, convert the output PDF file to image and return a Image object.

Example

```
>>> string = r'''\\documentclass[a4paper,12pt]{article}
... \\begin{document}
... \\title{Hello TensorCircuit!}
... \\end{document}'''
>>> tc.vis.render_pdf(string, "test.pdf", notebook=False)
>>> os.listdir()
['test.aux', 'test.log', 'test.pdf', 'test.tex']
```

Parameters

- **texcode** (*str*) – String of latex content

- **filename** (*Optional[str]*, *optional*) – File name, defaults to random UUID
str(uuid4())
- **latex** (*Optional[str]*, *optional*) – Executable Latex command, defaults to *pdflatex*
- **filepath** (*Optional[str]*, *optional*) – File path, defaults to current working place
os.getcwd()
- **notebook** (*bool*, *optional*) – [description], defaults to False

Returns if notebook is True, return *Image* object; otherwise return *None*

Return type *Optional[Image]*, defaults to *None*

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

tensorcircuit.abstractcircuit, 195
tensorcircuit.applications.dqas, 204
tensorcircuit.applications.graphdata, 208
tensorcircuit.applications.layers, 209
tensorcircuit.applications.utils, 220
tensorcircuit.applications.vags, 221
tensorcircuit.applications.van, 227
tensorcircuit.applications.vqes, 352
tensorcircuit.backends.backend_factory, 365
tensorcircuit.backends.jax_backend, 365
tensorcircuit.backends.numpy_backend, 394
tensorcircuit.backends.pytorch_backend, 421
tensorcircuit.backends.tensorflow_backend,
 449
tensorcircuit.basecircuit, 478
tensorcircuit.channels, 492
tensorcircuit.circuit, 499
tensorcircuit.compiler.qiskit_compiler, 534
tensorcircuit.cons, 535
tensorcircuit.densitymatrix, 539
tensorcircuit.experimental, 601
tensorcircuit.gates, 602
tensorcircuit.interfaces.numpy, 613
tensorcircuit.interfaces.scipy, 614
tensorcircuit.interfaces.tensorflow, 616
tensorcircuit.interfaces.tensortrans, 618
tensorcircuit.interfaces.torch, 620
tensorcircuit.keras, 622
tensorcircuit.mps_base, 635
tensorcircuit.mpscircuit, 639
tensorcircuit.noisemodel, 666
tensorcircuit.quantum, 668
tensorcircuit.results.counts, 697
tensorcircuit.results.readout_mitigation, 697
tensorcircuit.simplify, 700
tensorcircuit.templates.blocks, 701
tensorcircuit.templates.chems, 703
tensorcircuit.templates.dataset, 703
tensorcircuit.templates.graphs, 703
tensorcircuit.templates.measurements, 704
tensorcircuit.torchnn, 709
tensorcircuit.translation, 723
tensorcircuit.utils, 726
tensorcircuit.vis, 727

INDEX

Symbols

- `__init__()` (`tensorcircuit.applications.van.MADE method`), 227
- `__init__()` (`tensorcircuit.applications.van.MaskedConv2D method`), 256
- `__init__()` (`tensorcircuit.applications.van.MaskedLinear method`), 268
- `__init__()` (`tensorcircuit.applications.van.NMF method`), 280
- `__init__()` (`tensorcircuit.applications.van.PixelCNN method`), 310
- `__init__()` (`tensorcircuit.applications.van.ResidualBlock method`), 340
- `__init__()` (`tensorcircuit.applications.vqes.JointSchedule method`), 352
- `__init__()` (`tensorcircuit.applications.vqes.Linear method`), 352
- `__init__()` (`tensorcircuit.applications.vqes.VQNHE method`), 364
- `__init__()` (`tensorcircuit.backends.jax_backend.JaxBackend method`), 365
- `__init__()` (`tensorcircuit.backends.jax_backend.optax_optimizer method`), 393
- `__init__()` (`tensorcircuit.backends.numpy_backend.NumpyBackend method`), 394
- `__init__()` (`tensorcircuit.backends.pytorch_backend.PyTorchBackend method`), 421
- `__init__()` (`tensorcircuit.backends.pytorch_backend.torch_optimizer method`), 449
- `__init__()` (`tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method`), 449
- `__init__()` (`tensorcircuit.backends.tensorflow_backend.keras_optimizer method`), 477
- `__init__()` (`tensorcircuit.channels.KrausList method`), 492
- `__init__()` (`tensorcircuit.circuit.Circuit method`), 506
- `__init__()` (`tensorcircuit.densitymatrix.DMCircuit method`), 545
- `__init__()` (`tensorcircuit.densitymatrix.DMCircuit2 method`), 576
- `__init__()` (`tensorcircuit.gates.Gate method`), 602
- `__init__()` (`tensorcircuit.gates.GateF method`), 605
- `__init__()` (`tensorcircuit.gates.GateVF method`), 605
- `__init__()` (`tensorcircuit.keras.QuantumLayer method`), 622
- `__init__()` (`tensorcircuit.mps_base.FiniteMPS method`), 635
- `__init__()` (`tensorcircuit.mpscircuit.MPSCircuit method`), 645
- `__init__()` (`tensorcircuit.noisemodel.NoiseConf method`), 666
- `__init__()` (`tensorcircuit.quantum.QuAdjointVector method`), 670
- `__init__()` (`tensorcircuit.quantum.QuOperator method`), 674
- `__init__()` (`tensorcircuit.quantum.QuScalar method`), 677
- `__init__()` (`tensorcircuit.quantum.QuVector method`), 680
- `__init__()` (`tensorcircuit.results.readout_mitigation.ReadoutMit method`), 697
- `__init__()` (`tensorcircuit.templates.graphs.Grid2DCoord method`), 703
- `__init__()` (`tensorcircuit.torchnn.QuantumNet method`), 709

A

- `abs()` (`tensorcircuit.backends.jax_backend.JaxBackend method`), 365
- `abs()` (`tensorcircuit.backends.numpy_backend.NumpyBackend method`), 394

abs() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 421
abs() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 449
AbstractCircuit (class in *tensorcircuit.abstractcircuit*), 195
acos() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 365
acos() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 394
acos() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 421
acos() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 449
acosh() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 366
acosh() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 394
acosh() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 421
acosh() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 450
activity_regularizer (*tensorcircuit.applications.van.MADE property*), 227
activity_regularizer (*tensorcircuit.applications.van.MaskedConv2D property*), 256
activity_regularizer (*tensorcircuit.applications.van.MaskedLinear property*), 268
activity_regularizer (*tensorcircuit.applications.van.NMF property*), 280
activity_regularizer (*tensorcircuit.applications.van.PixelCNN property*), 310
activity_regularizer (*tensorcircuit.applications.van.ResidualBlock property*), 340
activity_regularizer (*tensorcircuit.applications.vqes.Linear property*), 352
activity_regularizer (*tensorcircuit.keras.QuantumLayer property*), 622
adaptive_vmap() (in module *tensorcircuit.experimental*), 601
add_axis_names() (*tensorcircuit.gates.Gate method*), 603
add_edge() (*tensorcircuit.gates.Gate method*), 603
add_loss() (*tensorcircuit.applications.van.MADE method*), 227
add_loss() (*tensorcircuit.applications.van.MaskedConv2D method*), 256
add_loss() (*tensorcircuit.applications.van.MaskedLinear method*), 268
add_loss() (*tensorcircuit.backends.TensorFlowBackend method*), 280
add_loss() (*tensorcircuit.applications.van.PixelCNN method*), 310
add_loss() (*tensorcircuit.applications.van.ResidualBlock method*), 340
add_loss() (*tensorcircuit.applications.vqes.Linear method*), 352
add_loss() (*tensorcircuit.applications.van.MADE method*), 227
add_loss() (*tensorcircuit.applications.van.PixelCNN method*), 310
add_loss() (*tensorcircuit.applications.van.ResidualBlock method*), 340
add_loss() (*tensorcircuit.applications.vqes.Linear method*), 352
add_metric() (*tensorcircuit.applications.van.MADE method*), 227
add_metric() (*tensorcircuit.applications.van.PixelCNN method*), 310
add_metric() (*tensorcircuit.applications.van.ResidualBlock method*), 340
add_metric() (*tensorcircuit.applications.vqes.Linear method*), 352
add_metric() (*tensorcircuit.applications.van.MADE method*), 227
add_metric() (*tensorcircuit.applications.van.PixelCNN method*), 310
add_metric() (*tensorcircuit.applications.van.ResidualBlock method*), 340
add_metric() (*tensorcircuit.applications.vqes.Linear method*), 352
add_module() (*tensorcircuit.torchnn.QuantumNet method*), 710
add_noise() (*tensorcircuit.noisemodel.NoiseConf method*), 666
add_update() (*tensorcircuit.applications.van.MADE method*), 228
add_update() (*tensorcircuit.applications.van.MaskedConv2D method*), 258
add_update() (*tensorcircuit.applications.van.MaskedLinear method*), 270
add_update() (*tensorcircuit.applications.van.NMF method*), 281
add_update() (*tensorcircuit.applications.van.PixelCNN method*), 311
add_update() (*tensorcircuit.applications.van.ResidualBlock method*), 341
add_update() (*tensorcircuit.applications.vqes.Linear method*), 352

method), 353

`add_update()` (*tensorcircuit.keras.QuantumLayer method*), 624

`add_variable()` (*tensorcircuit.applications.van.MADE method*), 228

`add_variable()` (*tensorcircuit.applications.van.MaskedConv2D method*), 258

`add_variable()` (*tensorcircuit.applications.van.MaskedLinear method*), 270

`add_variable()` (*tensorcircuit.applications.van.NMF method*), 282

`add_variable()` (*tensorcircuit.applications.van.PixelCNN method*), 311

`add_variable()` (*tensorcircuit.applications.van.ResidualBlock method*), 341

`add_variable()` (*tensorcircuit.applications.vqes.Linear method*), 354

`add_variable()` (*tensorcircuit.keras.QuantumLayer method*), 624

`add_weight()` (*tensorcircuit.applications.van.MADE method*), 228

`add_weight()` (*tensorcircuit.applications.van.MaskedConv2D method*), 258

`add_weight()` (*tensorcircuit.applications.van.MaskedLinear method*), 270

`add_weight()` (*tensorcircuit.applications.van.NMF method*), 282

`add_weight()` (*tensorcircuit.applications.van.PixelCNN method*), 312

`add_weight()` (*tensorcircuit.applications.van.ResidualBlock method*), 341

`add_weight()` (*tensorcircuit.applications.vqes.Linear method*), 354

`add_weight()` (*tensorcircuit.keras.QuantumLayer method*), 624

`addition()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 366

`addition()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 394

`addition()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 421

`addition()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 450

`adjoint()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 366

`adjoint()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 394

`adjoint()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 421

`adjoint()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 450

`adjoint()` (*tensorcircuit.gates.GateF method*), 605

`adjoint()` (*tensorcircuit.gates.GateVF method*), 605

`adjoint()` (*tensorcircuit.quantum.QuAdjointVector method*), 670

`adjoint()` (*tensorcircuit.quantum.QuOperator method*), 674

`adjoint()` (*tensorcircuit.quantum.QuScalar method*), 677

`adjoint()` (*tensorcircuit.quantum.QuVector method*), 680

`all_cols()` (*tensorcircuit.templates.graphs.Grid2DCoord method*), 703

`all_nodes_covered()` (*in module tensorcircuit.applications.graphdata*), 208

`all_rows()` (*tensorcircuit.templates.graphs.Grid2DCoord method*), 703

`all_zero_nodes()` (*tensorcircuit.basecircuit.BaseCircuit static method*), 478

`all_zero_nodes()` (*tensorcircuit.circuit.Circuit static method*), 506

`all_zero_nodes()` (*tensorcircuit.densitymatrix.DMCircuit static method*), 546

`all_zero_nodes()` (*tensorcircuit.densitymatrix.DMCircuit2 static method*), 577

`amplitude()` (*tensorcircuit.basecircuit.BaseCircuit method*), 478

`amplitude()` (*tensorcircuit.circuit.Circuit method*), 506

`amplitude()` (*tensorcircuit.densitymatrix.DMCircuit method*), 546

`amplitude()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 577

`amplitude()` (*tensorcircuit.mpscircuits.MPSCircuit method*), 645

`amplitude_encoding()` (*in module tensorcircuit.applications.utils*), 220

`amplitude_encoding()` (*in module tensorcircuit.templates.dataset*), 703

`amplitudedamping()` (*tensorcircuit.circuit.Circuit method*), 506

`amplitudedamping()` (*tensorcircuit.circuit.Circuit method*), 506

<code>cuit.densitymatrix.DMCircuit method), 546</code>		
<code>amplitudedamping() (in module tensorcircuit.densitymatrix.DMCircuit2 method), 577</code>	<code>(tensorcircuit.densitymatrix.DMCircuit method),</code>	
<code>amplitudedampingchannel() (in module tensorcircuit.channels), 492</code>		
<code>ANY() (tensorcircuit.circuit.Circuit method), 499</code>		
<code>any() (tensorcircuit.circuit.Circuit method), 506</code>		
<code>ANY() (tensorcircuit.densitymatrix.DMCircuit method), 539</code>		
<code>any() (tensorcircuit.densitymatrix.DMCircuit method), 546</code>		
<code>ANY() (tensorcircuit.densitymatrix.DMCircuit2 method), 570</code>		
<code>any() (tensorcircuit.densitymatrix.DMCircuit2 method), 577</code>		
<code>ANY() (tensorcircuit.mpscircuit.MPSCircuit method), 639</code>		
<code>any() (tensorcircuit.mpscircuit.MPSCircuit method), 645</code>		
<code>any_gate() (in module tensorcircuit.gates), 605</code>		
<code>any_local_measurements() (in module tensorcircuit.templates.measurements), 704</code>		
<code>any_measurements() (in module tensorcircuit.templates.measurements), 705</code>		
<code>anyHlayer() (in module tensorcircuit.applications.layers), 209</code>		
<code>anyIlayer() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyrxlayer() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyrylayer() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyrzlayer() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyswaplayer() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyswaplayer_bitflip_mc() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyxxlayer() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyxxlayer_bitflip_mc() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyxylayer() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyxylayer_bitflip_mc() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyxzlayer() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyxzlayer_bitflip_mc() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyyxlayer() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyyxlayer_bitflip_mc() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyyyxlayer() (in module tensorcircuit.applications.layers), 210</code>		
<code>anyyyxlayer_bitflip_mc() (in module tensorcircuit.applications.layers), 210</code>		
<code>cuit.applications.layers), 210</code>		
<code>anyyylayer() (in module tensorcircuit.applications.layers), 211</code>		
<code>anyyylayer_bitflip_mc() (in module tensorcircuit.applications.layers), 211</code>		
<code>anyyzlayer() (in module tensorcircuit.applications.layers), 211</code>		
<code>anyyzlayer_bitflip_mc() (in module tensorcircuit.applications.layers), 211</code>		
<code>anyzxlayer() (in module tensorcircuit.applications.layers), 211</code>		
<code>anyzxlayer_bitflip_mc() (in module tensorcircuit.applications.layers), 211</code>		
<code>anyzylayer() (in module tensorcircuit.applications.layers), 211</code>		
<code>anyzylayer_bitflip_mc() (in module tensorcircuit.applications.layers), 211</code>		
<code>anyzzlayer() (in module tensorcircuit.applications.layers), 211</code>		
<code>anyzzlayer_bitflip_mc() (in module tensorcircuit.applications.layers), 211</code>		
<code>append() (in module tensorcircuit.utils), 726</code>		
<code>append() (tensorcircuit.abstractcircuit.AbstractCircuit method), 195</code>		
<code>append() (tensorcircuit.basecircuit.BaseCircuit method), 478</code>		
<code>append() (tensorcircuit.channels.KrausList method), 492</code>		
<code>append() (tensorcircuit.circuit.Circuit method), 507</code>		
<code>append() (tensorcircuit.densitymatrix.DMCircuit method), 546</code>		
<code>append() (tensorcircuit.densitymatrix.DMCircuit2 method), 577</code>		
<code>append() (tensorcircuit.mpscircuit.MPSCircuit method), 646</code>		
<code>append_from_qir() (tensorcircuit.abstractcircuit.AbstractCircuit method), 195</code>		
<code>append_from_qir() (tensorcircuit.basecircuit.BaseCircuit method), 479</code>		
<code>append_from_qir() (tensorcircuit.circuit.Circuit method), 507</code>		
<code>append_from_qir() (tensorcircuit.densitymatrix.DMCircuit method), 547</code>		
<code>append_from_qir() (tensorcircuit.densitymatrix.DMCircuit2 method), 578</code>		
<code>append_from_qir() (tensorcircuit.mpscircuit.MPSCircuit method), 646</code>		
<code>apply() (tensorcircuit.basecircuit.BaseCircuit method), 479</code>		
<code>apply() (tensorcircuit.circuit.Circuit method), 507</code>		
<code>apply() (tensorcircuit.densitymatrix.DMCircuit method), 547</code>		

apply()	(<i>tensorcircuit.densitymatrix.DMCircuit2 method</i>), 578	<i>cuit.densitymatrix.DMCircuit static method</i>), 548
apply()	(<i>tensorcircuit.mpscircuits.MPSCircuit method</i>), 647	<i>apply_general_kraus_delayed()</i> (<i>tensorcircuit.densitymatrix.DMCircuit2 static method</i>), 579
apply()	(<i>tensorcircuit.torchnn.QuantumNet method</i>), 710	<i>apply_general_variable_gate_delayed()</i> (<i>tensorcircuit.abstractcircuit.AbstractCircuit static method</i>), 196
apply_adjacent_double_gate()	(<i>tensorcircuit.mpscircuits.MPSCircuit method</i>), 647	<i>apply_general_variable_gate_delayed()</i> (<i>tensorcircuit.basecircuits.BaseCircuit static method</i>), 479
apply_correction()	(<i>tensorcircuit.results.readout_mitigation.ReadoutMit method</i>), 697	<i>apply_general_variable_gate_delayed()</i> (<i>tensorcircuit.circuits.Circuit static method</i>), 508
apply_double_gate()	(<i>tensorcircuit.mpscircuits.MPSCircuit method</i>), 647	<i>apply_general_variable_gate_delayed()</i> (<i>tensorcircuit.densitymatrix.DMCircuit static method</i>), 548
apply_general_gate()	(<i>tensorcircuit.abstractcircuit.AbstractCircuit method</i>), 196	<i>apply_general_variable_gate_delayed()</i> (<i>tensorcircuit.densitymatrix.DMCircuit2 static method</i>), 579
apply_general_gate()	(<i>tensorcircuit.basecircuits.BaseCircuit method</i>), 479	<i>apply_general_variable_gate_delayed()</i> (<i>tensorcircuit.mpscircuits.MPSCircuit static method</i>), 648
apply_general_gate()	(<i>tensorcircuit.circuits.Circuit method</i>), 508	<i>apply_MP0()</i> (<i>tensorcircuit.mpscircuits.MPSCircuit method</i>), 647
apply_general_gate()	(<i>tensorcircuit.densitymatrix.DMCircuit method</i>), 548	<i>apply_nqubit_gate()</i> (<i>tensorcircuit.mpscircuits.MPSCircuit method</i>), 648
apply_general_gate()	(<i>tensorcircuit.densitymatrix.DMCircuit2 method</i>), 579	<i>apply_one_site_gate()</i> (<i>tensorcircuit.mps_base.FiniteMPS method</i>), 635
apply_general_gate()	(<i>tensorcircuit.mpscircuits.MPSCircuit method</i>), 647	<i>apply_qir_with_noise()</i> (<i>in module tensorcircuit.noisemodel</i>), 667
apply_general_gate_delayed()	(<i>tensorcircuit.abstractcircuit.AbstractCircuit static method</i>), 196	<i>apply_readout_mitigation()</i> (<i>tensorcircuit.results.readout_mitigation.ReadoutMit method</i>), 698
apply_general_gate_delayed()	(<i>tensorcircuit.basecircuits.BaseCircuit static method</i>), 479	<i>apply_single_gate()</i> (<i>tensorcircuit.mpscircuits.MPSCircuit method</i>), 648
apply_general_gate_delayed()	(<i>tensorcircuit.circuits.Circuit static method</i>), 508	<i>apply_transfer_operator()</i> (<i>tensorcircuit.mps_base.FiniteMPS method</i>), 635
apply_general_gate_delayed()	(<i>tensorcircuit.densitymatrix.DMCircuit static method</i>), 548	<i>apply_two_site_gate()</i> (<i>tensorcircuit.mps_base.FiniteMPS method</i>), 635
apply_general_gate_delayed()	(<i>tensorcircuit.densitymatrix.DMCircuit2 static method</i>), 579	<i>arange()</i> (<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 366
apply_general_gate_delayed()	(<i>tensorcircuit.mpscircuits.MPSCircuit static method</i>), 647	<i>arange()</i> (<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 394
apply_general_kraus()	(<i>tensorcircuit.circuits.Circuit method</i>), 508	<i>arange()</i> (<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 421
apply_general_kraus()	(<i>tensorcircuit.densitymatrix.DMCircuit method</i>), 548	<i>arange()</i> (<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 450
apply_general_kraus()	(<i>tensorcircuit.densitymatrix.DMCircuit2 method</i>), 579	<i>arg_alias()</i> (<i>in module tensorcircuit.utils</i>), 726
apply_general_kraus_delayed()	(<i>tensorcircuit.circuits.Circuit static method</i>), 508	<i>argmax()</i> (<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 366
apply_general_kraus_delayed()	(<i>tensorcircuit.densitymatrix.DMCircuit2 method</i>), 579	<i>argmax()</i> (<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 394

argmax() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*.*TensorBackend*.*Backend*.*method*), 450
argmin() (*tensorcircuit.backends.jax_backend.JaxBackend*.*func*) (in module *tensorcircuit.applications.vags*), 221
argmin() (*tensorcircuit.backends.numpy_backend.NumpyBackend*.*Backend*.*method*), 395
argmin() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*AbstractCircuit*.*method*), 196
argmin() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*.*Backend*.*AbstractCircuit*.*method*), 450
args_to_tensor() (in module *tensorcircuit.interfaces.tensortrans*), 618
array_to_tensor() (in module *tensorcircuit.gates*), 605
asin() (*tensorcircuit.backends.jax_backend.JaxBackend*.*method*), 366
asin() (*tensorcircuit.backends.numpy_backend.NumpyBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*BaseCircuit*.*method*), 479
asin() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*Circuit*.*method*), 508
asinh() (*tensorcircuit.backends.jax_backend.JaxBackend*.*method*), 367
asinh() (*tensorcircuit.backends.numpy_backend.NumpyBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*DMCircuit*.*method*), 548
asinh() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*DMCircuit2*.*method*), 579
asinh() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*MPSCircuit*.*method*), 648
asinh() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*BaseCircuit* (class in *tensorcircuit.basecircuit*), 478
asinh() (*tensorcircuit.backends.jax_backend.JaxBackend*.*method*), 367
asinh() (*tensorcircuit.backends.numpy_backend.NumpyBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*templates.blocks*), 701
asinh() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*bitfliplayer*), 711
asinh() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*bitfliplayer*), 711
assign() (*tensorcircuit.applications.vqes.VQNHE*.*method*), 364
atan() (*tensorcircuit.backends.jax_backend.JaxBackend*.*method*), 367
atan() (*tensorcircuit.backends.numpy_backend.NumpyBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*applications.layers*), 211
atan() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*bitfliplayer_mc*), 211
atan2() (*tensorcircuit.backends.jax_backend.JaxBackend*.*method*), 367
atan2() (*tensorcircuit.backends.numpy_backend.NumpyBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*bmatrix*), 606
atan2() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*bond_dimension*), 636
atan2() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*bond_dimensions*), 636
atan2() (*tensorcircuit.backends.jax_backend.JaxBackend*.*method*), 367
atan2() (*tensorcircuit.backends.numpy_backend.NumpyBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*broadcast_left_multiplication*), 422
atan2() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*broadcast_left_multiplication*), 422
atan2() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*broadcast_left_multiplication*), 422
atanh() (*tensorcircuit.backends.jax_backend.JaxBackend*.*method*), 367
atanh() (*tensorcircuit.backends.numpy_backend.NumpyBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*broadcast_right_multiplication*), 396
atanh() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*broadcast_right_multiplication*), 396
atanh() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*.*Backend*, *barrier_instruction*) (in *tensorcircuit.circuit*.*broadcast_right_multiplication*), 451

broadcast_right_multiplication() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 423

broadcast_right_multiplication() (*tensorcircuit.backends.tensorflow_backend.TensorBackend method*), 451

buffers() (*tensorcircuit.torchnn.QuantumNet method*), 711

build() (*tensorcircuit.applications.van.MADE method*), 229

build() (*tensorcircuit.applications.van.MaskedConv2D method*), 259

build() (*tensorcircuit.applications.van.MaskedLinear method*), 271

build() (*tensorcircuit.applications.van.NMF method*), 282

build() (*tensorcircuit.applications.van.PixelCNN method*), 312

build() (*tensorcircuit.applications.van.ResidualBlock method*), 342

build() (*tensorcircuit.applications.vqes.Linear method*), 354

build() (*tensorcircuit.keras.QuantumLayer method*), 625

C

call() (*tensorcircuit.applications.van.MADE method*), 230

call() (*tensorcircuit.applications.van.MaskedConv2D method*), 259

call() (*tensorcircuit.applications.van.MaskedLinear method*), 271

call() (*tensorcircuit.applications.van.NMF method*), 283

call() (*tensorcircuit.applications.van.PixelCNN method*), 313

call() (*tensorcircuit.applications.van.ResidualBlock method*), 342

call() (*tensorcircuit.applications.vqes.Linear method*), 355

call() (*tensorcircuit.keras.QuantumLayer method*), 625

cals_from_api() (*tensorcircuit.results.readout_mitigation.ReadoutMit method*), 698

cals_from_system() (*tensorcircuit.results.readout_mitigation.ReadoutMit method*), 698

canonicalize() (*tensorcircuit.mps_base.FiniteMPS method*), 636

cast() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 367

cast() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 396

cast() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 423

cast() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 451

ctrl() (*tensorcircuit.circuit.Circuit method*), 508

ccnot() (*tensorcircuit.densitymatrix.DMCircuit method*), 548

ccnot() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 579

ccnot() (*tensorcircuit.mpscircuit.MPSCircuit method*), 648

ccx() (*tensorcircuit.circuit.Circuit method*), 508

ccx() (*tensorcircuit.densitymatrix.DMCircuit method*), 548

ccx() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 579

ccx() (*tensorcircuit.mpscircuit.MPSCircuit method*), 648

center_position (*tensorcircuit.mps_base.FiniteMPS attribute*), 636

check_canonical() (*tensorcircuit.mps_base.FiniteMPS method*), 636

check_density_matrix() (*tensorcircuit.densitymatrix.DMCircuit static method*), 548

check_density_matrix() (*tensorcircuit.densitymatrix.DMCircuit2 static method*), 579

check_kraus() (*tensorcircuit.densitymatrix.DMCircuit static method*), 548

check_kraus() (*tensorcircuit.densitymatrix.DMCircuit2 static method*), 579

check_network() (*tensorcircuit.quantum.QuAdjointVector method*), 670

check_network() (*tensorcircuit.quantum.QuOperator method*), 674

check_network() (*tensorcircuit.quantum.QuScalar method*), 677

check_network() (*tensorcircuit.quantum.QuVector method*), 680

check_orthonormality() (*tensorcircuit.mps_base.FiniteMPS method*), 636

check_rep_transformation() (*in module tensorcircuit.channels*), 493

check_spaces() (*in module tensorcircuit.quantum*), 683

children() (*tensorcircuit.torchnn.QuantumNet method*), 711

choi_to_kraus() (*in module tensorcircuit.channels*), 493

choi_to_super() (*in module tensorcircuit.channels*), 493

cholesky() (*tensorcircuit*)

<code>cuit.backends.jax_backend.JaxBackend method), 368</code>	<code>cirqcnotgate()</code> (in module <i>cuit.applications.layers</i>), 213	<code>tensorcir-</code>
<code>cholesky()</code> (in module <i>cuit.backends.numpy_backend.NumpyBackend method</i>), 396	<code>cirqcnotlayer()</code> (in module <i>cuit.applications.layers</i>), 213	<code>tensorcir-</code>
<code>cholesky()</code> (in module <i>cuit.backends.pytorch_backend.PyTorchBackend method</i>), 423	<code>cirqHlayer()</code> (in module <i>cuit.applications.layers</i>), 211	<code>tensorcir-</code>
<code>cholesky()</code> (in module <i>cuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 452	<code>cirqrxlayer()</code> (in module <i>cuit.applications.layers</i>), 213	<code>tensorcir-</code>
<code>Circuit (class in tensorcircuit.circuit)</code> , 499	<code>cirqrylayer()</code> (in module <i>cuit.applications.layers</i>), 213	<code>tensorcir-</code>
<code>circuit_param</code> (in module <i>cuit.abstractcircuit.AbstractCircuit attribute</i>), 196	<code>cirqrzlayer()</code> (in module <i>cuit.applications.layers</i>), 213	<code>tensorcir-</code>
<code>circuit_param</code> (in module <i>cuit.basecircuit.BaseCircuit attribute</i>), 479	<code>cirqswapgate()</code> (in module <i>cuit.applications.layers</i>), 213	<code>tensorcir-</code>
<code>circuit_param</code> (in module <i>cuit.circuit.Circuit attribute</i>), 509	<code>cirqswaplayer()</code> (in module <i>cuit.applications.layers</i>), 213	<code>tensorcir-</code>
<code>circuit_param</code> (in module <i>cuit.densitymatrix.DMCircuit attribute</i>), 548	<code>cirqxxgate()</code> (in module <i>cuit.applications.layers</i>), 213	<code>tensorcir-</code>
<code>circuit_param</code> (in module <i>cuit.densitymatrix.DMCircuit2 attribute</i>), 579	<code>cirqxxlayer()</code> (in module <i>cuit.applications.layers</i>), 213	<code>tensorcir-</code>
<code>circuit_param</code> (in module <i>cuit.mpscircuit.MPSCircuit attribute</i>), 648	<code>cirqxzgate()</code> (in module <i>cuit.applications.layers</i>), 214	<code>tensorcir-</code>
<code>circuit_with_noise()</code> (in module <i>cuit.noisemodel</i>), 667	<code>cirqxzlayer()</code> (in module <i>cuit.applications.layers</i>), 214	<code>tensorcir-</code>
<code>cirqanyrxlayer()</code> (in module <i>cuit.applications.layers</i>), 211	<code>cirqyxgate()</code> (in module <i>cuit.applications.layers</i>), 214	<code>tensorcir-</code>
<code>cirqanyrylayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>cirqyxlayer()</code> (in module <i>cuit.applications.layers</i>), 214	<code>tensorcir-</code>
<code>cirqanyrzlayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>cirqyygate()</code> (in module <i>cuit.applications.layers</i>), 214	<code>tensorcir-</code>
<code>cirqanyswaplayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>cirqyylayer()</code> (in module <i>cuit.applications.layers</i>), 214	<code>tensorcir-</code>
<code>cirqanyxxlayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>cirqyzgate()</code> (in module <i>cuit.applications.layers</i>), 214	<code>tensorcir-</code>
<code>cirqanyxylayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>cirqyzlayer()</code> (in module <i>cuit.applications.layers</i>), 214	<code>tensorcir-</code>
<code>cirqanyxzlayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>cirqzxgate()</code> (in module <i>cuit.applications.layers</i>), 214	<code>tensorcir-</code>
<code>cirqanyyxlayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>cirqzxlayer()</code> (in module <i>cuit.applications.layers</i>), 214	<code>tensorcir-</code>
<code>cirqanyyylayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>cirqzygate()</code> (in module <i>cuit.applications.layers</i>), 214	<code>tensorcir-</code>
<code>cirqanyyzlayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>cirqzylayer()</code> (in module <i>cuit.applications.layers</i>), 215	<code>tensorcir-</code>
<code>cirqanyzxlayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>cirqzzgate()</code> (in module <i>cuit.applications.layers</i>), 215	<code>tensorcir-</code>
<code>cirqanyzylayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>cirqzzlayer()</code> (in module <i>cuit.applications.layers</i>), 215	<code>tensorcir-</code>
<code>cirqanyzzlayer()</code> (in module <i>cuit.applications.layers</i>), 212	<code>clear()</code> (<i>tensorcircuit.channels.KrausList method</i>), 492	
	<code>CNOT()</code> (<i>tensorcircuit.circuit.Circuit method</i>), 499	

cnot() (*tensorcircuit.circuit.Circuit* method), 509
 CNOT() (*tensorcircuit.densitymatrix.DMCircuit* method), 539
 cnot() (*tensorcircuit.densitymatrix.DMCircuit* method), 549
 CNOT() (*tensorcircuit.densitymatrix.DMCircuit2* method), 570
 cnot() (*tensorcircuit.densitymatrix.DMCircuit2* method), 580
 CNOT() (*tensorcircuit.mpscircuit.MPSCircuit* method), 639
 cnot() (*tensorcircuit.mpscircuit.MPSCircuit* method), 648
 color_svg() (in module *tensorcircuit.applications.utils*), 220
 coloring_copied_nodes() (*tensorcircuit.basecircuit.BaseCircuit* static method), 479
 coloring_copied_nodes() (*tensorcircuit.circuit.Circuit* static method), 509
 coloring_copied_nodes() (*tensorcircuit.densitymatrix.DMCircuit* static method), 549
 coloring_copied_nodes() (*tensorcircuit.densitymatrix.DMCircuit2* static method), 580
 coloring_nodes() (*tensorcircuit.basecircuit.BaseCircuit* static method), 479
 coloring_nodes() (*tensorcircuit.circuit.Circuit* static method), 509
 coloring_nodes() (*tensorcircuit.densitymatrix.DMCircuit* static method), 549
 coloring_nodes() (*tensorcircuit.densitymatrix.DMCircuit2* static method), 580
 compile() (*tensorcircuit.applications.van.MADE* method), 230
 compile() (*tensorcircuit.applications.van.NMF* method), 283
 compile() (*tensorcircuit.applications.van.PixelCNN* method), 313
 compose_tc_circuit_with_multiple_pools() (in module *tensorcircuit.applications.vags*), 221
 composedkraus() (in module *tensorcircuit.channels*), 494
 compute_dtype (*tensorcircuit.applications.van.MADE* property), 231
 compute_dtype (*tensorcircuit.applications.van.MaskedConv2D* property), 260
 compute_dtype (*tensorcircuit.applications.van.MaskedLinear* property), 272
 compute_dtype (*tensorcircuit.applications.van.NMF* property), 285
 compute_dtype (*tensorcircuit.applications.van.PixelCNN* property), 314
 compute_dtype (*tensorcircuit.applications.van.ResidualBlock* property), 343
 compute_dtype (*tensorcircuit.applications.vqes.Linear* property), 355
 compute_dtype (*tensorcircuit.keras.QuantumLayer* property), 625
 compute_loss() (*tensorcircuit.applications.van.MADE* method), 231
 compute_loss() (*tensorcircuit.applications.van.NMF* method), 285
 compute_loss() (*tensorcircuit.applications.van.PixelCNN* method), 315
 compute_mask() (*tensorcircuit.applications.van.MADE* method), 232
 compute_mask() (*tensorcircuit.applications.van.MaskedConv2D* method), 260
 compute_mask() (*tensorcircuit.applications.van.MaskedLinear* method), 272
 compute_mask() (*tensorcircuit.applications.van.NMF* method), 286
 compute_mask() (*tensorcircuit.applications.van.PixelCNN* method), 315
 compute_mask() (*tensorcircuit.applications.van.ResidualBlock* method), 343
 compute_mask() (*tensorcircuit.applications.vqes.Linear* method), 356
 compute_mask() (*tensorcircuit.keras.QuantumLayer* method), 625
 compute_metrics() (*tensorcircuit.applications.van.MADE* method), 232
 compute_metrics() (*tensorcircuit.applications.van.NMF* method), 286
 compute_metrics() (*tensorcircuit.applications.van.PixelCNN* method), 315
 compute_output_shape() (*tensorcircuit.applications.van.MADE* method), 233
 compute_output_shape() (*tensorcircuit.applications.van.MaskedConv2D* method), 260
 compute_output_shape() (*tensorcircuit.applications.van.MaskedLinear* method),

272	cond_measure()	(<i>tensorcircuit.circuit.Circuit</i> method), 509
compute_output_shape()	(<i>tensorcircuit.applications.van.NMF</i> method), 286	
compute_output_shape()	(<i>tensorcircuit.applications.van.PixelCNN</i> method), 316	
compute_output_shape()	(<i>tensorcircuit.applications.van.ResidualBlock</i> method), 344	
compute_output_shape()	(<i>tensorcircuit.applications.vqes.Linear</i> method), 356	
compute_output_shape()	(<i>tensorcircuit.keras.QuantumLayer</i> method), 625	
compute_output_signature()	(<i>tensorcircuit.applications.van.MADE</i> method), 233	
compute_output_signature()	(<i>tensorcircuit.applications.van.MaskedConv2D</i> method), 260	
compute_output_signature()	(<i>tensorcircuit.applications.van.MaskedLinear</i> method), 272	
compute_output_signature()	(<i>tensorcircuit.applications.van.NMF</i> method), 286	
compute_output_signature()	(<i>tensorcircuit.applications.van.PixelCNN</i> method), 316	
compute_output_signature()	(<i>tensorcircuit.applications.van.ResidualBlock</i> method), 344	
compute_output_signature()	(<i>tensorcircuit.applications.vqes.Linear</i> method), 356	
compute_output_signature()	(<i>tensorcircuit.keras.QuantumLayer</i> method), 625	
concat()	(<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 368	
concat()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 396	
concat()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 423	
concat()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 452	
cond()	(<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 368	
cond()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 396	
cond()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 423	
cond()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 452	
cond_measure()	(<i>tensorcircuit.abstractcircuit.AbstractCircuit</i> method), 196	
cond_measure()	(<i>tensorcircuit.basecircuit.BaseCircuit</i> method), 479	
	cond_measure()	(<i>tensorcircuit.densitymatrix.DMCircuit</i> method), 549
	cond_measure()	(<i>tensorcircuit.densitymatrix.DMCircuit2</i> method), 580
	cond_measure()	(<i>tensorcircuit.mpscircuits.MPSCircuit</i> method), 649
	cond_measurement()	(<i>tensorcircuit.abstractcircuit.AbstractCircuit</i> method), 197
	cond_measurement()	(<i>tensorcircuit.basecircuit.BaseCircuit</i> method), 480
	cond_measurement()	(<i>tensorcircuit.circuit.Circuit</i> method), 509
	cond_measurement()	(<i>tensorcircuit.densitymatrix.DMCircuit</i> method), 549
	cond_measurement()	(<i>tensorcircuit.densitymatrix.DMCircuit2</i> method), 580
	cond_measurement()	(<i>tensorcircuit.mpscircuits.MPSCircuit</i> method), 649
	conditional_gate()	(<i>tensorcircuit.abstractcircuit.AbstractCircuit</i> method), 197
	conditional_gate()	(<i>tensorcircuit.basecircuit.BaseCircuit</i> method), 480
	conditional_gate()	(<i>tensorcircuit.circuit.Circuit</i> method), 510
	conditional_gate()	(<i>tensorcircuit.densitymatrix.DMCircuit</i> method), 550
	conditional_gate()	(<i>tensorcircuit.densitymatrix.DMCircuit2</i> method), 581
	conditional_gate()	(<i>tensorcircuit.mpscircuits.MPSCircuit</i> method), 649
	conditional_gate()	(<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 368
	conditional_gate()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 396
	conj()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 424
	conj()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 452
	conj()	(<i>tensorcircuit.mps_base.FiniteMPS</i> method), 637
	conj()	(<i>tensorcircuit.mpscircuits.MPSCircuit</i> method), 650
	consecutive_swap()	(<i>tensorcircuit.mpscircuits.MPSCircuit</i> method), 650
	construct_matrix()	(in module <i>tensorcircuit.applications.vqes</i>), 365
	construct_matrix_tf()	(in module <i>tensorcircuit.applications.vqes</i>), 365

```

construct_matrix_v2() (in module tensorcircuit.applications.vqes), 365
construct_matrix_v3() (in module tensorcircuit.applications.vqes), 365
contract() (tensorcircuit.quantum.QuAdjointVector method), 670
contract() (tensorcircuit.quantum.QuOperator method), 674
contract() (tensorcircuit.quantum.QuScalar method), 677
contract() (tensorcircuit.quantum.QuVector method), 680
contraction_info_decorator() (in module tensorcircuit.cons), 535
contractor() (in module tensorcircuit.cons), 535
controlled() (tensorcircuit.gates.GateF method), 605
controlled() (tensorcircuit.gates.GateVF method), 605
convert_to_tensor() (tensorcircuit.backends.jax_backend.JaxBackend method), 368
convert_to_tensor() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 396
convert_to_tensor() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 424
convert_to_tensor() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 452
coo_sparse_matrix() (tensorcircuit.backends.jax_backend.JaxBackend method), 368
coo_sparse_matrix() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 396
coo_sparse_matrix() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 424
coo_sparse_matrix() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 452
coo_sparse_matrix_from_numpy() (tensorcircuit.backends.jax_backend.JaxBackend method), 368
coo_sparse_matrix_from_numpy() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 397
coo_sparse_matrix_from_numpy() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 424
coo_sparse_matrix_from_numpy() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 452
copy() (tensorcircuit.backends.jax_backend.JaxBackend
       method), 368
copy() (tensorcircuit.backends.numpy_backend.NumpyBackend
       method), 397
copy() (tensorcircuit.backends.pytorch_backend.PyTorchBackend
       method), 424
copy() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend
       method), 452
copy() (tensorcircuit.basecircuit.BaseCircuit static method), 480
copy() (tensorcircuit.channels.KrausList method), 492
copy() (tensorcircuit.circuit.Circuit static method), 510
copy() (tensorcircuit.densitymatrix.DMCircuit static method), 550
copy() (tensorcircuit.densitymatrix.DMCircuit2 static method), 581
copy() (tensorcircuit.gates.Gate method), 603
copy() (tensorcircuit.mps_base.FiniteMPS method), 637
copy() (tensorcircuit.mpscircuit.MPSCircuit method), 650
copy() (tensorcircuit.quantum.QuAdjointVector method), 671
copy() (tensorcircuit.quantum.QuOperator method), 675
copy() (tensorcircuit.quantum.QuScalar method), 678
copy() (tensorcircuit.quantum.QuVector method), 681
copy_without_tensor() (tensorcircuit.mpscircuit.MPSCircuit method), 650
correlation() (in module tensorcircuit.applications.vags), 221
correlation_from_counts() (in module tensorcircuit.quantum), 684
correlation_from_samples() (in module tensorcircuit.quantum), 684
cos() (tensorcircuit.backends.jax_backend.JaxBackend method), 369
cos() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 397
cos() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 424
cosd() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 453
cosh() (tensorcircuit.backends.jax_backend.JaxBackend method), 369
cosh() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 397
cosh() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 424
cosh() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 453
count() (tensorcircuit.channels.KrausList method), 492
count2vec() (in module tensorcircuit.results.counts), 697
count_d2s() (in module tensorcircuit.quantum), 684
count_params() (tensorcircuit.applications.van.MADE
    
```

<i>method)</i> , 233		
count_params() (tensorcircuit.applications.van.MaskedConv2D method), 261	(tensorcircuit.applications.van.MaskedConv2D method), 261	(tensorcircuit.applications.van.VQNHE method), 364
count_params() (tensorcircuit.applications.van.MaskedLinear method), 272	(tensorcircuit.applications.van.MaskedLinear method), 272	(tensorcircuit.applications.van.VQNHE method), 364
count_params() (tensorcircuit.applications.van.NMF method), 287	(tensorcircuit.applications.van.NMF method), 287	(tensorcircuit.applications.van.VQNHE method), 364
count_params() (tensorcircuit.applications.van.PixelCNN method), 316	(tensorcircuit.applications.van.PixelCNN method), 316	(tensorcircuit.applications.van.VQNHE method), 364
count_params() (tensorcircuit.applications.van.ResidualBlock method), 344	(tensorcircuit.applications.van.ResidualBlock method), 344	(tensorcircuit.applications.van.VQNHE method), 364
count_params() (tensorcircuit.applications.vqes.Linear method), 356	(tensorcircuit.applications.vqes.Linear method), 356	(tensorcircuit.applications.vqes.VQNHE method), 364
count_params() (tensorcircuit.keras.QuantumLayer method), 626	(tensorcircuit.keras.QuantumLayer method), 626	(tensorcircuit.applications.vqes.VQNHE method), 364
count_s2d() (in module tensorcircuit.quantum), 685		
count_t2v() (in module tensorcircuit.quantum), 685		
count_tuple2dict() (in module tensorcircuit.quantum), 685	(in module tensorcircuit.quantum), 685	(in module tensorcircuit.quantum), 685
count_vector2dict() (in module tensorcircuit.quantum), 685	(in module tensorcircuit.quantum), 685	(in module tensorcircuit.quantum), 685
counts_v2t() (in module tensorcircuit.quantum), 686		
CPHASE() (tensorcircuit.circuit.Circuit method), 499		
cphase() (tensorcircuit.circuit.Circuit method), 510		
CPHASE() (tensorcircuit.densitymatrix.DMCircuit method), 539	(tensorcircuit.densitymatrix.DMCircuit method), 539	(tensorcircuit.densitymatrix.DMCircuit method), 539
cphase() (tensorcircuit.densitymatrix.DMCircuit method), 550	(tensorcircuit.densitymatrix.DMCircuit method), 550	(tensorcircuit.densitymatrix.DMCircuit method), 550
CPHASE() (tensorcircuit.densitymatrix.DMCircuit2 method), 570	(tensorcircuit.densitymatrix.DMCircuit2 method), 570	(tensorcircuit.densitymatrix.DMCircuit2 method), 570
cphase() (tensorcircuit.densitymatrix.DMCircuit2 method), 581	(tensorcircuit.densitymatrix.DMCircuit2 method), 581	(tensorcircuit.densitymatrix.DMCircuit2 method), 581
CPHASE() (tensorcircuit.mpscircuit.MPSCircuit method), 639		
cphase() (tensorcircuit.mpscircuit.MPSCircuit method), 650		
cpu() (tensorcircuit.torchnn.QuantumNet method), 711		
CR() (tensorcircuit.circuit.Circuit method), 500		
cr() (tensorcircuit.circuit.Circuit method), 510		
CR() (tensorcircuit.densitymatrix.DMCircuit method), 539	(tensorcircuit.densitymatrix.DMCircuit method), 539	(tensorcircuit.densitymatrix.DMCircuit method), 539
cr() (tensorcircuit.densitymatrix.DMCircuit method), 550	(tensorcircuit.densitymatrix.DMCircuit method), 550	(tensorcircuit.densitymatrix.DMCircuit method), 550
CR() (tensorcircuit.densitymatrix.DMCircuit2 method), 570	(tensorcircuit.densitymatrix.DMCircuit2 method), 570	(tensorcircuit.densitymatrix.DMCircuit2 method), 570
cr() (tensorcircuit.densitymatrix.DMCircuit2 method), 581	(tensorcircuit.densitymatrix.DMCircuit2 method), 581	(tensorcircuit.densitymatrix.DMCircuit2 method), 581
CR() (tensorcircuit.mpscircuit.MPSCircuit method), 639		
cr() (tensorcircuit.mpscircuit.MPSCircuit method), 650		
cr_gate() (in module tensorcircuit.gates), 606		
create_circuit() (tensorcircuit.applications.vqes.VQNHE method), 364		
create_complex_model() (tensorcircuit.applications.vqes.VQNHE method), 364		
create_complex_rbm_model() (tensorcircuit.applications.vqes.VQNHE method), 364		
create_hea2_circuit() (tensorcircuit.applications.vqes.VQNHE method), 364		
create_hea_circuit() (tensorcircuit.applications.vqes.VQNHE method), 364		
create_hn_circuit() (tensorcircuit.applications.vqes.VQNHE method), 364		
create_model() (tensorcircuit.applications.vqes.VQNHE method), 364		
create_real_model() (tensorcircuit.applications.vqes.VQNHE method), 364		
create_real_rbm_model() (tensorcircuit.applications.vqes.VQNHE method), 364		
CRX() (tensorcircuit.circuit.Circuit method), 500		
crx() (tensorcircuit.circuit.Circuit method), 510		
CRX() (tensorcircuit.densitymatrix.DMCircuit method), 540		
crx() (tensorcircuit.densitymatrix.DMCircuit method), 550		
CRX() (tensorcircuit.densitymatrix.DMCircuit2 method), 571		
crx() (tensorcircuit.densitymatrix.DMCircuit2 method), 581		
CRX() (tensorcircuit.mpscircuit.MPSCircuit method), 639		
crx() (tensorcircuit.mpscircuit.MPSCircuit method), 650		
CRY() (tensorcircuit.circuit.Circuit method), 500		
cry() (tensorcircuit.circuit.Circuit method), 511		
CRY() (tensorcircuit.densitymatrix.DMCircuit method), 540		
cry() (tensorcircuit.densitymatrix.DMCircuit method), 550		
CRY() (tensorcircuit.densitymatrix.DMCircuit2 method), 571		
cry() (tensorcircuit.densitymatrix.DMCircuit2 method), 581		
CRY() (tensorcircuit.mpscircuit.MPSCircuit method), 639		
cry() (tensorcircuit.mpscircuit.MPSCircuit method), 650		

650
CRZ() (*tensorcircuit.circuit.Circuit* method), 500
crz() (*tensorcircuit.circuit.Circuit* method), 511
CRZ() (*tensorcircuit.densitymatrix.DMCircuit* method),
 540
crz() (*tensorcircuit.densitymatrix.DMCircuit* method),
 550
CRZ() (*tensorcircuit.densitymatrix.DMCircuit2* method),
 571
crz() (*tensorcircuit.densitymatrix.DMCircuit2* method),
 581
CRZ() (*tensorcircuit.mpscircuit.MPSCircuit* method),
 640
crz() (*tensorcircuit.mpscircuit.MPSCircuit* method),
 650
cswap() (*tensorcircuit.circuit.Circuit* method), 511
cswap() (*tensorcircuit.densitymatrix.DMCircuit*
 method), 551
cswap() (*tensorcircuit.densitymatrix.DMCircuit2*
 method), 582
cswap() (*tensorcircuit.mpscircuit.MPSCircuit* method),
 651
ctrl_str2ctrl_state() (in module *tensorcircuit.translation*), 723
CU() (*tensorcircuit.circuit.Circuit* method), 500
cu() (*tensorcircuit.circuit.Circuit* method), 511
CU() (*tensorcircuit.densitymatrix.DMCircuit* method),
 540
cu() (*tensorcircuit.densitymatrix.DMCircuit* method),
 551
CU() (*tensorcircuit.densitymatrix.DMCircuit2* method),
 571
cu() (*tensorcircuit.densitymatrix.DMCircuit2* method),
 582
CU() (*tensorcircuit.mpscircuit.MPSCircuit* method), 640
cu() (*tensorcircuit.mpscircuit.MPSCircuit* method), 651
cuda() (*tensorcircuit.torchnn.QuantumNet* method), 712
cumsum() (*tensorcircuit.backends.jax_backend.JaxBackend*
 method), 369
cumsum() (*tensorcircuit.backends.numpy_backend.NumpyBackend*
 method), 397
cumsum() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*
 method), 424
cumsum() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*
 method), 453
custom() (in module *tensorcircuit.cons*), 535
custom_stateful() (in module *tensorcircuit.cons*), 535
cvar() (in module *tensorcircuit.applications.vags*), 221
cx() (*tensorcircuit.circuit.Circuit* method), 511
cx() (*tensorcircuit.densitymatrix.DMCircuit* method),
 551
cx() (*tensorcircuit.densitymatrix.DMCircuit2* method),
 582
cx() (*tensorcircuit.mpscircuit.MPSCircuit* method), 651
CY() (*tensorcircuit.circuit.Circuit* method), 500
cy() (*tensorcircuit.circuit.Circuit* method), 511
CY() (*tensorcircuit.densitymatrix.DMCircuit* method),
 540
cy() (*tensorcircuit.densitymatrix.DMCircuit* method),
 551
CY() (*tensorcircuit.densitymatrix.DMCircuit2* method),
 571
cy() (*tensorcircuit.densitymatrix.DMCircuit2* method),
 582
CY() (*tensorcircuit.mpscircuit.MPSCircuit* method), 640
cy() (*tensorcircuit.mpscircuit.MPSCircuit* method), 651
CZ() (*tensorcircuit.circuit.Circuit* method), 500
cz() (*tensorcircuit.circuit.Circuit* method), 512
CZ() (*tensorcircuit.densitymatrix.DMCircuit* method),
 540
cz() (*tensorcircuit.densitymatrix.DMCircuit* method),
 551
CZ() (*tensorcircuit.densitymatrix.DMCircuit2* method),
 571
cz() (*tensorcircuit.densitymatrix.DMCircuit2* method),
 582
CZ() (*tensorcircuit.mpscircuit.MPSCircuit* method), 640
cz() (*tensorcircuit.mpscircuit.MPSCircuit* method), 651

D

d2s() (in module *tensorcircuit.cons*), 535
densitymatrix() (*tensorcircuit.densitymatrix.DMCircuit* method), 552
densitymatrix() (*tensorcircuit.densitymatrix.DMCircuit2* method),
 582
depolarizing() (*tensorcircuit.circuit.Circuit* method),
 512
depolarizing() (*tensorcircuit.densitymatrix.DMCircuit* method), 552
depolarizing() (*tensorcircuit.densitymatrix.DMCircuit2* method),
 583
depolarizing2() (*tensorcircuit.circuit.Circuit* method), 512
depolarizing_reference() (*tensorcircuit.circuit.Circuit* method), 512
depolarizing_channel() (in module *tensorcircuit.channels*), 494
deserialize_tensor() (*tensorcircuit.backends.jax_backend.JaxBackend*
 method), 369
deserialize_tensor() (*tensorcircuit.backends.numpy_backend.NumpyBackend*
 method), 397
deserialize_tensor() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*
 method), 425

<code>deserialize_tensor()</code>	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 453	<code>distribute_reduction_method</code>	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend distribute_reduction_method</i>), 317	<code>(tensorcircuit.property)</code> , 317
<code>device()</code>	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 369	<code>distribute_strategy</code>	(<i>tensorcircuit.backends.jax_backend.JaxBackend distribute_strategy</i>), 233	<code>(tensorcircuit.property)</code> , 233
<code>device()</code>	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 397	<code>distribute_strategy</code>	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend distribute_strategy</i>), 287	<code>(tensorcircuit.property)</code> , 287
<code>device()</code>	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 425	<code>distribute_strategy</code>	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend distribute_strategy</i>), 287	<code>(tensorcircuit.property)</code> , 287
<code>device()</code>	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 453	<code>divide()</code>	(<i>tensorcircuit.backends.jax_backend.JaxBackend divide()</i>), 370	
<code>device_move()</code>	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 369	<code>divide()</code>	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend divide()</i>), 398	
<code>device_move()</code>	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 398	<code>divide()</code>	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend divide()</i>), 426	
<code>device_move()</code>	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 425	<code>divide()</code>	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend divide()</i>), 454	
<code>device_move()</code>	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 453	<code>DMCircuit</code>	(class in <i>tensorcircuit.densitymatrix</i>), 539	
<code>diagflat()</code>	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 369	<code>DMCircuit2</code>	(class in <i>tensorcircuit.densitymatrix</i>), 570	
<code>diagflat()</code>	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 398	<code>double()</code>	(<i>tensorcircuit.torchnn.QuantumNet double()</i>), 712	
<code>diagflat()</code>	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 425	<code>double_qubits_initial()</code>	(in module <i>tensorcircuit.applications.vags</i>), 222	
<code>diagflat()</code>	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 453	<code>double_state()</code>	(in module <i>tensorcircuit.applications.vags</i>), 222	
<code>diagflat()</code>	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 369	<code>double_state()</code>	(in module <i>tensorcircuit.quantum</i>), 686	
<code>diagflat()</code>	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 398	<code>DQAS_search()</code>	(in module <i>tensorcircuit.dqas</i>), 204	
<code>diagflat()</code>	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 425	<code>DQAS_search_pmb()</code>	(in module <i>tensorcircuit.dqas</i>), 205	
<code>diagflat()</code>	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 453	<code>draw()</code>	(<i>tensorcircuit.abstractcircuit.AbstractCircuit draw()</i>), 197	
<code>diagonal()</code>	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 370	<code>draw()</code>	(<i>tensorcircuit.basecircuits.BaseCircuit draw()</i>), 481	
<code>diagonal()</code>	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 398	<code>draw()</code>	(<i>tensorcircuit.circuit.Circuit draw()</i>), 512	
<code>diagonal()</code>	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 425	<code>draw()</code>	(<i>tensorcircuit.densitymatrix.DMCircuit draw()</i>), 552	
<code>diagonal()</code>	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 453	<code>draw()</code>	(<i>tensorcircuit.densitymatrix.DMCircuit2 draw()</i>), 583	
<code>dict2graph()</code>	(in module <i>tensorcircuit.applications.graphdata</i>), 208	<code>draw()</code>	(<i>tensorcircuit.mpscircuits.MPSCircuit draw()</i>), 651	
<code>disable()</code>	(<i>tensorcircuit.gates.Gate disable()</i>), 603	<code>dtype</code>	(<i>tensorcircuit.applications.van.MADE dtype</i>), 233	
<code>distribute_reduction_method</code>	(<i>tensorcircuit.applications.van.MADE property</i>), 233	<code>dtype</code>	(<i>tensorcircuit.applications.van.MaskedConv2D property</i>), 261	
<code>distribute_reduction_method</code>	(<i>tensorcircuit.applications.van.NMF property</i>), 287	<code>dtype</code>	(<i>tensorcircuit.applications.van.MaskedLinear property</i>), 273	
<code>distribute_reduction_method</code>	(<i>tensorcircuit.applications.van.NMF property</i>), 287	<code>dtype</code>	(<i>tensorcircuit.applications.van.NMF dtype</i>), 287	
		<code>dtype</code>	(<i>tensorcircuit.applications.van.PixelCNN property</i>), 317	

dtype (*tensorcircuit.applications.van.ResidualBlock property*), 344
dtype (*tensorcircuit.applications.vqes.Linear property*), 356
dtype (*tensorcircuit.gates.Gate property*), 603
dtype (*tensorcircuit.keras.QuantumLayer property*), 626
dtype (*tensorcircuit.mps_base.FiniteMPS property*), 637
dtype() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 370
dtype() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 398
dtype() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 426
dtype() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 454
dtype_policy (*tensorcircuit.applications.van.MADE property*), 234
dtype_policy (*tensorcircuit.applications.van.MaskedConv2D property*), 261
dtype_policy (*tensorcircuit.applications.van.MaskedLinear property*), 273
dtype_policy (*tensorcircuit.applications.van.NMF property*), 287
dtype_policy (*tensorcircuit.applications.van.PixelCNN property*), 317
dtype_policy (*tensorcircuit.applications.van.ResidualBlock property*), 344
dtype_policy (*tensorcircuit.applications.vqes.Linear property*), 357
dtype_policy (*tensorcircuit.keras.QuantumLayer property*), 626
dump_patches (*tensorcircuit.torchnn.QuantumNet attribute*), 712
dynamic (*tensorcircuit.applications.van.MADE property*), 234
dynamic (*tensorcircuit.applications.van.MaskedConv2D property*), 261
dynamic (*tensorcircuit.applications.van.MaskedLinear property*), 273
dynamic (*tensorcircuit.applications.van.NMF property*), 287
dynamic (*tensorcircuit.applications.van.PixelCNN property*), 317
dynamic (*tensorcircuit.applications.van.ResidualBlock property*), 344
dynamic (*tensorcircuit.applications.vqes.Linear property*), 357
dynamic (*tensorcircuit.keras.QuantumLayer property*), 626
dynamics_matrix() (*in module tensorcircuit.experimental*), 601
dynamics_rhs() (*in module tensorcircuit.experimental*), 601

E

edges (*tensorcircuit.gates.Gate property*), 603
eigh() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 370
eigh() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 399
eigh() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 426
eigh() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 454
eigs() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 370
eigs() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 399
eigs() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 426
eigs() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 454
eigsh() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 371
eigsh() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 399
eigsh() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 427
eigsh() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 455
eigsh_lanczos() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 372
eigsh_lanczos() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 400
eigsh_lanczos() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 427
eigsh_lanczos() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 456
eigvalsh() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 373
eigvalsh() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 401
eigvalsh() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 428
eigvalsh() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 457

`einsum()` (*tensorcircuit.backends.jax_backend.JaxBackend*)
`evaluate_generator()` (*tensorcircuit.circuit.applications.van.MADE* method), 235
`method`), 374

`einsum()` (*tensorcircuit.backends.numpy_backend.NumpyBackend*)
`evaluate_generator()` (*tensorcircuit.circuit.applications.van.NMF* method), 289
`method`), 401

`einsum()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*)
`evaluate_generator()` (*tensorcircuit.circuit.applications.van.PixelCNN* method), 428
`method`), 428

`einsum()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*)
`evaluate_vag()` (in module *tensorcircuit.circuit.applications.vags*), 222

`eliminate_identities()` (in module *tensorcircuit.quantum*), 686

`energy()` (in module *tensorcircuit.applications.vags*), 222

`ensemble_maxcut_solution()` (in module *tensorcircuit.applications.graphdata*), 208

`entanglement_entropy()` (in module *tensorcircuit.applications.vags*), 222

`entropy()` (in module *tensorcircuit.applications.vags*), 222

`entropy()` (in module *tensorcircuit.quantum*), 686

`eps()` (*tensorcircuit.backends.jax_backend.JaxBackend*)
`method`), 374

`eps()` (*tensorcircuit.backends.numpy_backend.NumpyBackend*)
`method`), 401

`eps()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*)
`method`), 428

`eps()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*)
`method`), 428

`eqasm2tc()` (in module *tensorcircuit.translation*), 723

`erdos_graph_generator()` (in module *tensorcircuit.applications.graphdata*), 208

`eval()` (*tensorcircuit.quantum.QuAdjointVector*)
`method`), 671

`eval()` (*tensorcircuit.quantum.QuOperator* method), 675

`eval()` (*tensorcircuit.quantum.QuScalar* method), 678

`eval()` (*tensorcircuit.quantum.QuVector* method), 681

`eval()` (*tensorcircuit.torchnn.QuantumNet* method), 712

`eval_matrix()` (*tensorcircuit.quantum.QuAdjointVector*)
`method`), 671

`eval_matrix()` (*tensorcircuit.quantum.QuOperator* method), 675

`eval_matrix()` (*tensorcircuit.quantum.QuScalar* method), 678

`eval_matrix()` (*tensorcircuit.quantum.QuVector* method), 681

`evaluate()` (*tensorcircuit.applications.van.MADE* method), 234

`evaluate()` (*tensorcircuit.applications.van.NMF* method), 287

`evaluate()` (*tensorcircuit.applications.van.PixelCNN* method), 317

`evaluate_everyone()` (in module *tensorcircuit.applications.dqas*), 206

`evaluate_generator()` (*tensorcircuit.circuit.applications.van.MADE* method), 235
`method`), 235

`evaluate_generator()` (*tensorcircuit.circuit.applications.van.NMF* method), 289
`method`), 289

`evaluate_generator()` (*tensorcircuit.circuit.applications.van.PixelCNN* method), 222
`method`), 222

`evaluation()` (*tensorcircuit.applications.vqes.VQNHE* method), 364

`even1D()` (in module *tensorcircuit.applications.graphdata*), 208

`Even1D()` (in module *tensorcircuit.templates.graphs*), 703

`evol_kraus()` (in module *tensorcircuit.channels*), 494

`evol_superop()` (in module *tensorcircuit.channels*), 495

`example_block()` (in module *tensorcircuit.templates.blocks*), 702

`exp()` (*tensorcircuit.backends.jax_backend.JaxBackend*)
`method`), 374

`exp()` (*tensorcircuit.backends.numpy_backend.NumpyBackend*)
`method`), 401

`exp()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*)
`method`), 428

`exp()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*)
`method`), 457

`EXP()` (*tensorcircuit.circuit.Circuit* method), 501

`exp()` (*tensorcircuit.circuit.Circuit* method), 513

`EXP()` (*tensorcircuit.densitymatrix.DMCircuit* method), 540

`exp()` (*tensorcircuit.densitymatrix.DMCircuit* method), 552

`EXP()` (*tensorcircuit.densitymatrix.DMCircuit2* method), 571

`exp()` (*tensorcircuit.densitymatrix.DMCircuit2* method), 583

`EXP()` (*tensorcircuit.mpscircuits.MPSCircuit* method), 640

`exp()` (*tensorcircuit.mpscircuits.MPSCircuit* method), 652

`EXP1()` (*tensorcircuit.circuit.Circuit* method), 501

`exp1()` (*tensorcircuit.circuit.Circuit* method), 513

`EXP1()` (*tensorcircuit.densitymatrix.DMCircuit* method), 541

`exp1()` (*tensorcircuit.densitymatrix.DMCircuit* method), 552

`EXP1()` (*tensorcircuit.densitymatrix.DMCircuit2* method), 572

`exp1()` (*tensorcircuit.densitymatrix.DMCircuit2* method), 583

`EXP1()` (*tensorcircuit.mpscircuits.MPSCircuit* method), 640

`exp1()` (*tensorcircuit.mpscircuits.MPSCircuit* method), 652
`exp1_gate()` (in module *tensorcircuit.gates*), 607
`exp_forward()` (in module *tensorcircuit.applications.vags*), 222
`exp_gate()` (in module *tensorcircuit.gates*), 607
`expectation()` (in module *tensorcircuit.circuit*), 533
`expectation()` (in module *tensorcircuit.results.counts*), 697
`expectation()` (in module *tensorcircuit.abstractcircuit.AbstractCircuit* method), 197
`expectation()` (in module *tensorcircuit.basecircuits.BaseCircuit* method), 481
`expectation()` (in module *tensorcircuit.circuit.Circuit* method), 513
`expectation()` (in module *tensorcircuit.densitymatrix.DMCircuit* method), 552
`expectation()` (in module *tensorcircuit.densitymatrix.DMCircuit2* method), 583
`expectation()` (in module *tensorcircuit.mpscircuits.MPSCircuit* method), 652
`expectation()` (in module *tensorcircuit.results.readout_mitigation.ReadoutMit* method), 698
`expectation_before()` (in module *tensorcircuit.basecircuits.BaseCircuit* method), 481
`expectation_before()` (in module *tensorcircuit.circuit.Circuit* method), 514
`expectation_before()` (in module *tensorcircuit.densitymatrix.DMCircuit* method), 553
`expectation_before()` (in module *tensorcircuit.densitymatrix.DMCircuit2* method), 584
`expectation_noisy()` (in module *tensorcircuit.noisemodel*), 667
`expectation_ps()` (in module *tensorcircuit.abstractcircuit.AbstractCircuit* method), 198
`expectation_ps()` (in module *tensorcircuit.basecircuits.BaseCircuit* method), 481
`expectation_ps()` (in module *tensorcircuit.circuit.Circuit* method), 514
`expectation_ps()` (in module *tensorcircuit.densitymatrix.DMCircuit* method), 553
`expectation_ps()` (in module *tensorcircuit.densitymatrix.DMCircuit2* method), 584
`expectation_ps()` (in module *tensorcircuit.mpscircuits.MPSCircuit* method), 652
`experimental_contractor()` (in module *tensorcircuit.cons*), 535
`expm()` (in module *tensorcircuit.backends.jax_backend.JaxBackend* method), 374
`expm()` (in module *tensorcircuit.backends.numpy_backend.NumpyBackend* method), 401
`expm()` (in module *tensorcircuit.backends.pytorch_backend.PyTorchBackend* method), 428
`expm()` (in module *tensorcircuit.backends.tensorflow_backend.TensorFlowBackend* method), 457
`exponential_gate()` (in module *tensorcircuit.gates*), 607
`exponential_gate_unity()` (in module *tensorcircuit.gates*), 608
`extend()` (in module *tensorcircuit.channels.KrausList* method), 492
`extra_repr()` (in module *tensorcircuit.torchnn.QuantumNet* method), 712
`eye()` (in module *tensorcircuit.backends.jax_backend.JaxBackend* method), 374
`eye()` (in module *tensorcircuit.backends.numpy_backend.NumpyBackend* method), 401
`eye()` (in module *tensorcircuit.backends.pytorch_backend.PyTorchBackend* method), 428
`eye()` (in module *tensorcircuit.backends.tensorflow_backend.TensorFlowBackend* method), 457

F

`FakeModule` (class in *tensorcircuit.applications.utils*), 220
`fidelity()` (in module *tensorcircuit.applications.vags*), 222
`fidelity()` (in module *tensorcircuit.quantum*), 687
`finalize_state()` (in module *tensorcircuit.applications.van.MADE* method), 235
`finalize_state()` (in module *tensorcircuit.applications.van.MaskedConv2D* method), 261
`finalize_state()` (in module *tensorcircuit.applications.van.MaskedLinear* method), 273
`finalize_state()` (in module *tensorcircuit.applications.van.NMF* method), 289
`finalize_state()` (in module *tensorcircuit.applications.van.PixelCNN* method), 319
`finalize_state()` (in module *tensorcircuit.applications.van.ResidualBlock* method), 344
`finalize_state()` (in module *tensorcircuit.applications.vqes.Linear* method), 357
`finalize_state()` (in module *tensorcircuit.keras.QuantumLayer* method), 626
`FiniteMPS` (class in *tensorcircuit.mps_base*), 635
`fit()` (in module *tensorcircuit.applications.van.MADE* method), 235
`fit()` (in module *tensorcircuit.applications.van.NMF* method), 289

`fit()` (*tensorcircuit.applications.van.PixelCNN method*), 319
`fit_generator()` (*tensorcircuit.applications.van.MADE method*), 239
`fit_generator()` (*tensorcircuit.applications.van.NMF method*), 292
`fit_generator()` (*tensorcircuit.applications.van.PixelCNN method*), 322
`float()` (*tensorcircuit.torchnn.QuantumNet method*), 712
`forward()` (*tensorcircuit.torchnn.QuantumNet method*), 713
`FREDKIN()` (*tensorcircuit.circuit.Circuit method*), 501
`fredkin()` (*tensorcircuit.circuit.Circuit method*), 515
`FREDKIN()` (*tensorcircuit.densitymatrix.DMCircuit method*), 541
`fredkin()` (*tensorcircuit.densitymatrix.DMCircuit method*), 554
`FREDKIN()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 572
`fredkin()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 585
`FREDKIN()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 640
`fredkin()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 653
`free_energy()` (in module *tensorcircuit.applications.vags*), 222
`free_energy()` (in module *tensorcircuit.quantum*), 687
`fresh_edges()` (*tensorcircuit.gates.Gate method*), 603
`from_config()` (*tensorcircuit.applications.van.MADE class method*), 239
`from_config()` (*tensorcircuit.applications.van.MaskedConv2D class method*), 261
`from_config()` (*tensorcircuit.applications.van.MaskedLinear class method*), 273
`from_config()` (*tensorcircuit.applications.van.NMF class method*), 292
`from_config()` (*tensorcircuit.applications.van.PixelCNN class method*), 322
`from_config()` (*tensorcircuit.applications.van.ResidualBlock class method*), 344
`from_config()` (*tensorcircuit.applications.vqes.JointSchedule class method*), 352
`from_config()` (*tensorcircuit.applications.vqes.Linear class method*), 357
`from_config()` (*tensorcircuit.keras.QuantumLayer class method*), 626
`from_dlpack()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 374
`from_dlpack()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 401
`from_dlpack()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 429
`from_dlpack()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 457
`from_json()` (*tensorcircuit.abstractcircuit.AbstractCircuit class method*), 198
`from_json()` (*tensorcircuit.basecircuit.BaseCircuit class method*), 482
`from_json()` (*tensorcircuit.circuit.Circuit class method*), 515
`from_json()` (*tensorcircuit.densitymatrix.DMCircuit class method*), 554
`from_json()` (*tensorcircuit.densitymatrix.DMCircuit2 class method*), 585
`from_json()` (*tensorcircuit.mpscircuit.MPSCircuit class method*), 654
`from_json_file()` (*tensorcircuit.abstractcircuit.AbstractCircuit class method*), 199
`from_json_file()` (*tensorcircuit.basecircuit.BaseCircuit class method*), 482
`from_json_file()` (*tensorcircuit.circuit.Circuit class method*), 515
`from_json_file()` (*tensorcircuit.densitymatrix.DMCircuit class method*), 554
`from_json_file()` (*tensorcircuit.densitymatrix.DMCircuit2 class method*), 585
`from_json_file()` (*tensorcircuit.mpscircuit.MPSCircuit class method*), 654
`from_local_tensor()` (*tensorcircuit.quantum.QuAdjointVector class method*), 671
`from_local_tensor()` (*tensorcircuit.quantum.QuOperator class method*), 675
`from_local_tensor()` (*tensorcircuit.quantum.QuScalar class method*), 678
`from_local_tensor()` (*tensorcircuit.quantum.QuVector class method*), 681
`from_openqasm()` (*tensorcircuit.abstractcircuit.AbstractCircuit class*

method), 199

from_openqasm() (*tensorcircuit.basecircuit.BaseCircuit class method*), 483

from_openqasm() (*tensorcircuit.circuit.Circuit class method*), 516

from_openqasm() (*tensorcircuit.densitymatrix.DMCircuit class method*), 555

from_openqasm() (*tensorcircuit.densitymatrix.DMCircuit2 class method*), 586

from_openqasm() (*tensorcircuit.mpscircuit.MPSCircuit class method*), 654

from_openqasm_file() (*tensorcircuit.abstractcircuit.AbstractCircuit class method*), 199

from_openqasm_file() (*tensorcircuit.basecircuit.BaseCircuit class method*), 483

from_openqasm_file() (*tensorcircuit.circuit.Circuit class method*), 516

from_openqasm_file() (*tensorcircuit.densitymatrix.DMCircuit class method*), 555

from_openqasm_file() (*tensorcircuit.densitymatrix.DMCircuit2 class method*), 586

from_openqasm_file() (*tensorcircuit.mpscircuit.MPSCircuit class method*), 654

from_qir() (*tensorcircuit.abstractcircuit.AbstractCircuit class method*), 199

from_qir() (*tensorcircuit.basecircuit.BaseCircuit class method*), 483

from_qir() (*tensorcircuit.circuit.Circuit class method*), 516

from_qir() (*tensorcircuit.densitymatrix.DMCircuit class method*), 555

from_qir() (*tensorcircuit.densitymatrix.DMCircuit2 class method*), 586

from_qir() (*tensorcircuit.mpscircuit.MPSCircuit class method*), 654

from_qiskit() (*tensorcircuit.abstractcircuit.AbstractCircuit class method*), 200

from_qiskit() (*tensorcircuit.basecircuit.BaseCircuit class method*), 483

from_qiskit() (*tensorcircuit.circuit.Circuit class method*), 516

from_qiskit() (*tensorcircuit.densitymatrix.DMCircuit class method*), 555

from_qiskit() (*tensorcircuit.densitymatrix.DMCircuit2 class method*), 586

cuit.densitymatrix.DMCircuit2 class method), 586

from_qiskit() (*tensorcircuit.mpscircuit.MPSCircuit class method*), 655

from_qsim_file() (*tensorcircuit.abstractcircuit.AbstractCircuit class method*), 200

from_qsim_file() (*tensorcircuit.basecircuit.BaseCircuit class method*), 484

from_qsim_file() (*tensorcircuit.circuit.Circuit class method*), 517

from_qsim_file() (*tensorcircuit.densitymatrix.DMCircuit class method*), 556

from_qsim_file() (*tensorcircuit.densitymatrix.DMCircuit2 class method*), 587

from_qsim_file() (*tensorcircuit.mpscircuit.MPSCircuit class method*), 655

from_serial_dict() (*tensorcircuit.gates.Gate class method*), 603

from_tensor() (*tensorcircuit.quantum.QuAdjointVector class method*), 671

from_tensor() (*tensorcircuit.quantum.QuOperator class method*), 675

from_tensor() (*tensorcircuit.quantum.QuScalar class method*), 678

from_tensor() (*tensorcircuit.quantum.QuVector class method*), 681

front_from_nodes() (*tensorcircuit.basecircuit.BaseCircuit static method*), 484

front_from_nodes() (*tensorcircuit.circuit.Circuit static method*), 517

front_from_nodes() (*tensorcircuit.densitymatrix.DMCircuit static method*), 556

front_from_nodes() (*tensorcircuit.densitymatrix.DMCircuit2 static method*), 587

G

gapfilling() (*in module tensorcircuit.applications.vags*), 222

Gate (*class in tensorcircuit.gates*), 602

gate_aliases (*tensorcircuit.abstractcircuit.AbstractCircuit attribute*), 200

gate_aliases (*tensorcircuit.basecircuit.BaseCircuit attribute*), 484

gate_aliases (*tensorcircuit.circuit.Circuit* attribute), 517
gate_aliases (*tensorcircuit.densitymatrix.DMCircuit* attribute), 556
gate_aliases (*tensorcircuit.densitymatrix.DMCircuit2* attribute), 587
gate_aliases (*tensorcircuit.mpscircuits.MPSCircuit* attribute), 656
gate_count() (*tensorcircuit.abstractcircuit.AbstractCircuit* method), 200
gate_count() (*tensorcircuit.basecircuit.BaseCircuit* method), 484
gate_count() (*tensorcircuit.circuit.Circuit* method), 517
gate_count() (*tensorcircuit.densitymatrix.DMCircuit* method), 556
gate_count() (*tensorcircuit.densitymatrix.DMCircuit2* method), 587
gate_count() (*tensorcircuit.mpscircuits.MPSCircuit* method), 656
gate_name_trans() (in module *tensorcircuit.vis*), 727
gate_summary() (*tensorcircuit.abstractcircuit.AbstractCircuit* method), 201
gate_summary() (*tensorcircuit.basecircuit.BaseCircuit* method), 484
gate_summary() (*tensorcircuit.circuit.Circuit* method), 517
gate_summary() (*tensorcircuit.densitymatrix.DMCircuit* method), 557
gate_summary() (*tensorcircuit.densitymatrix.DMCircuit2* method), 587
gate_summary() (*tensorcircuit.mpscircuits.MPSCircuit* method), 656
gate_to_matrix() (in module *tensorcircuit.interfaces.tensortrans*), 619
gate_to_MPO() (*tensorcircuit.mpscircuits.MPSCircuit* class method), 656
gate_wrapper() (in module *tensorcircuit.gates*), 608
GateF (class in *tensorcircuit.gates*), 605
GateVF (class in *tensorcircuit.gates*), 605
gatewise_vqe_vag() (in module *tensorcircuit.applications.vags*), 223
gather1d() (*tensorcircuit.backends.jax_backend.JaxBackend* method), 374
gather1d() (*tensorcircuit.backends.numpy_backend.NumpyBackend* method), 402
gather1d() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend* method), 429
gather1d() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend* method), 457
general_args_to_backend() (in module *tensorcircuit.interfaces.tensortrans*), 619
general_args_to_numpy() (in module *tensorcircuit.interfaces.tensortrans*), 619
general_kraus() (*tensorcircuit.circuit.Circuit* method), 517
general_kraus() (*tensorcircuit.densitymatrix.DMCircuit* method), 557
general_kraus() (*tensorcircuit.densitymatrix.DMCircuit2* method), 587
generaldepolarizing() (*tensorcircuit.circuit.Circuit* method), 518
generaldepolarizing() (*tensorcircuit.densitymatrix.DMCircuit* method), 557
generaldepolarizing() (*tensorcircuit.densitymatrix.DMCircuit2* method), 587
generaldepolarizingchannel() (in module *tensorcircuit.channels*), 495
generate_any_double_gate_layer() (in module *tensorcircuit.applications.layers*), 215
generate_any_double_gate_layer_bitflip_mc() (in module *tensorcircuit.applications.layers*), 215
generate_any_gate_layer() (in module *tensorcircuit.applications.layers*), 215
generate_cirq_any_double_gate_layer() (in module *tensorcircuit.applications.layers*), 215
generate_cirq_any_gate_layer() (in module *tensorcircuit.applications.layers*), 215
generate_cirq_double_gate() (in module *tensorcircuit.applications.layers*), 215
generate_cirq_double_gate_layer() (in module *tensorcircuit.applications.layers*), 215
generate_cirq_gate_layer() (in module *tensorcircuit.applications.layers*), 215
generate_double_gate() (in module *tensorcircuit.applications.layers*), 215
generate_double_gate_layer() (in module *tensorcircuit.applications.layers*), 215
generate_double_gate_layer_bitflip() (in module *tensorcircuit.applications.layers*), 215
generate_double_gate_layer_bitflip_mc() (in module *tensorcircuit.applications.layers*), 215
generate_double_layer_block() (in module *tensorcircuit.applications.layers*), 216
generate_gate_layer() (in module *tensorcircuit.applications.layers*), 216
generate_local_hamiltonian() (in module *tensorcircuit.quantum*), 688

generate_qubits() (in module `tensorcircuit.applications.layers`), 216
 generate_random_circuit() (in module `tensorcircuit.applications.utils`), 220
 get_all_dangling() (`tensorcircuit.gates.Gate` method), 603
 get_all_edges() (`tensorcircuit.gates.Gate` method), 603
 get_all_nondangling() (`tensorcircuit.gates.Gate` method), 603
 get_axis_number() (`tensorcircuit.gates.Gate` method), 603
 get_backend() (in module `tensorcircuit.backends.backend_factory`), 365
 get_bond_dimensions() (`tensorcircuit.mpscircuit.MPSCircuit` method), 656
 get_buffer() (`tensorcircuit.torchnn.QuantumNet` method), 713
 get_center_position() (`tensorcircuit.mpscircuit.MPSCircuit` method), 656
 get_circuit_as_quoperator() (`tensorcircuit.circuit.Circuit` method), 518
 get_config() (`tensorcircuit.applications.van.MADE` method), 239
 get_config() (`tensorcircuit.applications.van.MaskedConv2D` method), 261
 get_config() (`tensorcircuit.applications.van.MaskedLinear` method), 273
 get_config() (`tensorcircuit.applications.van.NMF` method), 293
 get_config() (`tensorcircuit.applications.van.PixelCNN` method), 322
 get_config() (`tensorcircuit.applications.van.ResidualBlock` method), 345
 get_config() (`tensorcircuit.applications.vqes.JointSchedule` method), 352
 get_config() (`tensorcircuit.applications.vqes.Linear` method), 357
 get_config() (`tensorcircuit.keras.QuantumLayer` method), 626
 get_contractor() (in module `tensorcircuit.cons`), 535
 get_dimension() (`tensorcircuit.gates.Gate` method), 603
 get_dm_as_quoperator() (`tensorcircuit.densitymatrix.DMCircuit` method), 557
 get_dm_as_quoperator() (`tensorcircuit.densitymatrix.DMCircuit2` method), 588
 get_dm_as_quvector() (`tensorcircuit.densitymatrix.DMCircuit` method), 557
 get_dm_as_quvector() (`tensorcircuit.densitymatrix.DMCircuit2` method), 588
 get_dtype() (in module `tensorcircuit.cons`), 535
 get_edge() (`tensorcircuit.gates.Gate` method), 604
 get_extra_state() (`tensorcircuit.torchnn.QuantumNet` method), 713
 get_graph() (in module `tensorcircuit.applications.graphdata`), 208
 get_input_at() (`tensorcircuit.applications.van.MADE` method), 239
 get_input_at() (`tensorcircuit.applications.van.MaskedConv2D` method), 261
 get_input_at() (`tensorcircuit.applications.van.MaskedLinear` method), 273
 get_input_at() (`tensorcircuit.applications.van.NMF` method), 293
 get_input_at() (`tensorcircuit.applications.van.PixelCNN` method), 323
 get_input_at() (`tensorcircuit.applications.van.ResidualBlock` method), 345
 get_input_at() (`tensorcircuit.applications.vqes.Linear` method), 357
 get_input_at() (`tensorcircuit.keras.QuantumLayer` method), 627
 get_input_mask_at() (`tensorcircuit.applications.van.MADE` method), 239
 get_input_mask_at() (`tensorcircuit.applications.van.MaskedConv2D` method), 262
 get_input_mask_at() (`tensorcircuit.applications.van.MaskedLinear` method), 273
 get_input_mask_at() (`tensorcircuit.applications.van.NMF` method), 293
 get_input_mask_at() (`tensorcircuit.applications.van.PixelCNN` method), 323
 get_input_mask_at() (`tensorcircuit.applications.van.ResidualBlock` method), 345
 get_input_mask_at() (`tensorcircuit.applications.vqes.Linear` method), 357
 get_input_mask_at() (`tensorcircuit.keras.QuantumLayer` method), 627
 get_input_shape_at() (`tensorcircuit.applications.van.MADE` method), 240
 get_input_shape_at() (`tensorcircuit.applications.van.MaskedConv2D` method),

262
get_input_shape_at() (*tensorcircuit.applications.van.MaskedLinear method*), 274
get_input_shape_at() (*tensorcircuit.applications.van.NMF method*), 293
get_input_shape_at() (*tensorcircuit.applications.van.PixelCNN method*), 323
get_input_shape_at() (*tensorcircuit.applications.van.ResidualBlock method*), 345
get_input_shape_at() (*tensorcircuit.applications.vqes.Linear method*), 357
get_input_shape_at() (*tensorcircuit.keras.QuantumLayer method*), 627
get_layer() (*tensorcircuit.applications.van.MADE method*), 240
get_layer() (*tensorcircuit.applications.van.NMF method*), 293
get_layer() (*tensorcircuit.applications.van.PixelCNN method*), 323
get_matrix() (*tensorcircuit.results.readout_mitigation.ReadoutMit method*), 699
get_metrics_result() (*tensorcircuit.applications.van.MADE method*), 240
get_metrics_result() (*tensorcircuit.applications.van.NMF method*), 293
get_metrics_result() (*tensorcircuit.applications.van.PixelCNN method*), 323
get_norm() (*tensorcircuit.mpscircuit.MPSCircuit method*), 656
get_op_pool() (*in module tensorcircuit.applications.dqas*), 206
get_output_at() (*tensorcircuit.applications.van.MADE method*), 240
get_output_at() (*tensorcircuit.applications.van.MaskedConv2D method*), 262
get_output_at() (*tensorcircuit.applications.van.MaskedLinear method*), 274
get_output_at() (*tensorcircuit.applications.van.NMF method*), 294
get_output_at() (*tensorcircuit.applications.van.PixelCNN method*), 323
get_output_at() (*tensorcircuit.applications.van.ResidualBlock method*), 345
get_output_at() (*tensorcircuit.applications.vqes.Linear method*), 358
get_output_at() (*tensorcircuit.keras.QuantumLayer method*), 627
get_parameter() (*tensorcircuit.torchnn.QuantumNet method*), 713
get_positional_logical_mapping() (*tensorcircuit.abstractcircuit.AbstractCircuit method*), 201
get_positional_logical_mapping() (*tensorcircuit.basecircuit.BaseCircuit method*), 485
get_positional_logical_mapping() (*tensorcircuit.circuit.Circuit method*), 518
get_positional_logical_mapping() (*tensorcircuit.densitymatrix.DMCircuit method*), 557
get_positional_logical_mapping() (*tensorcircuit*

cuit.densitymatrix.DMCircuit2 method), 588

get_positional_logical_mapping() (*tensorcircuit.mpscircuit.MPSCircuit method*), 656

get_preset() (*in module tensorcircuit.applications.dqas*), 206

get_ps() (*in module tensorcircuit.templates.chems*), 703

get_quoperator() (*tensorcircuit.circuit.Circuit method*), 518

get_qvector() (*tensorcircuit.basecircuit.BaseCircuit method*), 485

get_qvector() (*tensorcircuit.circuit.Circuit method*), 518

get_qvector() (*tensorcircuit.densitymatrix.DMCircuit method*), 557

get_qvector() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 588

get_qvector() (*tensorcircuit.mpscircuit.MPSCircuit method*), 657

get_random_state() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 374

get_random_state() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 402

get_random_state() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 429

get_random_state() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 458

get_rank() (*tensorcircuit.gates.Gate method*), 604

get_state_as_qvector() (*tensorcircuit.circuit.Circuit method*), 518

get_submodule() (*tensorcircuit.torchnn.QuantumNet method*), 713

get_tensor() (*tensorcircuit.gates.Gate method*), 604

get_tensor() (*tensorcircuit.mps_base.FiniteMPS method*), 637

get_tensors() (*tensorcircuit.mpscircuit.MPSCircuit method*), 657

get_u_parameter() (*in module tensorcircuit.gates*), 608

get_var() (*in module tensorcircuit.applications.dqas*), 206

get_weight_paths() (*tensorcircuit.applications.van.MADE method*), 241

get_weight_paths() (*tensorcircuit.applications.van.NMF method*), 294

get_weight_paths() (*tensorcircuit.applications.van.PixelCNN method*), 324

get_weights() (*in module tensorcircuit.applications.dqas*), 206

get_weights() (*tensorcircuit.applications.van.MADE method*), 241

get_weights() (*tensorcircuit.applications.van.MaskedConv2D method*), 262

get_weights() (*tensorcircuit.applications.van.MaskedLinear method*), 274

get_weights() (*tensorcircuit.applications.van.NMF method*), 295

get_weights() (*tensorcircuit.applications.van.PixelCNN method*), 324

get_weights() (*tensorcircuit.applications.van.ResidualBlock method*), 346

get_weights() (*tensorcircuit.applications.vqes.Linear method*), 358

get_weights() (*tensorcircuit.keras.QuantumLayer method*), 627

get_weights_v2() (*in module tensorcircuit.applications.dqas*), 206

GHZ_vag() (*in module tensorcircuit.applications.vags*), 221

GHZ_vag_tfq() (*in module tensorcircuit.applications.vags*), 221

gibbs_state() (*in module tensorcircuit.applications.vags*), 223

gibbs_state() (*in module tensorcircuit.quantum*), 688

global_miti_readout_circ() (*tensorcircuit.results.readout_mitigation.ReadoutMit method*), 699

gmres() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 374

gmres() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 402

gmres() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 429

gmres() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 458

grad() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 376

grad() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 403

grad() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 430

grad() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 459

graph1D() (*in module tensorcircuit.applications.graphdata*), 208

Grid2D() (*in module tensorcircuit.applications.graphdata*), 208

Grid2D_entangling() (*in module tensorcircuit.templates.blocks*), 701

Grid2DCoord (*class in tensorcircuit.templates.graphs*), 703

H

 H() (*tensorcircuit.circuit.Circuit method*), 501

 h() (*tensorcircuit.circuit.Circuit method*), 518

 H() (*tensorcircuit.densitymatrix.DMCircuit method*), 541

 h() (*tensorcircuit.densitymatrix.DMCircuit method*), 557

 H() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 572

 h() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 588

 H() (*tensorcircuit.mpscircuit.MPSCircuit method*), 641

 h() (*tensorcircuit.mpscircuit.MPSCircuit method*), 657

 half() (*tensorcircuit.torchnn.QuantumNet method*), 714

 hamiltonian_evol() (*in module tensorcircuit.experimental*), 601

 has_dangling_edge() (*tensorcircuit.gates.Gate method*), 604

 has_nondangling_edge() (*tensorcircuit.gates.Gate method*), 604

 Heisenberg1Denergy() (*in module tensorcircuit.applications.utils*), 220

 heisenberg_hamiltonian() (*in module tensorcircuit.quantum*), 688

 heisenberg_measurements() (*in module tensorcircuit.applications.vags*), 223

 heisenberg_measurements() (*in module tensorcircuit.templates.measurements*), 705

 heisenberg_measurements_tc() (*in module tensorcircuit.applications.vags*), 223

 hessian() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 376

 hessian() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 403

 hessian() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 431

 hessian() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 459

 history_loss() (*in module tensorcircuit.applications.dqas*), 207

 Hlayer() (*in module tensorcircuit.applications.layers*), 209

I

 i() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 376

 i() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 403

 i() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 431

 i() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 459

 I() (*tensorcircuit.circuit.Circuit method*), 501

 i() (*tensorcircuit.circuit.Circuit method*), 519

 I() (*tensorcircuit.densitymatrix.DMCircuit method*), 541

 i() (*tensorcircuit.densitymatrix.DMCircuit method*), 558

 I() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 572

 i() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 588

 I() (*tensorcircuit.mpscircuit.MPSCircuit method*), 641

 i() (*tensorcircuit.mpscircuit.MPSCircuit method*), 657

 ided() (*tensorcircuit.gates.GateF method*), 605

 ided() (*tensorcircuit.gates.GateVF method*), 605

 identity() (*in module tensorcircuit.quantum*), 689

 Ilayer() (*in module tensorcircuit.applications.layers*), 209

 imag() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 376

 imag() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 404

 imag() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 431

 imag() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 459

 implicit_randc() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 376

 implicit_randc() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 404

 implicit_randc() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 431

 implicit_randc() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 459

 implicit_rannd() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 377

 implicit_rannd() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 404

 implicit_rannd() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 431

 implicit_rannd() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 459

 implicit_rannd() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 377

 implicit_rannd() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 404

 implicit_rannd() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 431

 implicit_rannd() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 459

 implicit_rannd() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 377

 implicit_rannd() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 404

 implicit_rannd() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 431

 implicit_rannd() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 459

 implicit_rannd() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 377

 implicit_rannd() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 404

 implicit_rannd() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 431

 implicit_rannd() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 459

 implicit_rannd() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 377

 implicit_rannd() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 404

 implicit_rannd() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 431

 implicit_rannd() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 459

<code>method), 432</code>	<code>initial_mapping()</code>	<i>(tensorcircuit.circuit.Circuit method), 519</i>
<code>implicit_randu()</code>	<code>initial_mapping()</code>	<i>(tensorcircuit.densitymatrix.DMCircuit method), 558</i>
<code>circuit.backends.tensorflow_backend.TensorFlowBackend method), 460</code>	<code>initial_mapping()</code>	<i>(tensorcircuit.densitymatrix.DMCircuit2 method), 588</i>
<code>in_space (tensorcircuit.quantum.QuAdjointVector property), 672</code>	<code>initial_mapping()</code>	<i>(tensorcircuit.mpscircuit.MPSCircuit method), 657</i>
<code>in_space (tensorcircuit.quantum.QuOperator property), 676</code>	<code>input (tensorcircuit.applications.van.MADE property), 241</code>	
<code>in_space (tensorcircuit.quantum.QuScalar property), 679</code>	<code>input (tensorcircuit.applications.van.MaskedConv2D property), 263</code>	
<code>in_space (tensorcircuit.quantum.QuVector property), 682</code>	<code>input (tensorcircuit.applications.van.MaskedLinear property), 275</code>	
<code>inbound_nodes (tensorcircuit.applications.van.MADE property), 241</code>	<code>input (tensorcircuit.applications.van.NMF property), 295</code>	
<code>inbound_nodes (tensorcircuit.applications.van.MaskedConv2D property), 263</code>	<code>input (tensorcircuit.applications.van.PixelCNN property), 324</code>	
<code>inbound_nodes (tensorcircuit.applications.van.MaskedLinear property), 275</code>	<code>input (tensorcircuit.applications.van.ResidualBlock property), 346</code>	
<code>inbound_nodes (tensorcircuit.applications.van.NMF property), 295</code>	<code>input (tensorcircuit.applications.vqes.Linear property), 359</code>	
<code>inbound_nodes (tensorcircuit.applications.van.PixelCNN property), 324</code>	<code>input (tensorcircuit.keras.QuantumLayer property), 628</code>	
<code>inbound_nodes (tensorcircuit.applications.van.ResidualBlock property), 346</code>	<code>input_mask (tensorcircuit.applications.van.MADE property), 241</code>	
<code>inbound_nodes (tensorcircuit.applications.vqes.Linear property), 359</code>	<code>input_mask (tensorcircuit.applications.van.MaskedConv2D property), 263</code>	
<code>inbound_nodes (tensorcircuit.keras.QuantumLayer property), 628</code>	<code>input_mask (tensorcircuit.applications.van.MaskedLinear property), 275</code>	
<code>index() (tensorcircuit.channels.KrausList method), 492</code>	<code>input_mask (tensorcircuit.applications.van.NMF property), 295</code>	
<code>index_update() (tensorcircuit.backends.jax_backend.JaxBackend method), 377</code>	<code>input_mask (tensorcircuit.applications.van.PixelCNN property), 325</code>	
<code>index_update() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 405</code>	<code>input_mask (tensorcircuit.applications.van.ResidualBlock property), 346</code>	
<code>index_update() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 432</code>	<code>input_mask (tensorcircuit.applications.vqes.Linear property), 359</code>	
<code>index_update() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 460</code>	<code>input_mask (tensorcircuit.keras.QuantumLayer property), 628</code>	
<code>infer_new_shape() (in module tensorcircuit.simplify), 700</code>	<code>input_shape (tensorcircuit.applications.van.MADE property), 242</code>	
<code>infer_new_size() (in module tensorcircuit.simplify), 701</code>	<code>input_shape (tensorcircuit.applications.van.MaskedConv2D property), 263</code>	
<code>initial_mapping() (tensorcircuit.abstractcircuit.AbstractCircuit method), 201</code>	<code>input_shape (tensorcircuit.applications.van.MaskedLinear property), 275</code>	
<code>initial_mapping() (tensorcircuit.basecircuit.BaseCircuit method), 485</code>	<code>input_shape (tensorcircuit.applications.van.NMF property), 295</code>	
	<code>input_shape (tensorcircuit.applications.van.PixelCNN</code>	

property), 325
input_shape (tensorcircuit.applications.van.ResidualBlock property), 347
input_shape (tensorcircuit.applications.vqes.Linear property), 359
input_shape (tensorcircuit.keras.QuantumLayer property), 628
input_spec (tensorcircuit.applications.van.MADE property), 242
input_spec (tensorcircuit.applications.van.MaskedConv2D property), 264
input_spec (tensorcircuit.applications.van.MaskedLinear property), 275
input_spec (tensorcircuit.applications.van.NMF property), 295
input_spec (tensorcircuit.applications.van.PixelCNN property), 325
input_spec (tensorcircuit.applications.van.ResidualBlock property), 347
input_spec (tensorcircuit.applications.vqes.Linear property), 359
input_spec (tensorcircuit.keras.QuantumLayer property), 629
inputs (tensorcircuit.abstractcircuit.AbstractCircuit attribute), 201
inputs (tensorcircuit.basecircuit.BaseCircuit attribute), 485
inputs (tensorcircuit.circuit.Circuit attribute), 519
inputs (tensorcircuit.densitymatrix.DMCircuit attribute), 558
inputs (tensorcircuit.densitymatrix.DMCircuit2 attribute), 589
inputs (tensorcircuit.mpscircuit.MPSCircuit attribute), 657
insert() (tensorcircuit.channels.KrausList method), 492
inv() (tensorcircuit.backends.jax_backend.JaxBackend method), 377
inv() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 405
inv() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 432
inv() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 461
inverse() (tensorcircuit.abstractcircuit.AbstractCircuit method), 201
inverse() (tensorcircuit.basecircuit.BaseCircuit method), 485
inverse() (tensorcircuit.circuit.Circuit method), 519
inverse() (tensorcircuit.densitymatrix.DMCircuit method), 558
inverse() (tensorcircuit.densitymatrix.DMCircuit2 method), 589
inverse() (tensorcircuit.mpscircuit.MPSCircuit method), 657
is_adjoint_vector() (tensorcircuit.quantum.QuAdjointVector method), 672
is_adjoint_vector() (tensorcircuit.quantum.QuOperator method), 676
is_adjoint_vector() (tensorcircuit.quantum.QuScalar method), 679
is_adjoint_vector() (tensorcircuit.quantum.QuVector method), 682
is_dm (tensorcircuit.basecircuit.BaseCircuit attribute), 485
is_dm (tensorcircuit.circuit.Circuit attribute), 519
is_dm (tensorcircuit.densitymatrix.DMCircuit attribute), 558
is_dm (tensorcircuit.densitymatrix.DMCircuit2 attribute), 589
is_hermitian_matrix() (in module tensorcircuit.channels), 495
is_m1mac() (in module tensorcircuit.utils), 726
is_mps (tensorcircuit.abstractcircuit.AbstractCircuit attribute), 201
is_mps (tensorcircuit.basecircuit.BaseCircuit attribute), 486
is_mps (tensorcircuit.circuit.Circuit attribute), 519
is_mps (tensorcircuit.densitymatrix.DMCircuit attribute), 558
is_mps (tensorcircuit.densitymatrix.DMCircuit2 attribute), 589
is_mps (tensorcircuit.mpscircuit.MPSCircuit attribute), 658
is_number() (in module tensorcircuit.utils), 726
is_scalar() (tensorcircuit.quantum.QuAdjointVector method), 672
is_scalar() (tensorcircuit.quantum.QuOperator method), 676
is_scalar() (tensorcircuit.quantum.QuScalar method), 679
is_scalar() (tensorcircuit.quantum.QuVector method), 682
is_sequence() (in module tensorcircuit.utils), 726
is_jax_backend() (tensorcircuit.backends.jax_backend.JaxBackend method), 377
is_sparse() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 405
is_sparse() (tensorcircuit.backends.pytorch_backend.PyTorchBackend

J
is_sparse() (*tensorcircuit.circuit.backends.tensorflow_backend.TensorFlowBackend method*), 461
is_tensor() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 378
is_tensor() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 405
is_tensor() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 432
is_tensor() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 461
is_valid() (*tensorcircuit.circuit.Circuit method*), 519
is_valid() (*tensorcircuit.mpscircuits.MPSCircuit method*), 658
is_vector() (*tensorcircuit.quantum.QuAdjointVector method*), 672
is_vector() (*tensorcircuit.quantum.QuOperator method*), 676
is_vector() (*tensorcircuit.quantum.QuScalar method*), 679
is_vector() (*tensorcircuit.quantum.QuVector method*), 682
ISWAP() (*tensorcircuit.circuit.Circuit method*), 501
iswap() (*tensorcircuit.circuit.Circuit method*), 520
ISWAP() (*tensorcircuit.densitymatrix.DMCircuit method*), 541
iswap() (*tensorcircuit.densitymatrix.DMCircuit method*), 558
ISWAP() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 572
iswap() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 589
ISWAP() (*tensorcircuit.mpscircuits.MPSCircuit method*), 641
iswap() (*tensorcircuit.mpscircuits.MPSCircuit method*), 658
iswap_gate() (*in module tensorcircuit.gates*), 608
item() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 378
item() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 405
item() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 432
item() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 461
J
jacbwd() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 378

K
jacbwds() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 405
jacbwds() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 433
jacbwds() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 461
jacfwd() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 378
jacfwd() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 405
jacfwd() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 433
jacfwd() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 461
jaxdev() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 378
jacrev() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 406
jacrev() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 433
jacrev() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 461
JaxBackend (*class in tensorcircuit.backends.jax_backend*), 365
jit() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 378
jit() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 406
jit() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 433
jit() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 462
jnp (*in module tensorcircuit.backends.jax_backend*), 393
JointSchedule (*class in tensorcircuit.applications.vqes*), 352
json2qir() (*in module tensorcircuit.translation*), 723
json_to_tensor() (*in module tensorcircuit.translation*), 723
jsp (*in module tensorcircuit.backends.jax_backend*), 393
jvp() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 379
jvp() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 406
jvp() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 433
jvp() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 462
K
keras_optimizer (*class in tensorcircuit.backends.tensorflow_backend*), 477
KerasLayer (*in module tensorcircuit.keras*), 622
kl_divergence() (*in module tensorcircuit.results.counts*), 697

kraus_identity_check() (in module `tensorcircuit.channels`), 495
kraus_to_choi() (in module `tensorcircuit.channels`), 496
kraus_to_super() (in module `tensorcircuit.channels`), 496
kraus_to_super_gate() (in module `tensorcircuit.channels`), 496
krausgate_to_krausmatrix() (in module `tensorcircuit.channels`), 497
`KrausList` (class in `tensorcircuit.channels`), 492
krausmatrix_to_krausgate() (in module `tensorcircuit.channels`), 497
kron() (`tensorcircuit.backends.jax_backend.JaxBackend` method), 379
kron() (`tensorcircuit.backends.numpy_backend.NumpyBackend` method), 406
kron() (`tensorcircuit.backends.pytorch_backend.PyTorchBackend` method), 434
kron() (`tensorcircuit.backends.tensorflow_backend.TensorFlowBackend` method), 462

L

lattice_graph() (in module `tensorcircuit.templates.graphs.Grid2DCoord` method), 704
layers (`tensorcircuit.applications.van.MADE` property), 242
layers (`tensorcircuit.applications.van.NMF` property), 296
layers (`tensorcircuit.applications.van.PixelCNN` property), 325
left_envs() (`tensorcircuit.mps_base.FiniteMPS` method), 637
left_shift() (`tensorcircuit.backends.jax_backend.JaxBackend` method), 379
left_shift() (`tensorcircuit.backends.numpy_backend.NumpyBackend` method), 406
left_shift() (`tensorcircuit.backends.pytorch_backend.PyTorchBackend` method), 434
left_shift() (`tensorcircuit.backends.tensorflow_backend.TensorFlowBackend` method), 462
left_transfer_operator() (`tensorcircuit.mps_base.FiniteMPS` method), 637
libjax (in module `tensorcircuit.backends.jax_backend`), 393
Line1D() (in module `tensorcircuit.templates.graphs`), 704
Linear (class in `tensorcircuit.applications.vqes`), 352

load() (`tensorcircuit.applications.vqes.VQNHE` method), 364
load_func() (in module `tensorcircuit.keras`), 633
load_state_dict() (in module `tensorcircuit.torchnn.QuantumNet` method), 715
load_weights() (`tensorcircuit.applications.van.MADE` method), 242
load_weights() (`tensorcircuit.applications.van.NMF` method), 296
load_weights() (`tensorcircuit.applications.van.PixelCNN` method), 325
local_miti_readout_circ() (in module `tensorcircuit.results.readout_mitigation.ReadoutMit` method), 699
local_miti_readout_circ_by_mask() (in module `tensorcircuit.results.readout_mitigation.ReadoutMit` method), 699
log() (`tensorcircuit.backends.jax_backend.JaxBackend` method), 379
log() (`tensorcircuit.backends.numpy_backend.NumpyBackend` method), 407
log() (`tensorcircuit.backends.pytorch_backend.PyTorchBackend` method), 434
log() (`tensorcircuit.backends.tensorflow_backend.TensorFlowBackend` method), 462
log_prob() (`tensorcircuit.applications.van.MADE` method), 243
log_prob() (`tensorcircuit.applications.van.NMF` method), 296
log_prob() (`tensorcircuit.applications.van.PixelCNN` method), 326
losses (`tensorcircuit.applications.van.MADE` property), 243
losses (`tensorcircuit.applications.van.MaskedConv2D` property), 264
losses (`tensorcircuit.applications.van.MaskedLinear` property), 276
losses (`tensorcircuit.applications.van.NMF` property), 296
losses (`tensorcircuit.applications.van.PixelCNN` property), 326
losses (`tensorcircuit.applications.van.ResidualBlock` property), 347
losses (`tensorcircuit.applications.vqes.Linear` property), 359
losses (`tensorcircuit.keras.QuantumLayer` property), 629

M

MADE (class in `tensorcircuit.applications.van`), 227
make_predict_function() (`tensorcircuit.applications.van.MADE` method), 244

make_predict_function() (*tensorcircuit.applications.van.NMF* method), 297
make_predict_function() (*tensorcircuit.applications.van.PixelCNN* method), 327
make_test_function() (*tensorcircuit.applications.van.MADE* method), 244
make_test_function() (*tensorcircuit.applications.van.NMF* method), 297
make_test_function() (*tensorcircuit.applications.van.PixelCNN* method), 327
make_train_function() (*tensorcircuit.applications.van.MADE* method), 244
make_train_function() (*tensorcircuit.applications.van.NMF* method), 298
make_train_function() (*tensorcircuit.applications.van.PixelCNN* method), 328
mapping_preprocess() (*tensorcircuit.results.readout_mitigation.ReadoutMit* method), 699
marginal_count() (in module *tensorcircuit.results.counts*), 697
MaskedConv2D (class in *tensorcircuit.applications.van*), 256
MaskedLinear (class in *tensorcircuit.applications.van*), 268
matmul() (*tensorcircuit.backends.jax_backend.JaxBackend* method), 379
matmul() (*tensorcircuit.backends.numpy_backend.NumpyBackend* method), 407
matmul() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend* method), 434
matmul() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend* method), 462
matrix() (*tensorcircuit.circuit.Circuit* method), 520
matrix_for_gate() (in module *tensorcircuit.gates*), 608
max() (*tensorcircuit.backends.jax_backend.JaxBackend* method), 380
max() (*tensorcircuit.backends.numpy_backend.NumpyBackend* method), 407
max() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend* method), 434
max() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend* method), 463
maxcut_measurements_tc() (in module *tensorcircuit.applications.vags*), 223
maxcut_solution_bruteforce() (in module *tensorcircuit.applications.graphdata*), 209
mean() (*tensorcircuit.backends.jax_backend.JaxBackend* method), 380
mean() (*tensorcircuit.backends.numpy_backend.NumpyBackend* method), 407
mean() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend* method), 434
mean() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend* method), 463
measure() (*tensorcircuit.basecircuit.BaseCircuit* method), 486
measure() (*tensorcircuit.circuit.Circuit* method), 520
measure() (*tensorcircuit.densitymatrix.DMCircuit* method), 559
measure() (*tensorcircuit.densitymatrix.DMCircuit2* method), 589
measure() (*tensorcircuit.mpscircuit.MPSCircuit* method), 658
measure_instruction() (*tensorcircuit.abstractcircuit.AbstractCircuit* method), 201
measure_instruction() (*tensorcircuit.basecircuit.BaseCircuit* method), 486
measure_instruction() (*tensorcircuit.circuit.Circuit* method), 520
measure_instruction() (*tensorcircuit.densitymatrix.DMCircuit* method), 559
measure_instruction() (*tensorcircuit.densitymatrix.DMCircuit2* method), 589
measure_instruction() (*tensorcircuit.mpscircuit.MPSCircuit* method), 658
measure_jit() (*tensorcircuit.basecircuit.BaseCircuit* method), 486
measure_jit() (*tensorcircuit.circuit.Circuit* method), 520
measure_jit() (*tensorcircuit.densitymatrix.DMCircuit* method), 559
measure_jit() (*tensorcircuit.densitymatrix.DMCircuit2* method), 589
measure_jit() (*tensorcircuit.mpscircuit.MPSCircuit* method), 658
measure_local_operator() (*tensorcircuit.mps_base.FiniteMPS* method), 637
measure_reference() (*tensorcircuit.circuit.Circuit* method), 520
measure_two_body_correlator() (*tensorcircuit.mps_base.FiniteMPS* method), 637
measurement_counts() (in module *tensorcircuit.quantum*), 689
measureBackend_results() (in module *tensorcircuit.quantum*), 690
meta_gate() (in module *tensorcircuit.gates*), 609
meta_vgate() (in module *tensorcircuit.gates*), 609
metrics (*tensorcircuit.applications.van.MADE* property), 245
metrics (*tensorcircuit.applications.van.MaskedConv2D* property), 265
metrics (*tensorcircuit.applications.van.MaskedLinear*

property), 276
`metrics` (*tensorcircuit.applications.van.NMF* *property*), 298
`metrics` (*tensorcircuit.applications.van.PixelCNN* *property*), 328
`metrics` (*tensorcircuit.applications.van.ResidualBlock* *property*), 348
`metrics` (*tensorcircuit.applications.vqes.Linear* *property*), 360
`metrics` (*tensorcircuit.keras.QuantumLayer* *property*), 630
`metrics_names` (*tensorcircuit.applications.van.MADE* *property*), 245
`metrics_names` (*tensorcircuit.applications.van.NMF* *property*), 299
`metrics_names` (*tensorcircuit.applications.van.PixelCNN* *property*), 329
`micro_sample()` (*in module* *tensorcircuit.applications.dqas*), 207
`mid_measure()` (*tensorcircuit.circuit.Circuit* *method*), 521
`mid_measurement()` (*tensorcircuit.circuit.Circuit* *method*), 521
`mid_measurement()` (*tensorcircuit.mpscircut.MPSCircuit* *method*), 658
`min()` (*tensorcircuit.backends.jax_backend.JaxBackend* *method*), 380
`min()` (*tensorcircuit.backends.numpy_backend.NumpyBackend* *method*), 407
`min()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend* *method*), 435
`min()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend* *method*), 463
`mitigate_probability()` (*tensorcircuit.results.readout_mitigation.ReadoutMit* *method*), 700
`mnist_amplitude_data()` (*in module* *tensorcircuit.applications.utils*), 220
`mnist_generator()` (*in module* *tensorcircuit.applications.utils*), 220
`mnist_pair_data()` (*in module* *tensorcircuit.templates.dataset*), 703
`mod()` (*tensorcircuit.backends.jax_backend.JaxBackend* *method*), 380
`mod()` (*tensorcircuit.backends.numpy_backend.NumpyBackend* *method*), 408
`mod()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend* *method*), 435
`mod()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend* *method*), 463
`model()` (*tensorcircuit.applications.van.MADE* *method*), 246
`module`

`tensorcircuit.abstractcircuit`, 195
`tensorcircuit.applications.dqas`, 204
`tensorcircuit.applications.graphdata`, 208
`tensorcircuit.applications.layers`, 209
`tensorcircuit.applications.utils`, 220
`tensorcircuit.applications.vags`, 221
`tensorcircuit.applications.van`, 227
`tensorcircuit.applications.vqes`, 352
`tensorcircuit.backends.backend_factory`, 365
`tensorcircuit.backends.jax_backend`, 365
`tensorcircuit.backends.numpy_backend`, 394
`tensorcircuit.backends.pytorch_backend`, 421
`tensorcircuit.backends.tensorflow_backend`, 449
`tensorcircuit.basecircuit`, 478
`tensorcircuit.channels`, 492
`tensorcircuit.circuit`, 499
`tensorcircuit.compiler.qiskit_compiler`, 534
`tensorcircuit.cons`, 535
`tensorcircuit.densitymatrix`, 539
`tensorcircuit.experimental`, 601
`tensorcircuit.gates`, 602
`tensorcircuit.interfaces.numpy`, 613
`tensorcircuit.interfaces.scipy`, 614
`tensorcircuit.interfaces.tensorflow`, 616
`tensorcircuit.interfaces.tensortrans`, 618
`tensorcircuit.interfaces.torch`, 620
`tensorcircuit.keras`, 622
`tensorcircuit.mps_base`, 635
`tensorcircuit.mpscircut`, 639
`tensorcircuit.noisemodel`, 666
`tensorcircuit.quantum`, 668
`tensorcircuit.results.counts`, 697
`tensorcircuit.results.readout_mitigation`, 697
`tensorcircuit.simplify`, 700
`tensorcircuit.templates.blocks`, 701
`tensorcircuit.templates.chems`, 703
`tensorcircuit.templates.dataset`, 703
`tensorcircuit.templates.graphs`, 703
`tensorcircuit.templates.measurements`, 704
`tensorcircuit.torchnn`, 709
`tensorcircuit.translation`, 723
`tensorcircuit.utils`, 726
`tensorcircuit.vis`, 727
 modules () (*tensorcircuit.torchnn.QuantumNet* *method*), 541
 MPO () (*tensorcircuit.circuit.Circuit* *method*), 502
 mpo () (*tensorcircuit.circuit.Circuit* *method*), 521
 MPO () (*tensorcircuit.densitymatrix.DMCircuit* *method*), 541

`mpo()` (*tensorcircuit.densitymatrix.DMCircuit method*), 559
`MPO()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 572
`mpo()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 590
`MPO()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 641
`mpo()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 658
`mpo_expectation()` (*in module tensorcircuit.templates.measurements*), 706
`mpo_gate()` (*in module tensorcircuit.gates*), 609
`MPO_to_gate()` (*tensorcircuit.mpscircuit.MPSCircuit class method*), 641
`mpogates` (*tensorcircuit.abstractcircuit.AbstractCircuit attribute*), 202
`mpogates` (*tensorcircuit.basecircuit.BaseCircuit attribute*), 486
`mpogates` (*tensorcircuit.circuit.Circuit attribute*), 521
`mpogates` (*tensorcircuit.densitymatrix.DMCircuit attribute*), 559
`mpogates` (*tensorcircuit.densitymatrix.DMCircuit2 attribute*), 590
`mpogates` (*tensorcircuit.mpscircuit.MPSCircuit attribute*), 659
`MPSCircuit` (*class in tensorcircuit.mpscircuit*), 639
`multi_training()` (*tensorcircuit.applications.vqes.VQNHE method*), 364
`MULTICONTROL()` (*tensorcircuit.circuit.Circuit method*), 502
`multicontrol()` (*tensorcircuit.circuit.Circuit method*), 521
`MULTICONTROL()` (*tensorcircuit.densitymatrix.DMCircuit method*), 541
`multicontrol()` (*tensorcircuit.densitymatrix.DMCircuit method*), 559
`MULTICONTROL()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 572
`multicontrol()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 590
`MULTICONTROL()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 641
`multicontrol()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 659
`multicontrol_gate()` (*in module tensorcircuit.gates*), 609
`multiply()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 380
`multiply()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 408
`multiply()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 435
`multiply()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 464
`mutual_information()` (*in module tensorcircuit.quantum*), 691

N

`naive_qml_vag()` (*in module tensorcircuit.applications.utils*), 220
`name` (*tensorcircuit.applications.van.MADE property*), 246
`name` (*tensorcircuit.applications.van.MaskedConv2D property*), 265
`name` (*tensorcircuit.applications.van.MaskedLinear property*), 277
`name` (*tensorcircuit.applications.van.NMF property*), 299
`name` (*tensorcircuit.applications.van.PixelCNN property*), 329
`name` (*tensorcircuit.applications.van.ResidualBlock property*), 348
`name` (*tensorcircuit.applications.vqes.Linear property*), 360
`name` (*tensorcircuit.gates.Gate property*), 604
`name` (*tensorcircuit.keras.QuantumLayer property*), 630
`name_scope` (*tensorcircuit.applications.van.MADE property*), 246
`name_scope` (*tensorcircuit.applications.van.MaskedConv2D property*), 265
`name_scope` (*tensorcircuit.applications.van.MaskedLinear property*), 277
`name_scope` (*tensorcircuit.applications.van.NMF property*), 299
`name_scope` (*tensorcircuit.applications.van.PixelCNN property*), 329
`name_scope` (*tensorcircuit.applications.van.ResidualBlock property*), 348
`name_scope` (*tensorcircuit.applications.vqes.Linear property*), 361
`name_scope` (*tensorcircuit.keras.QuantumLayer property*), 630
`named_buffers()` (*tensorcircuit.torchnn.QuantumNet method*), 715
`named_children()` (*tensorcircuit.torchnn.QuantumNet method*), 716
`named_modules()` (*tensorcircuit.torchnn.QuantumNet method*), 716

named_parameters()	(<i>tensorcircuit.torchnn.QuantumNet</i> method), 716	non_trainable_weights	(<i>tensorcircuit.applications.vqes.Linear</i> property), 361
newrange()	(<i>tensorcircuit.circuit.results.readout_mitigation.ReadoutMit</i> method), 700	non_trainable_weights	(<i>tensorcircuit.keras.QuantumLayer</i> property), 630
NMF (class in <i>tensorcircuit.applications.van</i>), 280		norm()	(<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 380
nodes (<i>tensorcircuit.quantum.QuAdjointVector</i> property), 672		norm()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 408
nodes (<i>tensorcircuit.quantum.QuOperator</i> property), 676		norm()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 435
nodes (<i>tensorcircuit.quantum.QuScalar</i> property), 679		norm()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 464
nodes (<i>tensorcircuit.quantum.QuVector</i> property), 682		norm()	(<i>tensorcircuit.quantum.QuAdjointVector</i> method), 672
nodes_to_adj() (in module <i>tensorcircuit.cons</i>), 535		norm()	(<i>tensorcircuit.quantum.QuOperator</i> method), 676
noise_forward() (in module <i>tensorcircuit.circuit.applications.vags</i>), 223		norm()	(<i>tensorcircuit.quantum.QuScalar</i> method), 679
NoiseConf (class in <i>tensorcircuit.noisemodel</i>), 666		norm()	(<i>tensorcircuit.quantum.QuVector</i> method), 682
noisyfy() (in module <i>tensorcircuit.applications.vags</i>), 223		normalize()	(<i>tensorcircuit.mpscircuits.MPSCircuit</i> method), 659
non_trainable_variables	(<i>tensorcircuit.circuit.applications.van.MADE</i> property), 246	normalized_count()	(in module <i>tensorcircuit.results.counts</i>), 697
non_trainable_variables	(<i>tensorcircuit.circuit.applications.van.MaskedConv2D</i> property), 265	np_interface()	(in module <i>tensorcircuit.interfaces.numpy</i>), 613
non_trainable_variables	(<i>tensorcircuit.circuit.applications.van.MaskedLinear</i> property), 277	num_to_tensor()	(in module <i>tensorcircuit.gates</i>), 609
non_trainable_variables	(<i>tensorcircuit.circuit.applications.van.NMF</i> property), 299	numpy()	(<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 381
non_trainable_variables	(<i>tensorcircuit.circuit.applications.van.PixelCNN</i> property), 329	numpy()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 408
non_trainable_variables	(<i>tensorcircuit.circuit.applications.van.ResidualBlock</i> property), 348	numpy()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 435
non_trainable_variables	(<i>tensorcircuit.circuit.applications.vqes.Linear</i> property), 361	numpy()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 464
non_trainable_variables	(<i>tensorcircuit.keras.QuantumLayer</i> property), 630	numpy_args_to_backend()	(in module <i>tensorcircuit.interfaces.tensortrans</i>), 619
non_trainable_weights	(<i>tensorcircuit.circuit.applications.van.MADE</i> property), 246	numpy_interface()	(in module <i>tensorcircuit.interfaces.numpy</i>), 613
non_trainable_weights	(<i>tensorcircuit.circuit.applications.van.MaskedConv2D</i> property), 265	numpy_to_tensor()	(in module <i>tensorcircuit.interfaces.tensortrans</i>), 620
non_trainable_weights	(<i>tensorcircuit.circuit.applications.van.MaskedLinear</i> property), 277	NumpyBackend	(class in <i>tensorcircuit.backends.numpy_backend</i>), 394
non_trainable_weights	(<i>tensorcircuit.circuit.applications.van.NMF</i> property), 300	O	
non_trainable_weights	(<i>tensorcircuit.circuit.applications.van.PixelCNN</i> property), 329	ocontrolled()	(<i>tensorcircuit.gates.GateF</i> method), 605
non_trainable_weights	(<i>tensorcircuit.circuit.applications.van.ResidualBlock</i> property), 348	ocontrolled()	(<i>tensorcircuit.gates.GateVF</i> method), 605

one_hot() (`tensorcircuit.backends.numpy_backend.NumpyBackend`)
 method), 408
one_hot() (`tensorcircuit.backends.pytorch_backend.PyTorchBackend`)
 method), 435
one_hot() (`tensorcircuit.backends.tensorflow_backend.TensorFlowBackend`)
 method), 464
onehot() (`tensorcircuit.backends.jax_backend.JaxBackend`)
 method), 381
onehot() (`tensorcircuit.backends.numpy_backend.NumpyBackend`)
 method), 408
onehot() (`tensorcircuit.backends.pytorch_backend.PyTorchBackend`)
 method), 435
onehot() (`tensorcircuit.backends.tensorflow_backend.TensorFlowBackend`)
 method), 464
ones() (`tensorcircuit.backends.jax_backend.JaxBackend`)
 method), 381
ones() (`tensorcircuit.backends.numpy_backend.NumpyBackend`)
 method), 408
ones() (`tensorcircuit.backends.pytorch_backend.PyTorchBackend`)
 method), 436
ones() (`tensorcircuit.backends.tensorflow_backend.TensorFlowBackend`)
 method), 464
op2tensor() (in module `tensorcircuit.quantum`), 692
op_protection() (`tensorcircuit.gates.Gate` method),
 604
operator_expectation() (in module `tensorcircuit.templates.measurements`), 706
optax (in module `tensorcircuit.backends.jax_backend`),
 393
optax_optimizer (class in `tensorcircuit.backends.jax_backend`), 393
optimizer (`tensorcircuit.backends.jax_backend.JaxBackend`)
 attribute), 381
optimizer (`tensorcircuit.backends.pytorch_backend.PyTorchBackend`)
 attribute), 436
optimizer (`tensorcircuit.backends.tensorflow_backend.TensorFlowBackend`)
 attribute), 464
ORX() (`tensorcircuit.circuit.Circuit` method), 502
orx() (`tensorcircuit.circuit.Circuit` method), 521
ORX() (`tensorcircuit.densitymatrix.DMCircuit` method),
 542
orx() (`tensorcircuit.densitymatrix.DMCircuit` method),
 559
ORX() (`tensorcircuit.densitymatrix.DMCircuit2` method),
 573
orx() (`tensorcircuit.densitymatrix.DMCircuit2` method),
 590
ORX() (`tensorcircuit.mpscircuit.MPSCircuit` method),
 641
orx() (`tensorcircuit.mpscircuit.MPSCircuit` method),
 659
ORY() (`tensorcircuit.circuit.Circuit` method), 502
ory() (`tensorcircuit.circuit.Circuit` method), 522
ORY() (`tensorcircuit.densitymatrix.DMCircuit` method),
 590
ORY() (`tensorcircuit.densitymatrix.DMCircuit` method),
 642
ORY() (`tensorcircuit.mpscircuit.MPSCircuit` method),
 659
ORZ() (`tensorcircuit.circuit.Circuit` method), 502
orz() (`tensorcircuit.densitymatrix.DMCircuit` method),
 560
ORZ() (`tensorcircuit.densitymatrix.DMCircuit2` method),
 573
orz() (`tensorcircuit.densitymatrix.DMCircuit2` method),
 679
out_space (`tensorcircuit.quantum.QuAdjointVector`
 property), 672
out_space (`tensorcircuit.quantum.QuOperator` prop-
 erty), 676
out_space (`tensorcircuit.quantum.QuScalar` property),
 679
outbound_nodes (`tensorcircuit.quantum.QuVector` property),
 682
outbound_nodes (`tensorcircuit.applications.van.MADE`
 property), 246
outbound_nodes (`tensorcircuit.applications.van.MaskedConv2D`
 property), 265
outbound_nodes (`tensorcircuit.applications.van.MaskedLinear` property),
 277
outbound_nodes (`tensorcircuit.applications.van.NMF`
 property), 300
outbound_nodes (`tensorcircuit.applications.van.PixelCNN`
 property), 330
outbound_nodes (`tensorcircuit.applications.van.ResidualBlock`
 property), 349
outbound_nodes (`tensorcircuit.applications.vqes.Linear`
 property), 361
outbound_nodes (`tensorcircuit.keras.QuantumLayer`
 property), 630

outer_product() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 381

outer_product() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 408

outer_product() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 436

outer_product() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 464

output (*tensorcircuit.applications.van.MADE property*), 246

output (*tensorcircuit.applications.van.MaskedConv2D property*), 265

output (*tensorcircuit.applications.van.MaskedLinear property*), 277

output (*tensorcircuit.applications.van.NMF property*), 300

output (*tensorcircuit.applications.van.PixelCNN property*), 330

output (*tensorcircuit.applications.van.ResidualBlock property*), 349

output (*tensorcircuit.applications.vqes.Linear property*), 361

output (*tensorcircuit.keras.QuantumLayer property*), 630

output_asis_loss() (*in module tensorcircuit.keras*), 634

output_mask (*tensorcircuit.applications.van.MADE property*), 246

output_mask (*tensorcircuit.backends.jax_backend.JaxBackend property*), 266

output_mask (*tensorcircuit.backends.numpy_backend.NumpyBackend property*), 277

output_mask (*tensorcircuit.applications.van.MaskedLinear property*), 300

output_mask (*tensorcircuit.applications.van.PixelCNN property*), 330

output_mask (*tensorcircuit.applications.van.ResidualBlock property*), 349

output_mask (*tensorcircuit.applications.vqes.Linear property*), 361

output_mask (*tensorcircuit.keras.QuantumLayer property*), 631

output_shape (*tensorcircuit.applications.van.MADE property*), 247

output_shape (*tensorcircuit.backends.jax_backend.JaxBackend property*), 266

output_shape (*tensorcircuit.backends.numpy_backend.NumpyBackend property*), 277

output_shape (*tensorcircuit.applications.van.NMF property*), 300

output_shape (*tensorcircuit.applications.van.PixelCNN property*), 330

output_shape (*tensorcircuit.applications.van.ResidualBlock property*), 349

output_shape (*tensorcircuit.applications.vqes.Linear property*), 361

output_shape (*tensorcircuit.keras.QuantumLayer property*), 630

ox() (*tensorcircuit.circuit.Circuit method*), 502

ox() (*tensorcircuit.circuit.Circuit method*), 522

ox() (*tensorcircuit.densitymatrix.DMCircuit method*), 542

ox() (*tensorcircuit.densitymatrix.DMCircuit method*), 560

ox() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 573

ox() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 591

ox() (*tensorcircuit.mpscircuit.MPSCircuit method*), 642

ox() (*tensorcircuit.mpscircuit.MPSCircuit method*), 659

oy() (*tensorcircuit.circuit.Circuit method*), 502

oy() (*tensorcircuit.circuit.Circuit method*), 522

oy() (*tensorcircuit.densitymatrix.DMCircuit method*), 542

oy() (*tensorcircuit.densitymatrix.DMCircuit method*), 560

oy() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 573

oy() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 591

oy() (*tensorcircuit.mpscircuit.MPSCircuit method*), 642

oy() (*tensorcircuit.mpscircuit.MPSCircuit method*), 659

oz() (*tensorcircuit.circuit.Circuit method*), 503

oz() (*tensorcircuit.densitymatrix.DMCircuit method*), 542

oz() (*tensorcircuit.densitymatrix.DMCircuit method*), 560

oz() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 573

oz() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 591

oz() (*tensorcircuit.mpscircuit.MPSCircuit method*), 642

oz() (*tensorcircuit.mpscircuit.MPSCircuit method*), 660

P

parallel_kernel() (*in module tensorcircuit*)

cuit.applications.dqas), 207

`parallel_qaoa_train()` (*in module tensorcircuit.applications.dqas*), 207

`parameter_shift_grad()` (*in module tensorcircuit.experimental*), 601

`parameter_shift_grad_v2()` (*in module tensorcircuit.experimental*), 602

`parameterized_local_measurements()` (*in module tensorcircuit.templates.measurements*), 707

`parameterized_measurements()` (*in module tensorcircuit.templates.measurements*), 707

`parameters()` (*tensorcircuit.torchnn.QuantumNet method*), 717

`partial_trace()` (*tensorcircuit.quantum.QuAdjointVector method*), 672

`partial_trace()` (*tensorcircuit.quantum.QuOperator method*), 676

`partial_trace()` (*tensorcircuit.quantum.QuScalar method*), 679

`partial_trace()` (*tensorcircuit.quantum.QuVector method*), 682

`paulistring()` (*in module tensorcircuit.applications.vqes*), 365

`PauliString2COO()` (*in module tensorcircuit.quantum*), 668

`PauliStringSum2COO()` (*in module tensorcircuit.quantum*), 669

`PauliStringSum2COO_numpy()` (*in module tensorcircuit.quantum*), 669

`PauliStringSum2COO_tf()` (*in module tensorcircuit.quantum*), 669

`PauliStringSum2Dense()` (*in module tensorcircuit.quantum*), 669

`perfect_sampling()` (*tensorcircuit.basecircuit.BaseCircuit method*), 486

`perfect_sampling()` (*tensorcircuit.circuit.Circuit method*), 522

`perfect_sampling()` (*tensorcircuit.densitymatrix.DMCircuit method*), 560

`perfect_sampling()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 591

`perm_matrix()` (*in module tensorcircuit.translation*), 723

`PHASE()` (*tensorcircuit.circuit.Circuit method*), 503

`phase()` (*tensorcircuit.circuit.Circuit method*), 523

`PHASE()` (*tensorcircuit.densitymatrix.DMCircuit method*), 543

`phase()` (*tensorcircuit.densitymatrix.DMCircuit method*), 560

`PHASE()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 574

`phase()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 591

`PHASE()` (*tensorcircuit.mpscircuits.MPSCircuit method*), 642

`phase()` (*tensorcircuit.mpscircuits.MPSCircuit method*), 660

`phase_gate()` (*in module tensorcircuit.gates*), 610

`phasedamping()` (*tensorcircuit.circuit.Circuit method*), 523

`phasedamping()` (*tensorcircuit.densitymatrix.DMCircuit method*), 561

`phasedamping()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 591

`phasedampingchannel()` (*in module tensorcircuit.channels*), 497

`physical_dimensions` (*tensorcircuit.mps_base.FiniteMPS property*), 638

`pivot()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 381

`pivot()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 408

`pivot()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 436

`pivot()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 464

`PixelCNN` (*class in tensorcircuit.applications.van*), 310

`plain_contractor()` (*in module tensorcircuit.cons*), 535

`plain_evaluation()` (*tensorcircuit.applications.vqes.VQNHE method*), 364

`pop()` (*tensorcircuit.channels.KrausList method*), 492

`position()` (*tensorcircuit.mps_base.FiniteMPS method*), 638

`position()` (*tensorcircuit.mpscircuits.MPSCircuit method*), 660

`post_select()` (*tensorcircuit.circuit.Circuit method*), 523

`post_selection()` (*tensorcircuit.circuit.Circuit method*), 523

`power()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 381

`power()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 409

`power()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 436

`power()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 464

`predict()` (*tensorcircuit.applications.van.MADE method*), 247

`predict()` (*tensorcircuit.applications.van.NMF method*), 300

`predict()` (*tensorcircuit.applications.van.PixelCNN method*), 330

`predict_generator()` (*tensorcircuit.applications.van.MADE method*), 248
`predict_generator()` (*tensorcircuit.applications.van.NMF method*), 302
`predict_generator()` (*tensorcircuit.applications.van.PixelCNN method*), 331
`predict_on_batch()` (*tensorcircuit.applications.van.MADE method*), 248
`predict_on_batch()` (*tensorcircuit.applications.van.NMF method*), 302
`predict_on_batch()` (*tensorcircuit.applications.van.PixelCNN method*), 332
`predict_step()` (*tensorcircuit.applications.van.MADE method*), 248
`predict_step()` (*tensorcircuit.applications.van.NMF method*), 302
`predict_step()` (*tensorcircuit.applications.van.PixelCNN method*), 332
`prepend()` (*tensorcircuit.abstractcircuit.AbstractCircuit method*), 202
`prepend()` (*tensorcircuit.basecircuit.BaseCircuit method*), 486
`prepend()` (*tensorcircuit.circuit.Circuit method*), 523
`prepend()` (*tensorcircuit.densitymatrix.DMCircuit method*), 561
`prepend()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 591
`prepend()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 660
`preset_byprob()` (*in module tensorcircuit.applications.dqas*), 207
`probability()` (*tensorcircuit.basecircuit.BaseCircuit method*), 486
`probability()` (*tensorcircuit.circuit.Circuit method*), 523
`probability()` (*tensorcircuit.densitymatrix.DMCircuit method*), 561
`probability()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 592
`probability_sample()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 382
`probability_sample()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 409
`probability_sample()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 436
`probability_sample()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 465
`proj_with_mps()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 660
`projector()` (*tensorcircuit.quantum.QuAdjointVector method*), 672
`projector()` (*tensorcircuit.quantum.QuVector method*), 682
`ps2coo_core()` (*in module tensorcircuit.quantum*), 692
`pseudo_contract_between()` (*in module tensorcircuit.simplify*), 701
`pytorch_interface()` (*in module tensorcircuit.interfaces.torch*), 620
`PyTorchBackend` (*class in tensorcircuit.backends.pytorch_backend*), 421

Q

`q()` (*in module tensorcircuit.applications.vags*), 223
`QAOA_block()` (*in module tensorcircuit.templates.blocks*), 702
`qaoa_block_vag()` (*in module tensorcircuit.applications.vags*), 223
`qaoa_block_vag_energy()` (*in module tensorcircuit.applications.vags*), 224
`qaoa_noise_vag()` (*in module tensorcircuit.applications.vags*), 224
`qaoa_simple_train()` (*in module tensorcircuit.applications.dqas*), 207
`qaoa_train()` (*in module tensorcircuit.applications.vags*), 224
`qaoa_vag()` (*in module tensorcircuit.applications.vags*), 225
`qaoa_vag_energy()` (*in module tensorcircuit.applications.vags*), 225
`qft()` (*in module tensorcircuit.templates.blocks*), 702
`qft_circuit()` (*in module tensorcircuit.applications.vags*), 225
`qft_qem_vag()` (*in module tensorcircuit.applications.vags*), 225
`qir2cirq()` (*in module tensorcircuit.translation*), 723
`qir2json()` (*in module tensorcircuit.translation*), 724
`qir2qiskit()` (*in module tensorcircuit.translation*), 724
`qir2tex()` (*in module tensorcircuit.vis*), 727
`qiskit2tc()` (*in module tensorcircuit.translation*), 725
`qiskit_compile()` (*in module tensorcircuit.compiler.qiskit_compiler*), 534
`qiskit_from_qasm_str_ordered_measure()` (*in module tensorcircuit.translation*), 725
`qng()` (*in module tensorcircuit.experimental*), 602
`qng2()` (*in module tensorcircuit.experimental*), 602
`qop_to_matrix()` (*in module tensorcircuit.interfaces.tensortrans*), 620
`qr()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 382

`qr()` (*tensorcircuit.backends.numpy_backend.NumpyBackend*)
 (*tensorcircuit.method*), 409

`qr()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*)
 (*tensorcircuit.method*), 437

`qr()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*)
 (*tensorcircuit.method*), 465

`QuAdjointVector` (*class in tensorcircuit.quantum*), 670

`quantum_constructor()` (*in module tensorcircuit.quantum*), 692

`quantum_mp_qaoa_vag()` (*in module tensorcircuit.applications.vags*), 225

`quantum_qaoa_vag()` (*in module tensorcircuit.applications.vags*), 225

`QuantumLayer` (*class in tensorcircuit.keras*), 622

`QuantumNet` (*class in tensorcircuit.torchnn*), 709

`quimb2qop()` (*in module tensorcircuit.quantum*), 693

`QuOperator` (*class in tensorcircuit.quantum*), 673

`quoperator()` (*tensorcircuit.circuit.Circuit method*), 523

`QuScalar` (*class in tensorcircuit.quantum*), 677

`QuVector` (*class in tensorcircuit.quantum*), 680

`quvector()` (*tensorcircuit.basecircuit.BaseCircuit method*), 486

`quvector()` (*tensorcircuit.circuit.Circuit method*), 524

`quvector()` (*tensorcircuit.densitymatrix.DMCircuit method*), 561

`quvector()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 592

R

`R()` (*tensorcircuit.circuit.Circuit method*), 503

`r()` (*tensorcircuit.circuit.Circuit method*), 524

`R()` (*tensorcircuit.densitymatrix.DMCircuit method*), 543

`r()` (*tensorcircuit.densitymatrix.DMCircuit method*), 561

`R()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 574

`r()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 592

`R()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 643

`r()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 660

`r_gate()` (*in module tensorcircuit.gates*), 610

`randn()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 382

`randn()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 409

`randn()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 437

`randn()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 466

`random()` (*tensorcircuit.mps_base.FiniteMPS class method*), 638

`random_single_qubit_gate()` (*in module tensorcircuit.gates*), 610

`random_random_split()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 382

`random_random_split()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 409

`random_random_split()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 437

`random_random_split()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 466

`random_two_qubit_gate()` (*in module tensorcircuit.gates*), 610

`random_uniform()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 382

`random_uniform()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 410

`random_uniform()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 437

`random_uniform()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 466

`readouterror_bs()` (*tensorcircuit.basecircuit.BaseCircuit method*), 487

`readouterror_bs()` (*tensorcircuit.circuit.Circuit method*), 524

`readouterror_bs()` (*tensorcircuit.densitymatrix.DMCircuit method*), 561

`readouterror_bs()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 592

`ReadoutMit` (*class in tensorcircuit.results.readout_mitigation*), 697

`real()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 382

`real()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 410

`real()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 438

`real()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 466

`recoverer()` (*tensorcircuit.applications.vqes.VQNHE method*), 364

`recursive_index()` (*in module tensorcircuit.applications.utils*), 220

`reduce_dimension()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 660

`reduce_edges()` (*in module tensorcircuit.applications.graphdata*), 209

`reduce_tensor_dimension()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 382

cuit.mpscircuit.MPSCircuit class method), renyi_free_energy() (in module tensorcircuit.applications.vags), 226

reduced_ansatz() (in module tensorcircuit.applications.graphdata), 209

reduced_cal_matrix() (tensorcircuit.results.readout_mitigation.ReadoutMit method), 700

reduced_density() (tensorcircuit.quantum.QuAdjointVector method), 673

reduced_density() (tensorcircuit.quantum.QuVector method), 683

reduced_density_matrix() (in module tensorcircuit.applications.vags), 226

reduced_density_matrix() (in module tensorcircuit.quantum), 693

register_backward_hook() (tensorcircuit.torchnn.QuantumNet method), 717

register_buffer() (tensorcircuit.torchnn.QuantumNet method), 717

register_forward_hook() (tensorcircuit.torchnn.QuantumNet method), 718

register_forward_pre_hook() (tensorcircuit.torchnn.QuantumNet method), 718

register_full_backward_hook() (tensorcircuit.torchnn.QuantumNet method), 718

register_load_state_dict_post_hook() (tensorcircuit.torchnn.QuantumNet method), 719

register_module() (tensorcircuit.torchnn.QuantumNet method), 719

register_parameter() (tensorcircuit.torchnn.QuantumNet method), 719

regular_graph_generator() (in module tensorcircuit.applications.graphdata), 209

regularization() (tensorcircuit.applications.van.MADE method), 249

regularization() (tensorcircuit.applications.van.MaskedLinear method), 278

relu() (tensorcircuit.backends.jax_backend.JaxBackend method), 383

relu() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 410

relu() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 438

relu() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 466

remove() (tensorcircuit.channels.KrausList method), 492

render_pdf() (in module tensorcircuit.vis), 728

renyi_entropy() (in module tensorcircuit.applications.vags), 226

renyi_entropy() (in module tensorcircuit.quantum), 693

renyi_free_energy() (in module tensorcircuit.quantum), 694

reorder_axes() (tensorcircuit.gates.Gate method), 604

reorder_edges() (tensorcircuit.gates.Gate method), 604

replace_inputs() (tensorcircuit.basecircuit.BaseCircuit method), 487

replace_inputs() (tensorcircuit.circuit.Circuit method), 524

replace_inputs() (tensorcircuit.densitymatrix.DMCircuit method), 562

replace_inputs() (tensorcircuit.densitymatrix.DMCircuit2 method), 592

replace_mps_inputs() (tensorcircuit.circuit.Circuit method), 524

repr2array() (in module tensorcircuit.applications.utils), 220

repr_op() (in module tensorcircuit.applications.dgas), 208

requires_grad_() (tensorcircuit.torchnn.QuantumNet method), 719

reset() (tensorcircuit.circuit.Circuit method), 525

reset() (tensorcircuit.densitymatrix.DMCircuit method), 562

reset() (tensorcircuit.densitymatrix.DMCircuit2 method), 592

reset_instruction() (tensorcircuit.abstractcircuit.AbstractCircuit method), 202

reset_instruction() (tensorcircuit.basecircuit.BaseCircuit method), 487

reset_instruction() (tensorcircuit.circuit.Circuit method), 525

reset_instruction() (tensorcircuit.densitymatrix.DMCircuit method), 562

reset_instruction() (tensorcircuit.densitymatrix.DMCircuit2 method), 592

reset_instruction() (tensorcircuit.mpscircuit.MPSCircuit method), 660

reset_metrics() (tensorcircuit.applications.van.MADE method), 249

reset_metrics() (tensorcircuit.applications.van.NMF method), 302

reset_metrics() (tensorcircuit.applications.van.PixelCNN method), 332

reset_states() (tensorcircuit.applications.van.MADE method), 249

reset_states() (tensorcircuit.applications.van.NMF method), 303

reset_states()	(<i>tensorcircuit</i> . <i>applications.van.PixelCNN</i> method), 332	611
resetchannel()	(in module <i>tensorcircuit.channels</i>), 497	<i>right_envs()</i> (<i>tensorcircuit.mps_base.FiniteMPS</i> method), 638
reshape()	(<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 383	<i>right_shift()</i> (<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 383
reshape()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 410	<i>right_shift()</i> (<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 411
reshape()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 438	<i>right_shift()</i> (<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 438
reshape()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 466	<i>right_shift()</i> (<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 467
reshape2()	(<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 383	<i>right_transfer_operator()</i> (<i>tensorcircuit.mps_base.FiniteMPS</i> method), 638
reshape2()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 410	<i>rq()</i> (<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 383
reshape2()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 438	<i>rq()</i> (<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 411
reshape2()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 466	<i>rq()</i> (<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 439
reshapem()	(<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 383	<i>TensorFlowBackend</i> (<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 467
reshapem()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 410	<i>run_eagerly</i> (<i>tensorcircuit.applications.van.MADE</i> property), 249
reshapem()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 438	<i>run_eagerly</i> (<i>tensorcircuit.applications.van.NMF</i> property), 303
reshapem()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 467	<i>run_eagerly</i> (<i>tensorcircuit.applications.van.PixelCNN</i> property), 332
reshuffle()	(in module <i>tensorcircuit.channels</i>), 497	<i>runtime_backend()</i> (in module <i>tensorcircuit.cons</i>), 536
ResidualBlock	(class in <i>tensorcircuit.applications.van</i>), 340	<i>runtime_contractor()</i> (in module <i>tensorcircuit.cons</i>), 536
return_partial()	(in module <i>tensorcircuit.utils</i>), 726	<i>runtime_dtype()</i> (in module <i>tensorcircuit.cons</i>), 536
reverse()	(<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 383	<i>rx()</i> (<i>tensorcircuit.circuit.Circuit</i> method), 503
reverse()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 410	<i>rx()</i> (<i>tensorcircuit.circuit.Circuit</i> method), 525
reverse()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 438	<i>rx()</i> (<i>tensorcircuit.densitymatrix.DMCircuit</i> method), 543
reverse()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 467	<i>rx()</i> (<i>tensorcircuit.densitymatrix.DMCircuit</i> method), 562
reshuffle()	(in module <i>tensorcircuit.channels</i>), 497	<i>rx()</i> (<i>tensorcircuit.densitymatrix.DMCircuit2</i> method), 574
ResidualBlock	(class in <i>tensorcircuit.applications.van</i>), 340	<i>rx()</i> (<i>tensorcircuit.densitymatrix.DMCircuit2</i> method), 593
return_partial()	(in module <i>tensorcircuit.utils</i>), 726	<i>rx()</i> (<i>tensorcircuit.mpscircuit.MPSCircuit</i> method), 643
reverse()	(<i>tensorcircuit.backends.jax_backend.JaxBackend</i> method), 383	<i>rx()</i> (<i>tensorcircuit.mpscircuit.MPSCircuit</i> method), 660
reverse()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend</i> method), 410	<i>rx_rx_block()</i> (in module <i>tensorcircuit.gates</i>), 611
reverse()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend</i> method), 438	<i>rx_ry_block()</i> (in module <i>tensorcircuit.gates</i>), 216
reverse()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend</i> method), 467	<i>rx_rz_block()</i> (in module <i>tensorcircuit.gates</i>), 216
reverse()	(<i>tensorcircuit.channels.KrausList</i> method), 492	<i>rx_rx_block()</i> (in module <i>tensorcircuit.layers</i>), 216
reverse_count()	(in module <i>tensorcircuit.results.counts</i>), 697	<i>rx_ry_block()</i> (in module <i>tensorcircuit.layers</i>), 216
rgate_theoretical()	(in module <i>tensorcircuit.gates</i>),	<i>rx_rz_block()</i> (in module <i>tensorcircuit.layers</i>), 216

rx_xx_block() (in module `tensorcircuit.applications.layers`), 216
rx_yy_block() (in module `tensorcircuit.applications.layers`), 216
rx_zz_block() (in module `tensorcircuit.applications.layers`), 216
rxylayer() (in module `tensorcircuit.applications.layers`), 216
RXX() (`tensorcircuit.circuit.Circuit` method), 503
rxx() (`tensorcircuit.circuit.Circuit` method), 525
RXX() (`tensorcircuit.densitymatrix.DMCircuit` method), 543
rxx() (`tensorcircuit.densitymatrix.DMCircuit` method), 562
RXX() (`tensorcircuit.densitymatrix.DMCircuit2` method), 574
rxx() (`tensorcircuit.densitymatrix.DMCircuit2` method), 593
RXX() (`tensorcircuit.mpscircuit.MPSCircuit` method), 643
rxx() (`tensorcircuit.mpscircuit.MPSCircuit` method), 661
rxx_gate() (in module `tensorcircuit.gates`), 611
RY() (`tensorcircuit.circuit.Circuit` method), 503
ry() (`tensorcircuit.circuit.Circuit` method), 525
RY() (`tensorcircuit.densitymatrix.DMCircuit` method), 543
ry() (`tensorcircuit.densitymatrix.DMCircuit` method), 562
RY() (`tensorcircuit.densitymatrix.DMCircuit2` method), 574
ry() (`tensorcircuit.densitymatrix.DMCircuit2` method), 593
RY() (`tensorcircuit.mpscircuit.MPSCircuit` method), 643
ry() (`tensorcircuit.mpscircuit.MPSCircuit` method), 661
ry_gate() (in module `tensorcircuit.gates`), 611
ry_rx_block() (in module `tensorcircuit.applications.layers`), 216
ry_ry_block() (in module `tensorcircuit.applications.layers`), 216
ry_rz_block() (in module `tensorcircuit.applications.layers`), 216
ry_xx_block() (in module `tensorcircuit.applications.layers`), 216
ry_yy_block() (in module `tensorcircuit.applications.layers`), 216
ry_zz_block() (in module `tensorcircuit.applications.layers`), 216
rylayer() (in module `tensorcircuit.applications.layers`), 216
RYY() (`tensorcircuit.circuit.Circuit` method), 503
ryy() (`tensorcircuit.circuit.Circuit` method), 525
RYY() (`tensorcircuit.densitymatrix.DMCircuit` method), 543
tensorcir- ryy() (`tensorcircuit.densitymatrix.DMCircuit` method), 562
tensorcir- RYY() (`tensorcircuit.densitymatrix.DMCircuit2` method), 574
tensorcir- ryy() (`tensorcircuit.densitymatrix.DMCircuit2` method), 593
RYY() (`tensorcircuit.mpscircuit.MPSCircuit` method), 643
ryy() (`tensorcircuit.mpscircuit.MPSCircuit` method), 661
ryy_gate() (in module `tensorcircuit.gates`), 612
RZ() (`tensorcircuit.circuit.Circuit` method), 504
rz() (`tensorcircuit.circuit.Circuit` method), 525
RZ() (`tensorcircuit.densitymatrix.DMCircuit` method), 543
rz() (`tensorcircuit.densitymatrix.DMCircuit` method), 562
RZ() (`tensorcircuit.densitymatrix.DMCircuit2` method), 574
rz() (`tensorcircuit.densitymatrix.DMCircuit2` method), 593
RZ() (`tensorcircuit.mpscircuit.MPSCircuit` method), 643
rz() (`tensorcircuit.mpscircuit.MPSCircuit` method), 661
rz_gate() (in module `tensorcircuit.gates`), 612
rz_rx_block() (in module `tensorcircuit.applications.layers`), 216
rz_ry_block() (in module `tensorcircuit.applications.layers`), 216
rz_rz_block() (in module `tensorcircuit.applications.layers`), 216
rz_xx_block() (in module `tensorcircuit.applications.layers`), 216
rz_yy_block() (in module `tensorcircuit.applications.layers`), 217
rz_zz_block() (in module `tensorcircuit.applications.layers`), 217
rzlayer() (in module `tensorcircuit.applications.layers`), 217
RZZ() (`tensorcircuit.circuit.Circuit` method), 504
rzz() (`tensorcircuit.circuit.Circuit` method), 525
RZZ() (`tensorcircuit.densitymatrix.DMCircuit` method), 544
rzz() (`tensorcircuit.densitymatrix.DMCircuit` method), 562
RZZ() (`tensorcircuit.densitymatrix.DMCircuit2` method), 575
rzz() (`tensorcircuit.densitymatrix.DMCircuit2` method), 593
RZZ() (`tensorcircuit.mpscircuit.MPSCircuit` method), 643
rzz() (`tensorcircuit.mpscircuit.MPSCircuit` method), 661
rzz_gate() (in module `tensorcircuit.gates`), 612

S

s() (*tensorcircuit.circuit.Circuit* method), 504
s() (*tensorcircuit.circuit.Circuit* method), 526
s() (*tensorcircuit.densitymatrix.DMCircuit* method), 544
s() (*tensorcircuit.densitymatrix.DMCircuit* method), 563
s() (*tensorcircuit.densitymatrix.DMCircuit2* method), 575
s() (*tensorcircuit.densitymatrix.DMCircuit2* method), 593
s() (*tensorcircuit.mpscircuit.MPSCircuit* method), 643
s() (*tensorcircuit.mpscircuit.MPSCircuit* method), 661
sample() (*tensorcircuit.applications.van.MADE* method), 249
sample() (*tensorcircuit.applications.van.NMF* method), 303
sample() (*tensorcircuit.applications.van.PixelCNN* method), 333
sample() (*tensorcircuit.basecircuit.BaseCircuit* method), 487
sample() (*tensorcircuit.circuit.Circuit* method), 526
sample() (*tensorcircuit.densitymatrix.DMCircuit* method), 563
sample() (*tensorcircuit.densitymatrix.DMCircuit2* method), 593
sample2all() (in module *tensorcircuit.quantum*), 694
sample2count() (in module *tensorcircuit.quantum*), 694
sample_bin2int() (in module *tensorcircuit.quantum*), 695
sample_expectation_ps() (*tensorcircuit.basecircuit.BaseCircuit* method), 488
sample_expectation_ps() (*tensorcircuit.circuit.Circuit* method), 526
sample_expectation_ps() (*tensorcircuit.densitymatrix.DMCircuit* method), 563
sample_expectation_ps() (*tensorcircuit.densitymatrix.DMCircuit2* method), 594
sample_expectation_ps_noisy() (in module *tensorcircuit.noisemodel*), 668
sample_int2bin() (in module *tensorcircuit.quantum*), 695
save() (*tensorcircuit.applications.van.MADE* method), 249
save() (*tensorcircuit.applications.van.NMF* method), 303
save() (*tensorcircuit.applications.van.PixelCNN* method), 333
save() (*tensorcircuit.applications.vqes.VQNHE* method), 364
save() (*tensorcircuit.mps_base.FiniteMPS* method), 638
save_func() (in module *tensorcircuit.keras*), 634
save_spec() (*tensorcircuit.applications.van.MADE* method), 250
save_spec() (*tensorcircuit.applications.van.NMF* method), 303
save_spec() (*tensorcircuit.applications.van.PixelCNN* method), 333
save_weights() (*tensorcircuit.applications.van.MADE* method), 250
save_weights() (*tensorcircuit.applications.van.NMF* method), 304
save_weights() (*tensorcircuit.applications.van.PixelCNN* method), 334
scatter() (*tensorcircuit.backends.jax_backend.JaxBackend* method), 383
scatter() (*tensorcircuit.backends.numpy_backend.NumpyBackend* method), 411
scatter() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend* method), 439
scatter() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend* method), 467
scipy_interface() (in module *tensorcircuit.interfaces.scipy*), 614
scipy_optimize_interface() (in module *tensorcircuit.interfaces.scipy*), 615
SD() (*tensorcircuit.circuit.Circuit* method), 504
sd() (*tensorcircuit.circuit.Circuit* method), 527
SD() (*tensorcircuit.densitymatrix.DMCircuit* method), 544
sd() (*tensorcircuit.densitymatrix.DMCircuit* method), 564
SD() (*tensorcircuit.densitymatrix.DMCircuit2* method), 575
sd() (*tensorcircuit.densitymatrix.DMCircuit2* method), 595
SD() (*tensorcircuit.mpscircuit.MPSCircuit* method), 644
sd() (*tensorcircuit.mpscircuit.MPSCircuit* method), 661
sgd() (*tensorcircuit.circuit.Circuit* method), 527
sgd() (*tensorcircuit.densitymatrix.DMCircuit* method), 564
sgd() (*tensorcircuit.densitymatrix.DMCircuit2* method), 595
sdg() (*tensorcircuit.mpscircuit.MPSCircuit* method), 662
searchsorted() (*tensorcircuit.backends.jax_backend.JaxBackend* method), 384
searchsorted() (*tensorcircuit.backends.numpy_backend.NumpyBackend* method), 411
searchsorted() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend* method), 439
searchsorted() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend* method), 468

select_gate() (tensorcircuit.abstractcircuit.AbstractCircuit method), 202

select_gate() (tensorcircuit.basecircuit.BaseCircuit method), 489

select_gate() (tensorcircuit.circuit.Circuit method), 528

select_gate() (tensorcircuit.densitymatrix.DMCircuit method), 565

select_gate() (tensorcircuit.densitymatrix.DMCircuit2 method), 595

select_gate() (tensorcircuit.mpscircuit.MPSCircuit method), 662

serialize_tensor() (tensorcircuit.backends.jax_backend.JaxBackend method), 384

serialize_tensor() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 411

serialize_tensor() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 439

serialize_tensor() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 468

set_backend() (in module tensorcircuit.cons), 536

set_contractor() (in module tensorcircuit.cons), 537

set_dtype() (in module tensorcircuit.cons), 537

set_extra_state() (tensorcircuit.torchnn.QuantumNet method), 719

set_function_backend() (in module tensorcircuit.cons), 537

set_function_contractor() (in module tensorcircuit.cons), 538

set_function_dtype() (in module tensorcircuit.cons), 538

set_name() (tensorcircuit.gates.Gate method), 604

set_op_pool() (in module tensorcircuit.applications.dgas), 208

set_random_state() (tensorcircuit.backends.jax_backend.JaxBackend method), 384

set_random_state() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 411

set_random_state() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 440

set_random_state() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 468

set_split_rules() (tensorcircuit.mpscircuit.MPSCircuit method), 662

set_tensor() (tensorcircuit.gates.Gate method), 604

set_tensornetwork_backend() (in module tensorcircuit.cons), 538

set_weights() (tensorcircuit.applications.van.MADE method), 251

set_weights() (tensorcircuit.applications.van.MaskedConv2D method), 266

set_weights() (tensorcircuit.applications.van.MaskedLinear method), 278

set_weights() (tensorcircuit.applications.van.NMF method), 305

set_weights() (tensorcircuit.applications.van.PixelCNN method), 335

set_weights() (tensorcircuit.applications.van.ResidualBlock method), 349

set_weights() (tensorcircuit.applications.vqes.Linear method), 361

set_weights() (tensorcircuit.keras.QuantumLayer method), 631

sexpps() (tensorcircuit.basecircuit.BaseCircuit method), 489

sexpps() (tensorcircuit.circuit.Circuit method), 528

sexpps() (tensorcircuit.densitymatrix.DMCircuit method), 565

sexpps() (tensorcircuit.densitymatrix.DMCircuit2 method), 596

sgates (tensorcircuit.abstractcircuit.AbstractCircuit attribute), 202

sgates (tensorcircuit.basecircuit.BaseCircuit attribute), 490

sgates (tensorcircuit.circuit.Circuit attribute), 529

sgates (tensorcircuit.densitymatrix.DMCircuit attribute), 566

sgates (tensorcircuit.densitymatrix.DMCircuit2 attribute), 597

sgates (tensorcircuit.mpscircuit.MPSCircuit attribute), 662

shape (tensorcircuit.gates.Gate property), 604

shape_concat() (tensorcircuit.backends.jax_backend.JaxBackend method), 384

shape_concat() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 411

shape_concat() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 440

shape_concat() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 468

shape_prod()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 384	sin()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 385
shape_prod()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 412	sin()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 412
shape_prod()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 440	sin()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 440
shape_prod()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 468	sin()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 469
shape_tensor()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 384	single_generator()	(in module <i>tensorcircuit.applications.dqas</i>), 208
shape_tensor()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 412	single_qubit_kraus_identity_check()	(in module <i>tensorcircuit.channels</i>), 498
shape_tensor()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 440	sinh()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 385
shape_tensor()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 468	sinh()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 412
shape_tuple()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 384	sinh()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 440
shape_tuple()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 412	sinh()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 469
shape_tuple()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 440	size()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 385
shape_tuple()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 468	size()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 412
share_memory()	(<i>tensorcircuit.torchnn.QuantumNet method</i>), 720	size()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 440
sigmoid()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 384	size()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 469
sigmoid()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 412	sizend()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 385
sigmoid()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 440	sizend()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 412
sigmoid()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 468	sizend()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 441
sign()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 385	slice()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 385
sign()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 412	slice()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 412
sign()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 440	slice()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 441
sign()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 469	slice()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 469
softmax()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 385	softmax()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 385
softmax()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 413	softmax()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 413
softmax()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 441	softmax()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 441
softmax()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 469	softmax()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 469

solve() (*tensorcircuit.backends.jax_backend.JaxBackend* `split_rules()` (*in module tensorcircuit.cons*), 538
 method), 386
 split_tensor() (*in module tensorcircuit.mpscircuit*),
solve() (*tensorcircuit.backends.numpy_backend.NumpyBackend* 666
 method), 413
 sqrt() (*tensorcircuit.backends.jax_backend.JaxBackend*
 method), 386
 sqrt() (*tensorcircuit.backends.numpy_backend.NumpyBackend*
 method), 413
 sqrt() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*
 method), 441
 sqrt() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*
 method), 470
sort() (*tensorcircuit.channels.KrausList* *method*), 492
sort_count() (*in module tensorcircuit.results.counts*),
 697
space (*tensorcircuit.quantum.QuAdjointVector* *property*), 673
space (*tensorcircuit.quantum.QuVector* *property*), 683
sparse_dense_matmul() (*tensorcircuit.backends.jax_backend.JaxBackend*
 method), 386
sparse_dense_matmul() (*tensorcircuit.backends.numpy_backend.NumpyBackend*
 method), 413
sparse_dense_matmul() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*
 method), 441
sparse_dense_matmul() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*
 method), 470
sparse_expectation() (*in module tensorcircuit.templates.measurements*), 708
sparse_shape (*tensorcircuit.gates.Gate* *property*), 604
sparse_shape() (*tensorcircuit.backends.jax_backend.JaxBackend*
 method), 386
sparse_shape() (*tensorcircuit.backends.numpy_backend.NumpyBackend*
 method), 413
sparse_shape() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*
 method), 442
sparse_shape() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*
 method), 470
spin_by_basis() (*in module tensorcircuit.quantum*),
 695
spin_glass_measurements() (*in module tensorcircuit.templates.measurements*), 708
split (*tensorcircuit.basecircuit.BaseCircuit* *attribute*),
 490
split (*tensorcircuit.circuit.Circuit* *attribute*), 529
split (*tensorcircuit.densitymatrix.DMCircuit* *attribute*),
 566
split (*tensorcircuit.densitymatrix.DMCircuit2* *attribute*),
 597
split_ansatz() (*in module tensorcircuit.applications.graphdata*), 209
 state() (*tensorcircuit.circuit.Circuit* *method*), 529
 state() (*tensorcircuit.densitymatrix.DMCircuit*
 method), 566
 state() (*tensorcircuit.densitymatrix.DMCircuit2*
 method), 597
 state() (*tensorcircuit.mpscircuit.MPSCircuit* *method*),
 662
 state_centric() (*in module tensorcircuit.templates.blocks*), 702
 state_dict() (*tensorcircuit.torchnn.QuantumNet*

method), 720

state_updates (*tensorcircuit.applications.van.MADE property*), 252

state_updates (*tensorcircuit.applications.van.NMF property*), 305

state_updates (*tensorcircuit.applications.van.PixelCNN property*), 335

stateful (*tensorcircuit.applications.van.MADE property*), 252

stateful (*tensorcircuit.applications.van.MaskedConv2D property*), 267

stateful (*tensorcircuit.applications.van.MaskedLinear property*), 278

stateful (*tensorcircuit.applications.van.NMF property*), 306

stateful (*tensorcircuit.applications.van.PixelCNN property*), 335

stateful (*tensorcircuit.applications.van.ResidualBlock property*), 350

stateful (*tensorcircuit.applications.vqes.Linear property*), 362

stateful (*tensorcircuit.keras.QuantumLayer property*), 632

stateful_randc() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 386

stateful_randc() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 414

stateful_randc() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 442

stateful_randc() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 470

stateful_randn() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 387

stateful_randn() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 414

stateful_randn() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 442

stateful_randn() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 471

stateful_randu() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 387

stateful_randu() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 414

std() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 387

std() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 443

std() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 471

stop_gradient() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 388

stop_gradient() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 415

stop_gradient() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 443

stop_gradient() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 471

stop_gradient() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 472

submodules (*tensorcircuit.applications.van.MADE property*), 252

submodules (*tensorcircuit.applications.van.MaskedConv2D property*), 267

submodules (*tensorcircuit.applications.van.MaskedLinear property*), 278

submodules (*tensorcircuit.applications.van.NMF property*), 306

submodules (*tensorcircuit.applications.van.PixelCNN property*), 335

submodules (*tensorcircuit.applications.van.ResidualBlock property*), 350

submodules (*tensorcircuit.applications.vqes.Linear property*), 362

submodules (*tensorcircuit.keras.QuantumLayer property*), 632

subsystem_edges (*tensorcircuit.quantum.QuAdjointVector property*), 673

subsystem_edges (*tensorcircuit.quantum.QuVector property*), 683

subtraction() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 388

subtraction() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 443

cuit.backends.numpy_backend.NumpyBackend method), 415

subtraction() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 443*

subtraction() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 472*

sum() (*tensorcircuit.backends.jax_backend.JaxBackend method), 388*

sum() (*tensorcircuit.backends.numpy_backend.NumpyBackend method), 415*

sum() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 443*

sum() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 472*

summary() (*tensorcircuit.applications.van.MADE method), 252*

summary() (*tensorcircuit.applications.van.NMF method), 306*

summary() (*tensorcircuit.applications.van.PixelCNN method), 336*

super_to_choi() (*in module tensorcircuit.channels*), 498

super_to_kraus() (*in module tensorcircuit.channels*), 498

supports_masking (*tensorcircuit.applications.van.MADE property*), 253

supports_masking (*tensorcircuit.applications.van.MaskedConv2D property*), 267

supports_masking (*tensorcircuit.applications.van.MaskedLinear property*), 279

supports_masking (*tensorcircuit.applications.van.NMF property*), 306

supports_masking (*tensorcircuit.applications.van.PixelCNN property*), 336

supports_masking (*tensorcircuit.applications.van.ResidualBlock property*), 350

supports_masking (*tensorcircuit.applications.vqes.Linear property*), 363

supports_masking (*tensorcircuit.keras.QuantumLayer property*), 632

svd() (*tensorcircuit.backends.jax_backend.JaxBackend method), 388*

svd() (*tensorcircuit.backends.numpy_backend.NumpyBackend method), 415*

svd() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 443*

svd() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 472*

method), 472

SWAP() (*tensorcircuit.circuit.Circuit method), 504*

swap() (*tensorcircuit.circuit.Circuit method), 529*

SWAP() (*tensorcircuit.densitymatrix.DMCircuit method), 544*

swap() (*tensorcircuit.densitymatrix.DMCircuit method), 566*

SWAP() (*tensorcircuit.densitymatrix.DMCircuit2 method), 575*

swap() (*tensorcircuit.densitymatrix.DMCircuit2 method), 597*

SWAP() (*tensorcircuit.mpscircuits.MPSCircuit method), 644*

swap() (*tensorcircuit.mpscircuits.MPSCircuit method), 652*

switch() (*tensorcircuit.backends.jax_backend.JaxBackend method), 389*

switch() (*tensorcircuit.backends.numpy_backend.NumpyBackend method), 416*

switch() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 444*

switch() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 473*

T

T() (*tensorcircuit.circuit.Circuit method), 504*

t() (*tensorcircuit.circuit.Circuit method), 529*

T() (*tensorcircuit.densitymatrix.DMCircuit method), 544*

t() (*tensorcircuit.densitymatrix.DMCircuit method), 566*

T() (*tensorcircuit.densitymatrix.DMCircuit2 method), 575*

t() (*tensorcircuit.densitymatrix.DMCircuit2 method), 597*

T() (*tensorcircuit.mpscircuits.MPSCircuit method), 644*

t() (*tensorcircuit.mpscircuits.MPSCircuit method), 663*

T_destination (*tensorcircuit.torchnn.QuantumNet attribute*), 709

tan() (*tensorcircuit.backends.jax_backend.JaxBackend method), 389*

tan() (*tensorcircuit.backends.numpy_backend.NumpyBackend method), 416*

tan() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 444*

tan() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 473*

tanh() (*tensorcircuit.backends.jax_backend.JaxBackend method), 389*

tanh() (*tensorcircuit.backends.numpy_backend.NumpyBackend method), 416*

tanh() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 445*

tanh() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 473*

taylorlnm()	(in module <code>tensorcircuit.applications.vags</code>)	226	<code>tensorcircuit.applications.vags</code>	module, 221
taylorlnm()	(in module <code>tensorcircuit.quantum</code>)	695	<code>tensorcircuit.applications.van</code>	module, 227
TD()	(<code>tensorcircuit.circuit.Circuit</code> method)	504	<code>tensorcircuit.applications.vqes</code>	module, 352
td()	(<code>tensorcircuit.circuit.Circuit</code> method)	530	<code>tensorcircuit.backends.backend_factory</code>	module, 365
TD()	(<code>tensorcircuit.densitymatrix.DMCircuit</code> method)	544	<code>tensorcircuit.backends.jax_backend</code>	module, 365
td()	(<code>tensorcircuit.densitymatrix.DMCircuit</code> method)	567	<code>tensorcircuit.backends.numpy_backend</code>	module, 394
TD()	(<code>tensorcircuit.densitymatrix.DMCircuit2</code> method)	575	<code>tensorcircuit.backends.pytorch_backend</code>	module, 421
td()	(<code>tensorcircuit.densitymatrix.DMCircuit2</code> method)	597	<code>tensorcircuit.backends.tensorflow_backend</code>	module, 449
TD()	(<code>tensorcircuit.mpscircuit.MPSCircuit</code> method)	644	<code>tensorcircuit.basecircuit</code>	module, 478
td()	(<code>tensorcircuit.mpscircuit.MPSCircuit</code> method)	663	<code>tensorcircuit.channels</code>	module, 492
tdg()	(<code>tensorcircuit.circuit.Circuit</code> method)	530	<code>tensorcircuit.circuit</code>	module, 499
tdg()	(<code>tensorcircuit.densitymatrix.DMCircuit</code> method)	567	<code>tensorcircuit.compiler.qiskit_compiler</code>	module, 534
tdg()	(<code>tensorcircuit.densitymatrix.DMCircuit2</code> method)	598	<code>tensorcircuit.cons</code>	module, 535
tdg()	(<code>tensorcircuit.mpscircuit.MPSCircuit</code> method)	663	<code>tensorcircuit.densitymatrix</code>	module, 539
tensor	(<code>tensorcircuit.gates.Gate</code> property)	604	<code>tensorcircuit.experimental</code>	module, 601
tensor_from_edge_order()	(<code>tensorcircuit.gates.Gate</code> method)	604	<code>tensorcircuit.gates</code>	module, 602
tensor_product()	(<code>tensorcircuit.quantum.QuAdjointVector</code> method)	673	<code>tensorcircuit.interfaces.numpy</code>	module, 613
tensor_product()	(<code>tensorcircuit.quantum.QuOperator</code> method)	676	<code>tensorcircuit.interfaces.scipy</code>	module, 614
tensor_product()	(<code>tensorcircuit.quantum.QuScalar</code> method)	679	<code>tensorcircuit.interfaces.tensorflow</code>	module, 616
tensor_product()	(<code>tensorcircuit.quantum.QuVector</code> method)	683	<code>tensorcircuit.interfaces.tensortrans</code>	module, 618
tensor_to_backend_jittable()	(in module <code>tensorcircuit.interfaces.tensortrans</code>)	620	<code>tensorcircuit.interfaces.torch</code>	module, 620
tensor_to_dlpack()	(in module <code>tensorcircuit.interfaces.tensortrans</code>)	620	<code>tensorcircuit.keras</code>	module, 622
tensor_to_dtype()	(in module <code>tensorcircuit.interfaces.tensortrans</code>)	620	<code>tensorcircuit.mps_base</code>	module, 635
tensor_to_json()	(in module <code>circuit.translation</code>)	725	<code>tensorcircuit.mpscircuit</code>	module, 639
tensor_to_numpy()	(in module <code>circuit.interfaces.tensortrans</code>)	620	<code>tensorcircuit.noisemodel</code>	module, 666
tensorcircuit.abstractcircuit	module, 195		<code>tensorcircuit.quantum</code>	module, 668
tensorcircuit.applications.dqas	module, 204		<code>tensorcircuit.results.counts</code>	module, 697
tensorcircuit.applications.graphdata	module, 208			
tensorcircuit.applications.layers	module, 209			
tensorcircuit.applications.utils	module, 220			

tensorcircuit.results.readout_mitigation
 module, 697

tensorcircuit.simplify
 module, 700

tensorcircuit.templates.blocks
 module, 701

tensorcircuit.templates.chems
 module, 703

tensorcircuit.templates.dataset
 module, 703

tensorcircuit.templates.graphs
 module, 703

tensorcircuit.templates.measurements
 module, 704

tensorcircuit.torchnn
 module, 709

tensorcircuit.translation
 module, 723

tensorcircuit.utils
 module, 726

tensorcircuit.vis
 module, 727

tensordot()
 (tensorcircuit.backends.jax_backend.JaxBackend
 method), 389

tensordot()
 (tensorcircuit.backends.numpy_backend.NumpyBackend
 method), 416

tensordot()
 (tensorcircuit.backends.pytorch_backend.PyTorchBackend
 method), 445

tensordot()
 (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend
 method), 473

tensorflow_interface()
 (in module tensorcircuit.interfaces.tensorflow), 616

TensorFlowBackend
 (class in tensorcircuit.backends.tensorflow_backend), 449

test_on_batch()
 (tensorcircuit.applications.van.MADE
 method), 253

test_on_batch()
 (tensorcircuit.applications.van.NMF
 method), 306

test_on_batch()
 (tensorcircuit.applications.van.PixelCNN
 method), 336

test_step()
 (tensorcircuit.applications.van.MADE
 method), 254

test_step()
 (tensorcircuit.applications.van.NMF
 method), 307

test_step()
 (tensorcircuit.applications.van.PixelCNN
 method), 337

tex()
 (tensorcircuit.abstractcircuit.AbstractCircuit
 method), 202

tex()
 (tensorcircuit.basecircuit.BaseCircuit
 method), 490

tex()
 (tensorcircuit.circuit.Circuit
 method), 530

tex()
 (tensorcircuit.densitymatrix.DMCircuit
 method), 567

tex()
 (tensorcircuit.densitymatrix.DMCircuit2
 method), 598

tex()
 (tensorcircuit.mpscircuit.MPSCircuit
 method), 663

tf_dtype()
 (in module tensorcircuit.interfaces.tensorflow), 617

tf_interface()
 (in module tensorcircuit.interfaces.tensorflow), 617

tf_wrapper()
 (in module tensorcircuit.interfaces.tensorflow), 617

TFIM1Denergy()
 (in module tensorcircuit.applications.utils), 220

tfim_measurements()
 (in module tensorcircuit.applications.vags), 226

tfim_measurements_tc()
 (in module tensorcircuit.applications.vags), 226

thermalrelaxation()
 (tensorcircuit.circuit.Circuit
 method), 530

thermalrelaxation()
 (tensorcircuit.densitymatrix.DMCircuit
 method), 567

thermalrelaxation()
 (tensorcircuit.densitymatrix.DMCircuit2
 method), 598

thermalrelaxationchannel()
 (in module tensorcircuit.channels), 498

tile()
 (tensorcircuit.backends.jax_backend.JaxBackend
 method), 389

tile()
 (tensorcircuit.backends.numpy_backend.NumpyBackend
 method), 417

tile()
 (tensorcircuit.backends.pytorch_backend.PyTorchBackend
 method), 445

tile()
 (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend
 method), 473

tn2qop()
 (in module tensorcircuit.quantum), 695

tn_greedy_contractor()
 (in module tensorcircuit.cons), 539

to()
 (tensorcircuit.torchnn.QuantumNet
 method), 720

to_circuit()
 (tensorcircuit.densitymatrix.DMCircuit
 method), 567

to_circuit()
 (tensorcircuit.densitymatrix.DMCircuit2
 method), 598

to_cirq()
 (tensorcircuit.abstractcircuit.AbstractCircuit
 method), 202

to_cirq()
 (tensorcircuit.basecircuit.BaseCircuit
 method), 490

to_cirq()
 (tensorcircuit.circuit.Circuit
 method), 530

to_cirq()
 (tensorcircuit.densitymatrix.DMCircuit
 method), 567

to_cirq()
 (tensorcircuit.densitymatrix.DMCircuit2
 method), 598

to_cirq()	(<i>tensorcircuit.mpscircuit.MPSCircuit method</i>), 663	to_openqasm()	(<i>tensorcircuit.abstractcircuit.AbstractCircuit method</i>), 203
to_dense()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 389	to_openqasm()	(<i>tensorcircuit.basecircuit.BaseCircuit method</i>), 491
to_dense()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 417	to_openqasm()	(<i>tensorcircuit.circuit.Circuit method</i>), 530
to_dense()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 445	to_openqasm()	(<i>tensorcircuit.densitymatrix.DMCircuit method</i>), 568
to_dense()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 474	to_openqasm()	(<i>tensorcircuit.densitymatrix.DMCircuit2 method</i>), 599
to_dlpack()	(<i>tensorcircuit.backends.jax_backend.JaxBackend method</i>), 390	to_openqasm()	(<i>tensorcircuit.mpscircuit.MPSCircuit method</i>), 663
to_dlpack()	(<i>tensorcircuit.backends.numpy_backend.NumpyBackend method</i>), 417	to_qir()	(<i>tensorcircuit.abstractcircuit.AbstractCircuit method</i>), 203
to_dlpack()	(<i>tensorcircuit.backends.pytorch_backend.PyTorchBackend method</i>), 445	to_qir()	(<i>tensorcircuit.basecircuit.BaseCircuit method</i>), 491
to_dlpack()	(<i>tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method</i>), 474	to_qir()	(<i>tensorcircuit.circuit.Circuit method</i>), 531
to_empty()	(<i>tensorcircuit.torchnn.QuantumNet method</i>), 722	to_qir()	(<i>tensorcircuit.densitymatrix.DMCircuit method</i>), 568
to_graphviz()	(<i>tensorcircuit.basecircuit.BaseCircuit method</i>), 490	to_qir()	(<i>tensorcircuit.mpscircuit.MPSCircuit method</i>), 663
to_graphviz()	(<i>tensorcircuit.circuit.Circuit method</i>), 530	to_qiskit()	(<i>tensorcircuit.abstractcircuit.AbstractCircuit method</i>), 203
to_graphviz()	(<i>tensorcircuit.densitymatrix.DMCircuit method</i>), 567	to_qiskit()	(<i>tensorcircuit.basecircuit.BaseCircuit method</i>), 491
to_graphviz()	(<i>tensorcircuit.densitymatrix.DMCircuit2 method</i>), 598	to_qiskit()	(<i>tensorcircuit.circuit.Circuit method</i>), 531
to_json()	(<i>tensorcircuit.abstractcircuit.AbstractCircuit method</i>), 202	to_qiskit()	(<i>tensorcircuit.densitymatrix.DMCircuit method</i>), 568
to_json()	(<i>tensorcircuit.applications.van.MADE method</i>), 254	to_qiskit()	(<i>tensorcircuit.densitymatrix.DMCircuit2 method</i>), 599
to_json()	(<i>tensorcircuit.applications.van.NMF method</i>), 307	to_qiskit()	(<i>tensorcircuit.mpscircuit.MPSCircuit method</i>), 664
to_json()	(<i>tensorcircuit.applications.van.PixelCNN method</i>), 337	to_serial_dict()	(<i>tensorcircuit.gates.Gate method</i>), 604
to_json()	(<i>tensorcircuit.basecircuit.BaseCircuit method</i>), 490	to_yaml()	(<i>tensorcircuit.applications.van.MADE method</i>), 254
to_json()	(<i>tensorcircuit.circuit.Circuit method</i>), 530	to_yaml()	(<i>tensorcircuit.applications.van.NMF method</i>), 307
to_json()	(<i>tensorcircuit.densitymatrix.DMCircuit method</i>), 567	to_yaml()	(<i>tensorcircuit.applications.van.PixelCNN method</i>), 337
to_json()	(<i>tensorcircuit.densitymatrix.DMCircuit2 method</i>), 598	TOFFOLI()	(<i>tensorcircuit.circuit.Circuit method</i>), 505
to_json()	(<i>tensorcircuit.mpscircuit.MPSCircuit method</i>), 663	toffoli()	(<i>tensorcircuit.circuit.Circuit method</i>), 531
		TOFFOLI()	(<i>tensorcircuit.densitymatrix.DMCircuit method</i>), 544
		toffoli()	(<i>tensorcircuit.densitymatrix.DMCircuit method</i>), 569
		TOFFOLI()	(<i>tensorcircuit.densitymatrix.DMCircuit2 method</i>), 575
		toffoli()	(<i>tensorcircuit.densitymatrix.DMCircuit2</i>

method), 599
TOFFOLI() (tensorcircuit.mpscircuit.MPSCircuit method), 644
toffoli() (tensorcircuit.mpscircuit.MPSCircuit method), 664
torch_interface() (in module tensorcircuit.interfaces.torch), 621
torch_optimizer (class in tensorcircuit.backends.pytorch_backend), 449
TorchLayer (in module tensorcircuit.torchnn), 723
torchlib (in module tensorcircuit.backends.pytorch_backend), 449
trace() (tensorcircuit.backends.jax_backend.JaxBackend method), 390
trace() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 417
trace() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 445
trace() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 474
trace() (tensorcircuit.quantum.QuAdjointVector method), 673
trace() (tensorcircuit.quantum.QuOperator method), 677
trace() (tensorcircuit.quantum.QuScalar method), 680
trace() (tensorcircuit.quantum.QuVector method), 683
trace_distance() (in module tensorcircuit.applications.vags), 226
trace_distance() (in module tensorcircuit.quantum), 696
trace_product() (in module tensorcircuit.quantum), 696
train() (tensorcircuit.torchnn.QuantumNet method), 722
train_on_batch() (tensorcircuit.applications.van.MADE method), 254
train_on_batch() (tensorcircuit.applications.van.NMF method), 308
train_on_batch() (tensorcircuit.applications.van.PixelCNN method), 338
train_qml_vag() (in module tensorcircuit.applications.utils), 221
train_step() (tensorcircuit.applications.van.MADE method), 255
train_step() (tensorcircuit.applications.van.NMF method), 308
train_step() (tensorcircuit.applications.van.PixelCNN method), 338
trainable (tensorcircuit.applications.van.MADE property), 255
trainable (tensorcircuit.applications.van.MaskedConv2D property), 267
trainable_variables (tensorcircuit.applications.van.MADE property), 255
trainable_variables (tensorcircuit.applications.van.MaskedConv2D property), 267
trainable_variables (tensorcircuit.applications.van.ResidualBlock property), 350
trainable_variables (tensorcircuit.applications.van.PixelCNN property), 339
trainable_variables (tensorcircuit.applications.van.MADE property), 255
trainable_weights (tensorcircuit.applications.van.MADE property), 255
trainable_weights (tensorcircuit.applications.van.PixelCNN property), 339
trainable_weights (tensorcircuit.applications.van.ResidualBlock property), 351
trainable_weights (tensorcircuit.applications.van.PixelCNN property), 338
trainable_weights (tensorcircuit.applications.van.MADE property), 255
trainable_weights (tensorcircuit.keras.QuantumLayer property), 632

t
training (*tensorcircuit.torchnn.QuantumNet attribute*), 722
training() (*tensorcircuit.applications.vqes.VQNHE method*), 364
transpose() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 390
transpose() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 417
transpose() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 446
transpose() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 474
tree_flatten() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 390
tree_flatten() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 418
tree_flatten() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 446
tree_flatten() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 474
tree_map() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 390
tree_map() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 418
tree_map() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 446
tree_map() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 474
tree_unflatten() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 390
tree_unflatten() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 418
tree_unflatten() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 446
tree_unflatten() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 475
Triangle2D() (*in module tensorcircuit.applications.graphdata*), 208
truncated_free_energy() (*in module tensorcircuit.applications.vags*), 226
truncated_free_energy() (*in module tensorcircuit.quantum*), 696
two2one() (*tensorcircuit.templates.graphs.Grid2DCoord method*), 704
type() (*tensorcircuit.torchnn.QuantumNet method*), 722

U

U() (*tensorcircuit.circuit.Circuit method*), 505
u() (*tensorcircuit.circuit.Circuit method*), 532
U() (*tensorcircuit.densitymatrix.DMCircuit method*), 545
u() (*tensorcircuit.densitymatrix.DMCircuit method*), 569
U() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 576
u() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 600
U() (*tensorcircuit.mpscircuits.MPSCircuit method*), 644
u() (*tensorcircuit.mpscircuits.MPSCircuit method*), 664
u_gate() (*in module tensorcircuit.gates*), 612
ubs() (*tensorcircuit.results.readout_mitigation.ReadoutMit method*), 700
unique_with_counts() (*tensorcircuit.backends.jax_backend.JaxBackend method*), 391
unique_with_counts() (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 418
unique_with_counts() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 446
unique_with_counts() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 475
unitary() (*tensorcircuit.circuit.Circuit method*), 532
unitary() (*tensorcircuit.densitymatrix.DMCircuit method*), 569
unitary() (*tensorcircuit.densitymatrix.DMCircuit2 method*), 600
unitary() (*tensorcircuit.mpscircuits.MPSCircuit method*), 665
unitary_design() (*in module tensorcircuit.applications.vags*), 226
unitary_design_block() (*in module tensorcircuit.applications.vags*), 226
unitary_kraus() (*tensorcircuit.circuit.Circuit method*), 532
unitary_kraus2() (*tensorcircuit.circuit.Circuit method*), 532
update() (*tensorcircuit.backends.jax_backend.optax_optimizer method*), 393
update() (*tensorcircuit.backends.pytorch_backend.torch_optimizer method*), 449

update() (*tensorcircuit.backends.tensorflow_backend.keras.variable*)
 (*tensorcircuit.backends.tensorflow_backend.keras.variable*.*dtype*), 477
updates (*tensorcircuit.applications.van.MADE* property), 255
updates (*tensorcircuit.applications.van.MaskedConv2D* property), 267
updates (*tensorcircuit.applications.van.MaskedLinear* property), 279
updates (*tensorcircuit.applications.van.NMF* property), 309
updates (*tensorcircuit.applications.van.PixelCNN* property), 339
updates (*tensorcircuit.applications.van.ResidualBlock* property), 351
updates (*tensorcircuit.applications.vqes.Linear* property), 363
updates (*tensorcircuit.keras.QuantumLayer* property), 632

V

validate_qml_vag() (in module *tensorcircuit.applications.utils*), 221
value_and_grad() (*tensorcircuit.backends.jax_backend.JaxBackend*.*method*), 391
value_and_grad() (*tensorcircuit.backends.numpy_backend.NumpyBackend*.*method*), 418
value_and_grad() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*.*method*), 446
value_and_grad() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*.*method*), 475
van_regularization() (in module *tensorcircuit.applications.dqas*), 208
van_sample() (in module *tensorcircuit.applications.dqas*), 208
variable_dtype (*tensorcircuit.applications.van.MADE* property), 255
variable_dtype (*tensorcircuit.applications.van.MaskedConv2D* property), 267
variable_dtype (*tensorcircuit.applications.van.MaskedLinear* property), 279
variable_dtype (*tensorcircuit.applications.van.NMF* property), 309
variable_dtype (*tensorcircuit.applications.van.PixelCNN* property), 339
variable_dtype (*tensorcircuit.applications.van.ResidualBlock* property), 351

variable_dtype (*tensorcircuit.keras.variable*)
 (*tensorcircuit.applications.vqes.Linear* property), 363
variable_dtype (*tensorcircuit.keras.QuantumLayer* property), 632
variables (*tensorcircuit.applications.van.MADE* property), 255
variables (*tensorcircuit.applications.van.MaskedConv2D* property), 267
variables (*tensorcircuit.applications.van.MaskedLinear* property), 279
variables (*tensorcircuit.applications.van.NMF* property), 309
variables (*tensorcircuit.applications.van.PixelCNN* property), 339
variables (*tensorcircuit.applications.van.ResidualBlock* property), 351
variables (*tensorcircuit.applications.vqes.Linear* property), 363
variables (*tensorcircuit.keras.QuantumLayer* property), 632
vec2count() (in module *tensorcircuit.results.counts*), 697
vectorized_value_and_grad() (*tensorcircuit.backends.jax_backend.JaxBackend*.*method*), 391
vectorized_value_and_grad() (*tensorcircuit.backends.numpy_backend.NumpyBackend*.*method*), 419
vectorized_value_and_grad() (*tensorcircuit.backends.pytorch_backend.PyTorchBackend*.*method*), 447
vectorized_value_and_grad() (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend*.*method*), 475
verbose_output() (in module *tensorcircuit.applications.dqas*), 208
vgates (*tensorcircuit.abstractcircuit.AbstractCircuit* attribute), 203
vgates (*tensorcircuit.basecircuit.BaseCircuit* attribute), 491
vgates (*tensorcircuit.circuit.Circuit* attribute), 532
vgates (*tensorcircuit.densitymatrix.DMCircuit* attribute), 569
vgates (*tensorcircuit.densitymatrix.DMCircuit2* attribute), 600
vgates (*tensorcircuit.mpscircuit.MPSCircuit* attribute), 665
vis_tex() (*tensorcircuit.abstractcircuit.AbstractCircuit* method), 203
vis_tex() (*tensorcircuit.basecircuit.BaseCircuit* method), 492
vis_tex() (*tensorcircuit.circuit.Circuit* method), 532
vis_tex() (*tensorcircuit.densitymatrix.DMCircuit*

method), 569
vis_tex() (tensorcircuit.densitymatrix.DMCircuit method), 600
vis_tex() (tensorcircuit.mpscircuit.MPSCircuit method), 665
vjp() (tensorcircuit.backends.jax_backend.JaxBackend method), 392
vjp() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 419
vjp() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 448
vjp() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 476
vmap() (tensorcircuit.backends.jax_backend.JaxBackend method), 392
vmap() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 420
vmap() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 448
vmap() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 476
void_generator() (in module tensorcircuit.applications.dqas), 208
vqe_energy() (in module tensorcircuit.applications.vqes), 365
vqe_energy_shortcut() (in module tensorcircuit.applications.vqes), 365
VQNHE (class in tensorcircuit.applications.vqes), 364
vvag() (tensorcircuit.backends.jax_backend.JaxBackend method), 392
vvag() (tensorcircuit.backends.numpy_backend.NumpyBackend method), 420
vvag() (tensorcircuit.backends.pytorch_backend.PyTorchBackend method), 448
vvag() (tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method), 477

W

wavefunction() (tensorcircuit.circuit.Circuit method), 532
wavefunction() (tensorcircuit.densitymatrix.DMCircuit method), 569
wavefunction() (tensorcircuit.densitymatrix.DMCircuit2 method), 600
wavefunction() (tensorcircuit.mpscircuit.MPSCircuit method), 665
wavefunction_to_tensors() (tensorcircuit.mpscircuit.MPSCircuit class method), 665
weights (tensorcircuit.applications.van.MADE property), 256
weights (tensorcircuit.applications.van.MaskedConv2D property), 267
weights (tensorcircuit.applications.van.MaskedLinear property), 279
weights (tensorcircuit.applications.van.NMF property), 309
weights (tensorcircuit.applications.van.PixelCNN property), 339
weights (tensorcircuit.applications.van.ResidualBlock property), 351
weights (tensorcircuit.applications.vqes.Linear property), 363
weights (tensorcircuit.keras.QuantumLayer property), 633
which_backend() (in module tensorcircuit.interfaces.tensortrans), 620
with_name_scope() (tensorcircuit.applications.van.MADE class method), 256
with_name_scope() (tensorcircuit.applications.van.MaskedConv2D class method), 268
with_name_scope() (tensorcircuit.applications.van.MaskedLinear class method), 279
with_name_scope() (tensorcircuit.applications.van.NMF class method), 309
with_name_scope() (tensorcircuit.applications.van.PixelCNN class method), 339
with_name_scope() (tensorcircuit.applications.van.ResidualBlock class method), 351
with_name_scope() (tensorcircuit.applications.vqes.Linear class method), 633
WROOT() (tensorcircuit.circuit.Circuit method), 505
wroot() (tensorcircuit.circuit.Circuit method), 533
WROOT() (tensorcircuit.densitymatrix.DMCircuit method), 545
wroot() (tensorcircuit.densitymatrix.DMCircuit method), 569
WROOT() (tensorcircuit.densitymatrix.DMCircuit2 method), 576
wroot() (tensorcircuit.densitymatrix.DMCircuit2 method), 600
WROOT() (tensorcircuit.mpscircuit.MPSCircuit method), 645
wroot() (tensorcircuit.mpscircuit.MPSCircuit method), 665

X

`x()` (*tensorcircuit.circuit.Circuit method*), 505
`x()` (*tensorcircuit.circuit.Circuit method*), 533
`x()` (*tensorcircuit.densitymatrix.DMCircuit method*), 545
`x()` (*tensorcircuit.densitymatrix.DMCircuit method*), 569
`x()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 576
`x()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 600
`x()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 645
`x()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 665
`xpu()` (*tensorcircuit.torchnn.QuantumNet method*), 722
`xx_rx_block()` (*in module tensorcircuit.applications.layers*), 217
`xx_ry_block()` (*in module tensorcircuit.applications.layers*), 217
`xx_rz_block()` (*in module tensorcircuit.applications.layers*), 217
`xx_xx_block()` (*in module tensorcircuit.applications.layers*), 217
`xx_yy_block()` (*in module tensorcircuit.applications.layers*), 217
`xx_zz_block()` (*in module tensorcircuit.applications.layers*), 217
`xxgate()` (*in module tensorcircuit.applications.layers*), 217
`xxlayer()` (*in module tensorcircuit.applications.layers*), 217
`xxlayer_bitflip()` (*in module tensorcircuit.applications.layers*), 217
`xxlayer_bitflip_mc()` (*in module tensorcircuit.applications.layers*), 217
`xygate()` (*in module tensorcircuit.applications.layers*), 217
`xylayer()` (*in module tensorcircuit.applications.layers*), 217
`xylayer_bitflip()` (*in module tensorcircuit.applications.layers*), 217
`xylayer_bitflip_mc()` (*in module tensorcircuit.applications.layers*), 217
`xzgate()` (*in module tensorcircuit.applications.layers*), 217
`xzlayer()` (*in module tensorcircuit.applications.layers*), 217
`xzlayer_bitflip()` (*in module tensorcircuit.applications.layers*), 218
`xzlayer_bitflip_mc()` (*in module tensorcircuit.applications.layers*), 218

Y

`y()` (*tensorcircuit.circuit.Circuit method*), 505
`y()` (*tensorcircuit.circuit.Circuit method*), 533
`y()` (*tensorcircuit.densitymatrix.DMCircuit method*), 545
`y()` (*tensorcircuit.densitymatrix.DMCircuit method*), 570

`y()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 576
`y()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 601
`y()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 645
`y()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 666
`yxgate()` (*in module tensorcircuit.applications.layers*), 218
`yxlayer()` (*in module tensorcircuit.applications.layers*), 218
`yxlayer_bitflip()` (*in module tensorcircuit.applications.layers*), 218
`yxlayer_bitflip_mc()` (*in module tensorcircuit.applications.layers*), 218
`yy_rx_block()` (*in module tensorcircuit.applications.layers*), 218
`yy_ry_block()` (*in module tensorcircuit.applications.layers*), 218
`yy_rz_block()` (*in module tensorcircuit.applications.layers*), 218
`yy_xx_block()` (*in module tensorcircuit.applications.layers*), 218
`yy_yy_block()` (*in module tensorcircuit.applications.layers*), 218
`yy_zz_block()` (*in module tensorcircuit.applications.layers*), 218
`yygate()` (*in module tensorcircuit.applications.layers*), 218
`yylayer()` (*in module tensorcircuit.applications.layers*), 218
`yylayer_bitflip()` (*in module tensorcircuit.applications.layers*), 218
`yylayer_bitflip_mc()` (*in module tensorcircuit.applications.layers*), 218
`yzgate()` (*in module tensorcircuit.applications.layers*), 218
`yzlayer()` (*in module tensorcircuit.applications.layers*), 218
`yzlayer_bitflip()` (*in module tensorcircuit.applications.layers*), 219
`yzlayer_bitflip_mc()` (*in module tensorcircuit.applications.layers*), 219

Z

`z()` (*tensorcircuit.circuit.Circuit method*), 505
`z()` (*tensorcircuit.circuit.Circuit method*), 533
`z()` (*tensorcircuit.densitymatrix.DMCircuit method*), 545
`z()` (*tensorcircuit.densitymatrix.DMCircuit method*), 570
`z()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 576
`z()` (*tensorcircuit.densitymatrix.DMCircuit2 method*), 601
`z()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 645
`z()` (*tensorcircuit.mpscircuit.MPSCircuit method*), 666

`zero_grad()` (*tensorcircuit.torchnn.QuantumNet method*), 723
`zeros()` (*tensorcircuit.backends.jax_backend.JaxBackend method*), 393
`zeros()` (*tensorcircuit.backends.numpy_backend.NumpyBackend method*), 421
`zeros()` (*tensorcircuit.backends.pytorch_backend.PyTorchBackend method*), 449
`zeros()` (*tensorcircuit.backends.tensorflow_backend.TensorFlowBackend method*), 477
`zxgate()` (*in module tensorcircuit.applications.layers*),
219
`zxlayer()` (*in module tensorcircuit.applications.layers*),
219
`zxlayer_bitflip()` (*in module tensorcircuit.applications.layers*), 219
`zxlayer_bitflip_mc()` (*in module tensorcircuit.applications.layers*), 219
`zygate()` (*in module tensorcircuit.applications.layers*),
219
`zylayer()` (*in module tensorcircuit.applications.layers*),
219
`zylayer_bitflip()` (*in module tensorcircuit.applications.layers*), 219
`zylayer_bitflip_mc()` (*in module tensorcircuit.applications.layers*), 219
`zz_rx_block()` (*in module tensorcircuit.applications.layers*), 219
`zz_ry_block()` (*in module tensorcircuit.applications.layers*), 219
`zz_rz_block()` (*in module tensorcircuit.applications.layers*), 219
`zz_xx_block()` (*in module tensorcircuit.applications.layers*), 219
`zz_yy_block()` (*in module tensorcircuit.applications.layers*), 219
`zz_zz_block()` (*in module tensorcircuit.applications.layers*), 219
`zzgate()` (*in module tensorcircuit.applications.layers*),
219
`zzlayer()` (*in module tensorcircuit.applications.layers*),
219
`zzlayer_bitflip()` (*in module tensorcircuit.applications.layers*), 220
`zzlayer_bitflip_mc()` (*in module tensorcircuit.applications.layers*), 220