QGOpt

Release 0.2a

Dec 07, 2022

Getting started:

QGOpt is an extension of TensorFlow optimizers on Riemannian manifolds that often arise in quantum mechanics. QGOpt allows to perform optimization on the following manifolds:

- Complex Stiefel manifold;
- Manifold of density matrices;
- Manifold of Choi matrices;
- Manifold of Hermitian matrices;
- Complex positive-definite cone;
- Manfiold of POVMs.

QGOpt includes Riemannian versions of popular first-order optimization algorithms that are used in deep learning.

One can use this library to perform quantum tomography of states and channels, to solve quantum control problems and optimize quantum unitary circuits, to perform entanglement renormalization, to solve different model identification problems, to optimize tensor networks with natural "quantum" constraints, etc.

CHAPTER 1

Installation

Make sure you have TensorFlow >= 2.0. One can install the package from GitHub (is recommended)

```
pip install git+https://github.com/LuchnikovI/QGOpt
```

or from pypi (might be different in comparison with the current state of master)

```
pip install QGOpt
```

CHAPTER 2

Quick Start: Quantum Gate decomposition

One can open this tutorial in Google Colab (is recommended)

In the given short tutorial, we show the basic steps of working with QGOpt. It is known that an arbitrary two-qubit unitary gate can be decomposed into a sequence of CNOT gates and one qubit gates as it is shown on the tensor diagram below (if the diagram is not displayed here, please open the notebook in Google Colab).**|renorm_layer.png|**

Local unitary gates are elements of the complex Stiefel manifold; thus, the decomposition can be found by minimizing Frobenius distance between a given two qubits unitary gate and its decomposition. In the beginning, let us import some libraries.

First, one needs to import all necessary libraries.

[ ]:
```
import tensorflow as tf  # tf 2.x
import matplotlib.pyplot as plt
import math

try:
    import QGOpt as qgo
except ImportError:
    !pip install git+https://github.com/LuchnikovI/QGOpt
    import QGOpt as qgo
```

Before considering the main part of the code that solves the problem of gate decomposition, we need to introduce a function that calculates the Kronecker product of two matrices:

[2]:
```
def kron(A, B):
    """
    Returns Kronecker product of two square matrices.

    Args:
        A: complex valued tf tensor of shape (dim1, dim1)
        B: complex valued tf tensor of shape (dim2, dim2)
```

(continues on next page)

(continued from previous page)

```
    Returns:
        complex valued tf tensor of shape (dim1 * dim2, dim1 * dim2),
        kronecker product of two matrices
    """

    dim1 = A.shape[-1]
    dim2 = B.shape[-1]
    AB = tf.transpose(tf.tensordot(A, B, axes=0), (0, 2, 1, 3))
    return tf.reshape(AB, (dim1 * dim2, dim1 * dim2))
```

Then we define an example of the complex Stiefel manifold:

[3]:
```
m = qgo.manifolds.StiefelManifold()
```

As a target gate that we want to decompose, we use a randomly generated one:

[4]:
```
U = m.random((4, 4), dtype=tf.complex128)
```

We initialize the initial set of local unitary gates $\{u_{ij}\}_{i,j=1}^{4,2}$ randomly as a 4th rank tensor:

[5]:
```
u = m.random((4, 2, 2, 2), dtype=tf.complex128)
```

The first two indices of this tensor enumerate a particular one-qubit gate, the last two indices are matrix indices of a gate. We turn this tensor into its real representation in order to make it suitable for an optimizer and wrap it into the TF variable:

[6]:
```
u = qgo.manifolds.complex_to_real(u)
u = tf.Variable(u)
```

We initialize the CNOT gate as follows:

[7]:
```
cnot = tf.constant([[1, 0, 0, 0],
                    [0, 1, 0, 0],
                    [0, 0, 0, 1],
                    [0, 0, 1, 0]], dtype=tf.complex128)
```

As a next step we initialize Riemannian Adam optimizer:

[9]:
```
lr = 0.2  # optimization step size
# we also pass an example of manifold
# to the optimizer in order to give information
# about constraints to the optimizer
opt = qgo.optimizers.RAdam(m, lr)
```

Finally, we ran part of code that calculate forward pass, gradients, and optimization step several times until convergence is reached:

[10]:
```
# this list will be filled by value of
# error per iteration
err_vs_iter = []

# optimization loop
for _ in range(500):
    with tf.GradientTape() as tape:
        # turning u back into its complex representation
        uc = qgo.manifolds.real_to_complex(u)
```

(continues on next page)

(continued from previous page)

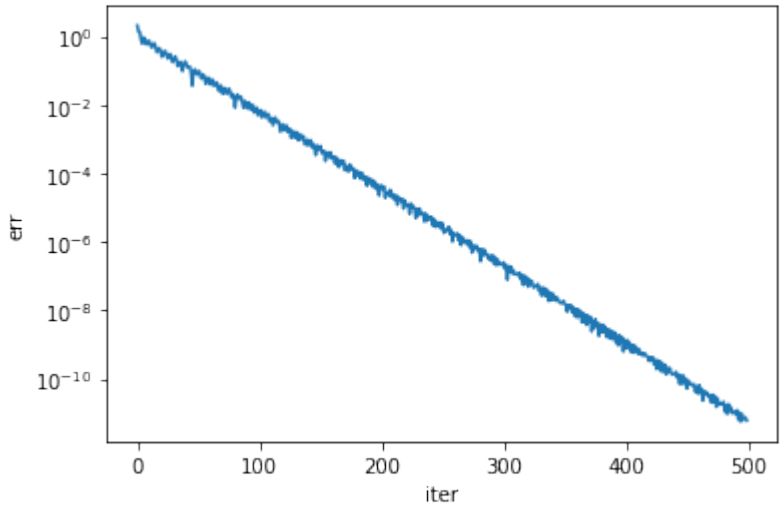
```
        # decomposition
        D = kron(uc[0, 0], uc[0, 1])
        D = cnot @ D
        D = kron(uc[1, 0], uc[1, 1])@ D
        D = cnot @ D
        D = kron(uc[2, 0], uc[2, 1])@ D
        D = cnot @ D
        D = kron(uc[3, 0], uc[3, 1]) @ D
        # loss function
        L = tf.linalg.norm(D - U) ** 2
        L = tf.math.real(L)
    # filling list with history of error
    err_vs_iter.append(tf.math.sqrt(L))
    # gradient from tape
    grad = tape.gradient(L, u)
    # optimization step
    opt.apply_gradients(zip([grad], [u]))
```

Finally, we plot how error decreases with time

[11]:
```
print('[0, 0] element of the trained gate {}'.format(D[0, 0].numpy()))
print('[0, 0] element of the true gate {}'.format(U[0, 0].numpy()))
plt.plot(err_vs_iter)
plt.yscale('log')
plt.xlabel('iter')
plt.ylabel('err')
```

```
[0, 0] element of the trained gate (-0.034378823704696526-0.46822585286096785j)
[0, 0] element of the true gate (-0.03437882370484857-0.4682258528614082j)
```

[11]:
```
Text(0, 0.5, 'err')
```



CHAPTER 3

API

3.1 Manifolds

3.2 Optimizers

3.3 Auxiliary functions

CHAPTER 4

Frequently asked questions

4.1 Is there a relation between complex matrix manifolds and real matrix manifolds?

One can represent any complex matrix $D = E + iF$ as a real matrix $\tilde{D} = \begin{pmatrix} E & F \\ -F & E \end{pmatrix}$. Then, matrix operations on matrices without and with tilde are related as follows:

$A + B \longleftrightarrow \tilde{A} + \tilde{B},\ AB \longleftrightarrow \tilde{A}\tilde{B},\ A^{\dagger} \longleftrightarrow \tilde{A}^{T}.$

Therefore, any complex manifold has a corresponding real one. For more details read

Sato, H., & Iwai, T. (2013). A Riemannian optimization approach to the matrix singular value decomposition. *SIAM Journal on Optimization*, 23(1), 188-212.

4.2 How to perform optimization over complex tensors and matrices?

To perform optimization over complex matrices and tensors, one needs to follow several simple rules. First of all, a value of a loss function, you want to optimize, must be real. Secondly, the class for TensorFlow optimizers works well only with real valued variables. Due to the class for Riemannian optimizers of QGOpt is inherited from the class for TensorFlow optimizers, one requires all input variables to be real. Normally a point from a manifold is represented by a complex matrix or tensor, but one can also consider a point as a real tensor. In general, we suggest the following scheme for variables initialization and optimization:

```
# Here we initialize an example of the complex Stiefel manifold.
m = qgo.manifolds.StiefelManifold()
# Here we initialize a unitary matrix by using an example of the
# complex Stiefel manifold (dtype = tf.complex64).
u = m.random((4, 4))
# Here we turn a complex matrix to its real representation
# (shape=(4, 4) --> shape=(4, 4, 2)).
# The last index enumerates real and imaginary parts.
```

(continues on next page)

(continued from previous page)

```
# (dtype=tf.complex64 --> dtype=tf.float32).
u = qgo.manifolds.complex_to_real(u)
# Here we turn u to tf.Variable, any Riemannian optimizer
# can perform optimization over u now, because it is
# real valued TensorFlow variable. Note also, that
# any Riemannian optimizer preserves all the constraints
# of a corresponding complex manifold.
u = tf.Variable(u)
```

After initialization of variables one can perform optimization step:

```
lr = 0.01 # optimization step size
opt = qgo.optimizers.RAdam(m, lr) # optimizer initialization

# Here we calculate the gradient and perform optimization step.
# Note, that in the body of a TensorFlow graph one can
# have complex-valued tensors. It is only important to
# have input variables and target function to be real.
tf.with tf.GradientTape() as tape:

   # Here we turn the real representation of a point on a manifold
   # back to the complex representation.
   # (shape=(4, 4, 2) --> shape=(4, 4)),
   # (dtype=tf.float32 --> dtype=tf.complex64)
   uc = qgo.manifolds.real_to_complex(u)

   # Here we calculate the value of a target function, we want to minimize.
   # Target function returns real value. If a target function returns an
   # imaginary value, then optimizer minimizes real part of a function.
   loss = target_function(uc)

# Here we calculate the gradient of a function.
grad = tape.gradient(loss, u)
# And perform an optimization step.
opt.apply_gradients(zip([grad], [u]))
```

CHAPTER 5

Entanglement renormalization

One can open this notebook in Google Colab (is recommended)

In the given tutorial, we show how the Riemannian optimization on the complex Stiefel manifold can be used to perform entanglement renormalization and find the ground state energy and the ground state itself of a many-body spin system at the point of quantum phase transition. First of all, let us import the necessary libraries.

[ ]:
```
import numpy as np
from scipy import integrate
import tensorflow as tf  # tf 2.x

try:
    import QGOpt as qgo
except ImportError:
    !pip install git+https://github.com/LuchnikovI/QGOpt
    import QGOpt as qgo

# TensorNetwork library
try:
    import tensornetwork as tn
except ImportError:
    !pip install tensornetwork
    import tensornetwork as tn

import matplotlib.pyplot as plt
from tqdm import tqdm
tn.set_default_backend("tensorflow")

# Fix random seed to make results reproducable.
tf.random.set_seed(42)
```

5.1 1. Renormalization layer

First of all, one needs to define a renormalization (mera) layer. We use ncon API from TensorNetwork library for these purposes. The function mera_layer takes unitary and isometric tensors (building blocks) and performs renormalization of a local Hamiltonian as it is shown on the tensor diagram below (if the diagram is not displayed here, please open the notebook in Google Colab). |renorm_layer.png| For more information about entanglement renormalization please see

Evenbly, G., & Vidal, G. (2009). Algorithms for entanglement renormalization. Physical Review B, 79(14), 144108.

Evenbly, G., & Vidal, G. (2014). Algorithms for entanglement renormalization: boundaries, impurities and interfaces. Journal of Statistical Physics, 157(4-5), 931-978.

For more information about ncon notation see for example

Pfeifer, R. N., Evenbly, G., Singh, S., & Vidal, G. (2014). NCON: A tensor network contractor for MATLAB. arXiv preprint arXiv:1402.0939.

[2]:
```
@tf.function
def mera_layer(H,
               U,
               U_conj,
               Z_left,
               Z_right,
               Z_left_conj,
               Z_right_conj):
    """
    Renormalizes local Hamiltonian.

    Args:
        H: complex valued tensor of shape (chi, chi, chi, chi),
            input two-side Hamiltonian (a local term).
        U: complex valued tensor of shape (chi ** 2, chi ** 2), disentangler
        U_conj: complex valued tensor of shape (chi ** 2, chi ** 2),
            conjugated disentangler.
        Z_left: complex valued tensor of shape (chi ** 3, new_chi),
            left isometry.
        Z_right: complex valued tensor of shape (chi ** 3, new_chi),
            right isometry.
        Z_left_conj: complex valued tensor of shape (chi ** 3, new_chi),
            left conjugated isometry.
        Z_right_conj: complex valued tensor of shape (chi ** 3, new_chi),
            right conjugated isometry.

    Returns:
        complex valued tensor of shape (new_chi, new_chi, new_chi, new_chi),
        renormalized two side hamiltonian.

    Notes:
        chi is the dimension of an index. chi increases with the depth of mera,␣
↪however,
        at some point, chi is cut to prevent exponential growth of indices
        dimensionality."""

    # index dimension before renormalization
    chi = tf.cast(tf.math.sqrt(tf.cast(tf.shape(U)[0], dtype=tf.float64)),
                  dtype=tf.int32)
```

(continues on next page)

(continued from previous page)

```
    # index dimension after renormalization
    chi_new = tf.shape(Z_left)[-1]

    # List of building blocks
    list_of_tensors = [tf.reshape(Z_left, (chi, chi, chi, chi_new)),
                       tf.reshape(Z_right, (chi, chi, chi, chi_new)),
                       tf.reshape(Z_left_conj, (chi, chi, chi, chi_new)),
                       tf.reshape(Z_right_conj, (chi, chi, chi, chi_new)),
                       tf.reshape(U, (chi, chi, chi, chi)),
                       tf.reshape(U_conj, (chi, chi, chi, chi)),
                       H]

    # structures (ncon notation) of three terms of ascending super operator
    net_struc_1 = [[1, 2, 3, -3], [9, 11, 12, -4], [1, 6, 7, -1],
                   [10, 11, 12, -2], [3, 9, 4, 8], [7, 10, 5, 8], [6, 5, 2, 4]]
    net_struc_2 = [[1, 2, 3, -3], [9, 11, 12, -4], [1, 2, 6, -1],
                   [10, 11, 12, -2], [3, 9, 4, 7], [6, 10, 5, 8], [5, 8, 4, 7]]
    net_struc_3 = [[1, 2, 3, -3], [9, 10, 12, -4], [1, 2, 5, -1],
                   [8, 11, 12, -2], [3, 9, 4, 6], [5, 8, 4, 7], [7, 11, 6, 10]]

    # sub-optimal contraction orders for three terms of ascending super operator
    con_ord_1 = [4, 5, 8, 6, 7, 1, 2, 3, 11, 12, 9, 10]
    con_ord_2 = [4, 7, 5, 8, 1, 2, 11, 12, 3, 6, 9, 10]
    con_ord_3 = [6, 7, 4, 11, 8, 12, 10, 9, 1, 2, 3, 5]

    # ncon
    term_1 = tn.ncon(list_of_tensors, net_struc_1, con_ord_1)
    term_2 = tn.ncon(list_of_tensors, net_struc_2, con_ord_2)
    term_3 = tn.ncon(list_of_tensors, net_struc_3, con_ord_3)

    return (term_1 + term_2 + term_3) / 3  # renormalized hamiltonian

# auxiliary functions that return initial isometries and disentanglers
@tf.function
def z_gen(chi, new_chi):
    """Returns random isometry.

    Args:
        chi: int number, input chi.
        new_chi: int number, output chi.

    Returns:
        complex valued tensor of shape (chi ** 3, new_chi)."""

    # one can use the complex Stiefel manfiold to generate a random isometry
    m = qgo.manifolds.StiefelManifold()
    return m.random((chi ** 3, new_chi), dtype=tf.complex128)


@tf.function
def u_gen(chi):
    """Returns the identity matrix of a given size (initial disentangler).

    Args:
        chi: int number.

    Returns:
```

(continues on next page)

(continued from previous page)

```
        complex valued tensor of shape (chi ** 2, chi ** 2)."""

    return tf.eye(chi ** 2, dtype=tf.complex128)
```

5.2 2. Transverse-field Ising (TFI) model hamiltonian and MERA building blocks

Here we define the Transverse-field Ising model Hamiltonian and building blocks (disentanglers and isometries) of MERA network that will be optimized.

First of all we initialize hyper parameters of MERA and TFI hamiltonian.

[3]:
```
max_chi = 4 # max bond dim
num_of_layers = 5 # number of MERA layers (corresponds to 2*3^5 = 486 spins)
h_x = 1 # value of transverse field in TFI model (h_x=1 is the critical field)
```

One needs to define Pauli matrices. Here all Pauli matrices are represented as one tensor of size $3 \times 2 \times 2$, where the first index enumerates a particular Pauli matrix, and the remaining two indices are matrix indices.

[4]:
```
sigma = tf.constant([[[1j*0, 1 + 1j*0], [1 + 1j*0, 0*1j]],
                     [[0*1j, -1j], [1j, 0*1j]],
                     [[1 + 0*1j, 0*1j], [0*1j, -1 + 0*1j]]], dtype=tf.complex128)
```

Here we define local term of the TFI hamiltonian.

[5]:
```
zz_term = tf.einsum('ij,kl->ikjl', sigma[2], sigma[2])
x_term = tf.einsum('ij,kl->ikjl', sigma[0], tf.eye(2, dtype=tf.complex128))
h = -zz_term - h_x * x_term
```

Here we define initial disentanglers, isometries, and state in the renormalized space.

[6]:
```
# disentangler U and isometry Z in the first MERA layer
U = u_gen(2)
Z = z_gen(2, max_chi)

# lists with disentanglers and isometries in the rest of the layers
U_list = [u_gen(max_chi) for _ in range(num_of_layers - 1)]
Z_list = [z_gen(max_chi, max_chi) for _ in range(num_of_layers - 1)]

# lists with all disentanglers and isometries
U_list = [U] + U_list
Z_list = [Z] + Z_list

# initial state in the renormalized space (low dimensional in comparison
# with the dimensionality of the initial problem)
psi = tf.ones((max_chi ** 2, 1), dtype=tf.complex128)
psi = psi / tf.linalg.norm(psi)

# converting disentanglers, isometries, and initial state to real
# representation (necessary for the further optimizer)
U_list = list(map(qgo.manifolds.complex_to_real, U_list))
Z_list = list(map(qgo.manifolds.complex_to_real, Z_list))
psi = qgo.manifolds.complex_to_real(psi)
```

(continues on next page)

(continued from previous page)

```
# wrapping disentanglers, isometries, and initial state into
# tf.Variable (necessary for the further optimizer)
U_var = list(map(tf.Variable, U_list))
Z_var = list(map(tf.Variable, Z_list))
psi_var = tf.Variable(psi)
```

5.3 3. Optimization of MERA

MERA parametrizes quantum state $\Psi(U, Z, \psi)$ of a spin system, where U is a set of disentanglers, Z is a set of isometries, and ψ is a state in the renormalized space. In order to find the ground state and its energy, we perform optimization of variational energy

$$\langle\Psi(U, Z, \psi)|H_{\mathrm{TFI}}|\Psi(U, Z, \psi)\rangle \rightarrow \min_{U,\, Z,\, \psi \in \mathrm{Stiefel\ manifold}}$$

First of all, we define the parameters of optimization. In order to achieve better convergence, we decrease the learning rate with the number of iteration according to the exponential law.

[7]:
```
iters = 3000 # number of iterations
lr_i = 0.6 # initial learning rate
lr_f = 0.05 # final learning rate

# learning rate is multiplied by this coefficient each iteration
decay = (lr_f / lr_i) ** (1 / iters)
```

Here we define an example of the complex Stiefel manifold necessary for Riemannian optimization and Riemannian Adam optimizer.

[8]:
```
m = qgo.manifolds.StiefelManifold()  # complex Stiefel manifold
opt = qgo.optimizers.RAdam(m, lr_i)  # Riemannian Adam
```

Finally, we perform an optimization loop.

[9]:
```
# this list will be filled by the value of variational energy per iteration
E_list = []

# optimization loop
for j in tqdm(range(iters)):

    # gradient calculation
    with tf.GradientTape() as tape:

        # convert real valued variables back to complex valued tensors
        U_var_c = list(map(qgo.manifolds.real_to_complex, U_var))
        Z_var_c = list(map(qgo.manifolds.real_to_complex, Z_var))
        psi_var_c = qgo.manifolds.real_to_complex(psi_var)

        # initial local Hamiltonian term
        h_renorm = h

        # renormalization of a local Hamiltonian term
        for i in range(len(U_var)):
            h_renorm = mera_layer(h_renorm,
```

(continues on next page)

(continued from previous page)

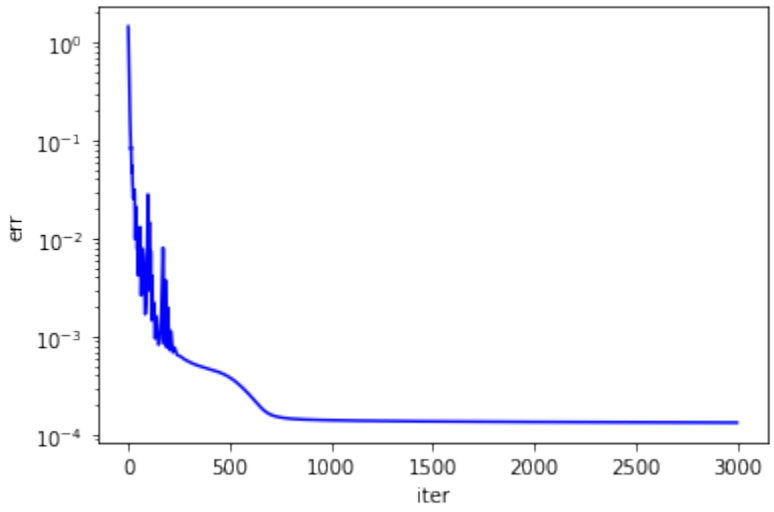
```
                                    U_var_c[i],
                                    tf.math.conj(U_var_c[i]),
                                    Z_var_c[i],
                                    Z_var_c[i],
                                    tf.math.conj(Z_var_c[i]),
                                    tf.math.conj(Z_var_c[i]))

        # renormalizad Hamiltonian (low dimensional)
        h_renorm = (h_renorm + tf.transpose(h_renorm, (1, 0, 3, 2))) / 2
        h_renorm = tf.reshape(h_renorm, (max_chi * max_chi, max_chi * max_chi))

        # energy
        E = tf.cast((tf.linalg.adjoint(psi_var_c) @ h_renorm @ psi_var_c),
                    dtype=tf.float64)[0, 0]

    # adding current variational energy to the list
    E_list.append(E)

    # gradients
    grad = tape.gradient(E, U_var + Z_var + [psi_var])

    # optimization step
    opt.apply_gradients(zip(grad, U_var + Z_var + [psi_var]))

    # learning rate update
    opt._set_hyper("learning_rate", opt._get_hyper("learning_rate") * decay)
```

```
100%|| 3000/3000 [06:21<00:00,  7.87it/s]
```

Here we compare exact ground state energy with MERA based value. We also plot how the difference between exact ground state energy and MERA-based energy evolves with the number of iteration.

[10]:
```
# exact value of ground state energy in the critical point
N = 2 * (3 ** num_of_layers) # number of spins (for 5 layers one has 486 spins)
E0_exact_fin = -2 * (1 / np.sin(np.pi / (2 * N))) / N # exact energy per spin

plt.yscale('log')
plt.xlabel('iter')
plt.ylabel('err')
plt.plot(E_list - tf.convert_to_tensor(([E0_exact_fin] * len(E_list))), 'b')
print("MERA energy:", E_list[-1].numpy())
print("Exact energy:", E0_exact_fin)
```

```
MERA energy: -1.2731094185716914
Exact energy: -1.2732417615356748
```


10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
err
0
500
1000
1500
2000
2500
3000
iter

CHAPTER 6

Quantum channel tomography

One can open this notebook in Google Colab (is recommended)

In this tutorial, we perform quantum channel tomography via Riemannian optimization. First two blocks of code (1. Many-qubit, informationally complete, positive operator-valued measure (IC POVM) and 2. Data set generation (measurement outcomes simulation)) are refered to data generation, third bock dedicated to tomography of a channel.

First, one needs to import all necessary libraries.

[1]:
```
import tensorflow as tf  # tf 2.x
import matplotlib.pyplot as plt
from math import sqrt

try:
    import QGOpt as qgo
except ImportError:
    !pip install git+https://github.com/LuchnikovI/QGOpt
    import QGOpt as qgo
```

6.1 1. Many-qubit, informationally complete, positive operator-valued measure (IC POVM)

Before generating measurement outcomes and performing quantum tomography, one needs to introduce POVM describing quantum measurements. For simplicity, we use one-qubit tetrahedral POVM and generalize it on a many-qubit case by taking tensor product between POVM elements, i.e. $\{M_\alpha\}_{\alpha=1}^4$ is the one-qubit tetrahedral POVM, $\{M_{\alpha_1}\otimes\cdots\otimes M_{\alpha_N}\}_{\alpha_1=1,\ldots,\alpha_N=1}^4$ is the many-qubits tetrahedral POVM.

[2]:
```
# Auxiliary function that returns Kronecker product between two
# POVM elements A and B
def kron(A, B):
    """Kronecker product of two POVM elements.
```

(continues on next page)

(continued from previous page)

```
    Args:
        A: complex valued tensor of shape (q, n, k).
        B: complex valued tensor of shape (p, m, l).

    Returns:
        complex valued tensor of shape (q * p, n * m, k * l)"""

    AB = tf.tensordot(A, B, axes=0)
    AB = tf.transpose(AB, (0, 3, 1, 4, 2, 5))
    shape = AB.shape
    AB = tf.reshape(AB, (shape[0] * shape[1],
                         shape[2] * shape[3],
                         shape[4] * shape[5]))
    return AB

# Pauli matrices
sigma_x = tf.constant([[0, 1], [1, 0]], dtype=tf.complex128)
sigma_y = tf.constant([[0 + 0j, -1j], [1j, 0 + 0j]], dtype=tf.complex128)
sigma_z = tf.constant([[1, 0], [0, -1]], dtype=tf.complex128)

# All Pauli matrices in one tensor of shape (3, 2, 2)
sigma = tf.concat([sigma_x[tf.newaxis],
                   sigma_y[tf.newaxis],
                   sigma_z[tf.newaxis]], axis=0)

# Coordinates of thetrahedron peaks (is needed to build tetrahedral POVM)
s0 = tf.constant([0, 0, 1], dtype=tf.complex128)
s1 = tf.constant([2 * sqrt(2) / 3, 0, -1/3], dtype=tf.complex128)
s2 = tf.constant([-sqrt(2) / 3, sqrt(2 / 3), -1 / 3], dtype=tf.complex128)
s3 = tf.constant([-sqrt(2) / 3, -sqrt(2 / 3), -1 / 3], dtype=tf.complex128)

# Coordinates of thetrahedron peaks in one tensor of shape (4, 3)
s = tf.concat([s0[tf.newaxis],
               s1[tf.newaxis],
               s2[tf.newaxis],
               s3[tf.newaxis]], axis=0)

# One qubit thetrahedral POVM
M = 0.25 * (tf.eye(2, dtype=tf.complex128) + tf.tensordot(s, sigma, axes=1))

n = 2  # number of qubits we experiment with

# M for many qubits
Mmq = M
for _ in range(n - 1):
    Mmq = kron(Mmq, M)
```

6.2 2. Data set generation (measurement outcomes simulation).

Here we generate a set of measurement outcomes (training set). First of all, we generate a random quantum channel with Kraus rank k by using the quotient manifold of Choi matrices. This quantum channel will be a target unknown one, that we want to reconstruct. Then we generate a set of random pure density matrices, pass them through the generated channel, and simulate measurements of output states. Results of measurements and initial states we write in

a data set.

[3]:
```
#=================Parameters==================#
num_of_meas = 600000  # number of measurements
k = 2  # Kraus rank (number of Kraus operators)
#=============================================#


# example of quotient manifold of Choi matrices
m = qgo.manifolds.ChoiMatrix()

# random parametrization of Choi matrix of kraus rank k
A = m.random((2 ** (2 * n), k), dtype=tf.complex128)

# corresponding Choi matrix
C = A @ tf.linalg.adjoint(A)

# corresponding quantum channel
C_resh = tf.reshape(C, (2 ** n, 2 ** n, 2 ** n, 2 ** n))
Phi = tf.transpose(C_resh, (1, 3, 0, 2))
Phi = tf.reshape(Phi, (2 ** (2 * n), 2 ** (2 * n)))

# random initial pure density matrices
psi_set = tf.random.normal((num_of_meas, 2 ** n, 2), dtype=tf.float64)
psi_set = qgo.manifolds.real_to_complex(psi_set)
psi_set = psi_set / tf.linalg.norm(psi_set, axis=-1, keepdims=True)
rho_in = psi_set[..., tf.newaxis] * tf.math.conj(psi_set[:, tf.newaxis])

# reshaping density matrices to vectors
rho_in_resh = tf.reshape(rho_in, (-1, 2 ** (2 * n)))

# output states (we pass initial density matrices trough a channel)
rho_out_resh = tf.tensordot(rho_in_resh, Phi, axes=[[1], [1]])
# reshaping output density matrices back to matrix form
rho_out = tf.reshape(rho_out_resh, (-1, 2 ** n, 2 ** n))

# Measurements simulation (by using Gumbel trick for sampling from a
# discrete distribution)
P = tf.cast(tf.einsum('qjk,pkj->pq', Mmq, rho_out), dtype=tf.float64)
eps = tf.random.uniform((num_of_meas, 2 ** (2 * n)), dtype=tf.float64)
eps = -tf.math.log(-tf.math.log(eps))
ind_set = tf.math.argmax(eps + tf.math.log(P), axis=-1)

# projectors that came true
M_set = tf.gather_nd(Mmq, ind_set[:, tf.newaxis])

# resulting dataset
data_set = [rho_in, M_set]
```

6.3 3. Data processing (tomography)

First, we define an example of the Choi matrices manifold:

[4]:
```
m = qgo.manifolds.ChoiMatrix()
```

The manifold of Choi matrices is represneted through the quadratic parametrization $C = AA^\dagger$ with qn equivalence

relation $A \sim AQ$, where Q is an arbitrary unitary matrix. Thus, we initialize a variable, that represents the parametrization of a Choi matrix:

[5]:
```
# random initial paramterization
a = m.random((2 ** (2 * n), 2 ** (2 * n)), dtype=tf.complex128)
# in order to make an optimizer works properly
# one need to turn a to real representation
a = qgo.manifolds.complex_to_real(a)
# variable
a = tf.Variable(a)
```

Then we initialize Riemannian Adam optimizer:

[6]:
```
lr = 0.07  # optimization step size
opt = qgo.optimizers.RAdam(m, lr)
```

Finally, we ran part of code that calculate forward pass, gradients, and optimization step several times until convergence is reached:

[7]:
```
# the list will be filled by value of J distance per iteration
j_distance = []

for _ in range(400):
    with tf.GradientTape() as tape:
        # complex representation of parametrization
        # shape=(2**2n, 2**2n)
        ac = qgo.manifolds.real_to_complex(a)

        # reshape parametrization
        # (2**2n, 2**2n) --> (2**n, 2**n, 2**2n)
        ac = tf.reshape(ac, (2**n, 2**n, 2**(2*n)))

        # Choi tensor (reshaped Choi matrix)
        c = tf.tensordot(ac, tf.math.conj(ac), [[2], [2]])

        # turning Choi tensor to the
        # corresponding quantum channel
        phi = tf.transpose(c, (1, 3, 0, 2))
        phi = tf.reshape(phi, (2**(2*n), 2**(2*n)))

        # reshape initial density
        # matrices to vectors
        rho_resh = tf.reshape(data_set[0], (num_of_meas, 2**(2*n)))

        # passing density matrices
        # through a quantum channel
        rho_out = tf.tensordot(phi,
                               rho_resh,
                               [[1], [1]])
        rho_out = tf.transpose(rho_out)
        rho_out = tf.reshape(rho_out, (num_of_meas, 2**n, 2**n))

        # probabilities of measurement outcomes
        # (povms is a set of POVM elements
        # came true of shape (N, 2**n, 2**n))
        p = tf.linalg.trace(data_set[1] @ rho_out)

        # negative log likelihood (to be minimized)
```

(continues on next page)

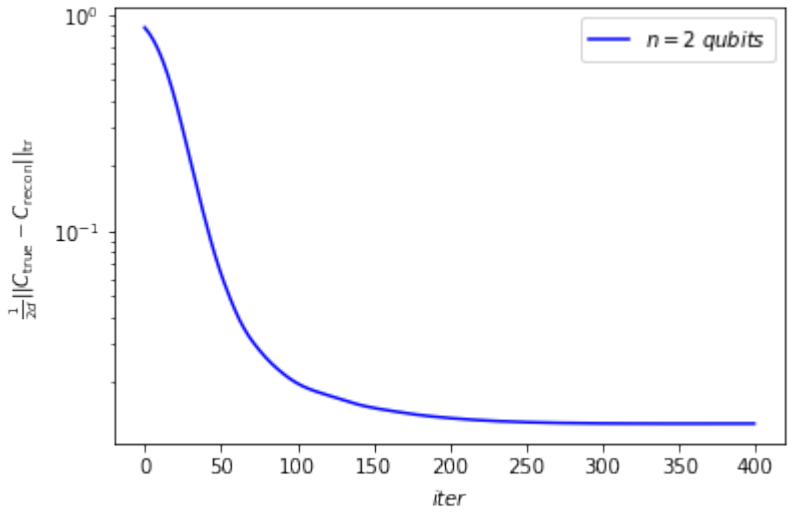
(continued from previous page)

```
        L = -tf.reduce_mean(tf.math.log(p))

    # filling j_distance list (for further plotting)
    j_distance.append(tf.reduce_sum(tf.abs(tf.linalg.eigvalsh(tf.reshape(c,
    (2 ** (2 * n), 2 ** (2 * n))) - C))) / (2 * (2 ** n)))
    # gradient
    grad = tape.gradient(L, a)
    # optimization step
    opt.apply_gradients(zip([grad], [a]))
```

Finally, we plot the dependance between J distance and iteration.

[14]:
```
plt.plot(j_distance, 'b')
plt.legend([r'$n=$' + str(n) + r'$\ qubits$'])
plt.yscale('log')
plt.ylabel(r'$\frac{1}{2d}||C_{\rm true} - C_{\rm recon}||_{\rm tr}$')
plt.xlabel(r'$iter$')
```

[14]:
```
Text(0.5, 0, '$iter$')
```



CHAPTER 7

Quantum state tomography

One can open this notebook in Google Colab (is recommended)

In this tutorial, we perform quantum state tomography via Riemannian optimization. First two blocks of a code (1. Many-qubit, informationally complete, positive operator-valued measure (IC POVM) and 2. Data set generation (measurement outcomes simulation)) are refered to data generation, third bock dedicated to tomography of a state.

First, one needs to import all necessary libraries.

[1]:
```
import tensorflow as tf  # tf 2.x
from math import sqrt

try:
    import QGOpt as qgo
except ImportError:
    !pip install git+https://github.com/LuchnikovI/QGOpt
    import QGOpt as qgo

import matplotlib.pyplot as plt
from tqdm import tqdm

# Fix random seed to make results reproducable.
tf.random.set_seed(42)
```

7.1 1. Many-qubit, informationally complete, positive operator-valued measure (IC POVM)

Before generating measurement outcomes and performing quantum tomography, one needs to introduce POVM describing quantum measurements. For simplicity, we use one-qubit tetrahedral POVM and generalize it on a many-qubit case by taking tensor product between POVM elements, i.e. $\{M_\alpha\}_{\alpha=1}^4$ is the one-qubit tetrahedral POVM, $\{M_{\alpha_1}\otimes\cdots\otimes M_{\alpha_N}\}_{\alpha_1=1,\ldots,\alpha_N=1}^4$ is the many-qubits tetrahedral POVM.

[2]:
```
# Auxiliary function that returns Kronecker product between two
# POVM elements A and B
def kron(A, B):
    """Kronecker product of two POVM elements.

    Args:
        A: complex valued tensor of shape (q, n, k).
        B: complex valued tensor of shape (p, m, l).

    Returns:
        complex valued tensor of shape (q * p, n * m, k * l)"""

    AB = tf.tensordot(A, B, axes=0)
    AB = tf.transpose(AB, (0, 3, 1, 4, 2, 5))
    shape = AB.shape
    AB = tf.reshape(AB, (shape[0] * shape[1],
                         shape[2] * shape[3],
                         shape[4] * shape[5]))
    return AB

# Pauli matrices
sigma_x = tf.constant([[0, 1], [1, 0]], dtype=tf.complex128)
sigma_y = tf.constant([[0 + 0j, -1j], [1j, 0 + 0j]], dtype=tf.complex128)
sigma_z = tf.constant([[1, 0], [0, -1]], dtype=tf.complex128)

# All Pauli matrices in one tensor of shape (3, 2, 2)
sigma = tf.concat([sigma_x[tf.newaxis],
                   sigma_y[tf.newaxis],
                   sigma_z[tf.newaxis]], axis=0)

# Coordinates of thetrahedron peaks (is needed to build tetrahedral POVM)
s0 = tf.constant([0, 0, 1], dtype=tf.complex128)
s1 = tf.constant([2 * sqrt(2) / 3, 0, -1/3], dtype=tf.complex128)
s2 = tf.constant([-sqrt(2) / 3, sqrt(2 / 3), -1 / 3], dtype=tf.complex128)
s3 = tf.constant([-sqrt(2) / 3, -sqrt(2 / 3), -1 / 3], dtype=tf.complex128)

# Coordinates of thetrahedron peaks in one tensor of shape (4, 3)
s = tf.concat([s0[tf.newaxis],
               s1[tf.newaxis],
               s2[tf.newaxis],
               s3[tf.newaxis]], axis=0)

# One qubit thetrahedral POVM
M = 0.25 * (tf.eye(2, dtype=tf.complex128) + tf.tensordot(s, sigma, axes=1))

n = 2  # number of qubits we experiment with

# M for n qubits (Mmq)
Mmq = M
for _ in range(n - 1):
    Mmq = kron(Mmq, M)
```

7.2 2. Data set generation (measurement outcomes simulation).

Here we generate a set of measurement outcomes (training set). First of all, we generate a random density matrix that is a target state we want to reconstruct. Then, we simulate measurement outcomes over the target state driven by many-qubits tetrahedral POVM introduced in the previous cell.

[3]:
```
#-----------------------------------------------------#
num_of_meas = 600000  # number of measurement outcomes
#-----------------------------------------------------#

# random target density matrix (to be reconstructed)
m = qgo.manifolds.DensityMatrix()
A = m.random((2 ** n, 2 ** n), dtype=tf.complex128)
rho_true = A @ tf.linalg.adjoint(A)

# measurements simulation (by using Gumbel trick for sampling from a
# discrete distribution)
P = tf.cast(tf.tensordot(Mmq, rho_true, [[1, 2], [1, 0]]), dtype=tf.float64)
eps = tf.random.uniform((num_of_meas, 2 ** (2 * n)), dtype=tf.float64)
eps = -tf.math.log(-tf.math.log(eps))
ind_set = tf.math.argmax(eps + tf.math.log(P), axis=-1)

# POVM elements came true (data set)
data_set = tf.gather_nd(Mmq, ind_set[:, tf.newaxis])
```

7.3 3. Data processing (tomography)

First, we define an example of the density matrices manifold:

[4]:
```
m = qgo.manifolds.DensityMatrix()
```

The manifold of density matrices is represneted through the quadratic parametrization $\varrho = AA^\dagger$ with an equivalence relation $A \sim AQ$, where Q is an arbitrary unitary matrix. Thus, we initialize a variable, that represents the parametrization of a density matrix:

[5]:
```
# random initial paramterization
a = m.random((2 ** n, 2 ** n), dtype=tf.complex128)
# in order to make an optimizer works properly
# one need to turn a to real representation
a = qgo.manifolds.complex_to_real(a)
# variable
a = tf.Variable(a)
```

Then we initialize Riemannian Adam optimizer:

[6]:
```
lr = 0.07  # optimization step size
opt = qgo.optimizers.RAdam(m, lr)
```

Finally, we ran part of code that calculate forward pass, gradients, and optimization step several times until convergence is reached:

[7]:
```
# the list will be filled by value of trace distance per iteration
trace_distance = []
```

(continues on next page)

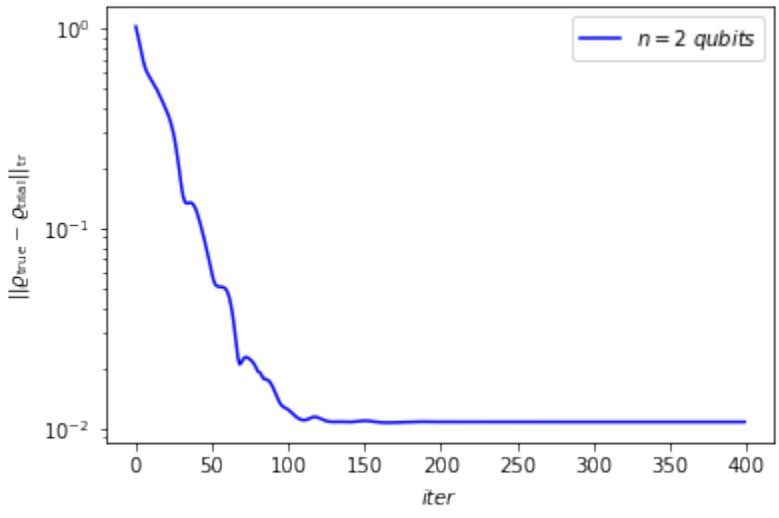
(continued from previous page)

```
for _ in range(400):
    with tf.GradientTape() as tape:
        # complex representation of parametrization
        # shape=(2**n, 2**n)
        ac = qgo.manifolds.real_to_complex(a)

        # density matrix
        rho_trial = ac @ tf.linalg.adjoint(ac)

        # probabilities of measurement outcomes
        p = tf.tensordot(rho_trial, data_set, [[0, 1], [2, 1]])
        p = tf.math.real(p)

        # negative log likelihood (to be minimized)
        L = -tf.reduce_mean(tf.math.log(p))

    # filling trace_distance list (for further plotting)
    trace_distance.append(tf.reduce_sum(tf.math.abs(tf.linalg.eigvalsh(rho_trial -␣
↪rho_true))))
    # gradient
    grad = tape.gradient(L, a)
    # optimization step
    opt.apply_gradients(zip([grad], [a]))
```

Here we plot trace distance vs number of iteration to validate the result

[8]:
```
plt.plot(trace_distance, 'b')
plt.legend([r'$n=$' + str(n) + r'$\ qubits$'])
plt.yscale('log')
plt.ylabel(r'$||\varrho_{\rm true} - \varrho_{\rm trial}||_{\rm tr}$')
plt.xlabel(r'$iter$')
```

[8]:
```
Text(0.5,0,'$iter$')
```



CHAPTER 8

Optimal POVM

One can open this tutorial in Google Colab (is recommended)

In the following tutorial, we show how to perform optimization over the manifold of POVMs by using the QGOpt library. It is known that measurements of a qubit induced by tetrahedral POVM allow reconstructing an unknown qubit state with a minimal variance if there is no prior information about a qubit state. Let us check this fact numerically using optimization over the manifold of POVMs. In the beginning, let us import some libraries.

[1]:
```
import tensorflow as tf  # tf 2.x
import matplotlib.pyplot as plt
import math

try:
    import QGOpt as qgo
except ImportError:
    !pip install git+https://github.com/LuchnikovI/QGOpt@Dev
    import QGOpt as qgo

# Fix random seed to make results reproducable.
tf.random.set_seed(42)
```

8.1 1. Prior information about a quantum state

We represent a prior probability distribution over a quantum state approximately, by using a set of samples from a prior distribution. Since tetrahedral POVM is optimal when there is no prior information about a state, we consider uniform distribution across the Bloch ball.

[2]:
```
#-----------------------------------------------------------------------#
num_of_samples = 10000  # number of samples representing prior information
#-----------------------------------------------------------------------#
```

(continues on next page)

(continued from previous page)

```
# Pauli matrices
sigma_x = tf.constant([[0, 1], [1, 0]], dtype=tf.complex128)
sigma_y = tf.constant([[0 + 0j, -1j], [1j, 0 + 0j]], dtype=tf.complex128)
sigma_z = tf.constant([[1, 0], [0, -1]], dtype=tf.complex128)

# All Pauli matrices in one tensor of shape (3, 2, 2)
sigma = tf.concat([sigma_x[tf.newaxis],
                   sigma_y[tf.newaxis],
                   sigma_z[tf.newaxis]], axis=0)

# Set of points distributed uniformly across Bloch ball
x = tf.random.normal((num_of_samples, 3), dtype=tf.float64)
x = x / tf.linalg.norm(x, axis=-1, keepdims=True)
x = tf.cast(x, dtype=tf.complex128)
u = tf.random.uniform((num_of_samples, 1), maxval=1, dtype=tf.float64)
u = u ** (1 / 3)
u = tf.cast(u, dtype=tf.complex128)
x = x * u

# Set of density matrices distributed uniformly across Bloch ball
# (prior information)
rho = 0.5 * (tf.eye(2, dtype=tf.complex128) + tf.tensordot(x, sigma, axes=1))
```

8.2 2. Search for the optimal POVM with given prior information about a state

Here we search for the optimal POVM via minimizing the variance of a posterior distribution over density matrices. First, we define an example of the POVMs manifold:

[3]:
```
m = qgo.manifolds.POVM()
```

The manifolds of POVMs is represented through the quadratic parametrization $M_i = A_i A_i^\dagger$ with an equivalence relation $A_i \sim A_i Q_i$, where Q_i is an arbitrary unitary matrix. Here, we initialize a variable that represents the parametrization of each element of POVM:

[4]:
```
# randon initial parametrization of POVM
A = m.random((4, 2, 2), dtype=tf.complex128)
# real representtion of A
A = qgo.manifolds.complex_to_real(A)
# tf.Variable to be tuned
A = tf.Variable(A)
```

Then we initialize Riemannian Adam optimizer:

[5]:
```
lr = 0.03
opt = qgo.optimizers.RAdam(m, lr)
```

Finally, we ran the part of code that calculates forward pass, gradients, and optimization step several times until convergence to the optimal point is reached:

[6]:
```
for i in range(1000):
    with tf.GradientTape() as tape:
```

(continues on next page)

(continued from previous page)

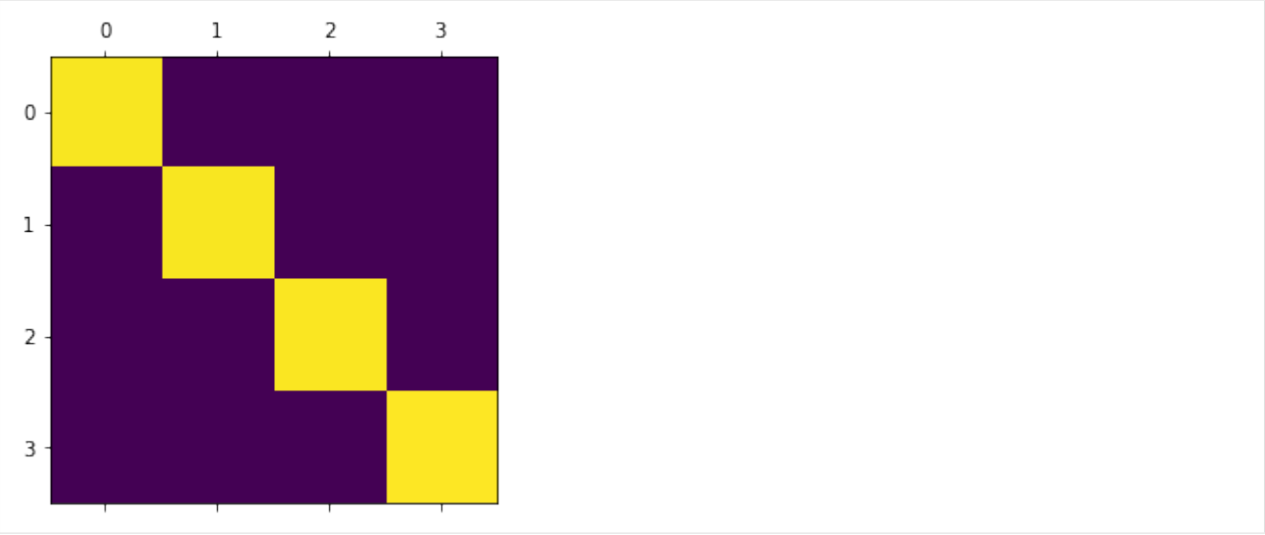
```
        # Complex representation of A
        Ac = qgo.manifolds.real_to_complex(A)
        # POVM from its parametrization
        povm = Ac @ tf.linalg.adjoint(Ac)
        # Inverce POVM (is needed to map a probability distribution to a density␣
↪matrix)
        povm_inv = tf.linalg.inv(tf.reshape(povm, (4, 4)))
        # Matrix T maps probability vector to four real parameters representing
        # a quantum state (equivalent to inverse POVM)
        T = tf.concat([tf.math.real(povm_inv[0, tf.newaxis]),
                       tf.math.real(povm_inv[3, tf.newaxis]),
                       tf.math.real(povm_inv[2, tf.newaxis]),
                       tf.math.imag(povm_inv[2, tf.newaxis])], axis=0)

        # POVM maps a quantum state to a probability vector
        p = tf.tensordot(rho, povm, axes=[[2], [1]])
        p = tf.transpose(p, (0, 2, 1, 3))
        p = tf.math.real(tf.linalg.trace(p))

        # Covariance matrix of a reconstructed density matrix
        cov = -p[:, tf.newaxis] * p[..., tf.newaxis]
        cov = cov + tf.linalg.diag(p ** 2)
        cov = cov + tf.linalg.diag(p * (1 - p))
        cov = tf.tensordot(T, cov, [[1], [1]])
        cov = tf.tensordot(cov, T, [[2], [1]])
        cov = tf.transpose(cov, (1, 0, 2))

        # Covariance matrix avaraged over prior distribution
        av_cov = tf.reduce_mean(cov, axis=0)

        # loss function (log volume of Covariance matrix)
        loss = tf.reduce_sum(tf.math.log(tf.linalg.svd(av_cov)[0][:-1]))
    grad = tape.gradient(loss, A)  # gradient
    opt.apply_gradients(zip([grad], [A]))  # minimization step
```

8.3 3. Verification

Here we check the resulting optimal POVM. For tetrahedral POVM one has the following relation $\mathrm{Tr}\left(M^\alpha M^\beta\right) = \frac{2\delta_{\alpha\beta}+1}{12}$. One can see, that this relation is almost true for a resulting POVM. The small error appears due to the approximate Monte-Carlo averaging of a covariance matric by using a set of samples from the prior distribution.

[7]:
```
cross = tf.tensordot(povm, povm, [[2], [1]])
cross = tf.transpose(cross, (0, 2, 1, 3))
cross = tf.linalg.trace(cross)
cross = tf.math.real(cross)
plt.matshow(cross)
print(cross)
```

```
tf.Tensor(
[[0.24927765 0.08337808 0.08333673 0.08328467]
 [0.08337808 0.24939711 0.08328523 0.08333633]
 [0.08333673 0.08328523 0.25029829 0.08337795]
 [0.08328467 0.08333633 0.08337795 0.25102899]], shape=(4, 4), dtype=float64)
```


0
1
2
3
0
1
2
3

CHAPTER 9

How to Contribute

9.1 Code style

All contributions should be formated according to the PEP8 standard. Slightly more than 80 characters can sometimes be tolerated if increased line width increases readability. All docstrings should follow Google Style Python Docstrings.

9.2 Dependencies

Make sure that you use Python >= 3.5 and have TensorFlow >= 2.0 installed.

9.3 Unit tests

After any change of QGOpt, one has to check whether all the tests run without errors. Currently, tests check optimization primitives (retractions, vector transports, etc.) for all manifolds and check optimizers' performance on the simple optimization problem on complex Stiefel manifold. For any new functionality, please provide suitable unit tests. Also, if you find a bug, consider adding a test that detects the bug before fixing it.

```
pytest
```

running all test files.

```
pytest test_manifolds.py
```

running tests of all manifolds except complex Stiefel manifold.

```
pytest test_stiefel.py
```

running tests of complex Stiefel manifold.

```
pytest test_optimizers.py
```

running tests of optimizers.