QuCumber Documentation

Release v1.3.2

PIQuIL

2021-01-11

INTRODUCTION

CHAPTER

ONE

INSTALLATION

QuCumber only supports Python 3 (specifically, 3.6 and up), not Python 2. If you are using Python 2, please update! You may also want to install PyTorch (https://pytorch.org/), if you have not already.

If you're running a reasonably up-to-date Linux or macOS system, PyTorch should be installed automatically when you install QuCumber with *pip*.

1.1 Windows

Windows 10 is recommended. PyTorch is required (following https://pytorch.org/get-started/locally/). One way for getting PyTorch is having Anaconda installed first and using the 64-bit graphical installer (https://repo.anaconda.com/archive/Anaconda3-2020.02-Windows-x86_64.exe).

Before you install Anaconda, make sure to have a LaTeX distribution installed, for example MiKTeX (https://miktex.org/download), as Matplotlib libraries require LaTeX for nice visualization in Python.

After the Anaconda installation, follow specific instructions on https://pytorch.org/get-started/locally/ to get the correct installation command for PyTorch, which is CUDA version dependent. For example, if your system does not have a GPU card, you will need the CPU version:

```
conda install pytorch torchvision cpuonly -c pytorch
```

To install QuCumber on Anaconda, start the Anaconda prompt, or navigate to the directory (through command prompt) where *pip.exe* is installed (usually `C:\Python\Scripts\pip.exe`) and then type:

```
pip.exe install qucumber
```

1.2 Linux / macOS

Open up a terminal, then type:

```
pip install qucumber
```

1.3 GitHub

Navigate to the qucumber page on GitHub (https://github.com/PIQuIL/QuCumber) and clone the repository by typing:

```
git clone https://github.com/PIQuIL/QuCumber.git
```

Navigate to the main directory and type:

```
python setup.py install
```

CHAPTER

TWO

THEORY

For a basic introduction to Restricted Boltzmann Machines, click here.

CHAPTER

THREE

DOWNLOAD THE TUTORIALS

Once you have installed QuCumber, we recommend going through our tutorial that is divided into two parts.

1. Training a wave function to reconstruct a positive-real wave function (i.e. no phase) from a transverse-field Ising model (TFIM) and then generating new data.
2. Training an wave function to reconstruct a complex wave function (i.e. with a phase) from a simple two qubit random state and then generating new data.

We have made interactive python notebooks that can be downloaded (along with the data required) here. Note that the linked examples are from the most recent stable release (relative to the version of the docs you're currently viewing), and may not match the examples shown in the following pages. It is recommended that you refer to documentation for the latest stable release: https://qucumber.readthedocs.io/en/stable/.

If you wish to simply view the static, non-interactive notebooks, continue to the next page of the documentation.

Alternatively, you can view interactive notebooks online at: 

, though they may be slow.

CHAPTER

FOUR

RECONSTRUCTION OF A POSITIVE-REAL WAVEFUNCTION

This tutorial shows how to reconstruct a **positive-real** wavefunction via training a *Restricted Boltzmann Machine* (RBM), the neural network behind QuCumber. The data used for training are σ^z measurements from a one-dimensional transverse-field Ising model (TFIM) with 10 sites at its critical point.

4.1 Transverse-field Ising model

The example dataset, located in `tfim1d_data.txt`, comprises 10,000 σ^z measurements from a one-dimensional TFIM with 10 sites at its critical point. The Hamiltonian for the TFIM is given by

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

where σ_i^z is the conventional spin-1/2 Pauli operator on site i. At the critical point, $J = h = 1$. By convention, spins are represented in binary notation with zero and one denoting the states spin-down and spin-up, respectively.

4.2 Using QuCumber to reconstruct the wavefunction

4.2.1 Imports

To begin the tutorial, first import the required Python packages.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

from qucumber.nn_states import PositiveWaveFunction
from qucumber.callbacks import MetricEvaluator

import qucumber.utils.training_statistics as ts
import qucumber.utils.data as data
import qucumber

# set random seed on cpu but not gpu, since we won't use gpu for this tutorial
qucumber.set_random_seed(1234, cpu=True, gpu=False)
```

The Python class `PositiveWaveFunction` contains generic properties of a RBM meant to reconstruct a positive-real wavefunction, the most notable one being the gradient function required for stochastic gradient descent.

To instantiate a `PositiveWaveFunction` object, one needs to specify the number of visible and hidden units in the RBM. The number of visible units, `num_visible`, is given by the size of the physical system, i.e. the number

of spins or qubits (10 in this case), while the number of hidden units, `num_hidden`, can be varied to change the expressiveness of the neural network.

Note: The optimal `num_hidden` : `num_visible` ratio will depend on the system. For the TFIM, having this ratio be equal to 1 leads to good results with reasonable computational effort.

4.2.2 Training

To evaluate the training in real time, we compute the fidelity between the true ground-state wavefunction of the system and the wavefunction that QuCumber reconstructs, $|\langle\psi|\psi_{RBM}\rangle|^2$, along with the Kullback-Leibler (KL) divergence (the RBM's cost function). As will be shown below, any custom function can be used to evaluate the training.

First, the training data and the true wavefunction of this system must be loaded using the `data` utility.

[2]:
```
psi_path = "tfim1d_psi.txt"
train_path = "tfim1d_data.txt"
train_data, true_psi = data.load_data(train_path, psi_path)
```

As previously mentioned, to instantiate a `PositiveWaveFunction` object, one needs to specify the number of visible and hidden units in the RBM; we choose them to be equal.

[3]:
```
nv = train_data.shape[-1]
nh = nv

nn_state = PositiveWaveFunction(num_visible=nv, num_hidden=nh, gpu=False)
```

If `gpu=True` (the default), QuCumber will attempt to run on a GPU if one is available (otherwise, QuCumber will default to CPU). If one `gpu=False`, QuCumber will run on the CPU.

Now we specify the hyperparameters of the training process:

1. `epochs`: the total number of training cycles that will be performed (default = 100)
2. `pbs` (`pos_batch_size`): the number of data points used in the positive phase of the gradient (default = 100)
3. `nbs` (`neg_batch_size`): the number of data points used in the negative phase of the gradient (default = 100)
4. `k`: the number of contrastive divergence steps (default = 1)
5. `lr`: the learning rate (default = 0.001)

   Note: For more information on the hyperparameters above, it is strongly encouraged that the user to read through the brief, but thorough theory document on RBMs located in the QuCumber documentation. One does not have to specify these hyperparameters, as their default values will be used without the user overwriting them. It is recommended to keep with the default values until the user has a stronger grasp on what these hyperparameters mean. The quality and the computational efficiency of the training will highly depend on the choice of hyperparameters. As such, playing around with the hyperparameters is almost always necessary.

For the TFIM with 10 sites, the following hyperparameters give excellent results:

[4]:
```
epochs = 500
pbs = 100
nbs = pbs
lr = 0.01
k = 10
```

For evaluating the training in real time, the `MetricEvaluator` is called every 100 epochs in order to calculate the training evaluators. The `MetricEvaluator` requires the following arguments:

1. `period`: the frequency of the training evaluators being calculated (e.g. `period=100` means that the `MetricEvaluator` will do an evaluation every 100 epochs)
2. A dictionary of functions you would like to reference to evaluate the training (arguments required for these functions are keyword arguments placed after the dictionary)

The following additional arguments are needed to calculate the fidelity and KL divergence in the `training_statistics` utility:

- `target_psi`: the true wavefunction of the system
- `space`: the Hilbert space of the system

The training evaluators can be printed out via the `verbose=True` statement.

Although the fidelity and KL divergence are excellent training evaluators, they are not practical to calculate in most cases; the user may not have access to the target wavefunction of the system, nor may generating the Hilbert space of the system be computationally feasible. However, evaluating the training in real time is extremely convenient.

Any custom function that the user would like to use to evaluate the training can be given to the `MetricEvaluator`, thus avoiding having to calculate fidelity and/or KL divergence. Any custom function given to `MetricEvaluator` must take the neural-network state (in this case, the `PositiveWaveFunction` object) and keyword arguments. As an example, we define a custom function `psi_coefficient`, which is the fifth coefficient of the reconstructed wavefunction multiplied by a parameter A.

[5]:
```
def psi_coefficient(nn_state, space, A, **kwargs):
    norm = nn_state.compute_normalization(space).sqrt_()
    return A * nn_state.psi(space)[0][4] / norm
```

Now the Hilbert space of the system can be generated for the fidelity and KL divergence.

[6]:
```
period = 10
space = nn_state.generate_hilbert_space()
```

Now the training can begin. The `PositiveWaveFunction` object has a property called `fit` which takes care of this. `MetricEvaluator` must be passed to the `fit` function in a list (`callbacks`).

[7]:
```
callbacks = [
    MetricEvaluator(
        period,
        {"Fidelity": ts.fidelity, "KL": ts.KL, "A_rbm_5": psi_coefficient},
        target=true_psi,
        verbose=True,
        space=space,
        A=3.0,
    )
]

nn_state.fit(
    train_data,
    epochs=epochs,
    pos_batch_size=pbs,
    neg_batch_size=nbs,
    lr=lr,
    k=k,
    callbacks=callbacks,
    time=True,
)
```

```
Epoch: 10       Fidelity = 0.500444     KL = 1.434037   A_rbm_5 = 0.111008
Epoch: 20       Fidelity = 0.570243     KL = 1.098804   A_rbm_5 = 0.140842
Epoch: 30       Fidelity = 0.681689     KL = 0.712384   A_rbm_5 = 0.192823
Epoch: 40       Fidelity = 0.781095     KL = 0.457683   A_rbm_5 = 0.222722
Epoch: 50       Fidelity = 0.840074     KL = 0.326949   A_rbm_5 = 0.239039
Epoch: 60       Fidelity = 0.875057     KL = 0.252105   A_rbm_5 = 0.239344
Epoch: 70       Fidelity = 0.895826     KL = 0.211282   A_rbm_5 = 0.239159
Epoch: 80       Fidelity = 0.907819     KL = 0.190410   A_rbm_5 = 0.245369
Epoch: 90       Fidelity = 0.914834     KL = 0.177129   A_rbm_5 = 0.238663
Epoch: 100      Fidelity = 0.920255     KL = 0.167432   A_rbm_5 = 0.246280
Epoch: 110      Fidelity = 0.924585     KL = 0.158587   A_rbm_5 = 0.244731
Epoch: 120      Fidelity = 0.928158     KL = 0.150159   A_rbm_5 = 0.236318
Epoch: 130      Fidelity = 0.932489     KL = 0.140405   A_rbm_5 = 0.243707
Epoch: 140      Fidelity = 0.936930     KL = 0.130399   A_rbm_5 = 0.242923
Epoch: 150      Fidelity = 0.941502     KL = 0.120001   A_rbm_5 = 0.246340
Epoch: 160      Fidelity = 0.946511     KL = 0.108959   A_rbm_5 = 0.243519
Epoch: 170      Fidelity = 0.951172     KL = 0.098144   A_rbm_5 = 0.235464
Epoch: 180      Fidelity = 0.955645     KL = 0.088780   A_rbm_5 = 0.237005
Epoch: 190      Fidelity = 0.959723     KL = 0.080219   A_rbm_5 = 0.234366
Epoch: 200      Fidelity = 0.962512     KL = 0.074663   A_rbm_5 = 0.227764
Epoch: 210      Fidelity = 0.965615     KL = 0.068804   A_rbm_5 = 0.233611
Epoch: 220      Fidelity = 0.967394     KL = 0.065302   A_rbm_5 = 0.233936
Epoch: 230      Fidelity = 0.969286     KL = 0.061641   A_rbm_5 = 0.230911
Epoch: 240      Fidelity = 0.970506     KL = 0.059283   A_rbm_5 = 0.225389
Epoch: 250      Fidelity = 0.971461     KL = 0.057742   A_rbm_5 = 0.233186
Epoch: 260      Fidelity = 0.973525     KL = 0.053430   A_rbm_5 = 0.225180
Epoch: 270      Fidelity = 0.975005     KL = 0.050646   A_rbm_5 = 0.228983
Epoch: 280      Fidelity = 0.976041     KL = 0.048451   A_rbm_5 = 0.231805
Epoch: 290      Fidelity = 0.977197     KL = 0.046058   A_rbm_5 = 0.232667
Epoch: 300      Fidelity = 0.977386     KL = 0.045652   A_rbm_5 = 0.239462
Epoch: 310      Fidelity = 0.979153     KL = 0.042036   A_rbm_5 = 0.232371
Epoch: 320      Fidelity = 0.979264     KL = 0.041764   A_rbm_5 = 0.224176
Epoch: 330      Fidelity = 0.981203     KL = 0.037786   A_rbm_5 = 0.231017
Epoch: 340      Fidelity = 0.982122     KL = 0.035848   A_rbm_5 = 0.233144
Epoch: 350      Fidelity = 0.982408     KL = 0.035287   A_rbm_5 = 0.239080
Epoch: 360      Fidelity = 0.983737     KL = 0.032537   A_rbm_5 = 0.232325
Epoch: 370      Fidelity = 0.984651     KL = 0.030705   A_rbm_5 = 0.233523
Epoch: 380      Fidelity = 0.985230     KL = 0.029546   A_rbm_5 = 0.235031
Epoch: 390      Fidelity = 0.985815     KL = 0.028345   A_rbm_5 = 0.235860
Epoch: 400      Fidelity = 0.986262     KL = 0.027459   A_rbm_5 = 0.240407
Epoch: 410      Fidelity = 0.986678     KL = 0.026623   A_rbm_5 = 0.229870
Epoch: 420      Fidelity = 0.987422     KL = 0.025197   A_rbm_5 = 0.235147
Epoch: 430      Fidelity = 0.987339     KL = 0.025400   A_rbm_5 = 0.227832
Epoch: 440      Fidelity = 0.988037     KL = 0.023930   A_rbm_5 = 0.237405
Epoch: 450      Fidelity = 0.988104     KL = 0.023838   A_rbm_5 = 0.241163
Epoch: 460      Fidelity = 0.988751     KL = 0.022605   A_rbm_5 = 0.233818
Epoch: 470      Fidelity = 0.988836     KL = 0.022364   A_rbm_5 = 0.241944
Epoch: 480      Fidelity = 0.989127     KL = 0.021844   A_rbm_5 = 0.235669
Epoch: 490      Fidelity = 0.989361     KL = 0.021288   A_rbm_5 = 0.242225
Epoch: 500      Fidelity = 0.989816     KL = 0.020486   A_rbm_5 = 0.232313
Total time elapsed during training: 87.096 s
```

All of these training evaluators can be accessed after the training has completed. The code below shows this, along with plots of each training evaluator as a function of epoch (training cycle number).

[8]:
```
# Note that the key given to the *MetricEvaluator* must be
# what comes after callbacks[0].
```

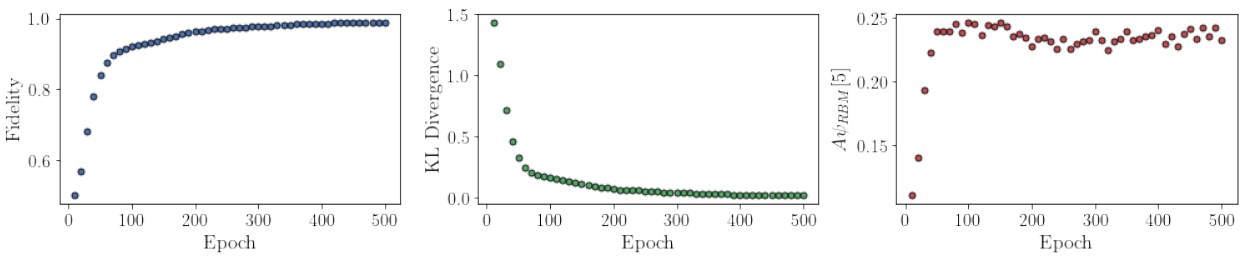
(continues on next page)

(continued from previous page)

```
fidelities = callbacks[0].Fidelity

# Alternatively, we can use the usual dictionary/list subsripting
# syntax. This is useful in cases where the name of the
# metric contains special characters or spaces.
KLs = callbacks[0]["KL"]
coeffs = callbacks[0]["A_rbm_5"]

epoch = np.arange(period, epochs + 1, period)
```

[9]:
```
# Some parameters to make the plots look nice
params = {
    "text.usetex": True,
    "font.family": "serif",
    "legend.fontsize": 14,
    "figure.figsize": (10, 3),
    "axes.labelsize": 16,
    "xtick.labelsize": 14,
    "ytick.labelsize": 14,
    "lines.linewidth": 2,
    "lines.markeredgewidth": 0.8,
    "lines.markersize": 5,
    "lines.marker": "o",
    "patch.edgecolor": "black",
}
plt.rcParams.update(params)
plt.style.use("seaborn-deep")
```

[10]:
```
# Plotting
fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(14, 3))
ax = axs[0]
ax.plot(epoch, fidelities, "o", color="C0", markeredgecolor="black")
ax.set_ylabel(r"Fidelity")
ax.set_xlabel(r"Epoch")

ax = axs[1]
ax.plot(epoch, KLs, "o", color="C1", markeredgecolor="black")
ax.set_ylabel(r"KL Divergence")
ax.set_xlabel(r"Epoch")

ax = axs[2]
ax.plot(epoch, coeffs, "o", color="C2", markeredgecolor="black")
ax.set_ylabel(r"$A\psi_{RBM}[5]$")
ax.set_xlabel(r"Epoch")

plt.tight_layout()
plt.show()
```



It should be noted that one could have just ran `nn_state.fit(train_samples)`, which uses the default hyperparameters and no training evaluators.

To demonstrate how important it is to find the optimal hyperparameters for a certain system, restart this notebook and comment out the original `fit` statement, then uncomment and run the cell below.

[11]:
```
# nn_state.fit(train_samples)
```

Using the non-default hyperparameters produced a fidelity of approximately 0.989, while the default hyperparameters yield approximately 0.523!

The trained RBM can be saved to a pickle file with the name `saved_params.pt` for future use:

[12]:
```
nn_state.save("saved_params.pt")
```

This saves the weights, visible biases and hidden biases as torch tensors under the following keys: `weights`, `visible_bias`, `hidden_bias`.

CHAPTER

FIVE

RECONSTRUCTION OF A COMPLEX WAVEFUNCTION

In this tutorial, a walkthrough of how to reconstruct a **complex** wavefunction via training a *Restricted Boltzmann Machine* (RBM), the neural network behind QuCumber, will be presented.

5.1 The wavefunction to be reconstructed

The simple wavefunction below describing two qubits (coefficients stored in `qubits_psi.txt`) will be reconstructed.

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$$

where the exact values of α, β, γ and δ used for this tutorial are

$$\begin{aligned}\alpha &= 0.2861 + 0.0539i \\ \beta &= 0.3687 - 0.3023i \\ \gamma &= -0.1672 - 0.3529i \\ \delta &= -0.5659 - 0.4639i\end{aligned}$$

The example dataset, `qubits_train.txt`, comprises of 500 σ measurements made in various bases (X, Y and Z). A corresponding file containing the bases for each data point in `qubits_train.txt`, `qubits_train_bases.txt`, is also required. As per convention, spins are represented in binary notation with zero and one denoting spin-down and spin-up, respectively.

5.2 Using qucumber to reconstruct the wavefunction

5.2.1 Imports

To begin the tutorial, first import the required Python packages.

[1]:
```
import numpy as np
import torch
import matplotlib.pyplot as plt

from qucumber.nn_states import ComplexWaveFunction

from qucumber.callbacks import MetricEvaluator

import qucumber.utils.unitaries as unitaries
import qucumber.utils.cplx as cplx
```

(continues on next page)

(continued from previous page)

```
import qucumber.utils.training_statistics as ts
import qucumber.utils.data as data
import qucumber

# set random seed on cpu but not gpu, since we won't use gpu for this tutorial
qucumber.set_random_seed(1234, cpu=True, gpu=False)
```

The Python class `ComplexWaveFunction` contains generic properties of a RBM meant to reconstruct a complex wavefunction, the most notable one being the gradient function required for stochastic gradient descent.

To instantiate a `ComplexWaveFunction` object, one needs to specify the number of visible and hidden units in the RBM. The number of visible units, `num_visible`, is given by the size of the physical system, i.e. the number of spins or qubits (2 in this case), while the number of hidden units, `num_hidden`, can be varied to change the expressiveness of the neural network.

Note: The optimal `num_hidden` : `num_visible` ratio will depend on the system. For the two-qubit wavefunction described above, good results can be achieved when this ratio is 1.

On top of needing the number of visible and hidden units, a `ComplexWaveFunction` object requires the user to input a dictionary containing the unitary operators (2x2) that will be used to rotate the qubits in and out of the computational basis, Z, during the training process. The `unitaries` utility will take care of creating this dictionary.

The `MetricEvaluator` class and `training_statistics` utility are built-in amenities that will allow the user to evaluate the training in real time.

Lastly, the `cplx` utility allows QuCumber to be able to handle complex numbers as they are not currently supported by PyTorch.

5.2.2 Training

To evaluate the training in real time, the fidelity between the true wavefunction of the system and the wavefunction that QuCumber reconstructs, $|\langle\psi|\psi_{RBM}\rangle|^2$, will be calculated along with the Kullback-Leibler (KL) divergence (the RBM's cost function). First, the training data and the true wavefunction of this system need to be loaded using the `data` utility.

[2]:
```
train_path = "qubits_train.txt"
train_bases_path = "qubits_train_bases.txt"
psi_path = "qubits_psi.txt"
bases_path = "qubits_bases.txt"

train_samples, true_psi, train_bases, bases = data.load_data(
    train_path, psi_path, train_bases_path, bases_path
)
```

The file `qubits_bases.txt` contains every unique basis in the `qubits_train_bases.txt` file. Calculation of the full KL divergence in every basis requires the user to specify each unique basis.

As previously mentioned, a `ComplexWaveFunction` object requires a dictionary that contains the unitary operators that will be used to rotate the qubits in and out of the computational basis, Z, during the training process. In the case of the provided dataset, the unitaries required are the well-known H, and K gates. The dictionary needed can be created with the following command.

[3]:
```
unitary_dict = unitaries.create_dict()
# unitary_dict = unitaries.create_dict(<unitary_name>=torch.tensor([[real part],
```

(continues on next page)

(continued from previous page)

```
#                                                                    [imaginary part]],
#                                                                   dtype=torch.double)
```

If the user wishes to add their own unitary operators from their experiment to `unitary_dict`, uncomment the block above. When `unitaries.create_dict()` is called, it will contain the identity and the H and K gates by default under the keys "Z", "X" and "Y", respectively.

The number of visible units in the RBM is equal to the number of qubits. The number of hidden units will also be taken to be the number of visible units.

[4]:
```
nv = train_samples.shape[-1]
nh = nv

nn_state = ComplexWaveFunction(
    num_visible=nv, num_hidden=nh, unitary_dict=unitary_dict, gpu=False
)
```

If `gpu=True` (the default), QuCumber will attempt to run on a GPU if one is available (if one is not available, QuCumber will fall back to CPU). If one wishes to guarantee that QuCumber runs on the CPU, add the flag `gpu=False` in the `ComplexWaveFunction` object instantiation.

Now the hyperparameters of the training process can be specified.

1. `epochs`: the total number of training cycles that will be performed (default = 100)
2. `pos_batch_size`: the number of data points used in the positive phase of the gradient (default = 100)
3. `neg_batch_size`: the number of data points used in the negative phase of the gradient (default = `pos_batch_size`)
4. `k`: the number of contrastive divergence steps (default = 1)
5. `lr`: the learning rate (default = 0.001)

   Note: For more information on the hyperparameters above, it is strongly encouraged that the user to read through the brief, but thorough theory document on RBMs. One does not have to specify these hyperparameters, as their default values will be used without the user overwriting them. It is recommended to keep with the default values until the user has a stronger grasp on what these hyperparameters mean. The quality and the computational efficiency of the training will highly depend on the choice of hyperparameters. As such, playing around with the hyperparameters is almost always necessary.

The two-qubit example in this tutorial should be extremely easy to train, regardless of the choice of hyperparameters. However, the hyperparameters below will be used.

[5]:
```
epochs = 500
pbs = 100  # pos_batch_size
nbs = pbs  # neg_batch_size
lr = 0.1
k = 10
```

For evaluating the training in real time, the `MetricEvaluator` will be called to calculate the training evaluators every 10 epochs. The `MetricEvaluator` requires the following arguments.

1. `period`: the frequency of the training evaluators being calculated (e.g. `period=200` means that the `MetricEvaluator` will compute the desired metrics every 200 epochs)
2. A dictionary of functions you would like to reference to evaluate the training (arguments required for these functions are keyword arguments placed after the dictionary)

The following additional arguments are needed to calculate the fidelity and KL divergence in the `training_statistics` utility.

- `target_psi` (the true wavefunction of the system)
- `space` (the entire Hilbert space of the system)

The training evaluators can be printed out via the `verbose=True` statement.

Although the fidelity and KL divergence are excellent training evaluators, they are not practical to calculate in most cases; the user may not have access to the target wavefunction of the system, nor may generating the Hilbert space of the system be computationally feasible. However, evaluating the training in real time is extremely convenient.

Any custom function that the user would like to use to evaluate the training can be given to the `MetricEvaluator`, thus avoiding having to calculate fidelity and/or KL divergence. As an example, functions that calculate the the norm of each of the reconstructed wavefunction's coefficients are presented. Any custom function given to `MetricEvaluator` must take the neural-network state (in this case, the `ComplexWaveFunction` object) and keyword arguments. Although the given example requires the Hilbert space to be computed, the scope of the `MetricEvaluator`'s ability to be able to handle any function should still be evident.

[6]:
```
def alpha(nn_state, space, **kwargs):
    rbm_psi = nn_state.psi(space)
    normalization = nn_state.normalization(space).sqrt_()
    alpha_ = cplx.norm(
        torch.tensor([rbm_psi[0][0], rbm_psi[1][0]], device=nn_state.device)
        / normalization
    )

    return alpha_


def beta(nn_state, space, **kwargs):
    rbm_psi = nn_state.psi(space)
    normalization = nn_state.normalization(space).sqrt_()
    beta_ = cplx.norm(
        torch.tensor([rbm_psi[0][1], rbm_psi[1][1]], device=nn_state.device)
        / normalization
    )

    return beta_


def gamma(nn_state, space, **kwargs):
    rbm_psi = nn_state.psi(space)
    normalization = nn_state.normalization(space).sqrt_()
    gamma_ = cplx.norm(
        torch.tensor([rbm_psi[0][2], rbm_psi[1][2]], device=nn_state.device)
        / normalization
    )

    return gamma_


def delta(nn_state, space, **kwargs):
    rbm_psi = nn_state.psi(space)
    normalization = nn_state.normalization(space).sqrt_()
    delta_ = cplx.norm(
        torch.tensor([rbm_psi[0][3], rbm_psi[1][3]], device=nn_state.device)
        / normalization
```

(continues on next page)

(continued from previous page)

```
    )

    return delta_
```

Now the basis of the Hilbert space of the system must be generated in order to compute the fidelity, KL divergence, and the dictionary of functions the user would like to compute. These metrics will be evaluated every `period` epochs, which is a parameter that must be given to the `MetricEvaluator`.

Note that some of the coefficients are not being evaluated as they are commented out. This is simply to avoid cluttering the output, and may be uncommented by the user.

[7]:
```
period = 25
space = nn_state.generate_hilbert_space()

callbacks = [
    MetricEvaluator(
        period,
        {
            "Fidelity": ts.fidelity,
            "KL": ts.KL,
            "norm": alpha,
            # "norm": beta,
            # "norm": gamma,
            # "norm": delta,
        },
        target=true_psi,
        bases=bases,
        verbose=True,
        space=space,
    )
]
```

Now the training can begin. The `ComplexWaveFunction` object has a function called `fit` which takes care of this.

[8]:
```
nn_state.fit(
    train_samples,
    epochs=epochs,
    pos_batch_size=pbs,
    neg_batch_size=nbs,
    lr=lr,
    k=k,
    input_bases=train_bases,
    callbacks=callbacks,
    time=True,
)
```

```
Epoch: 25       Fidelity = 0.940240     KL = 0.032256   norm = 0.258429
Epoch: 50       Fidelity = 0.974944     KL = 0.017143   norm = 0.260490
Epoch: 75       Fidelity = 0.984727     KL = 0.012232   norm = 0.270684
Epoch: 100      Fidelity = 0.987769     KL = 0.010389   norm = 0.269163
Epoch: 125      Fidelity = 0.988929     KL = 0.009581   norm = 0.261813
Epoch: 150      Fidelity = 0.989075     KL = 0.009273   norm = 0.271764
Epoch: 175      Fidelity = 0.989197     KL = 0.008928   norm = 0.267943
Epoch: 200      Fidelity = 0.989451     KL = 0.008817   norm = 0.259327
Epoch: 225      Fidelity = 0.990894     KL = 0.007215   norm = 0.269941
Epoch: 250      Fidelity = 0.991517     KL = 0.006804   norm = 0.261673
```

(continues on next page)

(continued from previous page)

```
Epoch: 275      Fidelity = 0.991808     KL = 0.006408   norm = 0.261002
Epoch: 300      Fidelity = 0.992318     KL = 0.005788   norm = 0.274654
Epoch: 325      Fidelity = 0.992078     KL = 0.005881   norm = 0.266831
Epoch: 350      Fidelity = 0.991938     KL = 0.006020   norm = 0.262980
Epoch: 375      Fidelity = 0.991670     KL = 0.006181   norm = 0.270877
Epoch: 400      Fidelity = 0.992082     KL = 0.005945   norm = 0.255576
Epoch: 425      Fidelity = 0.992678     KL = 0.005130   norm = 0.259746
Epoch: 450      Fidelity = 0.993102     KL = 0.004702   norm = 0.259373
Epoch: 475      Fidelity = 0.993109     KL = 0.004765   norm = 0.255803
Epoch: 500      Fidelity = 0.992805     KL = 0.004785   norm = 0.261486
Total time elapsed during training: 49.059 s
```

All of these training evaluators can be accessed after the training has completed, as well. The code below shows this, along with plots of each training evaluator versus the training cycle number (epoch).

[9]:
```
# Note that the key given to the *MetricEvaluator* must be
# what comes after callbacks[0].
fidelities = callbacks[0].Fidelity

# Alternatively, we may use the usual dictionary/list subscripting
# syntax. This is useful in cases where the name of the metric
# may contain special characters or spaces.
KLs = callbacks[0]["KL"]
coeffs = callbacks[0]["norm"]
epoch = np.arange(period, epochs + 1, period)
```

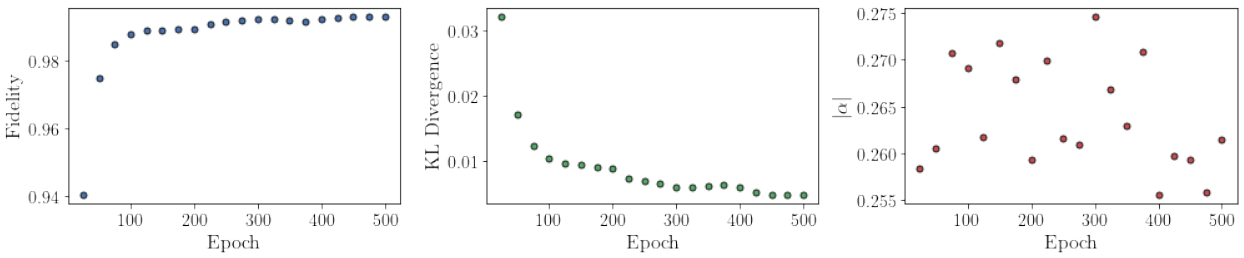
[10]:
```
# Some parameters to make the plots look nice
params = {
    "text.usetex": True,
    "font.family": "serif",
    "legend.fontsize": 14,
    "figure.figsize": (10, 3),
    "axes.labelsize": 16,
    "xtick.labelsize": 14,
    "ytick.labelsize": 14,
    "lines.linewidth": 2,
    "lines.markeredgewidth": 0.8,
    "lines.markersize": 5,
    "lines.marker": "o",
    "patch.edgecolor": "black",
}
plt.rcParams.update(params)
plt.style.use("seaborn-deep")
```

[11]:
```
fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(14, 3))
ax = axs[0]
ax.plot(epoch, fidelities, "o", color="C0", markeredgecolor="black")
ax.set_ylabel(r"Fidelity")
ax.set_xlabel(r"Epoch")

ax = axs[1]
ax.plot(epoch, KLs, "o", color="C1", markeredgecolor="black")
ax.set_ylabel(r"KL Divergence")
ax.set_xlabel(r"Epoch")

ax = axs[2]
```

(continues on next page)

(continued from previous page)

```
ax.plot(epoch, coeffs, "o", color="C2", markeredgecolor="black")
ax.set_ylabel(r"$\vert\alpha\vert$")
ax.set_xlabel(r"Epoch")

plt.tight_layout()
plt.show()
```



It should be noted that one could have just run `nn_state.fit(train_samples)` using the default hyperparameters and no training evaluators, which would induce different convergence behavior.

At the end of the training process, the network parameters (the weights, visible biases, and hidden biases) are stored in the `ComplexWaveFunction` object. One can save them to a pickle file, which will be called `saved_params.pt`, with the following command.

[12]:
```
nn_state.save("saved_params.pt")
```

This saves the weights, visible biases and hidden biases as torch tensors under the following keys: `weights`, `visible_bias`, `hidden_bias`.

CHAPTER

SIX

RECONSTRUCTION OF A DENSITY MATRIX

In this tutorial, a walkthrough of how to reconstruct a density matrix via training a pair of modified *Restricted Boltzmann Machines* is presented

6.1 The density matrix to be reconstructed

The density matrix that will be be reconstructed is the density matrix associated with the 2-qubit W state

$$|\psi\rangle = \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

so that

$$\rho = |\psi\rangle\langle\psi|$$

with global depolarization probability $p_{dep} = 0.5$ such that

$$\rho_{new} = (1 - p_{dep})\,\rho + \frac{p_{dep}}{2^N} I$$

where I is the identity matrix, representing the maximally mixed state.

The example dataset, `N2_W_state_100_samples_data.txt`, is comprised of 900 σ^z measurements, 100 in each of the 3^2 permutations of two of the bases X, Y and Z. A corresponding file containing the bases for each data point, `N2_W_state_100_samples_bases.txt`, is also required.

In this tutorial is also included versions with 1000 measurements in each basis, to illustrate the improvements to reconstruction fidelity of a larger data set. The measurements and bases are stored in `N2_W_state_1000_samples_data.txt`, and `N2_W_state_1000_samples_bases.txt` respectively.

The set of all 3^2 bases in which measurements are made is stored in `N2_IC_bases.txt`. Finally, the real and imaginary parts of the matrix are stored in `N2_W_state_target_real.txt` and `N2_W_state_target_imag.txt` respectively. As per convention, spins are represented in binary notation with zero and one denoting spin-up and spin-down, respectively.

6.2 Using QuCumber to reconstruct the density matrix

6.2.1 Imports

To begin the tutorial, first import the required Python packages.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

import torch

from qucumber.nn_states import DensityMatrix

from qucumber.callbacks import MetricEvaluator
import qucumber.utils.unitaries as unitaries

import qucumber.utils.training_statistics as ts
import qucumber.utils.data as data
import qucumber

# set random seed on cpu but not gpu, since we won't use gpu for this tutorial
qucumber.set_random_seed(1234, cpu=True, gpu=False)
```

The Python class `DensityMatrix` contains the properties of an RBM needed to reconstruct the density matrix, as demonstrated in this paper here.

To instantiate a `DensityMatrix` object, one needs to specify the number of visible, hidden and auxiliary units in the RBM. The number of visible units, `num_visible`, is given by the size of the physical system, i.e. the number of spins or qubits (2 in this case). On the other hand, the number of hidden units, `num_hidden`, can be varied to change the expressiveness of the neural network, and the number of auxiliary units, `num_aux`, can be varied depending on the extent of purification required of the system.

On top of needing the number of visible, hidden and auxiliary units, a `DensityMatrix` object requires the user to input a dictionary containing the unitary operators (2x2) that will be used to rotate the qubits in and out of the computational basis, Z, during the training process. The `unitaries` utility will take care of creating this dictionary.

The `MetricEvaluator` class and `training_statistics` utility are built-in amenities that will allow the user to evaluate the training in real time.

6.2.2 Training

To evaluate the training in real time, the fidelity between the true wavefunction of the system and the wavefunction that QuCumber reconstructs,

$$\mathrm{Tr}\left(\sqrt{\sqrt{\rho_{RBM}}\rho\sqrt{\rho_{RBM}}}\right)$$

will be calculated along with the Kullback-Leibler (KL) divergence (the RBM's cost function). First, the training data and the true wavefunction of this system need to be loaded using the `data` utility.

[2]:
```
train_path = "N2_W_state_100_samples_data.txt"
train_bases_path = "N2_W_state_100_samples_bases.txt"
matrix_path_real = "N2_W_state_target_real.txt"
matrix_path_imag = "N2_W_state_target_imag.txt"
bases_path = "N2_IC_bases.txt"


train_samples, true_matrix, train_bases, bases = data.load_data_DM(
    train_path, matrix_path_real, matrix_path_imag, train_bases_path, bases_path
)
```

The file `N2_IC_bases.txt` contains every unique basis in the `N2_W_state_100_samples_bases.txt` file. Calculation of the full KL divergence in every basis requires the user to specify each unique basis.

As previously mentioned, a `DensityMatrix` object requires a dictionary that contains the unitary operators that will be used to rotate the qubits in and out of the computational basis, Z, during the training process. In the case of the provided dataset, the unitaries required are the well-known H, and K gates. The dictionary needed can be created with the following command.

[3]:
```
unitary_dict = unitaries.create_dict()
# unitary_dict = unitaries.create_dict(unitary_name=torch.tensor([[real part],
#                                                                 [imaginary part]],
#                                                                dtype=torch.double)
```

If the user wishes to add their own unitary operators from their experiment to `unitary_dict`, uncomment the block above. When `unitaries.create_dict()` is called, it will contain the identity and the H and K gates by default under the keys "Z", "X" and "Y", respectively.

The number of visible units in the RBM is equal to the number of qubits. The number of hidden units will also be taken to be the number of visible units.

[4]:
```
nv = train_samples.shape[-1]
nh = na = nv

nn_state = DensityMatrix(
    num_visible=nv, num_hidden=nh, num_aux=na, unitary_dict=unitary_dict, gpu=False
)
```

The number of visible, hidden, and auxiliary units must now be specified. These are given by `nv`, `nh` and `na` respectively. The number of visible units is equal to the size of the system. The hidden and auxiliary units are hyperparameters that must be provided by the user. With these, a `DensityMatrix` object can be instantiated.

Now the hyperparameters of the training process can be specified.

1. `epochs`: the total number of training cycles that will be performed (default = 100)
2. `pos_batch_size`: the number of data points used in the positive phase of the gradient (default = 100)
3. `neg_batch_size`: the number of data points used in the negative phase of the gradient (default = `pos_batch_size`)
4. `k`: the number of contrastive divergence steps (default = 1)
5. `lr`: coefficient that scales the default value of the (non-constant) learning rate of the Adadelta algorithm (default = 1)

Extra hyperparameters that we will be passing to the learning rate scheduler: 6. `lr_drop_epochs`: the number of epochs after which to decay the learning rate by `lr_drop_factor` 7. `lr_drop_factor`: the factor by which the learning rate drops at `lr_drop_epoch` or all epochs in `lr_drop_epoch` if it is a list

Set `lr_drop_factor` to 1.0 to maintain constant "base" learning rate for Adadelta optimization. The choice shown here is optimized for this tutorial, but can (and should) be varied according to each instance

Note: For more information on the hyperparameters above, it is strongly encouraged that the user to read through the brief, but thorough theory document on RBMs. One does not have to specify these hyperparameters, as their default values will be used without the user overwriting them. It is recommended to keep with the default values until the user has a stronger grasp on what these hyperparameters mean. The quality and the computational efficiency of the training will highly depend on the choice of hyperparameters. As such, playing around with the hyperparameters is almost always necessary.

[5]:
```
epochs = 500
pbs = 100  # pos_batch_size
nbs = pbs  # neg_batch_size
lr = 10
```

(continues on next page)

(continued from previous page)

```
k = 10
lr_drop_epoch = 125
lr_drop_factor = 0.5
```

For evaluating the training in real time, the `MetricEvaluator` will be called to calculate the training evaluators every `period` epochs. The `MetricEvaluator` requires the following arguments.

1. `period`: the frequency of the training evaluators being calculated (e.g. `period=200` means that the `MetricEvaluator` will compute the desired metrics every 200 epochs)
2. A dictionary of functions you would like to reference to evaluate the training (arguments required for these functions are keyword arguments placed after the dictionary)

The following additional arguments are needed to calculate the fidelity and KL divergence in the `training_statistics` utility.

- `target_matrix` (the true density matrix of the system)
- `space` (the entire Hilbert space of the system)

The training evaluators can be printed out via the `verbose=True` statement.

Although the fidelity and KL divergence are excellent training evaluators, they are not practical to calculate in most cases; the user may not have access to the target wavefunction of the system, nor may generating the Hilbert space of the system be computationally feasible. However, evaluating the training in real time is extremely convenient.

Any custom function that the user would like to use to evaluate the training can be given to the `MetricEvaluator`, thus avoiding having to calculate fidelity and/or KL divergence. As an example, a function to compute the partition function of the current density matrix is presented. Any custom function given to `MetricEvaluator` must take the neural-network state (in this case, the `Density` object) and keyword arguments. Although the given example requires the Hilbert space to be computed, the scope of the `MetricEvaluator`'s ability to be able to handle any function should still be evident.

[6]:
```
def partition(nn_state, space, **kwargs):
    return nn_state.rbm_am.partition(space)
```

Now the Hilbert space of the system must be generated for the fidelity and KL divergence and the dictionary of functions the user would like to compute every `period` epochs must be given to the `MetricEvaluator`.

[7]:
```
period = 25
space = nn_state.generate_hilbert_space()

callbacks = [
    MetricEvaluator(
        period,
        {
            "Fidelity": ts.fidelity,
            "KL": ts.KL,
            # "Partition Function": partition,
        },
        target=true_matrix,
        bases=bases,
        verbose=True,
        space=space,
    )
]
```

Now the training can begin. The `DensityMatrix` object has a function called `fit` which takes care of this.

[8]:
```
nn_state.fit(
    data=train_samples,
    input_bases=train_bases,
    epochs=epochs,
    pos_batch_size=pbs,
    neg_batch_size=nbs,
    lr=lr,
    k=k,
    bases=bases,
    callbacks=callbacks,
    time=True,
    optimizer=torch.optim.Adadelta,
    scheduler=torch.optim.lr_scheduler.StepLR,
    scheduler_args={"step_size": lr_drop_epoch, "gamma": lr_drop_factor},
)
```

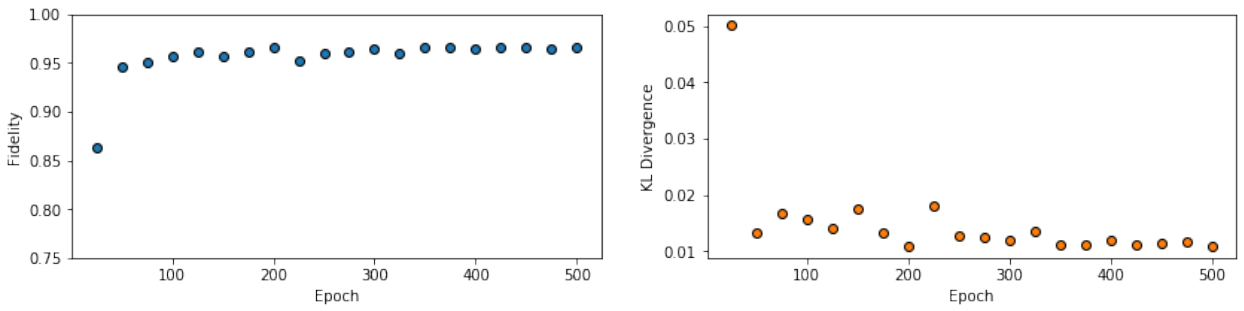
```
Epoch: 25       Fidelity = 0.863061     KL = 0.050122
Epoch: 50       Fidelity = 0.946054     KL = 0.013152
Epoch: 75       Fidelity = 0.950760     KL = 0.016754
Epoch: 100      Fidelity = 0.957204     KL = 0.015601
Epoch: 125      Fidelity = 0.960522     KL = 0.013925
Epoch: 150      Fidelity = 0.957411     KL = 0.017446
Epoch: 175      Fidelity = 0.961068     KL = 0.013377
Epoch: 200      Fidelity = 0.966589     KL = 0.010754
Epoch: 225      Fidelity = 0.951836     KL = 0.017970
Epoch: 250      Fidelity = 0.960255     KL = 0.012612
Epoch: 275      Fidelity = 0.961232     KL = 0.012477
Epoch: 300      Fidelity = 0.963946     KL = 0.011832
Epoch: 325      Fidelity = 0.959750     KL = 0.013571
Epoch: 350      Fidelity = 0.965108     KL = 0.011095
Epoch: 375      Fidelity = 0.965353     KL = 0.011048
Epoch: 400      Fidelity = 0.963568     KL = 0.011941
Epoch: 425      Fidelity = 0.966334     KL = 0.011148
Epoch: 450      Fidelity = 0.965549     KL = 0.011321
Epoch: 475      Fidelity = 0.965054     KL = 0.011573
Epoch: 500      Fidelity = 0.965568     KL = 0.010963
Total time elapsed during training: 175.240 s
```

All of these training evaluators can be accessed after the training has completed, as well. The code below shows this, along with plots of each training evaluator versus the training cycle number (epoch).

[9]:
```
fidelities = callbacks[0]["Fidelity"]
KLs = callbacks[0]["KL"]
epoch = np.arange(period, epochs + 1, period)
```

[10]:
```
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(14, 3))
ax = axs[0]
ax.plot(epoch, fidelities, "o", color="C0", markeredgecolor="black")
ax.set_ylabel(r"Fidelity")
ax.set_xlabel(r"Epoch")
ax.set_ylim(0.75, 1.00)

ax = axs[1]
ax.plot(epoch, KLs, "o", color="C1", markeredgecolor="black")
ax.set_ylabel(r"KL Divergence")
ax.set_xlabel(r"Epoch")
```

[10]:
```
Text(0.5, 0, 'Epoch')
```



This saves the weights, and biases of the two internal RBMs as dictionaries containing torch tensors.

[11]:
```
nn_state.save("saved_params_W_state.pt")
```

CHAPTER

SEVEN

SAMPLING AND CALCULATING OBSERVABLES

7.1 Generate new samples

Firstly, to generate meaningful data, an RBM needs to be trained. Please refer to the tutorials 1 and 2 on training an RBM-based Neural-Network-State if doing so using QuCumber is unclear. A Neural-Network-State (`nn_state`) of a positive-real wavefunction describing a transverse-field Ising model (TFIM) with 10 sites has already been trained in the first tutorial, with the parameters of the machine saved here as `saved_params.pt`. The `autoload` function can be employed here to instantiate the corresponding `PositiveWaveFunction` object from the saved `nn_state` parameters.

[1]:
```
import numpy as np
import torch
import matplotlib.pyplot as plt

import qucumber
from qucumber.nn_states import PositiveWaveFunction, DensityMatrix
from qucumber.observables import ObservableBase
from qucumber.observables.pauli import flip_spin
from qucumber.utils import cplx

from quantum_ising_chain import TFIMChainEnergy, Convergence

# set random seed on cpu but not gpu, since we won't use gpu for this tutorial
qucumber.set_random_seed(1234, cpu=True, gpu=False)
nn_state = PositiveWaveFunction.autoload("saved_params.pt", gpu=False)
```

A `PositiveWaveFunction` object has a property called `sample` that allows us to sample the learned distribution of TFIM chains. The it takes the following arguments (along with a few others which are not relevant for our purposes):

1. `k`: the number of Gibbs steps to perform to generate the new samples. Increasing this number will produce samples closer to the learned distribution, but will require more computation.
2. `num_samples`: the number of new data points to be generated

[2]:
```
new_samples = nn_state.sample(k=100, num_samples=10000)
print(new_samples)
```

```
tensor([[1., 0., 1.,  ..., 1., 1., 1.],
        [0., 0., 0.,  ..., 0., 0., 1.],
        [0., 0., 0.,  ..., 1., 0., 1.],
        ...,
        [0., 0., 0.,  ..., 0., 1., 1.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 1., 1.,  ..., 0., 0., 0.]], dtype=torch.float64)
```

7.1.1 Magnetization

With the newly generated samples, the user can now easily calculate observables that do not require any information associated with the wavefunction and hence the `nn_state`. These are observables which are diagonal in the computational (Pauli Z) basis. A great example of this is the magnetization (in the Z direction). To calculate the magnetization, the newly-generated samples must be converted to ± 1 from 1 and 0, respectively. The function below does the trick.

[3]:
```
def to_pm1(samples):
    return samples.mul(2.0).sub(1.0)
```

Now, the (absolute) magnetization in the Z-direction is calculated as follows.

[4]:
```
def Magnetization(samples):
    return to_pm1(samples).mean(1).abs().mean()


magnetization = Magnetization(new_samples).item()

print("Magnetization = %.5f" % magnetization)
```

```
Magnetization = 0.55752
```

The exact value for the magnetization is 0.5610.

The magnetization and the newly-generated samples can also be saved to a pickle file along with the `nn_state` parameters in the *PositiveWaveFunction* object.

[5]:
```
nn_state.save(
    "saved_params_and_new_data.pt",
    metadata={"samples": new_samples, "magnetization": magnetization},
)
```

The `metadata` argument in the `save` function takes in a dictionary of data that you would like to save alongside the `nn_state` parameters.

7.2 Calculate an observable using the *Observable* module

7.2.1 Magnetization (again)

QuCumber provides the `Observable` module to simplify estimation of expectations and variances of observables in memory efficient ways. To start off, we'll repeat the above example using the `SigmaZ` Observable module provided with QuCumber.

[6]:
```
from qucumber.observables import SigmaZ
```

We'll compute the absolute magnetization again, for the sake of comparison with the previous example. We want to use the samples drawn earlier to perform this estimate, so we use the `statistics_from_samples` function:

[7]:
```
sz = SigmaZ(absolute=True)
sz.statistics_from_samples(nn_state, new_samples)
```

[7]:
```
{'mean': 0.5575200000000005,
 'variance': 0.09791724132414,
 'std_error': 0.0031291730748576373,
 'num_samples': 10000}
```

With this function we get the variance and standard error for free. Now you may be asking: "That's not too difficult, I could have computed those myself!". The power of the `Observable` module comes from the fact that it simplifies estimation of these values over a large number of samples. The `statistics` function computes these statistics by generating the samples internally. Let's see it in action:

[8]:
```
%time sz.statistics(nn_state, num_samples=10000, burn_in=100)
# just think of burn_in as being equivalent to k for now
```

```
CPU times: user 1.74 s, sys: 0 ns, total: 1.74 s
Wall time: 504 ms
```

[8]:
```
{'mean': 0.5534800000000003,
 'variance': 0.09726161576157935,
 'std_error': 0.0031186794603097535,
 'num_samples': 10000}
```

Let's consider what is taking place under the hood at the moment. The `statistics` function is drawing 10000 samples from the given `nn_state`, and cycling it through the visible and hidden layers for 100 Block Gibbs steps before computing the statistics. This means that, at any given time it has to hold a matrix with 10000 rows and 10 (the number of lattice sites) columns in memory, which becomes infeasible for large lattices or if we want to use more samples to bring our standard error down. To bypass this issue, the `statistics` function allows us to specify the number of Markov Chains to evolve using the `nn_state`, and will sample from these chains multiple times to produce enough samples. It takes the following arguments:

- `num_samples`: the number of samples to generate internally
- `num_chains`: the number of Markov chains to run in parallel (default = 0, meaning `num_chains = num_samples`)
- `burn_in`: the number of Gibbs steps to perform before recording any samples (default = 1000)
- `steps`: the number of Gibbs steps to perform between each sample; increase this to reduce the autocorrelation between samples (default = 1)
- `initial_state`: the initial state of the Markov Chain. If given, `num_chains` will be ignored. (default = `None`)
- `overwrite`: Whether to overwrite the `initial_state` tensor, with the updated state of the Markov chain. (default = `False`)

The `statistics` function will also return a dictionary containing the mean, standard error (of the mean), the variance, and the total number of samples that were drawn with the keys `mean`, `std_error`, `variance`, and `num_samples` respectively.

[9]:
```
%time sz.statistics(nn_state, num_samples=10000, num_chains=1000, burn_in=100,␣
↪steps=2)
```

```
CPU times: user 331 ms, sys: 0 ns, total: 331 ms
Wall time: 83.8 ms
```

[9]:
```
{'mean': 0.55116,
 'variance': 0.09716837123712346,
 'std_error': 0.003117184165831776,
 'num_samples': 10000}
```

Recall that, earlier, we had produced a batch of samples `new_samples` which we assumed were already converged to the equilibrium after 100 Gibbs steps. We can use these pre-computed samples to skip the "burn-in" phase like so:

[10]:
```
%%time
sz.statistics(
```

(continues on next page)

(continued from previous page)

```
    nn_state, num_samples=10000, burn_in=0, steps=2, initial_state=new_samples[:1000,␣
↪:]
)
```

```
CPU times: user 73.9 ms, sys: 0 ns, total: 73.9 ms
Wall time: 21.6 ms
```

[10]:
```
{'mean': 0.5508999999999998,
 'variance': 0.09669885988598853,
 'std_error': 0.003109644029241748,
 'num_samples': 10000}
```

We only took the first 1000 samples from `new_samples` in order to keep the number of Markov Chains the same as the previous cell. Notice how much time was saved by initializing the chains using pre-equilibrated samples. In fact, one could save even more time by skipping the generation of pre-equilibrated samples and using the `nn_state`'s training data instead!

In addition to using less memory (since the matrix held in memory is now of size `num_chains` x `num_sites` = 1000 x 10), using fewer chains also produced a decent speed boost! Next, we'll try increasing the total number of drawn samples:

[11]:
```
sz.statistics(nn_state, num_samples=int(1e7), num_chains=1000, burn_in=100, steps=2)
```

[11]:
```
{'mean': 0.5505633800000023,
 'variance': 0.09801013840398955,
 'std_error': 9.900006990098014e-05,
 'num_samples': 10000000}
```

Note how much we reduced our standard error just by increasing the number of drawn samples. Finally, we can also draw samples of measurements **of the observable** using the `sample` function:

[12]:
```
sz.sample(nn_state, k=100, num_samples=50)
```

[12]:
```
tensor([[0.2000, 1.0000, 0.0000, 0.4000, 0.2000, 0.4000, 0.4000, 0.4000, 0.6000,
         1.0000, 1.0000, 0.6000, 0.4000, 0.0000, 0.4000, 0.4000, 0.8000, 1.0000,
         0.0000, 0.4000, 0.0000, 0.0000, 0.8000, 0.8000, 0.8000, 0.8000, 0.6000,
         0.8000, 0.6000, 1.0000, 0.4000, 0.4000, 0.4000, 0.4000, 0.8000, 0.2000,
         0.2000, 0.0000, 0.8000, 0.4000, 0.6000, 0.0000, 0.2000, 1.0000, 0.4000,
         0.8000, 0.2000, 0.4000, 1.0000, 0.6000]], dtype=torch.float64)
```

Note that this function does not perform any fancy sampling tricks like `statistics` and is therefore susceptible to "Out of Memory" errors.

7.2.2 TFIM Energy

Some observables cannot be computed directly from samples, but instead depend on the `nn_state` as previously mentioned. For example, the magnetization of the TFIM simply depends on the samples the user gives as input. While we did provide the `nn_state` as an argument when calling `statistics_from_samples`, `SigmaZ` ignores it. The TFIM energy, on the other hand, is much more complicated. Consider the TFIM Hamiltonian:

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

As our `nn_state` was trained in the Z-basis, the off-diagonal transverse-field term is impossible to compute just from the samples; we need to know the value of the wavefunction for each sample as well. An example for the computation of the energy is provided in the python file `quantum_ising_chain.py`, which takes advantage of QuCumber's `Observable` module.

`quantum_ising_chain.py` comprises of a class that computes the energy of a TFIM (`TFIMChainEnergy`) that inherits properties from the `Observable` module. To instantiate a `TFIMChainEnergy` object, the $\frac{h}{J}$ value must be specified. The trained `nn_state` parameters are from the first tutorial, where the example data was from the TFIM with 10 sites at its critical point ($\frac{h}{J} = 1$).

[13]:
```
h = 1

tfim_energy = TFIMChainEnergy(h)
```

To go ahead and calculate the mean energy and its standard error from the previously generated samples from this tutorial (`new_samples`), the `statistics_from_samples` function in the `Observable` module is called upon.

[14]:
```
energy_stats = tfim_energy.statistics_from_samples(nn_state, new_samples)
print("Mean: %.4f" % energy_stats["mean"], "+/- %.4f" % energy_stats["std_error"])
print("Variance: %.4f" % energy_stats["variance"])
```

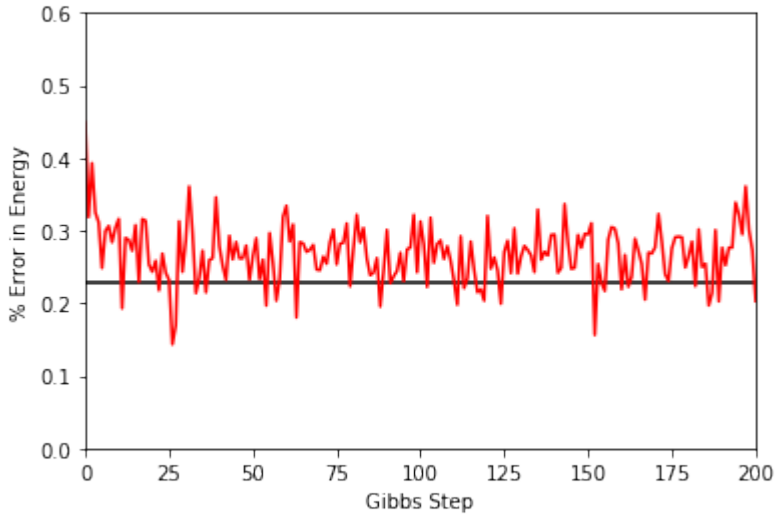
```
Mean: -1.2353 +/- 0.0005
Variance: 0.0022
```

The exact value for the energy is -1.2381.

To illustrate how quickly the energy converges as a function of the sampling step (i.e. the number of Gibbs steps to perform to generate a new batch of samples), `steps`, the `Convergence` function in `quantum_ising_chain.py` will do the trick. `Convergence` creates a batch of random samples initially, which is then used to generate a new batch of samples from the `nn_state`. The TFIM energy will be calculated at every Gibbs step. Note that this function is not available in the QuCumber API; it is only used here as an illustrative example.

[15]:
```
steps = 200
num_samples = 10000

dict_observables = Convergence(nn_state, tfim_energy, num_samples, steps)

energy = dict_observables["energies"]
err_energy = dict_observables["error"]

step = np.arange(steps + 1)

E0 = -1.2381

ax = plt.axes()
ax.plot(step, abs((E0 - energy) / E0) * 100, color="red")
ax.hlines(abs((E0 - energy_stats["mean"]) / E0) * 100, 0, 200, color="black")
ax.set_xlim(0, steps)
ax.set_ylim(0, 0.6)
ax.set_xlabel("Gibbs Step")
ax.set_ylabel("% Error in Energy")
```

[15]:
```
Text(0, 0.5, '% Error in Energy')
```



One can see a brief transient period in the magnetization observable, before the state of the machine "warms up" to equilibrium (this explains the `burn_in` argument we saw earlier). After that, the values fluctuate around the estimated mean (the horizontal black line).

7.2.3 Combining observables

One may also add / subtract and multiply observables with each other or with real numbers. To illustrate this, we will build an alternative implementation of the TFIM energy observable. First, we will introduce the built-in `NeighbourInteraction` observable:

[16]:
```
from qucumber.observables import NeighbourInteraction
```

The TFIM chain we trained the `nn_state` on did not have periodic boundary conditions, so `periodic_bcs=False`. Meanwhile, `c` specifies the distance between interacting spins, that is, a given site will only interact with a site `c` places away from itself; we set this to 1 as the TFIM chain has nearest-neighbour interactions.

[17]:
```
nn_inter = NeighbourInteraction(periodic_bcs=False, c=1)
```

Next, we need the `SigmaX` observable, which computes the magnetization in the X-direction:

[18]:
```
from qucumber.observables import SigmaX
```

Next, we build the Hamiltonian, setting $h = J = 1$:

[19]:
```
h = J = 1
sx = SigmaX()
tfim = -J * nn_inter - h * sx
```

The same statistics of this new TFIM observable can also be calculated.

[20]:
```
new_tfim_stats = tfim.statistics_from_samples(nn_state, new_samples)
print("Mean: %.4f" % new_tfim_stats["mean"], "+/- %.4f" % new_tfim_stats["std_error"])
print("Variance: %.4f" % new_tfim_stats["variance"])
```

```
Mean: -1.2353 +/- 0.0005
Variance: 0.0022
```

The statistics above match with those computed earlier.

7.2.4 Rényi Entropy and the Swap operator

We can estimate the second Rényi Entropy using the Swap operator as shown by Hastings et al. (2010). The second Rényi Entropy, in terms of the expectation of the Swap operator is given by:

$$S_2(A) = -\ln\langle \text{Swap}_A \rangle$$

where A is the subset of the lattice for which we wish to compute the Rényi entropy.

[21]:
```
from qucumber.observables import SWAP
```

As an example, we will take the region A consist of sites 0 through 4 (inclusive).

[22]:
```
A = [0, 1, 2, 3, 4]
swap = SWAP(A)

swap_stats = swap.statistics_from_samples(nn_state, new_samples)
print("Mean: %.4f" % swap_stats["mean"], "+/- %.4f" % swap_stats["std_error"])
print("Variance: %.4f" % swap_stats["variance"])
```

```
Mean: 0.7838 +/- 0.0061
Variance: 0.3663
```

The second Rényi Entropy can be computed directly from the sample mean. The standard error of the entropy, from first-order error analysis, is given by the standard error of the Swap operator divided by the mean of the Swap operator.

[23]:
```
S_2 = -np.log(swap_stats["mean"])
S_2_error = abs(swap_stats["std_error"] / swap_stats["mean"])


print("S_2: %.4f" % S_2, "+/- %.4f" % S_2_error)
```

```
S_2: 0.2437 +/- 0.0077
```

7.2.5 Writing custom diagonal observables

QuCumber has a built-in module called `Observable` which makes it easy for the user to compute any arbitrary observable from the `nn_state`. To see the the `Observable` module in action, an example (diagonal) observable called `PIQuIL`, which inherits properties from the `Observable` module, is shown below.

The `PIQuIL` observable takes a σ^z measurement at a site and multiplies it by the measurement two sites away from it. There is also a parameter, P , that determines the strength of each of these interactions. For example, for the dataset $(-1, 1, 1, -1), (1, 1, 1, 1)$ and $(1, 1, -1, 1)$ with $P = 2$, the `PIQuIL` for each data point would be $(2(-1 \times 1) + 2(1 \times -1) = -4)$, $(2(1 \times 1) + 2(1 \times 1) = 4)$ and $(2(1 \times -1) + 2(1 \times 1) = 0)$, respectively.

[24]:
```
class PIQuIL(ObservableBase):
    def __init__(self, P):
        self.name = "PIQuIL"
        self.symbol = "Q"
        self.P = P

    # Required : function that calculates the PIQuIL. Must be named "apply"
    def apply(self, nn_state, samples):
```

(continues on next page)

(continued from previous page)

```
        samples = to_pm1(samples)
        interaction_ = 0.0
        for i in range(samples.shape[-1] - 2):
            interaction_ += self.P * samples[:, i] * samples[:, i + 2]

        return interaction_


P = 0.05
piquil = PIQuIL(P)
```

The `apply` function is contained in the `Observable` module, but is overwritten here. The `apply` function in `Observable` will compute the observable itself and must take in the `nn_state` and a batch of samples as arguments. Thus, any new class inheriting from `Observable` that the user would like to define must contain a function called `apply` that calculates this new observable. For more details on `apply`, we refer to the documentation:

Although the `PIQuIL` observable could technically be computed without the first argument of `apply` since it does not ever use the `nn_state`, we still include it in the list of arguments in order to conform to the interface provided in the `ObservableBase` class.

Since we have already generated new samples of data, the `PIQuIL` observable's mean, standard error and variance on the new data can be calculated with the `statistics_from_samples` function in the `Observable` module. The user must simply provide the `nn_state` and the samples as arguments.

[25]:
```
piquil_stats1 = piquil.statistics_from_samples(nn_state, new_samples)
```

The `statistics_from_samples` function returns a dictionary containing the mean, standard error and the variance with the keys "mean", "std_error" and "variance", respectively.

[26]:
```
print(
    "Mean PIQuIL: %.4f" % piquil_stats1["mean"], "+/- %.4f" % piquil_stats1["std_error
↪"]
)
print("Variance: %.4f" % piquil_stats1["variance"])
```

```
Mean PIQuIL: 0.1762 +/- 0.0016
Variance: 0.0244
```

Exercise: We notice that the `PIQuIL` observable is essentially a scaled next-nearest-neighbours interaction. (a) Construct an equivalent `Observable` object algebraically in a similar manner to the TFIM observable constructed above. (b) Compute the statistics of this observable on `new_samples`, and compare to those computed using the `PIQuIL` observable.

[27]:
```
# solve the above exercise here
```

7.2.6 Writing off-diagonal observables

Now, as the `PIQuIL` observable was diagonal, it was fairly easy to write. Things get a bit more complicated once we consider off-diagonal observables, as we'd need to make use of information about the quantum state itself. In general, computing an observable exactly with respect to the state ρ requires performing a trace:

$$\langle O \rangle = \mathrm{Tr}\left[O\rho\right] = \sum_i \langle i|O\rho|i\rangle = \sum_{ij} \langle i|O|j\rangle\langle j|\rho|i\rangle$$

where $\{|i\rangle\}_i, \{|j\rangle\}_j$ are two orthonormal bases spanning the Hilbert space. Multiplying the numerator and denominator by $\langle i|\rho|i\rangle$ gives:

$$\langle O\rangle = \sum_i \langle i|\rho|i\rangle \sum_j \frac{\langle j|\rho|i\rangle}{\langle i|\rho|i\rangle}\langle i|O|j\rangle = \sum_i \rho_{ii} \sum_j \frac{\rho_{ji}}{\rho_{ii}} O_{ij} = \sum_i \rho_{ii}\mathcal{O}_i$$

Hence, computing the expectation of the observable O with respect to ρ, amounts to estimating the so-called "local-estimator" $\mathcal{O}$ with respect to the probability distribution $\{\rho_{ii}\}_i$. Setting $\{|i\rangle\}_i$ to our computational basis states $\{|\sigma\rangle\}$, we note that, as we are able to draw samples from ρ in the computational basis using our `nn_state`, we can easily estimate the expectation of O:

$$\langle O\rangle = \sum_\sigma \rho_{\sigma\sigma}\mathcal{O}(\sigma) \approx \frac{1}{|\mathcal{D}|}\sum_{\sigma\in\mathcal{D}} \mathcal{O}(\sigma)$$

where $\mathcal{D}$ denotes the set of drawn samples. Recall that the local-estimator is:

$$\mathcal{O}(\sigma) = \sum_{\sigma'} \frac{\rho(\sigma',\sigma)}{\rho(\sigma,\sigma)} O(\sigma,\sigma')$$

which, in the case of a pure state $\rho = |\psi\rangle\langle\psi|$, reduces to:

$$\mathcal{O}(\sigma) = \sum_{\sigma'} \frac{\psi(\sigma')}{\psi(\sigma)} O(\sigma,\sigma')$$

The task of the `apply` function, is actually to compute the local-estimator, given a sample σ. Ideally, this function would take into account the structure of O in order to perform this computation efficiently, and avoid iterating through every entry of the wavefunction of density matrix unnecessarily.

It should be noted that, though the Neural-Network-States provided by QuCumber do not give normalized probability estimates, this is not an issue for computing the local-estimator, as the normalization constant cancels out.

As an example, we will write a simplified version of the `SigmaX` observable. But first, let's see what the statistics of the official version of `SigmaX` are, for the sake of later comparison:

[28]:
```
sx.statistics_from_samples(nn_state, new_samples)
```

[28]:
```
{'mean': 0.7293210861294865,
 'variance': 0.07933831206407158,
 'std_error': 0.002816705736566594,
 'num_samples': 10000}
```

[29]:
```
class MySigmaX(ObservableBase):
    def __init__(self):
        self.name = "SigmaX"
        self.symbol = "X"

    def apply(self, nn_state, samples):
        samples = samples.to(device=nn_state.device)

        # vectors of shape: (2, num_samples,)
        denom = cplx.conjugate(nn_state.psi(samples))
        numer_sum = torch.zeros_like(denom)

        for i in range(samples.shape[-1]):  # sum over spin sites
            samples_ = flip_spin(i, samples.clone())  # flip the spin at site i

            # compute the numerator of the importance and add it to the running sum
```

(continues on next page)

(continued from previous page)

```
            numer = cplx.conjugate(nn_state.psi(samples_))
            numer_sum.add_(numer)

        mag = cplx.elementwise_division(numer_sum, denom)

        # take real part (imaginary part should be approximately zero)
        # and divide by number of spins
        return cplx.real(mag).div_(samples.shape[-1])
```

[30]:
```
MySigmaX().statistics_from_samples(nn_state, new_samples)
```

[30]:
```
{'mean': 0.7293210861294865,
 'variance': 0.07933831206407158,
 'std_error': 0.002816705736566594,
 'num_samples': 10000}
```

We're on the right track! The only remaining problem is generalizing this to work with mixed states. Note that in both expressions of the local-estimator, we need to compute a ratio dependent on σ and σ'. The Neural-Network-States provided by QuCumber implement the functions `importance_sampling_weight`, `importance_sampling_numerator`, and `importance_sampling_denominator` in order to simplify writing observables for both pure and mixed states.

In simple cases, we'd only need to make use of `importance_sampling_weight`, however, note that, since the denominator can be factored out of the summation, it is more efficient to compute the numerator and denominator separately in order to avoid duplicating work. Let's update our version of the X-magnetization observable to support mixed states:

[31]:
```
class MySigmaX(ObservableBase):
    def __init__(self):
        self.name = "SigmaX"
        self.symbol = "X"

    def apply(self, nn_state, samples):
        samples = samples.to(device=nn_state.device)

        # vectors of shape: (2, num_samples,)
        denom = nn_state.importance_sampling_denominator(samples)
        numer_sum = torch.zeros_like(denom)

        for i in range(samples.shape[-1]):  # sum over spin sites
            samples_ = flip_spin(i, samples.clone())  # flip the spin at site i

            # compute the numerator of the importance and add it to the running sum
            numer = nn_state.importance_sampling_numerator(samples_, samples)
            numer_sum.add_(numer)

        mag = cplx.elementwise_division(numer_sum, denom)

        # take real part (imaginary part should be approximately zero)
        # and divide by number of spins
        return cplx.real(mag).div_(samples.shape[-1])
```

[32]:
```
MySigmaX().statistics_from_samples(nn_state, new_samples)
```

[32]:
```
{'mean': 0.7293210861294865,
 'variance': 0.07933831206407158,
```

(continues on next page)

(continued from previous page)

```
 'std_error': 0.002816705736566594,
 'num_samples': 10000}
```

Note that not much has actually changed in our code. In fact, one can often write local-estimators for observables assuming a pure-state, and then later easily generalize their code to support mixed states using the abstract functions discussed earlier. As a final sanity check, let's try estimating the statistics of `MySigmaX` for a randomly initialized `DensityMatrix`, and compare the output to that of the official implementation:

[33]:
```
mixed_nn_state = DensityMatrix(nn_state.num_visible, gpu=False)

(
    sx.statistics_from_samples(mixed_nn_state, new_samples),
    MySigmaX().statistics_from_samples(mixed_nn_state, new_samples),
)
```

[33]:
```
({'mean': 0.9930701314464155,
  'variance': 0.010927188103705533,
  'std_error': 0.0010453319139730468,
  'num_samples': 10000},
 {'mean': 0.9930701314464155,
  'variance': 0.010927188103705533,
  'std_error': 0.0010453319139730468,
  'num_samples': 10000})
```

7.3 Estimating Statistics of Many Observables Simultaneously

One may often be concerned with estimating the statistics of many observables simultaneously. In order to avoid excess memory usage, it makes sense to reuse the same set of samples to estimate each observable. When we need a large number of samples however, we run into the same issue mentioned earlier: we may run out of memory storing the samples. QuCumber provides a `System` object to keep track of multiple observables and estimate their statistics efficiently.

[34]:
```
from qucumber.observables import System
from pprint import pprint
```

At this point we must make a quick aside: internally, `System` keeps track of multiple observables through their `name` field (which we saw in the definition of the `PIQuIL` observable). This name is returned by Python's built-in `repr` function, which is automatically called when we try to display an `Observable` object in Jupyter:

[35]:
```
piquil
```

[35]:
```
PIQuIL
```

[36]:
```
tfim
```

[36]:
```
((-1 * NeighbourInteraction(periodic_bcs=False, c=1)) + -(1 * SigmaX))
```

Note how the TFIM energy observable's name is quite complicated, due to the fact that we constructed it algebraically as opposed to the `PIQuIL` observable which was built from scratch and manually assigned a name. In order to assign a name to `tfim`, we do the following:

[37]:
```
tfim.name = "TFIM"
tfim
```

[37]:
```
TFIM
```

Now, back to `System`. We'd like to create a `System` object which keeps track of the absolute magnetization, the energy of the chain, the Swap observable (of region A, as defined earlier), and finally, the `PIQuIL` observable.

[38]:
```
tfim_system = System(sz, tfim, swap, piquil)
```

[39]:
```
pprint(tfim_system.statistics_from_samples(nn_state, new_samples))
```

```
{'PIQuIL': {'mean': 0.1762100000000003,
            'num_samples': 10000,
            'std_error': 0.001561328717706924,
            'variance': 0.024377473647363472},
 'SWAP': {'mean': 0.7837510693478925,
          'num_samples': 10000,
          'std_error': 0.006052467264446535,
          'variance': 0.3663235998719692},
 'SigmaZ': {'mean': 0.5575200000000005,
            'num_samples': 10000,
            'std_error': 0.0031291730748576373,
            'variance': 0.09791724132414},
 'TFIM': {'mean': -1.2352610861294844,
          'num_samples': 10000,
          'std_error': 0.0004669027817740233,
          'variance': 0.0021799820762832 12}}
```

These all match with the values computed earlier. Next, we will compute these statistics from fresh samples drawn from the `nn_state`:

[40]:
```
%%time
pprint(
    tfim_system.statistics(
        nn_state, num_samples=10000, num_chains=1000, burn_in=100, steps=2
    )
)
```

```
{'PIQuIL': {'mean': 0.17354,
            'num_samples': 10000,
            'std_error': 0.001551072990784856,
            'variance': 0.02405827422742278},
 'SWAP': {'mean': 0.7824233758221596,
          'num_samples': 10000,
          'std_error': 0.006105647043859781,
          'variance': 0.3727892582419368},
 'SigmaZ': {'mean': 0.5508199999999999,
            'num_samples': 10000,
            'std_error': 0.0031138886284257233,
            'variance': 0.09696302390239034},
 'TFIM': {'mean': -1.2351832610706792,
          'num_samples': 10000,
          'std_error': 0.0004658592121667226,
          'variance': 0.0021704969 15879073}}
CPU times: user 741 ms, sys: 0 ns, total: 741 ms
Wall time: 192 ms
```

Compare this to computing these statistics on each observable individually:

[41]:
```
%%time
pprint(
    {
        obs.name: obs.statistics(
            nn_state, num_samples=10000, num_chains=1000, burn_in=100, steps=2
        )
        for obs in [piquil, swap, sz, tfim]
    }
)
```

```
{'PIQuIL': {'mean': 0.17636999999999997,
            'num_samples': 10000,
            'std_error': 0.0015554112769374036,
            'variance': 0.024193042404240445},
 'SWAP': {'mean': 0.7788363185216998,
          'num_samples': 10000,
          'std_error': 0.005804524946475529,
          'variance': 0.3369250985425674},
 'SigmaZ': {'mean': 0.55804,
            'num_samples': 10000,
            'std_error': 0.0031169387820002294,
            'variance': 0.09715307370737074},
 'TFIM': {'mean': -1.2345632953955774,
          'num_samples': 10000,
          'std_error': 0.0004844496452134716,
          'variance': 0.0023469145874745853}}
CPU times: user 1.59 s, sys: 0 ns, total: 1.59 s
Wall time: 401 ms
```

Note the slowdown. This is, as mentioned before, due to the fact that the `System` object uses *the same samples* to estimate statistics for *all* of the observables it is keeping track of.

7.3.1 Template for your custom observable

Here is a generic template for you to try using the `Observable` module yourself.

[42]:
```
class YourObservable(ObservableBase):
    def __init__(self, your_constants):
        self.your_constants = your_constants
        self.name = "Observable_Name"

        # The algebraic symbol representing this Observable.
        # Returned by Python's built-in str() function
        self.symbol = "O"

    def apply(self, nn_state, samples):
        # arguments of "apply" must be in this order

        # calculate your observable for each data point
        obs = torch.tensor([42] * len(samples))

        # make sure the observables are on the same device and have the
        # same dtype as the samples
        obs = obs.to(samples)
```

(continues on next page)

(continued from previous page)

```
        # return a torch tensor containing the observable values
        return obs
```

CHAPTER

EIGHT

TRAINING WHILE MONITORING OBSERVABLES

As seen in the first tutorial that went through reconstructing the wavefunction describing the TFIM with 10 sites at its critical point, the user can evaluate the training in real time with the `MetricEvaluator` and custom functions. What is most likely more impactful in many cases is to calculate an observable, like the energy, during the training process. This is slightly more computationally involved than using the `MetricEvaluator` to evaluate functions because observables require that samples be drawn from the RBM.

Luckily, QuCumber also has a module very similar to the `MetricEvaluator`, but for observables. This is called the `ObservableEvaluator`. This tutorial uses the `ObservableEvaluator` to calculate the energy during the training on the TFIM data in the first tutorial. We will use the same training hyperparameters as before.

It is assumed that the user has worked through Tutorial 3 beforehand. Recall that `quantum_ising_chain.py` contains the `TFIMChainEnergy` class that inherits from the `Observable` module. The exact ground-state energy is -1.2381.

[1]:
```
import os.path

import numpy as np
import matplotlib.pyplot as plt

from qucumber.nn_states import PositiveWaveFunction
from qucumber.callbacks import ObservableEvaluator

import qucumber
import qucumber.utils.data as data

from quantum_ising_chain import TFIMChainEnergy

# set random seed on cpu but not gpu, since we won't use gpu for this tutorial
qucumber.set_random_seed(1234, cpu=True, gpu=False)
```

[2]:
```
train_data = data.load_data(
    os.path.join("..", "Tutorial1_TrainPosRealWaveFunction", "tfim1d_data.txt")
)[0]

nv = train_data.shape[-1]
nh = nv

nn_state = PositiveWaveFunction(num_visible=nv, num_hidden=nh, gpu=False)

epochs = 1000
pbs = 100  # pos_batch_size
nbs = 200  # neg_batch_size
lr = 0.01
```

(continues on next page)

(continued from previous page)

```
k = 10

period = 100

h = 1
num_samples = 10000
burn_in = 100
steps = 100

tfim_energy = TFIMChainEnergy(h)
```

Now, the `ObservableEvaluator` can be called. The `ObservableEvaluator` requires the following arguments.

1. `period`: the frequency of the training evaluators being calculated (e.g. `period=200` means that the `MetricEvaluator` will compute the desired metrics every 200 epochs)
2. A list of `Observable` objects you would like to reference to evaluate the training (arguments required for generating samples to calculate the observables are keyword arguments placed after the list). The `ObservableEvaluator` uses a `System` object (discussed in the previous tutorial) under the hood in order to estimate statistics efficiently.

The following additional arguments are needed to calculate the statistics on the generated samples during training (these are the arguments of the `statistics` function in the `Observable` module, minus the `nn_state` argument; this gets passed in as an argument to `fit`). For more detail on these arguments, refer to either the previous tutorial or the documentation for `Observable.statistics`.

- `num_samples`: the number of samples to generate internally
- `num_chains`: the number of Markov chains to run in parallel (default = 0)
- `burn_in`: the number of Gibbs steps to perform before recording any samples (default = 1000)
- `steps`: the number of Gibbs steps to perform between each sample (default = 1)

The training evaluators can be printed out by setting the `verbose` keyword argument to `True`.

[3]:
```
callbacks = [
    ObservableEvaluator(
        period,
        [tfim_energy],
        verbose=True,
        num_samples=num_samples,
        burn_in=burn_in,
        steps=steps,
    )
]

nn_state.fit(
    train_data,
    epochs=epochs,
    pos_batch_size=pbs,
    neg_batch_size=nbs,
    lr=lr,
    k=k,
    callbacks=callbacks,
)
```

```
Epoch: 100
  TFIMChainEnergy:
    mean: -1.193284     variance: 0.023108      std_error: 0.001520
Epoch: 200
  TFIMChainEnergy:
    mean: -1.217176     variance: 0.012590      std_error: 0.001122
Epoch: 300
  TFIMChainEnergy:
    mean: -1.225789     variance: 0.007857      std_error: 0.000886
Epoch: 400
  TFIMChainEnergy:
    mean: -1.229849     variance: 0.005336      std_error: 0.000730
Epoch: 500
  TFIMChainEnergy:
    mean: -1.231192     variance: 0.004132      std_error: 0.000643
Epoch: 600
  TFIMChainEnergy:
    mean: -1.233709     variance: 0.003314      std_error: 0.000576
Epoch: 700
  TFIMChainEnergy:
    mean: -1.234858     variance: 0.002687      std_error: 0.000518
Epoch: 800
  TFIMChainEnergy:
    mean: -1.234655     variance: 0.002244      std_error: 0.000474
Epoch: 900
  TFIMChainEnergy:
    mean: -1.235693     variance: 0.001981      std_error: 0.000445
Epoch: 1000
  TFIMChainEnergy:
    mean: -1.235892     variance: 0.001680      std_error: 0.000410
```

The `callbacks` list returns a list of dictionaries. The mean, standard error and the variance at each epoch can be accessed as follows:

[4]:
```
# Note that the name of the observable class that the user makes
# must be what comes after callbacks[0].
energies = callbacks[0].TFIMChainEnergy.mean

# Alternatively, we can use the usual dictionary/list subscripting
# syntax, which is useful in the case where the observable's name
# contains special characters or spaces
errors = callbacks[0]["TFIMChainEnergy"].std_error
variance = callbacks[0]["TFIMChainEnergy"]["variance"]
```

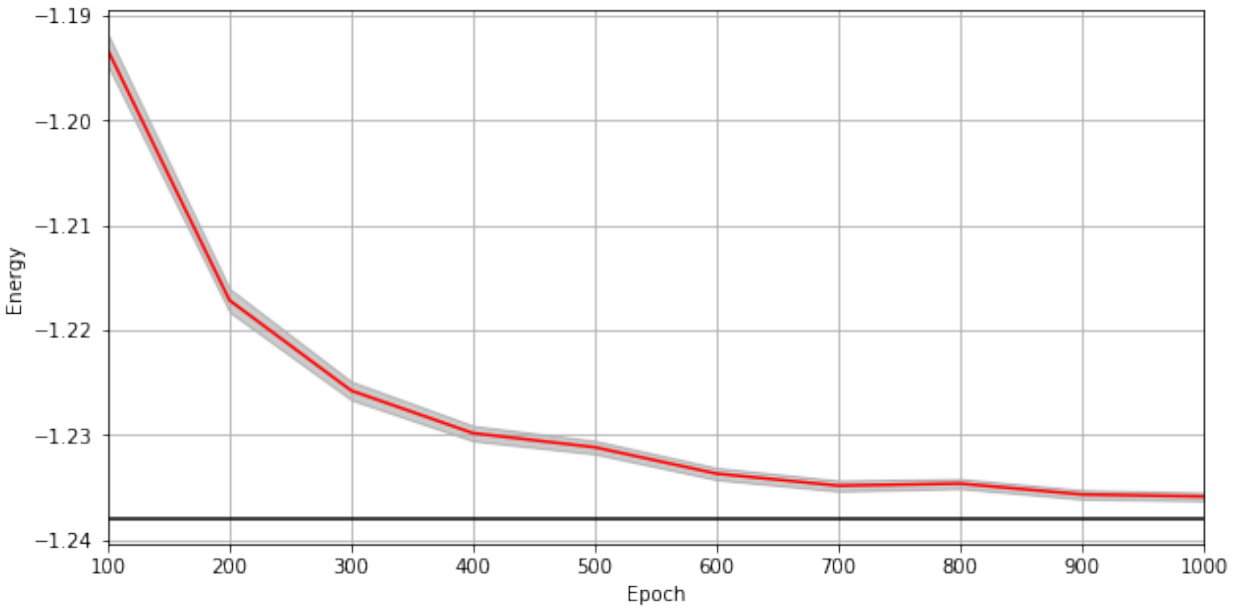
A plot of the energy as a function of the training cycle is presented below.

[5]:
```
epoch = np.arange(period, epochs + 1, period)

E0 = -1.2381

plt.figure(figsize=(10, 5))
ax = plt.axes()
ax.plot(epoch, energies, color="red")
ax.set_xlim(period, epochs)
ax.axhline(E0, color="black")
ax.fill_between(epoch, energies - errors, energies + errors, alpha=0.2, color="black")
ax.set_xlabel("Epoch")
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel("Energy")
ax.grid()
```



CHAPTER

NINE

WAVEFUNCTION RBM

class qucumber.rbm.**BinaryRBM**(**args: Any*, ***kwargs: Any*)

Bases: torch.nn.Module

effective_energy(*v*)

The effective energies of the given visible states.

$$\mathcal{E}(v) = -\sum_j b_j v_j - \sum_i \log\left[1 + \exp\left(c_i + \sum_j W_{ij} v_j\right)\right]$$

Parameters **v** (*torch.Tensor*) – The visible states.

Returns The effective energies of the given visible states.

Return type torch.Tensor

effective_energy_gradient(*v*, *reduce=True*)

The gradients of the effective energies for the given visible states.

Parameters

- **v** (*torch.Tensor*) – The visible states.
- **reduce** (*bool*) – If *True*, will sum over the gradients resulting from each visible state. Otherwise will return a batch of gradient vectors.

Returns Will return a vector (or matrix if *reduce=False* and multiple visible states were given as a matrix) containing the gradients for all parameters (computed on the given visible states v).

Return type torch.Tensor

gibbs_steps(*k*, *initial_state*, *overwrite=False*)

Performs k steps of Block Gibbs sampling. One step consists of sampling the hidden state h from the conditional distribution $p(h \mid v)$, and sampling the visible state v from the conditional distribution $p(v \mid h)$.

Parameters

- **k** (*int*) – Number of Block Gibbs steps.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chains.
- **overwrite** (*bool*) – Whether to overwrite the initial_state tensor. Exception: If initial_state is not on the same device as the RBM, it will NOT be overwritten.

Returns Returns the visible states after k steps of Block Gibbs sampling

Return type torch.Tensor

initialize_parameters(*zero_weights=False*)
Randomize the parameters of the RBM

partition(*space*)
Compute the partition function of the RBM.

Parameters **space** (*torch.Tensor*) – A rank 2 tensor of the visible space.

Returns The value of the partition function evaluated at the current state of the RBM.

Return type torch.Tensor

prob_h_given_v(*v*, *out=None*)
Given a visible unit configuration, compute the probability vector of the hidden units being on.

Parameters

- **h** (*torch.Tensor*) – The hidden unit.
- **out** (*torch.Tensor*) – The output tensor to write to.

Returns The probability of hidden units being active given the visible state.

Return type torch.Tensor

prob_v_given_h(*h*, *out=None*)
Given a hidden unit configuration, compute the probability vector of the visible units being on.

Parameters

- **h** (*torch.Tensor*) – The hidden unit
- **out** (*torch.Tensor*) – The output tensor to write to.

Returns The probability of visible units being active given the hidden state.

Return type torch.Tensor

sample_h_given_v(*v*, *out=None*)
Sample/generate a hidden state given a visible state.

Parameters

- **h** (*torch.Tensor*) – The visible state.
- **out** (*torch.Tensor*) – The output tensor to write to.

Returns The sampled hidden state.

Return type torch.Tensor

sample_v_given_h(*h*, *out=None*)
Sample/generate a visible state given a hidden state.

Parameters

- **h** (*torch.Tensor*) – The hidden state.
- **out** (*torch.Tensor*) – The output tensor to write to.

Returns The sampled visible state.

Return type torch.Tensor

CHAPTER

TEN

DENSITY MATRIX RBM

class qucumber.rbm.**PurificationRBM**(**args: Any*, ***kwargs: Any*)

Bases: torch.nn.Module

An RBM with a hidden and "auxiliary" layer, each separately connected to the visible units

Parameters

- **num_visible** (*int*) – The number of visible units, i.e. the size of the system
- **num_hidden** (*int*) – The number of units in the hidden layer
- **num_aux** (*int*) – The number of units in the auxiliary purification layer
- **zero_weights** (*bool*) – Whether or not to initialize the weights to zero
- **gpu** (*bool*) – Whether to perform computations on the default gpu.

effective_energy(*v*, *a=None*)

Computes the equivalent of the "effective energy" for the RBM. If *a* is *None*, will analytically trace out the auxiliary units.

Parameters

- **v** (*torch.Tensor*) – The current state of the visible units. Shape (b, n_v) or (n_v,).
- **a** (*torch.Tensor or None*) – The current state of the auxiliary units. Shape (b, n_a) or (n_a,).

Returns The "effective energy" of the RBM. Shape (b,) or (1,).

Return type torch.Tensor

effective_energy_gradient(*v*, *reduce=True*)

The gradients of the effective energies for the given visible states.

Parameters

- **v** (*torch.Tensor*) – The visible states.
- **reduce** (*bool*) – If *True*, will sum over the gradients resulting from each visible state. Otherwise will return a batch of gradient vectors.

Returns Will return a vector (or matrix if *reduce=False* and multiple visible states were given as a matrix) containing the gradients for all parameters (computed on the given visible states v).

Return type torch.Tensor

gamma(*v*, *vp*, *eta=1*, *expand=True*)

Calculates elements of the $\Gamma^{(\eta)}$ matrix, where $\eta = \pm$. If *expand* is *True*, will return a complex matrix $A_{ij} = \langle\sigma_i|\Gamma^{(\eta)}|\sigma'_j\rangle$. Otherwise will return a complex vector $A_i = \langle\sigma_i|\Gamma^{(\eta)}|\sigma'_i\rangle$.

Parameters

- **v** (*torch.Tensor*) – A batch of visible states, σ.
- **vp** (*torch.Tensor*) – The other batch of visible states, σ'.
- **eta** (*int*) – Determines which gamma matrix elements to compute.
- **expand** (*bool*) – Whether to return a matrix (*True*) or a vector (*False*). Ignored if both inputs are vectors, in which case, a scalar is returned.

Returns The matrix element given by $\langle\sigma|\Gamma^{(\eta)}|\sigma'\rangle$

Return type torch.Tensor

gamma_grad(*v*, *vp*, *eta=1*, *expand=False*)

Calculates elements of the gradient of the $\Gamma^{(\eta)}$ matrix, where $\eta = \pm$.

Parameters

- **v** (*torch.Tensor*) – A batch of visible states, σ
- **vp** (*torch.Tensor*) – The other batch of visible states, σ'
- **eta** (*int*) – Determines which gamma matrix elements to compute.
- **expand** (*bool*) – Whether to return a rank-3 tensor (*True*) or a matrix (*False*).

Returns The matrix element given by $\langle\sigma|\nabla_\lambda\Gamma^{(\eta)}|\sigma'\rangle$

Return type torch.Tensor

gibbs_steps(*k*, *initial_state*, *overwrite=False*)

Perform k steps of Block Gibbs sampling. One step consists of sampling the hidden and auxiliary states from the visible state, and then sampling the visible state from the hidden and auxiliary states

Parameters

- **k** (*int*) – The number of Block Gibbs steps
- **initial_state** (*torch.Tensor*) – The initial visible state
- **overwrite** (*bool*) – Whether to overwrite the initial_state tensor. Exception: If initial_state is not on the same device as the RBM, it will NOT be overwritten.

Returns Returns the visible states after k steps of Block Gibbs sampling

Return type torch.Tensor

initialize_parameters(*zero_weights=False*)

Initialize the parameters of the RBM

Parameters **zero_weights** (*bool*) – Whether or not to initialize the weights to zero

mixing_term(*v*)

Describes the extent of mixing in the system, $V_\theta = \frac{1}{2}U_\theta\sigma + d_\theta$

Parameters **v** (*torch.Tensor*) – The visible state of the system

Returns The term describing the mixing of the system

Return type torch.Tensor

partition(*space*)

Computes the partition function

Parameters **space** (*torch.Tensor*) – The Hilbert space of the visible units

Returns The partition function

Return type torch.Tensor

prob_a_given_v(*v*, *out=None*)
Given a visible unit configuration, compute the probability vector of the auxiliary units being on

Parameters

- **v** (*torch.Tensor*) – The visible units
- **out** (*torch.Tensor*) – The output tensor to write to

Returns The probability of the auxiliary units being active given the visible state

Rtype torch.Tensor

prob_h_given_v(*v*, *out=None*)
Given a visible unit configuration, compute the probability vector of the hidden units being on

Parameters

- **v** (*torch.Tensor*) – The visible units
- **out** (*torch.Tensor*) – The output tensor to write to

Returns The probability of the hidden units being active given the visible state

Rtype torch.Tensor

prob_v_given_ha(*h*, *a*, *out=None*)
Given a hidden and auxiliary unit configuration, compute the probability vector of the hidden units being on

Parameters

- **h** (*torch.Tensor*) – The hidden units
- **a** (*torch.Tensor*) – The auxiliary units
- **out** (*torch.Tensor*) – The output tensor to write to

Returns The probability of the visible units being active given the hidden and auxiliary states

Rtype torch.Tensor

sample_a_given_v(*v*, *out=None*)
Sample/generate an auxiliary state given a visible state

Parameters

- **v** (*torch.Tensor*) – The visible state
- **out** (*torch.Tensor*) – The output tensor to write to

Returns The sampled auxiliary state

Return type torch.Tensor

sample_h_given_v(*v*, *out=None*)
Sample/generate a hidden state given a visible state

Parameters

- **v** (*torch.Tensor*) – The visible state
- **out** (*torch.Tensor*) – The output tensor to write to

Returns The sampled hidden state

Return type torch.Tensor

sample_v_given_ha(*h*, *a*, *out=None*)
Sample/generate a visible state given the hidden and auxiliary states

Parameters

- **h** (*torch.Tensor*) – The hidden state
- **a** (*torch.Tensor*) – The auxiliary state
- **out** (*torch.Tensor*) – The output tensor to write to

Returns The sampled visible state

Return type torch.Tensor

CHAPTER

ELEVEN

QUANTUM STATES

11.1 Positive WaveFunction

class qucumber.nn_states.**PositiveWaveFunction**(*num_visible*, *num_hidden=None*, *gpu=True*, *module=None*)

Bases: *qucumber.nn_states.WaveFunctionBase*

Class capable of learning wavefunctions with no phase.

Parameters

- **num_visible** (*int*) – The number of visible units, ie. the size of the system being learned.
- **num_hidden** (*int*) – The number of hidden units in the internal RBM. Defaults to the number of visible units.
- **gpu** (*bool*) – Whether to perform computations on the default GPU.
- **module** (qucumber.rbm.BinaryRBM) – An instance of a BinaryRBM module to use for density estimation. Will be copied to the default GPU if *gpu=True* (if it isn't already there). If *None*, will initialize a BinaryRBM from scratch.

amplitude(*v*)

Compute the (unnormalized) amplitude of a given vector/matrix of visible states.

$$\text{amplitude}(\sigma) = |\psi_\lambda(\sigma)| = e^{-\mathcal{E}_\lambda(\sigma)/2}$$

Parameters **v** (*torch.Tensor*) – visible states σ

Returns Matrix/vector containing the amplitudes of v

Return type torch.Tensor

static autoload(*location*, *gpu=True*)

Initializes a NeuralState from the parameters in the given location.

Parameters

- **location** (*str or file*) – The location to load the model parameters from.
- **gpu** (*bool*) – Whether the returned model should be on the GPU.

Returns A new NeuralState initialized from the given parameters. The returned NeuralState will be of whichever type this function was called on. An error may be thrown if the loaded parameters correspond to a different type of NeuralState than the caller.

compute_batch_gradients(*k*, *samples_batch*, *neg_batch*, **args*, ***kwargs*)

Compute the gradients of a batch of the training data (*samples_batch*).

Parameters

- **k** (*int*) – Number of contrastive divergence steps in training.
- **samples_batch** (*torch.Tensor*) – Batch of the input samples.
- **neg_batch** (*torch.Tensor*) – Batch of the input samples for computing the negative phase.
- ***args** – Ignored.
- ****kwargs** – Ignored.

Returns A single-element list containing the gradients calculated with a Gibbs sampled negative phase update

Return type list[torch.Tensor]

compute_exact_gradients(*samples_batch*, *space*, *bases_batch=None*)
Computes the gradients of the parameters, using exact sampling for the negative phase update instead of Gibbs sampling

Parameters

- **samples_batch** (*torch.Tensor*) – The measurements
- **space** (*torch.Tensor*) – A rank 2 tensor of the entire visible space.
- **bases_batch** (*numpy.ndarray*) – The bases in which the measurements are made

Returns A two-element list containing the amplitude and phase RBM gradients calculated with an exact negative phase update

Return type list[torch.Tensor]

compute_exact_grads(*samples_batch*, *space*, **args*, ***kwargs*)
Computes the gradients of the parameters, using exact sampling for the negative phase update instead of Gibbs sampling

Parameters

- **samples_batch** (*torch.Tensor*) – The measurements
- **space** (*torch.Tensor*) – A rank 2 tensor of the entire visible space.
- ***args** – Ignored.
- ****kwargs** – Ignored.

Returns A single-element list containing the gradients calculated with an exact negative phase update

Return type list[torch.Tensor]

compute_normalization(*space*)
Alias for *normalization*

property device
The device that the model is on.

fit(*data*, *epochs=100*, *pos_batch_size=100*, *neg_batch_size=None*, *k=1*, *lr=0.001*, *progbar=False*, *starting_epoch=1*, *time=False*, *callbacks=None*, *optimizer=torch.optim.SGD*, *optimizer_args=None*, *scheduler=None*, *scheduler_args=None*, ***kwargs*)
Train the NeuralState.

Parameters

- **data** (*numpy.ndarray*) – The training samples

- **epochs** (*int*) – The number of full training passes through the dataset. Technically, this specifies the index of the *last* training epoch, which is relevant if *starting_epoch* is being set.
- **pos_batch_size** (*int*) – The size of batches for the positive phase taken from the data.
- **neg_batch_size** (*int*) – The size of batches for the negative phase taken from the data. Defaults to *pos_batch_size*.
- **k** (*int*) – The number of contrastive divergence steps.
- **lr** (*float*) – Learning rate
- **input_bases** (*numpy.ndarray*) – The measurement bases for each sample. Must be provided if training a ComplexWaveFunction or DensityMatrix.
- **progbar** (*bool or str*) – Whether or not to display a progress bar. If "notebook" is passed, will use a Jupyter notebook compatible progress bar.
- **starting_epoch** (*int*) – The epoch to start from. Useful if continuing training from a previous state.
- **callbacks** (*list[qucumber.callbacks.CallbackBase]*) – Callbacks to run while training.
- **optimizer** (*torch.optim.Optimizer*) – The constructor of a torch optimizer.
- **scheduler** – The constructor of a torch scheduler
- **optimizer_args** (*dict*) – Arguments to pass to the optimizer
- **scheduler_args** (*dict*) – Arguments to pass to the scheduler
- ****kwargs** – Ignored; exists for backwards compatibility.

generate_hilbert_space(*size=None*, *device=None*)

Generates Hilbert space of dimension 2^{size}.

Parameters

- **size** (*int*) – The size of each element of the Hilbert space. Defaults to the number of visible units.
- **device** – The device to create the Hilbert space matrix on. Defaults to the device this model is on.

Returns A tensor with all the basis states of the Hilbert space.

Return type torch.Tensor

gradient(*v*, **args*, ***kwargs*)

Compute the gradient of the effective energy for a batch of states.

$\nabla_\lambda \mathcal{E}_\lambda(\sigma)$

Parameters

- **v** (*torch.Tensor*) – visible states σ
- ***args** – Ignored.
- ****kwargs** – Ignored.

Returns A two-element list containing the gradients of the effective energy. The second element will always be zero.

Return type list[torch.Tensor]

importance_sampling_denominator(*v*)
Compute the denominator of the weight of an arbitrary sample, with respect to the sample *v*.

In the case of a mixed state, this quantity is $\rho(\sigma, \sigma)$, while in the pure case it is $\psi(\sigma')$

Parameters **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the denominator of the weights with respect to σ

Return type torch.Tensor

importance_sampling_numerator(*vp*, *v*)
Compute the numerator of the weight of sample *vp*, with respect to the sample *v*.

In the case of a mixed state, this quantity is $\rho(\sigma', \sigma)$, while in the pure case it is $\psi(\sigma')$

Parameters

- **vp** (*torch.Tensor*) – A batch containing the samples σ'
- **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the numerator of the weights of σ' with respect to σ

Return type torch.Tensor

importance_sampling_weight(*vp*, *v*)
Compute the weight of sample *vp*, with respect to the sample *v*.

In the case of a mixed state, this ratio is:

$$\frac{\rho(\sigma', \sigma)}{\rho(\sigma, \sigma)}$$

While in the pure case:

$$\frac{\psi(\sigma')}{\psi(\sigma)}$$

Parameters

- **vp** (*torch.Tensor*) – A batch containing the samples σ'
- **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the weights of σ' with respect to σ

Return type torch.Tensor

load(*location*)
Loads the NeuralState parameters from the given location ignoring any metadata stored in the file. Overwrites the NeuralState's parameters.

Note: The NeuralState object on which this function is called must have the same parameter shapes as the one who's parameters are being loaded.

Parameters **location** (*str or file*) – The location to load the NeuralState parameters from.

property max_size
Maximum size of the Hilbert space for full enumeration

property networks
A list of the names of the internal RBMs.

normalization(*space*)
Compute the normalization constant of the state. In the case of a pure state, this is the norm of the unnormalized wavefunction. In the case of a mixed state, this is the trace of the unnormalized density matrix.

$$Z_\lambda = \sum_\sigma p_\lambda(\sigma)$$

Parameters **space** (*torch.Tensor*) – A rank 2 tensor of the entire visible space.

phase(*v*)
Compute the phase of a given vector/matrix of visible states.

In the case of a PositiveWaveFunction, the phase is just zero.

Parameters **v** (*torch.Tensor*) – visible states σ

Returns Matrix/vector containing the phases of v

Return type torch.Tensor

positive_phase_gradients(*samples_batch*, **args*, ***kwargs*)
Computes the positive phase of the gradients of the parameters.

Parameters

- **samples_batch** (*torch.Tensor*) – The measurements
- ***args** – Ignored.
- ****kwargs** – Ignored.

Returns A two-element list containing the gradients of the effective energy. The second element will always be zero.

Return type list[torch.Tensor]

probability(*v*, *Z=1.0*)
Evaluates the probability of the given vector(s) of visible states. Assumes the visible states were measured in the computational basis.

Parameters

- **v** (*torch.Tensor*) – The visible states.
- **Z** (*float*) – The partition function / normalization constant. Defaults to 1, producing unnormalized probabilities.

Returns The probability of the given vector(s) of visible units.

Return type torch.Tensor

psi(*v*)
Compute the (unnormalized) wavefunction of a given vector/matrix of visible states.

$$\psi_\lambda(\sigma) = e^{-\mathcal{E}_\lambda(\sigma)/2}$$

Parameters **v** (*torch.Tensor*) – visible states σ

Returns Complex object containing the value of the wavefunction for each visible state

Return type torch.Tensor

property rbm_am
The RBM to be used to learn the wavefunction amplitude.

reinitialize_parameters()
Randomize the parameters of the internal RBMs.

sample(*k*, *num_samples=1*, *initial_state=None*, *overwrite=False*)
Performs k steps of Block Gibbs sampling. One step consists of sampling the hidden state h from the conditional distribution $p_\lambda(h|v)$, and sampling the visible state v from the conditional distribution $p_\lambda(v|h)$.

Parameters

- **k** (*int*) – Number of Block Gibbs steps.
- **num_samples** (*int*) – The number of samples to generate.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chains. If given, *num_samples* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the initial_state tensor, if it is provided.

save(*location*, *metadata=None*)
Saves the NeuralState parameters to the given location along with any given metadata.

Parameters

- **location** (*str or file*) – The location to save the data.
- **metadata** (*dict*) – Any extra metadata to store alongside the NeuralState parameters.

property stop_training
If *True*, will not train.

If this property is set to *True* during the training cycle, training will terminate once the current batch or epoch ends (depending on when *stop_training* was set).

subspace_vector(*num*, *size=None*, *device=None*)
Generates a single vector from the Hilbert space of dimension 2^{size}.

Parameters

- **size** (*int*) – The size of each element of the Hilbert space.
- **num** (*int*) – The specific vector to return from the Hilbert space. Since the Hilbert space can be represented by the set of binary strings of length *size*, *num* is equivalent to the decimal representation of the returned vector.
- **device** – The device to create the vector on. Defaults to the device this model is on.

Returns A state from the Hilbert space.

Return type torch.Tensor

11.2 Complex WaveFunction

class qucumber.nn_states.**ComplexWaveFunction**(*num_visible*, *num_hidden=None*, *unitary_dict=None*, *gpu=False*, *module=None*)
Bases: *qucumber.nn_states.WaveFunctionBase*

Class capable of learning wavefunctions with a non-zero phase.

Parameters

- **num_visible** (*int*) – The number of visible units, ie. the size of the system being learned.
- **num_hidden** (*int*) – The number of hidden units in both internal RBMs. Defaults to the number of visible units.
- **unitary_dict** (*dict[str, torch.Tensor]*) – A dictionary mapping unitary names to their matrix representations.
- **gpu** (*bool*) – Whether to perform computations on the default GPU.
- **module** (qucumber.rbm.BinaryRBM) – An instance of a BinaryRBM module to use for density estimation; The given RBM object will be used to estimate the amplitude of the wavefunction, while a copy will be used to estimate the phase of the wavefunction. Will be copied to the default GPU if *gpu=True* (if it isn't already there). If *None*, will initialize the BinaryRBMs from scratch.

am_grads(*v*)

Computes the gradients of the amplitude RBM for given input states

Parameters **v** (*torch.Tensor*) – The input state, σ

Returns The gradients of all amplitude RBM parameters

Return type torch.Tensor

amplitude(*v*)

Compute the (unnormalized) amplitude of a given vector/matrix of visible states.

$$\text{amplitude}(\sigma) = |\psi_{\lambda\mu}(\sigma)| = e^{-\mathcal{E}_\lambda(\sigma)/2}$$

Parameters **v** (*torch.Tensor*) – visible states σ.

Returns Vector containing the amplitudes of the given states.

Return type torch.Tensor

static autoload(*location*, *gpu=False*)

Initializes a NeuralState from the parameters in the given location.

Parameters

- **location** (*str or file*) – The location to load the model parameters from.
- **gpu** (*bool*) – Whether the returned model should be on the GPU.

Returns A new NeuralState initialized from the given parameters. The returned NeuralState will be of whichever type this function was called on. An error may be thrown if the loaded parameters correspond to a different type of NeuralState than the caller.

compute_batch_gradients(*k*, *samples_batch*, *neg_batch*, *bases_batch=None*)

Compute the gradients of a batch of the training data (*samples_batch*).

If measurements are taken in bases other than the reference basis, a list of bases (*bases_batch*) must also be provided.

Parameters

- **k** (*int*) – Number of contrastive divergence steps in training.
- **samples_batch** (*torch.Tensor*) – Batch of the input samples.
- **neg_batch** (*torch.Tensor*) – Batch of the input samples for computing the negative phase.

- **bases_batch** (*numpy.ndarray*) – Batch of the input bases corresponding to the samples in *samples_batch*.

Returns A two-element list containing the amplitude and phase RBM gradients calculated with a Gibbs sampled negative phase update

Return type list[torch.Tensor]

compute_exact_gradients(*samples_batch*, *space*, *bases_batch=None*)
Computes the gradients of the parameters, using exact sampling for the negative phase update instead of Gibbs sampling

Parameters

- **samples_batch** (*torch.Tensor*) – The measurements
- **space** (*torch.Tensor*) – A rank 2 tensor of the entire visible space.
- **bases_batch** (*numpy.ndarray*) – The bases in which the measurements are made

Returns A two-element list containing the amplitude and phase RBM gradients calculated with an exact negative phase update

Return type list[torch.Tensor]

compute_normalization(*space*)
Alias for *normalization*

property device
The device that the model is on.

fit(*data*, *epochs=100*, *pos_batch_size=100*, *neg_batch_size=None*, *k=1*, *lr=0.001*, *input_bases=None*, *progbar=False*, *starting_epoch=1*, *time=False*, *callbacks=None*, *optimizer=torch.optim.SGD*, *optimizer_args=None*, *scheduler=None*, *scheduler_args=None*, ***kwargs*)
Train the NeuralState.

Parameters

- **data** (*numpy.ndarray*) – The training samples
- **epochs** (*int*) – The number of full training passes through the dataset. Technically, this specifies the index of the *last* training epoch, which is relevant if *starting_epoch* is being set.
- **pos_batch_size** (*int*) – The size of batches for the positive phase taken from the data.
- **neg_batch_size** (*int*) – The size of batches for the negative phase taken from the data. Defaults to *pos_batch_size*.
- **k** (*int*) – The number of contrastive divergence steps.
- **lr** (*float*) – Learning rate
- **input_bases** (*numpy.ndarray*) – The measurement bases for each sample. Must be provided if training a ComplexWaveFunction or DensityMatrix.
- **progbar** (*bool or str*) – Whether or not to display a progress bar. If "notebook" is passed, will use a Jupyter notebook compatible progress bar.
- **starting_epoch** (*int*) – The epoch to start from. Useful if continuing training from a previous state.
- **callbacks** (*list[qucumber.callbacks.CallbackBase]*) – Callbacks to run while training.

- **optimizer** (*torch.optim.Optimizer*) – The constructor of a torch optimizer.
- **scheduler** – The constructor of a torch scheduler
- **optimizer_args** (*dict*) – Arguments to pass to the optimizer
- **scheduler_args** (*dict*) – Arguments to pass to the scheduler
- ****kwargs** – Ignored; exists for backwards compatibility.

generate_hilbert_space(*size=None*, *device=None*)
Generates Hilbert space of dimension 2^{size}.

Parameters

- **size** (*int*) – The size of each element of the Hilbert space. Defaults to the number of visible units.
- **device** – The device to create the Hilbert space matrix on. Defaults to the device this model is on.

Returns A tensor with all the basis states of the Hilbert space.

Return type torch.Tensor

gradient(*samples*, *bases=None*)
Compute the gradient of a batch of sample, measured in given bases.

Parameters

- **sample** (*numpy.ndarray*) – A batch of samples to compute the gradient of.
- **basis** (*numpy.ndarray or list[str] or None*) – A batch of bases.

Returns A list of 2 tensors containing the accumulated gradients of each of the internal RBMs.

Return type list[torch.Tensor]

importance_sampling_denominator(*v*)
Compute the denominator of the weight of an arbitrary sample, with respect to the sample *v*.

In the case of a mixed state, this quantity is $\rho(\sigma, \sigma)$, while in the pure case it is $\psi(\sigma')$

Parameters **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the denominator of the weights with respect to σ

Return type torch.Tensor

importance_sampling_numerator(*vp*, *v*)
Compute the numerator of the weight of sample *vp*, with respect to the sample *v*.

In the case of a mixed state, this quantity is $\rho(\sigma', \sigma)$, while in the pure case it is $\psi(\sigma')$

Parameters

- **vp** (*torch.Tensor*) – A batch containing the samples σ'
- **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the numerator of the weights of σ' with respect to σ

Return type torch.Tensor

importance_sampling_weight(*vp*, *v*)
Compute the weight of sample *vp*, with respect to the sample *v*.

In the case of a mixed state, this ratio is:

$$\frac{\rho(\sigma', \sigma)}{\rho(\sigma, \sigma)}$$

While in the pure case:

$$\frac{\psi(\sigma')}{\psi(\sigma)}$$

Parameters

- **vp** (*torch.Tensor*) – A batch containing the samples σ'
- **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the weights of σ' with respect to σ

Return type torch.Tensor

load(*location*)

Loads the NeuralState parameters from the given location ignoring any metadata stored in the file. Overwrites the NeuralState's parameters.

Note: The NeuralState object on which this function is called must have the same parameter shapes as the one who's parameters are being loaded.

Parameters **location** (*str or file*) – The location to load the NeuralState parameters from.

property max_size

Maximum size of the Hilbert space for full enumeration

property networks

A list of the names of the internal RBMs.

normalization(*space*)

Compute the normalization constant of the state. In the case of a pure state, this is the norm of the unnormalized wavefunction. In the case of a mixed state, this is the trace of the unnormalized density matrix.

$$Z_\lambda = \sum_\sigma p_\lambda(\sigma)$$

Parameters **space** (*torch.Tensor*) – A rank 2 tensor of the entire visible space.

ph_grads(*v*)

Computes the gradients of the phase RBM for given input states

Parameters **v** (*torch.Tensor*) – The input state, σ

Returns The gradients of all phase RBM parameters

Return type torch.Tensor

phase(*v*)

Compute the phase of a given vector/matrix of visible states.

$$\text{phase}(\sigma) = -\mathcal{E}_\mu(\sigma)/2$$

Parameters **v** (*torch.Tensor*) – visible states σ.

Returns Vector containing the phases of the given states.

Return type torch.Tensor

positive_phase_gradients(*samples_batch*, *bases_batch=None*)
Computes the positive phase of the gradients of the parameters.

Parameters

- **samples_batch** (*torch.Tensor*) – The measurements
- **bases_batch** (*numpy.ndarray*) – The bases in which the measurements are made

Returns A two-element list containing the amplitude and phase RBM gradients

Return type list[torch.Tensor]

probability(*v*, *Z=1.0*)
Evaluates the probability of the given vector(s) of visible states. Assumes the visible states were measured in the computational basis.

Parameters

- **v** (*torch.Tensor*) – The visible states.
- **Z** (*float*) – The partition function / normalization constant. Defaults to 1, producing unnormalized probabilities.

Returns The probability of the given vector(s) of visible units.

Return type torch.Tensor

psi(*v*)
Compute the (unnormalized) wavefunction of a given vector/matrix of visible states.

$$\psi_{\lambda\mu}(\sigma) = e^{-[\mathcal{E}_\lambda(\sigma)+i\mathcal{E}_\mu(\sigma)]/2}$$

Parameters **v** (*torch.Tensor*) – visible states σ

Returns Complex object containing the value of the wavefunction for each visible state

Return type torch.Tensor

property rbm_am
The RBM to be used to learn the wavefunction amplitude.

property rbm_ph
RBM used to learn the wavefunction phase.

reinitialize_parameters()
Randomize the parameters of the internal RBMs.

rotated_gradient(*basis*, *sample*)
Computes the gradients rotated into the measurement basis

Parameters

- **basis** (*numpy.ndarray*) – The bases in which the measurement is made
- **sample** (*torch.Tensor*) – The measurement (either 0 or 1)

Returns A list of two tensors, representing the rotated gradients of the amplitude and phase RBMS

Return type list[torch.Tensor, torch.Tensor]

sample(*k*, *num_samples=1*, *initial_state=None*, *overwrite=False*)
Performs k steps of Block Gibbs sampling. One step consists of sampling the hidden state h from the conditional distribution $p_\lambda(h|v)$, and sampling the visible state v from the conditional distribution $p_\lambda(v|h)$.

Parameters

- **k** (*int*) – Number of Block Gibbs steps.
- **num_samples** (*int*) – The number of samples to generate.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chains. If given, *num_samples* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the initial_state tensor, if it is provided.

save(*location*, *metadata=None*)
Saves the NeuralState parameters to the given location along with any given metadata.

Parameters

- **location** (*str or file*) – The location to save the data.
- **metadata** (*dict*) – Any extra metadata to store alongside the NeuralState parameters.

property stop_training
If *True*, will not train.

If this property is set to *True* during the training cycle, training will terminate once the current batch or epoch ends (depending on when *stop_training* was set).

subspace_vector(*num*, *size=None*, *device=None*)
Generates a single vector from the Hilbert space of dimension 2^{size}.

Parameters

- **size** (*int*) – The size of each element of the Hilbert space.
- **num** (*int*) – The specific vector to return from the Hilbert space. Since the Hilbert space can be represented by the set of binary strings of length *size*, *num* is equivalent to the decimal representation of the returned vector.
- **device** – The device to create the vector on. Defaults to the device this model is on.

Returns A state from the Hilbert space.

Return type torch.Tensor

11.3 Density Matrix

class qucumber.nn_states.**DensityMatrix**(*num_visible*, *num_hidden=None*, *num_aux=None*, *unitary_dict=None*, *gpu=False*, *module=None*)
Bases: *qucumber.nn_states.NeuralStateBase*

Parameters

- **num_visible** (*int*) – The number of visible units, i.e. the size of the system
- **num_hidden** (*int*) – The number of units in the hidden layer
- **num_aux** (*int*) – The number of units in the purification layer
- **unitary_dict** (*dict[str, torch.Tensor]*) – A dictionary associating bases with their unitary rotations

- **gpu** (*bool*) – Whether to perform computations on the default gpu.

am_grads(*v*)
Computes the gradients of the amplitude RBM for given input states

Parameters **v** (*torch.Tensor*) – The first input state, σ

Returns The gradients of all amplitude RBM parameters

Return type torch.Tensor

static autoload(*location*, *gpu=False*)
Initializes a NeuralState from the parameters in the given location.

Parameters

- **location** (*str or file*) – The location to load the model parameters from.
- **gpu** (*bool*) – Whether the returned model should be on the GPU.

Returns A new NeuralState initialized from the given parameters. The returned NeuralState will be of whichever type this function was called on. An error may be thrown if the loaded parameters correspond to a different type of NeuralState than the caller.

compute_batch_gradients(*k*, *samples_batch*, *neg_batch*, *bases_batch=None*)
Compute the gradients of a batch of the training data (*samples_batch*).

If measurements are taken in bases other than the reference basis, a list of bases (*bases_batch*) must also be provided.

Parameters

- **k** (*int*) – Number of contrastive divergence steps in training.
- **samples_batch** (*torch.Tensor*) – Batch of the input samples.
- **neg_batch** (*torch.Tensor*) – Batch of the input samples for computing the negative phase.
- **bases_batch** (*numpy.ndarray*) – Batch of the input bases corresponding to the samples in *samples_batch*.

Returns A two-element list containing the amplitude and phase RBM gradients calculated with a Gibbs sampled negative phase update

Return type list[torch.Tensor]

compute_exact_gradients(*samples_batch*, *space*, *bases_batch=None*)
Computes the gradients of the parameters, using exact sampling for the negative phase update instead of Gibbs sampling

Parameters

- **samples_batch** (*torch.Tensor*) – The measurements
- **space** (*torch.Tensor*) – A rank 2 tensor of the entire visible space.
- **bases_batch** (*numpy.ndarray*) – The bases in which the measurements are made

Returns A two-element list containing the amplitude and phase RBM gradients calculated with an exact negative phase update

Return type list[torch.Tensor]

compute_normalization(*space*)
Alias for *normalization*

property device
The device that the model is on.

fit(*data*, *epochs=100*, *pos_batch_size=100*, *neg_batch_size=None*, *k=1*, *lr=1*, *input_bases=None*, *progbar=False*, *starting_epoch=1*, *time=False*, *callbacks=None*, *optimizer=torch.optim.SGD*, *optimizer_args=None*, *scheduler=None*, *scheduler_args=None*, ***kwargs*)
Train the NeuralState.

Parameters

- **data** (*numpy.ndarray*) – The training samples
- **epochs** (*int*) – The number of full training passes through the dataset. Technically, this specifies the index of the *last* training epoch, which is relevant if *starting_epoch* is being set.
- **pos_batch_size** (*int*) – The size of batches for the positive phase taken from the data.
- **neg_batch_size** (*int*) – The size of batches for the negative phase taken from the data. Defaults to *pos_batch_size*.
- **k** (*int*) – The number of contrastive divergence steps.
- **lr** (*float*) – Learning rate
- **input_bases** (*numpy.ndarray*) – The measurement bases for each sample. Must be provided if training a ComplexWaveFunction or DensityMatrix.
- **progbar** (*bool or str*) – Whether or not to display a progress bar. If "notebook" is passed, will use a Jupyter notebook compatible progress bar.
- **starting_epoch** (*int*) – The epoch to start from. Useful if continuing training from a previous state.
- **callbacks** (*list[qucumber.callbacks.CallbackBase]*) – Callbacks to run while training.
- **optimizer** (*torch.optim.Optimizer*) – The constructor of a torch optimizer.
- **scheduler** – The constructor of a torch scheduler
- **optimizer_args** (*dict*) – Arguments to pass to the optimizer
- **scheduler_args** (*dict*) – Arguments to pass to the scheduler
- ****kwargs** – Ignored; exists for backwards compatibility.

generate_hilbert_space(*size=None*, *device=None*)
Generates Hilbert space of dimension 2^{size}.

Parameters

- **size** (*int*) – The size of each element of the Hilbert space. Defaults to the number of visible units.
- **device** – The device to create the Hilbert space matrix on. Defaults to the device this model is on.

Returns A tensor with all the basis states of the Hilbert space.

Return type torch.Tensor

gradient(*samples*, *bases=None*)
Compute the gradient of a batch of sample, measured in given bases.

Parameters

- **sample** (*numpy.ndarray*) – A batch of samples to compute the gradient of.
- **basis** (*numpy.ndarray or list[str] or None*) – A batch of bases.

Returns A list of 2 tensors containing the accumulated gradients of each of the internal RBMs.

Return type list[torch.Tensor]

importance_sampling_denominator(*v*)

Compute the denominator of the weight of an arbitrary sample, with respect to the sample *v*.

In the case of a mixed state, this quantity is $\rho(\sigma, \sigma)$, while in the pure case it is $\psi(\sigma')$

Parameters **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the denominator of the weights with respect to σ

Return type torch.Tensor

importance_sampling_numerator(*vp*, *v*)

Compute the numerator of the weight of sample *vp*, with respect to the sample *v*.

In the case of a mixed state, this quantity is $\rho(\sigma', \sigma)$, while in the pure case it is $\psi(\sigma')$

Parameters

- **vp** (*torch.Tensor*) – A batch containing the samples σ'
- **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the numerator of the weights of σ' with respect to σ

Return type torch.Tensor

importance_sampling_weight(*vp*, *v*)

Compute the weight of sample *vp*, with respect to the sample *v*.

In the case of a mixed state, this ratio is:

$$\frac{\rho(\sigma', \sigma)}{\rho(\sigma, \sigma)}$$

While in the pure case:

$$\frac{\psi(\sigma')}{\psi(\sigma)}$$

Parameters

- **vp** (*torch.Tensor*) – A batch containing the samples σ'
- **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the weights of σ' with respect to σ

Return type torch.Tensor

load(*location*)

Loads the NeuralState parameters from the given location ignoring any metadata stored in the file. Overwrites the NeuralState's parameters.

Note: The NeuralState object on which this function is called must have the same parameter shapes as the one who's parameters are being loaded.

Parameters **location** (*str or file*) – The location to load the NeuralState parameters from.

property max_size
Maximum size of the Hilbert space for full enumeration

property networks
A list of the names of the internal RBMs.

normalization(*space*)
Compute the normalization constant of the state. In the case of a pure state, this is the norm of the unnormalized wavefunction. In the case of a mixed state, this is the trace of the unnormalized density matrix.

$$Z_\lambda = \sum_\sigma p_\lambda(\sigma)$$

Parameters **space** (*torch.Tensor*) – A rank 2 tensor of the entire visible space.

ph_grads(*v*)
Computes the gradients of the phase RBM for given input states

Parameters **v** (*torch.Tensor*) – The first input state, σ

Returns The gradients of all phase RBM parameters

Return type torch.Tensor

pi(*v*, *vp*, *expand=True*)
Calculates elements of the Π matrix. If *expand* is *True*, will return a complex matrix $A_{ij} = \langle\sigma_i|\Pi|\sigma'_j\rangle$. Otherwise will return a complex vector $A_i = \langle\sigma_i|\Pi|\sigma'_i\rangle$.

Parameters

- **v** (*torch.Tensor*) – A batch of visible states, σ.
- **vp** (*torch.Tensor*) – The other batch of visible state, σ'.
- **expand** (*bool*) – Whether to return a matrix (*True*) or a vector (*False*).

Returns The matrix elements given by $\langle\sigma|\Pi|\sigma'\rangle$

Return type torch.Tensor

pi_grad(*v*, *vp*, *phase=False*, *expand=False*)

Calculates the gradient of the Π matrix with respect to the amplitude RBM parameters for two input states

Parameters

- **v** (*torch.Tensor*) – One of the visible states, σ
- **vp** (*torch.Tensor*) – The other visible state, :math`sigma``
- **phase** (*bool*) – Whether to compute the gradients for the phase RBM (*True*) or the amplitude RBM (*False*)

Returns The matrix element of the gradient given by $\langle\sigma|\nabla_\lambda\Pi|\sigma'\rangle$

Return type torch.Tensor

positive_phase_gradients(*samples_batch*, *bases_batch=None*)
Computes the positive phase of the gradients of the parameters.

Parameters

- **samples_batch** (*torch.Tensor*) – The measurements
- **bases_batch** (*numpy.ndarray*) – The bases in which the measurements are made

Returns A two-element list containing the amplitude and phase RBM gradients

Return type list[torch.Tensor]

probability(*v*, *Z=1.0*)

Evaluates the probability of the given vector(s) of visible states. Assumes the visible states were measured in the computational basis.

Parameters

- **v** (*torch.Tensor*) – The visible states.
- **Z** (*float*) – The partition function / normalization constant. Defaults to 1, producing unnormalized probabilities.

Returns The probability of the given vector(s) of visible units.

Return type torch.Tensor

property rbm_am

The RBM to be used to learn the wavefunction amplitude.

property rbm_ph

RBM used to learn the wavefunction phase.

reinitialize_parameters()

Randomize the parameters of the internal RBMs.

rho(*v*, *vp=None*, *expand=True*)

Computes the matrix elements of the (unnormalized) density matrix. If *expand* is *True*, will return a complex matrix $A_{ij} = \langle \sigma_i | \widetilde{\rho} | \sigma'_j \rangle$. Otherwise will return a complex vector $A_i = \langle \sigma_i | \widetilde{\rho} | \sigma'_i \rangle$.

Parameters

- **v** (*torch.Tensor*) – One of the visible states, σ.
- **vp** (*torch.Tensor*) – The other visible state, σ'. If *None*, will be set to *v*.
- **expand** (*bool*) – Whether to return a matrix (*True*) or a vector (*False*).

Returns The elements of the current density matrix $\langle \sigma | \widetilde{\rho} | \sigma' \rangle$

Return type torch.Tensor

rotated_gradient(*basis*, *sample*)

Computes the gradients rotated into the measurement basis

Parameters

- **basis** (*numpy.ndarray*) – The bases in which the measurement is made
- **sample** (*torch.Tensor*) – The measurement (either 0 or 1)

Returns A list of two tensors, representing the rotated gradients of the amplitude and phase RBMs

Return type list[torch.Tensor, torch.Tensor]

sample(*k*, *num_samples=1*, *initial_state=None*, *overwrite=False*)

Performs k steps of Block Gibbs sampling. One step consists of sampling the hidden state h from the conditional distribution $p_\lambda(h|v)$, and sampling the visible state v from the conditional distribution $p_\lambda(v|h)$.

Parameters

- **k** (*int*) – Number of Block Gibbs steps.
- **num_samples** (*int*) – The number of samples to generate.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chains. If given, *num_samples* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the initial_state tensor, if it is provided.

save(*location*, *metadata=None*)
Saves the NeuralState parameters to the given location along with any given metadata.

Parameters

- **location** (*str or file*) – The location to save the data.
- **metadata** (*dict*) – Any extra metadata to store alongside the NeuralState parameters.

property stop_training
If *True*, will not train.

If this property is set to *True* during the training cycle, training will terminate once the current batch or epoch ends (depending on when *stop_training* was set).

subspace_vector(*num*, *size=None*, *device=None*)
Generates a single vector from the Hilbert space of dimension 2^{size}.

Parameters

- **size** (*int*) – The size of each element of the Hilbert space.
- **num** (*int*) – The specific vector to return from the Hilbert space. Since the Hilbert space can be represented by the set of binary strings of length *size*, *num* is equivalent to the decimal representation of the returned vector.
- **device** – The device to create the vector on. Defaults to the device this model is on.

Returns A state from the Hilbert space.

Return type torch.Tensor

11.4 Abstract WaveFunction

Note: This is an Abstract Base Class, it is not meant to be used directly. The following API reference is mostly for developers.

class qucumber.nn_states.**WaveFunctionBase**
Bases: *qucumber.nn_states.NeuralStateBase*

Abstract Base Class for WaveFunctions.

amplitude(*v*)
Compute the (unnormalized) amplitude of a given vector/matrix of visible states.

$$\text{amplitude}(\sigma) = |\psi(\sigma)|$$

Parameters **v** (*torch.Tensor*) – visible states σ

Returns Matrix/vector containing the amplitudes of v

Return type torch.Tensor

importance_sampling_denominator(*v*)
Compute the denominator of the weight of an arbitrary sample, with respect to the sample *v*.

In the case of a mixed state, this quantity is $\rho(\sigma, \sigma)$, while in the pure case it is $\psi(\sigma')$

Parameters **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the denominator of the weights with respect to σ

Return type torch.Tensor

importance_sampling_numerator(*vp*, *v*)
Compute the numerator of the weight of sample *vp*, with respect to the sample *v*.

In the case of a mixed state, this quantity is $\rho(\sigma', \sigma)$, while in the pure case it is $\psi(\sigma')$

Parameters

- **vp** (*torch.Tensor*) – A batch containing the samples σ'
- **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the numerator of the weights of σ' with respect to σ

Return type torch.Tensor

abstract phase(*v*)
Compute the phase of a given vector/matrix of visible states.

$$\text{phase}(\sigma)$$

Parameters **v** (*torch.Tensor*) – visible states σ

Returns Matrix/vector containing the phases of v

Return type torch.Tensor

psi(*v*)
Compute the (unnormalized) wavefunction of a given vector/matrix of visible states.

$$\psi(\sigma)$$

Parameters **v** (*torch.Tensor*) – visible states σ

Returns Complex object containing the value of the wavefunction for each visible state

Return type torch.Tensor

reinitialize_parameters()
Randomize the parameters of the internal RBMs.

11.5 Abstract NeuralState

Note: This is an Abstract Base Class, it is not meant to be used directly. The following API reference is mostly for developers.

class qucumber.nn_states.**NeuralStateBase**
Bases: abc.ABC

Abstract Base Class for Neural Network Quantum States.

abstract static autoload(*location*, *gpu=False*)
Initializes a NeuralState from the parameters in the given location.

Parameters

- **location** (*str or file*) – The location to load the model parameters from.
- **gpu** (*bool*) – Whether the returned model should be on the GPU.

Returns A new NeuralState initialized from the given parameters. The returned NeuralState will be of whichever type this function was called on. An error may be thrown if the loaded parameters correspond to a different type of NeuralState than the caller.

compute_batch_gradients(*k*, *samples_batch*, *neg_batch*, *bases_batch=None*)
Compute the gradients of a batch of the training data (*samples_batch*).

If measurements are taken in bases other than the reference basis, a list of bases (*bases_batch*) must also be provided.

Parameters

- **k** (*int*) – Number of contrastive divergence steps in training.
- **samples_batch** (*torch.Tensor*) – Batch of the input samples.
- **neg_batch** (*torch.Tensor*) – Batch of the input samples for computing the negative phase.
- **bases_batch** (*numpy.ndarray*) – Batch of the input bases corresponding to the samples in *samples_batch*.

Returns A two-element list containing the amplitude and phase RBM gradients calculated with a Gibbs sampled negative phase update

Return type list[torch.Tensor]

compute_exact_gradients(*samples_batch*, *space*, *bases_batch=None*)
Computes the gradients of the parameters, using exact sampling for the negative phase update instead of Gibbs sampling

Parameters

- **samples_batch** (*torch.Tensor*) – The measurements
- **space** (*torch.Tensor*) – A rank 2 tensor of the entire visible space.
- **bases_batch** (*numpy.ndarray*) – The bases in which the measurements are made

Returns A two-element list containing the amplitude and phase RBM gradients calculated with an exact negative phase update

Return type list[torch.Tensor]

compute_normalization(*space*)
Alias for *normalization*

abstract property device
The device that the model is on.

fit(*data*, *epochs=100*, *pos_batch_size=100*, *neg_batch_size=None*, *k=1*, *lr=0.001*, *input_bases=None*, *progbar=False*, *starting_epoch=1*, *time=False*, *callbacks=None*, *optimizer=torch.optim.SGD*, *optimizer_args=None*, *scheduler=None*, *scheduler_args=None*, ***kwargs*)
Train the NeuralState.

Parameters

- **data** (*numpy.ndarray*) – The training samples

- **epochs** (*int*) – The number of full training passes through the dataset. Technically, this specifies the index of the *last* training epoch, which is relevant if *starting_epoch* is being set.
- **pos_batch_size** (*int*) – The size of batches for the positive phase taken from the data.
- **neg_batch_size** (*int*) – The size of batches for the negative phase taken from the data. Defaults to *pos_batch_size*.
- **k** (*int*) – The number of contrastive divergence steps.
- **lr** (*float*) – Learning rate
- **input_bases** (*numpy.ndarray*) – The measurement bases for each sample. Must be provided if training a ComplexWaveFunction or DensityMatrix.
- **progbar** (*bool or str*) – Whether or not to display a progress bar. If "notebook" is passed, will use a Jupyter notebook compatible progress bar.
- **starting_epoch** (*int*) – The epoch to start from. Useful if continuing training from a previous state.
- **callbacks** (*list[qucumber.callbacks.CallbackBase]*) – Callbacks to run while training.
- **optimizer** (*torch.optim.Optimizer*) – The constructor of a torch optimizer.
- **scheduler** – The constructor of a torch scheduler
- **optimizer_args** (*dict*) – Arguments to pass to the optimizer
- **scheduler_args** (*dict*) – Arguments to pass to the scheduler
- ****kwargs** – Ignored; exists for backwards compatibility.

generate_hilbert_space(*size=None*, *device=None*)

Generates Hilbert space of dimension 2^{size}.

Parameters

- **size** (*int*) – The size of each element of the Hilbert space. Defaults to the number of visible units.
- **device** – The device to create the Hilbert space matrix on. Defaults to the device this model is on.

Returns A tensor with all the basis states of the Hilbert space.

Return type torch.Tensor

gradient(*samples*, *bases=None*)

Compute the gradient of a batch of sample, measured in given bases.

Parameters

- **sample** (*numpy.ndarray*) – A batch of samples to compute the gradient of.
- **basis** (*numpy.ndarray or list[str] or None*) – A batch of bases.

Returns A list of 2 tensors containing the accumulated gradients of each of the internal RBMs.

Return type list[torch.Tensor]

abstract importance_sampling_denominator(*v*)

Compute the denominator of the weight of an arbitrary sample, with respect to the sample *v*.

In the case of a mixed state, this quantity is $\rho(\sigma, \sigma)$, while in the pure case it is $\psi(\sigma')$

Parameters **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the denominator of the weights with respect to σ

Return type torch.Tensor

abstract importance_sampling_numerator(*vp*, *v*)

Compute the numerator of the weight of sample *vp*, with respect to the sample *v*.

In the case of a mixed state, this quantity is $\rho(\sigma', \sigma)$, while in the pure case it is $\psi(\sigma')$

Parameters

- **vp** (*torch.Tensor*) – A batch containing the samples σ'
- **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the numerator of the weights of σ' with respect to σ

Return type torch.Tensor

importance_sampling_weight(*vp*, *v*)

Compute the weight of sample *vp*, with respect to the sample *v*.

In the case of a mixed state, this ratio is:

$$\frac{\rho(\sigma', \sigma)}{\rho(\sigma, \sigma)}$$

While in the pure case:

$$\frac{\psi(\sigma')}{\psi(\sigma)}$$

Parameters

- **vp** (*torch.Tensor*) – A batch containing the samples σ'
- **v** (*torch.Tensor*) – A batch containing the samples σ

Returns A complex tensor containing the weights of σ' with respect to σ

Return type torch.Tensor

load(*location*)

Loads the NeuralState parameters from the given location ignoring any metadata stored in the file. Overwrites the NeuralState's parameters.

Note: The NeuralState object on which this function is called must have the same parameter shapes as the one who's parameters are being loaded.

Parameters **location** (*str or file*) – The location to load the NeuralState parameters from.

property max_size

Maximum size of the Hilbert space for full enumeration

abstract property networks

A list of the names of the internal RBMs.

normalization(*space*)
Compute the normalization constant of the state. In the case of a pure state, this is the norm of the unnormalized wavefunction. In the case of a mixed state, this is the trace of the unnormalized density matrix.

$$Z_\lambda = \sum_\sigma p_\lambda(\sigma)$$

Parameters **space** (*torch.Tensor*) – A rank 2 tensor of the entire visible space.

positive_phase_gradients(*samples_batch*, *bases_batch=None*)
Computes the positive phase of the gradients of the parameters.

Parameters

- **samples_batch** (*torch.Tensor*) – The measurements
- **bases_batch** (*numpy.ndarray*) – The bases in which the measurements are made

Returns A two-element list containing the amplitude and phase RBM gradients

Return type list[torch.Tensor]

probability(*v*, *Z=1.0*)
Evaluates the probability of the given vector(s) of visible states. Assumes the visible states were measured in the computational basis.

Parameters

- **v** (*torch.Tensor*) – The visible states.
- **Z** (*float*) – The partition function / normalization constant. Defaults to 1, producing unnormalized probabilities.

Returns The probability of the given vector(s) of visible units.

Return type torch.Tensor

abstract property rbm_am
The RBM to be used to learn the wavefunction amplitude.

reinitialize_parameters()
Randomize the parameters of the internal RBMs.

sample(*k*, *num_samples=1*, *initial_state=None*, *overwrite=False*)
Performs k steps of Block Gibbs sampling. One step consists of sampling the hidden state h from the conditional distribution $p_\lambda(h|v)$, and sampling the visible state v from the conditional distribution $p_\lambda(v|h)$.

Parameters

- **k** (*int*) – Number of Block Gibbs steps.
- **num_samples** (*int*) – The number of samples to generate.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chains. If given, *num_samples* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the initial_state tensor, if it is provided.

save(*location*, *metadata=None*)
Saves the NeuralState parameters to the given location along with any given metadata.

Parameters

- **location** (*str or file*) – The location to save the data.
- **metadata** (*dict*) – Any extra metadata to store alongside the NeuralState parameters.

property stop_training
If *True*, will not train.

If this property is set to *True* during the training cycle, training will terminate once the current batch or epoch ends (depending on when *stop_training* was set).

subspace_vector(*num*, *size=None*, *device=None*)
Generates a single vector from the Hilbert space of dimension 2^{size}.

Parameters

- **size** (*int*) – The size of each element of the Hilbert space.
- **num** (*int*) – The specific vector to return from the Hilbert space. Since the Hilbert space can be represented by the set of binary strings of length *size*, *num* is equivalent to the decimal representation of the returned vector.
- **device** – The device to create the vector on. Defaults to the device this model is on.

Returns A state from the Hilbert space.

Return type torch.Tensor

CHAPTER

TWELVE

CALLBACKS

12.1 Base Callback

class qucumber.callbacks.**CallbackBase**

Base class for callbacks.

on_batch_end(*nn_state*, *epoch*, *batch*)

Called at the end of each batch.

Parameters

- **nn_state** (qucumber.nn_states.WaveFunctionBase) – The WaveFunction being trained.
- **epoch** (*int*) – The current epoch.
- **batch** (*int*) – The current batch index.

on_batch_start(*nn_state*, *epoch*, *batch*)

Called at the start of each batch.

Parameters

- **nn_state** (qucumber.nn_states.WaveFunctionBase) – The WaveFunction being trained.
- **epoch** (*int*) – The current epoch.
- **batch** (*int*) – The current batch index.

on_epoch_end(*nn_state*, *epoch*)

Called at the end of each epoch.

Parameters

- **nn_state** (qucumber.nn_states.WaveFunctionBase) – The WaveFunction being trained.
- **epoch** (*int*) – The current epoch.

on_epoch_start(*nn_state*, *epoch*)

Called at the start of each epoch.

Parameters

- **nn_state** (qucumber.nn_states.WaveFunctionBase) – The WaveFunction being trained.
- **epoch** (*int*) – The current epoch.

on_train_end(*nn_state*)
Called at the end of the training cycle.

Parameters **nn_state** (qucumber.nn_states.WaveFunctionBase) – The WaveFunction being trained.

on_train_start(*nn_state*)
Called at the start of the training cycle.

Parameters **nn_state** (qucumber.nn_states.WaveFunctionBase) – The WaveFunction being trained.

12.2 LambdaCallback

class qucumber.callbacks.**LambdaCallback**(*on_train_start=None*, *on_train_end=None*, *on_epoch_start=None*, *on_epoch_end=None*, *on_batch_start=None*, *on_batch_end=None*)
Class for creating simple callbacks.

This callback is constructed using the passed functions that will be called at the appropriate time.

Parameters

- **on_train_start** (*callable or None*) – A function to be called at the start of the training cycle. Must follow the same signature as *CallbackBase.on_train_start*.
- **on_train_end** (*callable or None*) – A function to be called at the end of the training cycle. Must follow the same signature as *CallbackBase.on_train_end*.
- **on_epoch_start** (*callable or None*) – A function to be called at the start of every epoch. Must follow the same signature as *CallbackBase.on_epoch_start*.
- **on_epoch_end** (*callable or None*) – A function to be called at the end of every epoch. Must follow the same signature as *CallbackBase.on_epoch_end*.
- **on_batch_start** (*callable or None*) – A function to be called at the start of every batch. Must follow the same signature as *CallbackBase.on_batch_start*.
- **on_batch_end** (*callable or None*) – A function to be called at the end of every batch. Must follow the same signature as *CallbackBase.on_batch_end*.

12.3 ModelSaver

class qucumber.callbacks.**ModelSaver**(*period*, *folder_path*, *file_name*, *save_initial=True*, *metadata=None*, *metadata_only=False*)
Callback which allows model parameters (along with some metadata) to be saved to disk at regular intervals.

This callback is called at the end of each epoch. If *save_initial* is *True*, will also be called at the start of the training cycle.

Parameters

- **period** (*int*) – Frequency of model saving (in epochs).
- **folder_path** (*str*) – The directory in which to save the files
- **file_name** (*str*) – The name of the output files. Should be a format string with one blank, which will be filled with either the epoch number or the word "initial".

- **save_initial** (*bool*) – Whether to save the initial parameters (and metadata).
- **metadata** (*callable or dict or None*) – The metadata to save to disk with the model parameters Can be either a function or a dictionary. In the case of a function, it must take 2 arguments the RBM being trained, and the current epoch number, and then return a dictionary containing the metadata to be saved.
- **metadata_only** (*bool*) – Whether to save *only* the metadata to disk.

12.4 Logger

class qucumber.callbacks.**Logger**(*period*, *logger_fn=<built-in function print>*, *msg_gen=None*, ***msg_gen_kwargs*)

Callback which logs output at regular intervals.

This callback is called at the end of each epoch.

Parameters

- **period** (*int*) – Logging frequency (in epochs).
- **logger_fn** (*callable*) – The function used for logging. Must take 1 string as an argument. Defaults to the standard *print* function.
- **msg_gen** (*callable*) – A callable which generates the string to be logged. Must take 2 positional arguments: the RBM being trained and the current epoch. It must also be able to take some keyword arguments.
- ****kwargs** – Keyword arguments which will be passed to *msg_gen*.

12.5 EarlyStopping

class qucumber.callbacks.**EarlyStopping**(*period*, *tolerance*, *patience*, *evaluator_callback*, *quantity_name*, *criterion='relative'*)

Stop training once the model stops improving.

There are three different stopping criteria available:

relative, which computes the relative change between the two model evaluation steps:

$$\left|\frac{M_{t-p} - M_t}{M_{t-p}}\right| < \epsilon$$

absolute computes the absolute change:

$$|M_{t-p} - M_t| < \epsilon$$

variance computes the absolute change, but scales the change by the standard deviation of the quantity of interest, such that the tolerance, *epsilon* can now be interpreted as the "number of standard deviations":

$$\left|\frac{M_{t-p} - M_t}{\sigma_{t-p}}\right| < \epsilon$$

where M_t is the metric value at the current evaluation (time t), p is the "patience" parameter, σ_t is the standard deviation of the metric, and ϵ is the tolerance.

This callback is called at the end of each epoch.

Parameters

- **period** (*int*) – Frequency with which the callback checks whether training has converged (in epochs).
- **tolerance** (*float*) – The maximum relative change required to consider training as having converged.
- **patience** (*int*) – How many intervals to wait before claiming the training has converged.
- **evaluator_callback** (*MetricEvaluator* or *ObservableEvaluator*) – An instance of *MetricEvaluator* or *ObservableEvaluator* which computes the metric that we want to check for convergence.
- **quantity_name** (*str*) – The name of the metric/observable stored in *evaluator_callback*.
- **criterion** (*str*) – The stopping criterion to use. Must be one of the following: *relative*, *absolute*, *variance*.

12.6 VarianceBasedEarlyStopping

class qucumber.callbacks.**VarianceBasedEarlyStopping**(*period*, *tolerance*, *patience*, *evaluator_callback*, *quantity_name*, *variance_name=None*)

Deprecated since version 1.2: Use *EarlyStopping* instead.

Stop training once the model stops improving. This is a variation on the *EarlyStopping* class which takes the variance of the metric into account.

The specific criterion for stopping is:

$$\left| \frac{M_{t-p} - M_t}{\sigma_{t-p}} \right| < \kappa$$

where M_t is the metric value at the current evaluation (time t), p is the "patience" parameter, σ_t is the standard deviation of the metric, and κ is the tolerance.

This callback is called at the end of each epoch.

Parameters

- **period** (*int*) – Frequency with which the callback checks whether training has converged (in epochs).
- **tolerance** (*float*) – The maximum (standardized) change required to consider training as having converged.
- **patience** (*int*) – How many intervals to wait before claiming the training has converged.
- **evaluator_callback** (*MetricEvaluator* or *ObservableEvaluator*) – An instance of *MetricEvaluator* or *ObservableEvaluator* which computes the metric/observable that we want to check for convergence.
- **quantity_name** (*str*) – The name of the metric/observable stored in *evaluator_callback*.
- **variance_name** (*str*) – The name of the variance stored in *evaluator_callback*. Ignored, exists for backward compatibility.

12.7 MetricEvaluator

class qucumber.callbacks.**MetricEvaluator**(*period*, *metrics*, *verbose=False*, *log=None*, ***metric_kwargs*)

Evaluate and hold on to the results of the given metric(s).

This callback is called at the end of each epoch.

Note: Since callbacks are given to *fit* as a list, they will be called in a deterministic order. It is therefore recommended that instances of *MetricEvaluator* be among the first callbacks in the list passed to *fit*, as one would often use it in conjunction with other callbacks like *EarlyStopping* which may depend on *MetricEvaluator* having been called.

Parameters

- **period** (*int*) – Frequency with which the callback evaluates the given metric(s).
- **metrics** (*dict(str, callable)*) – A dictionary of callables where the keys are the names of the metrics and the callables take the NeuralState being trained as their positional argument, along with some keyword arguments. The metrics are evaluated and put into an internal dictionary structure resembling the structure of *metrics*.
- **verbose** (*bool*) – Whether to print metrics to stdout.
- **log** (*str*) – A filepath to log metric values to in CSV format.
- ****metric_kwargs** – Keyword arguments to be passed to *metrics*.

__getattr__(*metric*)

Return an array of all recorded values of the given metric.

Parameters **metric** (*str*) – The metric to retrieve.

Returns The past values of the metric.

Return type numpy.ndarray

__getitem__(*metric*)

Alias for *__getattr__* to enable subscripting.

__len__()

Return the number of timesteps that metrics have been evaluated for.

Return type int

clear_history()

Delete all metric values the instance is currently storing.

property epochs

Return a list of all epochs that have been recorded.

Return type numpy.ndarray

get_value(*name*, *index=None*)

Retrieve the value of the desired metric from the given timestep.

Parameters

- **name** (*str*) – The name of the metric to retrieve.

- **index** (*int or None*) – The index/timestep from which to retrieve the metric. Negative indices are supported. If None, will just get the most recent value.

property names
The names of the tracked metrics.

Return type list[str]

12.8 ObservableEvaluator

class qucumber.callbacks.**ObservableEvaluator**(*period*, *observables*, *verbose=False*, *log=None*, ***sampling_kwargs*)
Evaluate and hold on to the results of the given observable(s).

This callback is called at the end of each epoch.

Note: Since callback are given to *fit* as a list, they will be called in a deterministic order. It is therefore recommended that instances of *ObservableEvaluator* be among the first callbacks in the list passed to *fit*, as one would often use it in conjunction with other callbacks like *EarlyStopping* which may depend on *ObservableEvaluator* having been called.

Parameters

- **period** (*int*) – Frequency with which the callback evaluates the given observables(s).
- **observables** (*list (qucumber.observables.ObservableBase)*) – A list of Observables. Observable statistics are evaluated by sampling the NeuralState. Note that observables that have the same name will conflict, and precedence will be given to the one which appears later in the list.
- **verbose** (*bool*) – Whether to print metrics to stdout.
- **log** (*str*) – A filepath to log metric values to in CSV format.
- ***sampling_kwargs** – Keyword arguments to be passed to *Observable.statistics*. Ex. *num_samples*, *num_chains*, *burn_in*, *steps*.

__getattr__(*observable*)
Return an ObservableStatistics containing recorded statistics of the given observable.

Parameters **observable** (*str*) – The observable to retrieve.

Returns The past values of the observable.

Return type ObservableStatistics

__getitem__(*observable*)
Alias for *__getattr__* to enable subscripting.

__len__()
Return the number of timesteps that observables have been evaluated for.

Return type int

clear_history()
Delete all statistics the instance is currently storing.

property epochs
Return a list of all epochs that have been recorded.

Return type numpy.ndarray

get_value(*name*, *index=None*)
Retrieve the statistics of the desired observable from the given timestep.

Parameters

- **name** (*str*) – The name of the observable to retrieve.
- **index** (*int or None*) – The index/timestep from which to retrieve the observable. Negative indices are supported. If None, will just get the most recent value.

Return type dict(str, float)

property names
The names of the tracked observables.

Return type list[str]

12.9 LivePlotting

class qucumber.callbacks.**LivePlotting**(*period*, *evaluator_callback*, *quantity_name*, *error_name=None*, *total_epochs=None*, *smooth=True*)
Plots metrics/observables.

This callback is called at the end of each epoch.

Parameters

- **period** (*int*) – Frequency with which the callback updates the plots (in epochs).
- **evaluator_callback** (*MetricEvaluator* or *ObservableEvaluator*) – An instance of *MetricEvaluator* or *ObservableEvaluator* which computes the metric/observable that we want to plot.
- **quantity_name** (*str*) – The name of the metric/observable stored in *evaluator_callback*.
- **error_name** (*str*) – The name of the error stored in *evaluator_callback*.

12.10 Timer

class qucumber.callbacks.**Timer**(*verbose=True*)
Callback which records the training time.

This callback is always called at the start and end of training. It will run at the end of an epoch or batch if the given model's *stop_training* property is set to True.

Parameters **verbose** (*bool*) – Whether to print the elapsed time at the end of training.

CHAPTER

THIRTEEN

OBSERVABLES

13.1 Pauli Operators

class qucumber.observables.**SigmaZ**(*absolute=False*)

Bases: *qucumber.observables.ObservableBase*

The σ_z observable.

Computes the average magnetization in the Z direction of a spin chain.

Parameters **absolute** (*bool*) – Specifies whether to estimate the absolute magnetization.

apply(*nn_state*, *samples*)

Computes the average magnetization along Z of each sample given a batch of samples.

Assumes that the computational basis that the NeuralState was trained on was the Z basis.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState that drew the samples.
- **samples** (*torch.Tensor*) – A batch of samples to calculate the observable on. Must be using the $\sigma_i = 0, 1$ convention.

property name

The name of the Observable.

sample(*nn_state*, *k*, *num_samples=1*, *initial_state=None*, *overwrite=False*)

Draws samples of the *observable* using the given NeuralState.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState to draw samples from.
- **k** (*int*) – The number of Gibbs Steps to perform before drawing a sample.
- **num_samples** (*int*) – The number of samples to draw.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chain. If given, *num_samples* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the *initial_state* tensor, if provided, with the updated state of the Markov chain.

Returns The samples drawn through this observable.

Return type torch.Tensor

statistics(*nn_state*, *num_samples*, *num_chains=0*, *burn_in=1000*, *steps=1*, *initial_state=None*, *overwrite=False*)

Estimates the expected value, variance, and the standard error of the observable over the distribution defined by the NeuralState.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState to draw samples from.
- **num_samples** (*int*) – The number of samples to draw. The actual number of samples drawn may be slightly higher if *num_samples % num_chains != 0*.
- **num_chains** (*int*) – The number of Markov chains to run in parallel; if 0 or greater than *num_samples*, will use a number of chains equal to *num_samples*. This is not recommended in the case where a *num_samples* is large, as this may use up all the available memory. Ignored if *initial_state* is provided.
- **burn_in** (*int*) – The number of Gibbs Steps to perform before recording any samples.
- **steps** (*int*) – The number of Gibbs Steps to take between each sample.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chain. If given, *num_chains* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the *initial_state* tensor, if provided, with the updated state of the Markov chain.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable. Also outputs the total number of drawn samples (key: "num_samples").

Return type dict(str, float)

statistics_from_samples(*nn_state*, *samples*)

Estimates the expected value, variance, and the standard error of the observable using the given samples.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState that drew the samples.
- **samples** (*torch.Tensor*) – A batch of sample states to calculate the observable on.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable. Also outputs the total number of drawn samples (key: "num_samples").

Return type dict(str, float)

property symbol

The algebraic symbol representing the Observable.

class qucumber.observables.**SigmaX**(*absolute=False*)

Bases: *qucumber.observables.ObservableBase*

The σ_x observable

Computes the average magnetization in the X direction of a spin chain.

Parameters **absolute** (*bool*) – Specifies whether to estimate the absolute magnetization.

apply(*nn_state*, *samples*)

Computes the average magnetization along X of each sample in the given batch of samples.

Assumes that the computational basis that the NeuralState was trained on was the Z basis.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState that drew the samples.
- **samples** (*torch.Tensor*) – A batch of samples to calculate the observable on. Must be using the $\sigma_i = 0, 1$ convention.

property name
The name of the Observable.

sample(*nn_state*, *k*, *num_samples=1*, *initial_state=None*, *overwrite=False*)
Draws samples of the *observable* using the given NeuralState.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState to draw samples from.
- **k** (*int*) – The number of Gibbs Steps to perform before drawing a sample.
- **num_samples** (*int*) – The number of samples to draw.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chain. If given, *num_samples* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the *initial_state* tensor, if provided, with the updated state of the Markov chain.

Returns The samples drawn through this observable.

Return type torch.Tensor

statistics(*nn_state*, *num_samples*, *num_chains=0*, *burn_in=1000*, *steps=1*, *initial_state=None*, *overwrite=False*)
Estimates the expected value, variance, and the standard error of the observable over the distribution defined by the NeuralState.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState to draw samples from.
- **num_samples** (*int*) – The number of samples to draw. The actual number of samples drawn may be slightly higher if *num_samples % num_chains != 0*.
- **num_chains** (*int*) – The number of Markov chains to run in parallel; if 0 or greater than *num_samples*, will use a number of chains equal to *num_samples*. This is not recommended in the case where a *num_samples* is large, as this may use up all the available memory. Ignored if *initial_state* is provided.
- **burn_in** (*int*) – The number of Gibbs Steps to perform before recording any samples.
- **steps** (*int*) – The number of Gibbs Steps to take between each sample.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chain. If given, *num_chains* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the *initial_state* tensor, if provided, with the updated state of the Markov chain.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable. Also outputs the total number of drawn samples (key: "num_samples").

Return type dict(str, float)

statistics_from_samples(*nn_state*, *samples*)
Estimates the expected value, variance, and the standard error of the observable using the given samples.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState that drew the samples.
- **samples** (*torch.Tensor*) – A batch of sample states to calculate the observable on.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable. Also outputs the total number of drawn samples (key: "num_samples").

Return type dict(str, float)

property symbol
The algebraic symbol representing the Observable.

class qucumber.observables.**SigmaY**(*absolute=False*)
Bases: *qucumber.observables.ObservableBase*

The σ_y observable

Computes the average magnetization in the Y direction of a spin chain.

Parameters **absolute** (*bool*) – Specifies whether to estimate the absolute magnetization.

apply(*nn_state*, *samples*)
Computes the average magnetization along Y of each sample in the given batch of samples.

Assumes that the computational basis that the NeuralState was trained on was the Z basis.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState that drew the samples.
- **samples** (*torch.Tensor*) – A batch of samples to calculate the observable on. Must be using the $\sigma_i = 0, 1$ convention.

property name
The name of the Observable.

sample(*nn_state*, *k*, *num_samples=1*, *initial_state=None*, *overwrite=False*)
Draws samples of the *observable* using the given NeuralState.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState to draw samples from.
- **k** (*int*) – The number of Gibbs Steps to perform before drawing a sample.
- **num_samples** (*int*) – The number of samples to draw.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chain. If given, *num_samples* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the *initial_state* tensor, if provided, with the updated state of the Markov chain.

Returns The samples drawn through this observable.

Return type torch.Tensor

statistics(*nn_state*, *num_samples*, *num_chains=0*, *burn_in=1000*, *steps=1*, *initial_state=None*, *overwrite=False*)

Estimates the expected value, variance, and the standard error of the observable over the distribution defined by the NeuralState.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState to draw samples from.
- **num_samples** (*int*) – The number of samples to draw. The actual number of samples drawn may be slightly higher if *num_samples % num_chains != 0*.
- **num_chains** (*int*) – The number of Markov chains to run in parallel; if 0 or greater than *num_samples*, will use a number of chains equal to *num_samples*. This is not recommended in the case where a *num_samples* is large, as this may use up all the available memory. Ignored if *initial_state* is provided.
- **burn_in** (*int*) – The number of Gibbs Steps to perform before recording any samples.
- **steps** (*int*) – The number of Gibbs Steps to take between each sample.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chain. If given, *num_chains* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the *initial_state* tensor, if provided, with the updated state of the Markov chain.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable. Also outputs the total number of drawn samples (key: "num_samples").

Return type dict(str, float)

statistics_from_samples(*nn_state*, *samples*)

Estimates the expected value, variance, and the standard error of the observable using the given samples.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState that drew the samples.
- **samples** (*torch.Tensor*) – A batch of sample states to calculate the observable on.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable. Also outputs the total number of drawn samples (key: "num_samples").

Return type dict(str, float)

property symbol

The algebraic symbol representing the Observable.

13.2 Neighbour Interactions

class qucumber.observables.**NeighbourInteraction**(*periodic_bcs=False*, *c=1*)

Bases: *qucumber.observables.ObservableBase*

The $\sigma_i^z \sigma_{i+c}^z$ observable

Computes the c^{th} nearest neighbour interaction for a spin chain with either open or periodic boundary conditions.

Parameters

- **periodic_bcs** (*bool*) – Specifies whether the system has periodic boundary conditions.
- **c** (*int*) – Interaction distance.

apply(*nn_state*, *samples*)

Computes the energy of this neighbour interaction for each sample given a batch of samples.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState that drew the samples.
- **samples** (*torch.Tensor*) – A batch of samples to calculate the observable on. Must be using the $\sigma_i = 0, 1$ convention.

property name

The name of the Observable.

sample(*nn_state*, *k*, *num_samples=1*, *initial_state=None*, *overwrite=False*)

Draws samples of the *observable* using the given NeuralState.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState to draw samples from.
- **k** (*int*) – The number of Gibbs Steps to perform before drawing a sample.
- **num_samples** (*int*) – The number of samples to draw.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chain. If given, *num_samples* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the *initial_state* tensor, if provided, with the updated state of the Markov chain.

Returns The samples drawn through this observable.

Return type torch.Tensor

statistics(*nn_state*, *num_samples*, *num_chains=0*, *burn_in=1000*, *steps=1*, *initial_state=None*, *overwrite=False*)

Estimates the expected value, variance, and the standard error of the observable over the distribution defined by the NeuralState.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState to draw samples from.
- **num_samples** (*int*) – The number of samples to draw. The actual number of samples drawn may be slightly higher if *num_samples % num_chains != 0*.

- **num_chains** (*int*) – The number of Markov chains to run in parallel; if 0 or greater than *num_samples*, will use a number of chains equal to *num_samples*. This is not recommended in the case where a *num_samples* is large, as this may use up all the available memory. Ignored if *initial_state* is provided.
- **burn_in** (*int*) – The number of Gibbs Steps to perform before recording any samples.
- **steps** (*int*) – The number of Gibbs Steps to take between each sample.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chain. If given, *num_chains* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the *initial_state* tensor, if provided, with the updated state of the Markov chain.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable. Also outputs the total number of drawn samples (key: "num_samples").

Return type dict(str, float)

statistics_from_samples(*nn_state*, *samples*)

Estimates the expected value, variance, and the standard error of the observable using the given samples.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState that drew the samples.
- **samples** (*torch.Tensor*) – A batch of sample states to calculate the observable on.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable. Also outputs the total number of drawn samples (key: "num_samples").

Return type dict(str, float)

property symbol

The algebraic symbol representing the Observable.

13.3 Abstract Observable

Note: This is an Abstract Base Class, it is not meant to be used directly. The following API reference is mostly for developers.

class qucumber.observables.**ObservableBase**

Bases: abc.ABC

Base class for observables.

abstract apply(*nn_state*, *samples*)

Computes the value of the local-estimator of the observable O, for a batch of samples $\{\sigma\}$:

$$\mathcal{O}(\sigma) = \sum_{\sigma'} \frac{\rho(\sigma', \sigma)}{\rho(\sigma, \sigma)} O(\sigma, \sigma') = \sum_{\sigma'} \frac{\psi(\sigma')}{\psi(\sigma)} O(\sigma, \sigma')$$

This function must not perform any averaging for statistical purposes, as the proper analysis is delegated to the specialized *statistics* and *statistics_from_samples* methods.

Must be implemented by any subclasses. Refer to the tutorial on Observables to see an example.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState that drew the samples.
- **samples** (*torch.Tensor*) – A batch of sample states to calculate the observable on.

Returns The value of the observable of each given basis state.

Return type torch.Tensor

property name
The name of the Observable.

sample(*nn_state*, *k*, *num_samples=1*, *initial_state=None*, *overwrite=False*)
Draws samples of the *observable* using the given NeuralState.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState to draw samples from.
- **k** (*int*) – The number of Gibbs Steps to perform before drawing a sample.
- **num_samples** (*int*) – The number of samples to draw.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chain. If given, *num_samples* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the *initial_state* tensor, if provided, with the updated state of the Markov chain.

Returns The samples drawn through this observable.

Return type torch.Tensor

statistics(*nn_state*, *num_samples*, *num_chains=0*, *burn_in=1000*, *steps=1*, *initial_state=None*, *overwrite=False*)
Estimates the expected value, variance, and the standard error of the observable over the distribution defined by the NeuralState.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState to draw samples from.
- **num_samples** (*int*) – The number of samples to draw. The actual number of samples drawn may be slightly higher if *num_samples % num_chains != 0*.
- **num_chains** (*int*) – The number of Markov chains to run in parallel; if 0 or greater than *num_samples*, will use a number of chains equal to *num_samples*. This is not recommended in the case where a *num_samples* is large, as this may use up all the available memory. Ignored if *initial_state* is provided.
- **burn_in** (*int*) – The number of Gibbs Steps to perform before recording any samples.
- **steps** (*int*) – The number of Gibbs Steps to take between each sample.
- **initial_state** (*torch.Tensor*) – The initial state of the Markov Chain. If given, *num_chains* will be ignored.
- **overwrite** (*bool*) – Whether to overwrite the *initial_state* tensor, if provided, with the updated state of the Markov chain.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable. Also outputs the total number of drawn samples (key: "num_samples").

Return type dict(str, float)

statistics_from_samples (*nn_state*, *samples*)

Estimates the expected value, variance, and the standard error of the observable using the given samples.

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The NeuralState that drew the samples.
- **samples** (*torch.Tensor*) – A batch of sample states to calculate the observable on.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable. Also outputs the total number of drawn samples (key: "num_samples").

Return type dict(str, float)

property symbol

The algebraic symbol representing the Observable.

CHAPTER

FOURTEEN

TRAINING STATISTICS

qucumber.utils.training_statistics.**KL**(*nn_state*, *target*, *space=None*, *bases=None*, ***kwargs*)

A function for calculating the KL divergence averaged over every given basis.

$$KL(P_{target}|P_{RBM}) = -\sum_{x\in\mathcal{H}} P_{target}(x)\log\left(\frac{P_{RBM}(x)}{P_{target}(x)}\right)$$

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The neural network state.
- **target** (*torch.Tensor or dict(str, torch.Tensor)*) – The true state (wavefunction or density matrix) of the system. Can be a dictionary with each value being the state represented in a different basis, and the key identifying the basis.
- **space** (*torch.Tensor*) – The basis elements of the Hilbert space of the system $\mathcal{H}$. The ordering of the basis elements must match with the ordering of the coefficients given in *target*. If *None*, will generate them using the provided *nn_state*.
- **bases** (*numpy.ndarray*) – An array of unique bases. If given, the KL divergence will be computed for each basis and the average will be returned.
- ****kwargs** – Extra keyword arguments that may be passed. Will be ignored.

Returns The KL divergence.

Return type float

qucumber.utils.training_statistics.**NLL**(*nn_state*, *samples*, *space=None*, *sample_bases=None*, ***kwargs*)

A function for calculating the negative log-likelihood (NLL).

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The neural network state.
- **samples** (*torch.Tensor*) – Samples to compute the NLL on.
- **space** (*torch.Tensor*) – The basis elements of the Hilbert space of the system $\mathcal{H}$. If *None*, will generate them using the provided *nn_state*.
- **sample_bases** (*numpy.ndarray*) – An array of bases where measurements were taken.
- ****kwargs** – Extra keyword arguments that may be passed. Will be ignored.

Returns The Negative Log-Likelihood.

Return type float

qucumber.utils.training_statistics.**fidelity**(*nn_state*, *target*, *space=None*, ***kwargs*)

Calculates the square of the overlap (fidelity) between the reconstructed state and the true state (both in the computational basis).

$$F = |\langle\psi_{RBM}|\psi_{target}\rangle|^2 = \left(\mathrm{tr}[\sqrt{\sqrt{\rho_{RBM}}\rho_{target}\sqrt{\rho_{RBM}}}]\right)^2$$

Parameters

- **nn_state** (qucumber.nn_states.NeuralStateBase) – The neural network state.
- **target** (*torch.Tensor*) – The true state of the system.
- **space** (*torch.Tensor*) – The basis elements of the Hilbert space of the system $\mathcal{H}$. The ordering of the basis elements must match with the ordering of the coefficients given in *target*. If *None*, will generate them using the provided *nn_state*.
- ****kwargs** – Extra keyword arguments that may be passed. Will be ignored.

Returns The fidelity.

Return type float

CHAPTER

FIFTEEN

COMPLEX ALGEBRA

qucumber.utils.cplx.**absolute_value**(*x*)
Returns the complex absolute value elementwise.

Parameters **x** (*torch.Tensor*) – A complex tensor.

Returns A real tensor.

Return type torch.Tensor

qucumber.utils.cplx.**conj**(*x*)
Returns the element-wise complex conjugate of the argument.

Parameters **x** (*torch.Tensor*) – A complex tensor.

Returns The complex conjugate of x.

Return type torch.Tensor

qucumber.utils.cplx.**conjugate**(*x*)
Returns the conjugate transpose of the argument.

In the case of a scalar or vector, only the complex conjugate is taken. In the case of a rank-2 or higher tensor, the complex conjugate is taken, then the first two indices of the tensor are swapped.

Parameters **x** (*torch.Tensor*) – A complex tensor.

Returns The conjugate of x.

Return type torch.Tensor

qucumber.utils.cplx.**einsum**(*equation*, *a*, *b*, *real_part=True*, *imag_part=True*)
Complex-aware version of *einsum*. See the torch documentation for more details.

Parameters

- **equation** (*str*) – The index equation. Passed directly to *torch.einsum*.
- **a** (*torch.Tensor*) – A complex tensor.
- **b** (*torch.Tensor*) – A complex tensor.
- **real_part** (*bool*) – Whether to compute and return the real part of the result.
- **imag_part** (*bool*) – Whether to compute and return the imaginary part of the result.

Returns The Einstein summation of the input tensors performed according to the given equation. If both *real_part* and *imag_part* are true, the result will be a complex tensor, otherwise a real tensor.

Return type torch.Tensor

qucumber.utils.cplx.**elementwise_division**(*x*, *y*)
Element-wise division of x by y.

Parameters

- **x** (*torch.Tensor*) – A complex tensor.
- **y** (*torch.Tensor*) – A complex tensor.

Return type torch.Tensor

qucumber.utils.cplx.**elementwise_mult**(*x*, *y*)
Alias for *scalar_mult()*.

qucumber.utils.cplx.**imag**(*x*)
Returns the imaginary part of a complex tensor.

Parameters **x** (*torch.Tensor*) – The complex tensor

Returns The imaginary part of x; will have one less dimension than x.

Return type torch.Tensor

qucumber.utils.cplx.**inner_prod**(*x*, *y*)
A function that returns the inner product of two complex vectors, x and y (<x|y>).

Parameters

- **x** (*torch.Tensor*) – A complex vector.
- **y** (*torch.Tensor*) – A complex vector.

Raises **ValueError** – If x and y are not complex vectors with their first dimensions being 2, then the function will not execute.

Returns The inner product, $\langle x|y\rangle$.

Return type torch.Tensor

qucumber.utils.cplx.**inverse**(*z*)
Returns the multiplicative inverse of z. Acts elementwise.

Parameters **z** (*torch.Tensor*) – The complex tensor.

Returns 1 / z

Return type torch.Tensor

qucumber.utils.cplx.**kronecker_prod**(*x*, *y*)
Returns the tensor / Kronecker product of 2 complex matrices, x and y.

Parameters

- **x** (*torch.Tensor*) – A complex matrix.
- **y** (*torch.Tensor*) – A complex matrix.

Raises **ValueError** – If x and y do not have 3 dimensions or their first dimension is not 2, the function cannot execute.

Returns The Kronecker product of x and y, $x \otimes y$.

Return type torch.Tensor

qucumber.utils.cplx.**make_complex**(*x*, *y=None*)
A function that creates a torch compatible complex tensor.

Note: x and y must have the same shape.

Parameters

- **x** (*torch.Tensor or numpy.ndarray*) – The real part or a complex numpy array. If a numpy array, will ignore y.
- **y** (*torch.Tensor*) – The imaginary part. Can be *None*, in which case, the resulting complex tensor will have imaginary part equal to zero.

Returns The tensor $[x, y] = x + iy$.

Return type torch.Tensor

qucumber.utils.cplx.**matmul**(*x*, *y*)

A function that computes complex matrix-matrix and matrix-vector products.

Note: If one wishes to do matrix-vector products, the vector must be the second argument (y).

Parameters

- **x** (*torch.Tensor*) – A complex matrix.
- **y** (*torch.Tensor*) – A complex vector or matrix.

Returns The product between x and y.

Return type torch.Tensor

qucumber.utils.cplx.**norm**(*x*)

Returns the norm of the argument.

Parameters **x** (*torch.Tensor*) – A complex scalar.

Returns $|x|$.

Return type torch.Tensor

qucumber.utils.cplx.**norm_sqr**(*x*)

Returns the squared norm of the argument.

Parameters **x** (*torch.Tensor*) – A complex scalar.

Returns $|x|^2$.

Return type torch.Tensor

qucumber.utils.cplx.**numpy**(*x*)

Converts a complex torch tensor into a numpy array

Parameters **x** (*torch.Tensor*) – The tensor to convert.

Returns A complex numpy array containing the data from x.

Return type numpy.ndarray

qucumber.utils.cplx.**outer_prod**(*x*, *y*)

A function that returns the outer product of two complex vectors, x and y.

Parameters

- **x** (*torch.Tensor*) – A complex vector.
- **y** (*torch.Tensor*) – A complex vector.

Raises **ValueError** – If x and y are not complex vectors with their first dimensions being 2, then an error will be raised.

Returns The outer product between x and y, $|x\rangle\langle y|$.

Return type torch.Tensor

qucumber.utils.cplx.**real**(*x*)

Returns the real part of a complex tensor.

Parameters **x** (*torch.Tensor*) – The complex tensor

Returns The real part of *x*; will have one less dimension than *x*.

Return type torch.Tensor

qucumber.utils.cplx.**scalar_divide**(*x*, *y*)

Divides *x* by *y*. If *x* and *y* have the same shape, then acts elementwise. If *y* is a complex scalar, then performs a scalar division.

Parameters

- **x** (*torch.Tensor*) – The numerator (a complex tensor).
- **y** (*torch.Tensor*) – The denominator (a complex tensor).

Returns x / y

Return type torch.Tensor

qucumber.utils.cplx.**scalar_mult**(*x*, *y*, *out=None*)

A function that computes the product between complex matrices and scalars, complex vectors and scalars or two complex scalars.

Parameters

- **x** (*torch.Tensor*) – A complex scalar, vector or matrix.
- **y** (*torch.Tensor*) – A complex scalar, vector or matrix.

Returns The product between *x* and *y*. Either overwrites *out*, or returns a new tensor.

Return type torch.Tensor

qucumber.utils.cplx.**sigmoid**(*x*, *y*)

Returns the sigmoid function of a complex number. Acts elementwise.

Parameters

- **x** (*torch.Tensor*) – The real part of the complex number
- **y** (*torch.Tensor*) – The imaginary part of the complex number

Returns The complex sigmoid of $x + iy$

Return type torch.Tensor

CHAPTER

SIXTEEN

DATA HANDLING

qucumber.utils.data.**extract_refbasis_samples**(*train_samples*, *train_bases*)

Extract the reference basis samples from the data.

Parameters

- **train_samples** (*torch.Tensor*) – The training samples.
- **train_bases** (*numpy.ndarray*) – The bases of the training samples.

Returns The samples in the data that are only in the reference basis.

Return type torch.Tensor

qucumber.utils.data.**load_data**(*tr_samples_path*, *tr_psi_path=None*, *tr_bases_path=None*, *bases_path=None*)

Load the data required for training.

Parameters

- **tr_samples_path** (*str*) – The path to the training data.
- **tr_psi_path** (*str*) – The path to the target/true wavefunction.
- **tr_bases_path** (*str*) – The path to the basis data.
- **bases_path** (*str*) – The path to a file containing all possible bases used in the tr_bases_path file.

Returns A list of all input parameters.

Return type list

qucumber.utils.data.**load_data_DM**(*tr_samples_path*, *tr_mtx_real_path=None*, *tr_mtx_imag_path=None*, *tr_bases_path=None*, *bases_path=None*)

Load the data required for training.

Parameters

- **tr_samples_path** (*str*) – The path to the training data.
- **tr_mtx_real_path** (*str*) – The path to the real part of the density matrix
- **tr_mtx_imag_path** (*str*) – The path to the imaginary part of the density matrix
- **tr_bases_path** (*str*) – The path to the basis data.
- **bases_path** (*str*) – The path to a file containing all possible bases used in the tr_bases_path file.

Returns A list of all input parameters, with the real and imaginary parts of the target density matrix (if provided) combined into one complex matrix.

Return type list

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

q

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

V

W