QuCumber Documentation

Release v1.1.0

PIQuIL

2019-05-17

INTRODUCTION

ONE

INSTALLATION

QuCumber only supports Python 3, not Python 2. If you are using Python 2, please update! You may also want to install PyTorch v1.0 (https://pytorch.org/), if you have not already.

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

1.1 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
```

1.2 Windows

Navigate to the directory (through command prompt) where pip.exe is installed (usually `C:\Python\Scripts\pip.exe`) and type:

```
pip.exe install qucumber
```

1.3 Linux / macOS

Open up a terminal, then type:

```
pip install qucumber
```

THEORY

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

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: launch binder , though they may be slow.

RECONSTRUCTION OF A POSITIVE-REAL WAVEFUNCTION

In this tutorial, a walkthrough of how to reconstruct a **positive-real** wavefunction via training a *Restricted Boltzmann Machine* (RBM), the neural network behind QuCumber, will be presented. The data used for training will be 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 of 10,000 σ^z measurements from a one-dimensional transverse-field Ising model (TFIM) with 10 sites at its critical point. The Hamiltonian for the transverse-field Ising model (TFIM) is given by

$$\mathcal{H} = -J \sum_i \sigma_i^z \sigma_{i+1}^z - h \sum_i \sigma_i^x \tag{4.1}$$

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

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, the fidelity between the true ground-state 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). It will also be shown that 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. These two quantities equal will be kept equal.

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

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

By default, QuCumber will attempt to run on a GPU if one is available (if one is not available, QuCumber will default to CPU). If one wishes to run QuCumber on a CPU, add the flag `gpu=False` in the `PositiveWaveFunction` object instantiation (i.e. uncomment the line above).

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 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   # pos_batch_size
     nbs = 200   # neg_batch_size
     lr = 0.01
     k = 10
```

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

1. `period`: the frequency of the training evaluators being calculated is controlled by the `period` argument (e.g. `period=200` means that the `MetricEvaluator` will update the user 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 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, the function to be passed to the `MetricEvaluator` will be the fifth coefficient of the reconstructed wavefunction multiplied by a parameter, A.

```python
[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 and the dictionary of functions the user would like to compute every `period` epochs can be given to the `MetricEvaluator`.

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

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`).

```python
[7]: callbacks = [
         MetricEvaluator(
             period,
             {"Fidelity": ts.fidelity, "KL": ts.KL, "A_rbm_5": psi_coefficient},
             target_psi=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,
     )
Epoch: 10        Fidelity = 0.534941     KL = 1.260048    A_rbm_5 = 0.116531
Epoch: 20        Fidelity = 0.636866     KL = 0.857609    A_rbm_5 = 0.169683
Epoch: 30        Fidelity = 0.736682     KL = 0.575945    A_rbm_5 = 0.208133
Epoch: 40        Fidelity = 0.794755     KL = 0.440940    A_rbm_5 = 0.226640
```

(continues on next page)

```
Epoch: 50      Fidelity = 0.831520    KL = 0.353924    A_rbm_5 = 0.234840
Epoch: 60      Fidelity = 0.862574    KL = 0.282083    A_rbm_5 = 0.238088
Epoch: 70      Fidelity = 0.885912    KL = 0.233535    A_rbm_5 = 0.242870
Epoch: 80      Fidelity = 0.900656    KL = 0.203658    A_rbm_5 = 0.237143
Epoch: 90      Fidelity = 0.911208    KL = 0.182475    A_rbm_5 = 0.237343
Epoch: 100     Fidelity = 0.918912    KL = 0.166443    A_rbm_5 = 0.231324
Epoch: 110     Fidelity = 0.926300    KL = 0.151879    A_rbm_5 = 0.240801
Epoch: 120     Fidelity = 0.932397    KL = 0.137685    A_rbm_5 = 0.230534
Epoch: 130     Fidelity = 0.938690    KL = 0.124218    A_rbm_5 = 0.231577
Epoch: 140     Fidelity = 0.944510    KL = 0.111655    A_rbm_5 = 0.225222
Epoch: 150     Fidelity = 0.949794    KL = 0.100720    A_rbm_5 = 0.227332
Epoch: 160     Fidelity = 0.954023    KL = 0.092029    A_rbm_5 = 0.222866
Epoch: 170     Fidelity = 0.958097    KL = 0.084130    A_rbm_5 = 0.226016
Epoch: 180     Fidelity = 0.961439    KL = 0.077638    A_rbm_5 = 0.228211
Epoch: 190     Fidelity = 0.963652    KL = 0.073456    A_rbm_5 = 0.224256
Epoch: 200     Fidelity = 0.965993    KL = 0.068942    A_rbm_5 = 0.221013
Epoch: 210     Fidelity = 0.967833    KL = 0.065555    A_rbm_5 = 0.220742
Epoch: 220     Fidelity = 0.969344    KL = 0.062550    A_rbm_5 = 0.219263
Epoch: 230     Fidelity = 0.970622    KL = 0.059982    A_rbm_5 = 0.220068
Epoch: 240     Fidelity = 0.971988    KL = 0.057298    A_rbm_5 = 0.217463
Epoch: 250     Fidelity = 0.973524    KL = 0.054281    A_rbm_5 = 0.221264
Epoch: 260     Fidelity = 0.974584    KL = 0.052146    A_rbm_5 = 0.218502
Epoch: 270     Fidelity = 0.975617    KL = 0.049899    A_rbm_5 = 0.217135
Epoch: 280     Fidelity = 0.976913    KL = 0.047288    A_rbm_5 = 0.218298
Epoch: 290     Fidelity = 0.978150    KL = 0.044736    A_rbm_5 = 0.214478
Epoch: 300     Fidelity = 0.979212    KL = 0.042611    A_rbm_5 = 0.219678
Epoch: 310     Fidelity = 0.980419    KL = 0.040171    A_rbm_5 = 0.220057
Epoch: 320     Fidelity = 0.981252    KL = 0.038439    A_rbm_5 = 0.219286
Epoch: 330     Fidelity = 0.982365    KL = 0.036267    A_rbm_5 = 0.221942
Epoch: 340     Fidelity = 0.982638    KL = 0.035651    A_rbm_5 = 0.217697
Epoch: 350     Fidelity = 0.983759    KL = 0.033440    A_rbm_5 = 0.220134
Epoch: 360     Fidelity = 0.984440    KL = 0.032067    A_rbm_5 = 0.219768
Epoch: 370     Fidelity = 0.985045    KL = 0.030901    A_rbm_5 = 0.217321
Epoch: 380     Fidelity = 0.985576    KL = 0.029898    A_rbm_5 = 0.222066
Epoch: 390     Fidelity = 0.985822    KL = 0.029499    A_rbm_5 = 0.220483
Epoch: 400     Fidelity = 0.986610    KL = 0.027838    A_rbm_5 = 0.216224
Epoch: 410     Fidelity = 0.986830    KL = 0.027407    A_rbm_5 = 0.219407
Epoch: 420     Fidelity = 0.987248    KL = 0.026607    A_rbm_5 = 0.218464
Epoch: 430     Fidelity = 0.987026    KL = 0.027048    A_rbm_5 = 0.222123
Epoch: 440     Fidelity = 0.987700    KL = 0.025744    A_rbm_5 = 0.220681
Epoch: 450     Fidelity = 0.988019    KL = 0.025160    A_rbm_5 = 0.219184
Epoch: 460     Fidelity = 0.988265    KL = 0.024712    A_rbm_5 = 0.219010
Epoch: 470     Fidelity = 0.988460    KL = 0.024316    A_rbm_5 = 0.214715
Epoch: 480     Fidelity = 0.988744    KL = 0.023759    A_rbm_5 = 0.214839
Epoch: 490     Fidelity = 0.988667    KL = 0.023881    A_rbm_5 = 0.218261
Epoch: 500     Fidelity = 0.988650    KL = 0.024005    A_rbm_5 = 0.210195
```

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).

```
[8]:  # Note that the key given to the *MetricEvaluator* must be
      # what comes after callbacks[0].
      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.savefig("fid_KL.pdf")
      plt.show()
```
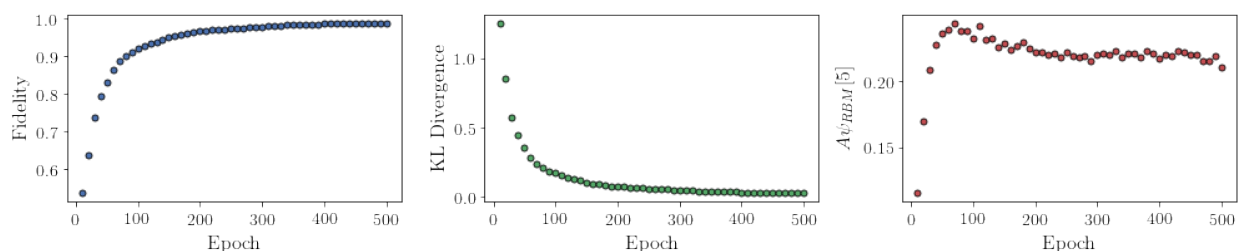

It should be noted that one could have just ran `nn_state.fit(train_samples)` and just used the default hyperparameters and no training evaluators.

4.2. Using QuCumber to reconstruct the wavefunction 11

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 and uncomment the one below. The default hyperparameters will be used instead. Using the non-default hyperparameters yielded a fidelity of approximately 0.994, while the default hyperparameters yielded a fidelity of approximately 0.523!

The RBM's parameters will also be saved for future use in other tutorials. They can be saved to a pickle file with the name `saved_params.pt` with the code below.

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

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

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.

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle \tag{5.1}$$

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

$$\alpha = 0.2861 + 0.0539i \tag{5.2}$$
$$\beta = 0.3687 - 0.3023i \tag{5.3}$$
$$\gamma = -0.1672 - 0.3529i \tag{5.4}$$
$$\delta = -0.5659 - 0.4639i. \tag{5.5}$$

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)

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

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 are yielded 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. Currently, PyTorch does not support complex numbers.

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.

```python
[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.

```python
[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 = nv

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

By 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. Set `gpu=True` in the line above to run this tutorial on a GPU.

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 = 100
     pbs = 50  # pos_batch_size
     nbs = 50  # neg_batch_size
     lr = 0.1
     k = 5
```

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.compute_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.compute_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.compute_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.compute_normalization(space).sqrt_()
         delta_ = cplx.norm(
             torch.tensor([rbm_psi[0][3], rbm_psi[1][3]], device=nn_state.device)
             / normalization
         )

         return delta_
```

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

```
[7]: period = 2
     space = nn_state.generate_hilbert_space(nv)

     callbacks = [
         MetricEvaluator(
             period,
             {
                 "Fidelity": ts.fidelity,
                 "KL": ts.KL,
                 "norm": alpha,
                 # "norm": beta,
                 # "norm": gamma,
                 # "norm": delta,
             },
             target_psi=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,
     )
```

```
Epoch: 2      Fidelity = 0.623747    KL = 0.226386    norm = 0.272518
Epoch: 4      Fidelity = 0.744691    KL = 0.142639    norm = 0.248872
Epoch: 6      Fidelity = 0.818254    KL = 0.094584    norm = 0.263589
Epoch: 8      Fidelity = 0.867098    KL = 0.067506    norm = 0.278453
Epoch: 10     Fidelity = 0.900217    KL = 0.051592    norm = 0.281094
Epoch: 12     Fidelity = 0.922993    KL = 0.041311    norm = 0.276052
Epoch: 14     Fidelity = 0.937807    KL = 0.034972    norm = 0.274676
Epoch: 16     Fidelity = 0.947232    KL = 0.030543    norm = 0.283873
Epoch: 18     Fidelity = 0.955277    KL = 0.027313    norm = 0.278906
Epoch: 20     Fidelity = 0.959930    KL = 0.025034    norm = 0.290271
Epoch: 22     Fidelity = 0.963333    KL = 0.023719    norm = 0.296183
Epoch: 24     Fidelity = 0.969419    KL = 0.021086    norm = 0.276108
Epoch: 26     Fidelity = 0.972300    KL = 0.020200    norm = 0.290305
Epoch: 28     Fidelity = 0.974777    KL = 0.018635    norm = 0.284231
Epoch: 30     Fidelity = 0.976208    KL = 0.017865    norm = 0.282036
Epoch: 32     Fidelity = 0.978382    KL = 0.016862    norm = 0.282498
Epoch: 34     Fidelity = 0.980578    KL = 0.015977    norm = 0.279435
Epoch: 36     Fidelity = 0.980983    KL = 0.015545    norm = 0.277835
Epoch: 38     Fidelity = 0.982651    KL = 0.014751    norm = 0.280070
Epoch: 40     Fidelity = 0.983155    KL = 0.014353    norm = 0.276912
Epoch: 42     Fidelity = 0.983996    KL = 0.013827    norm = 0.278844
Epoch: 44     Fidelity = 0.982731    KL = 0.015100    norm = 0.305219
Epoch: 46     Fidelity = 0.984791    KL = 0.013417    norm = 0.293674
```

(continues on next page)

```
Epoch: 48       Fidelity = 0.985395     KL = 0.012845     norm = 0.280658
Epoch: 50       Fidelity = 0.986767     KL = 0.012093     norm = 0.277599
Epoch: 52       Fidelity = 0.987795     KL = 0.011650     norm = 0.278886
Epoch: 54       Fidelity = 0.987057     KL = 0.011843     norm = 0.271735
Epoch: 56       Fidelity = 0.987125     KL = 0.011552     norm = 0.280304
Epoch: 58       Fidelity = 0.987295     KL = 0.011382     norm = 0.288229
Epoch: 60       Fidelity = 0.988201     KL = 0.011201     norm = 0.266736
Epoch: 62       Fidelity = 0.989181     KL = 0.010504     norm = 0.288520
Epoch: 64       Fidelity = 0.989308     KL = 0.010293     norm = 0.292218
Epoch: 66       Fidelity = 0.989321     KL = 0.009901     norm = 0.282069
Epoch: 68       Fidelity = 0.989347     KL = 0.009836     norm = 0.275723
Epoch: 70       Fidelity = 0.989494     KL = 0.009838     norm = 0.293840
Epoch: 72       Fidelity = 0.990115     KL = 0.009225     norm = 0.282556
Epoch: 74       Fidelity = 0.990199     KL = 0.009095     norm = 0.278911
Epoch: 76       Fidelity = 0.989979     KL = 0.009214     norm = 0.273241
Epoch: 78       Fidelity = 0.989633     KL = 0.009275     norm = 0.274384
Epoch: 80       Fidelity = 0.989972     KL = 0.008976     norm = 0.275430
Epoch: 82       Fidelity = 0.989920     KL = 0.008871     norm = 0.285605
Epoch: 84       Fidelity = 0.991177     KL = 0.008183     norm = 0.282607
Epoch: 86       Fidelity = 0.991249     KL = 0.008095     norm = 0.276934
Epoch: 88       Fidelity = 0.990857     KL = 0.008273     norm = 0.272151
Epoch: 90       Fidelity = 0.990802     KL = 0.008071     norm = 0.280823
Epoch: 92       Fidelity = 0.991090     KL = 0.007838     norm = 0.279963
Epoch: 94       Fidelity = 0.990995     KL = 0.007861     norm = 0.275772
Epoch: 96       Fidelity = 0.990326     KL = 0.008202     norm = 0.289882
Epoch: 98       Fidelity = 0.991012     KL = 0.007690     norm = 0.277037
Epoch: 100      Fidelity = 0.991736     KL = 0.007292     norm = 0.275516
```

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]
      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.savefig("complex_fid_KL.pdf")
      plt.show()
```
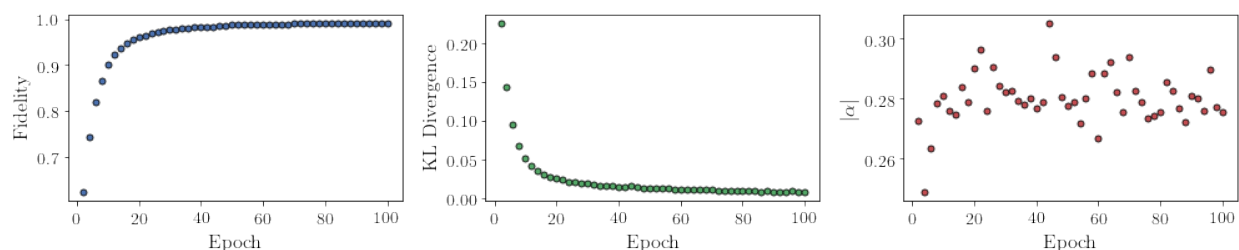

It should be noted that one could have just ran `nn_state.fit(train_samples)` and just used the default hyperparameters and no training evaluators.

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 with the following keys: "weights", "visible_bias", "hidden_bias".

5.2. Using qucumber to reconstruct the wavefunction 19

SAMPLING AND CALCULATING OBSERVABLES

6.1 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 if how to train an RBM using QuCumber is unclear. An RBM with 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 RBM parameters.

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

     from qucumber.nn_states import PositiveWaveFunction

     from qucumber.observables import ObservableBase

     from quantum_ising_chain import TFIMChainEnergy, Convergence

     nn_state = PositiveWaveFunction.autoload("saved_params.pt")
```

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

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

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

6.1.1 Magnetization

With the newly generated samples, the user can now easliy calculate observables that do not require any details associated with the RBM. A great example of this is the magnetization. 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 = 0.55246
```

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 RBM 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 RBM parameters.

6.2 Calculate an observable using the *Observable* module

6.2.1 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': 0.5524600000000002,
      'variance': 0.09724167256725606,
      'std_error': 0.0031183597061156376}
```

With this function we get the variance and standard error for free. Now you may be asking: "That isn't 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 1.77 s, sys: 5.24 ms, total: 1.78 s
Wall time: 582 ms
```

```
[8]: {'mean': 0.5486800000000001,
      'variance': 0.10007226482647404,
      'std_error': 0.0031634200610490227}
```

Let's consider what is taking place under the hood at the moment. The `statistics` function is drawing 10000 samples from the given RBM 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 RBM, 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)

The `statistics` function will also return a dictionary containing the mean, standard error (of the mean) and the variance with the keys "mean", "std_error" and "variance", respectively.

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

```
CPU times: user 292 ms, sys: 0 ns, total: 292 ms
Wall time: 75.9 ms
```

```
[9]: {'mean': 0.548,
      'variance': 0.09837783778377833,
      'std_error': 0.0031365241555546537}
```

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

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

```
[10]: {'mean': 0.5508113799999957,
       'variance': 0.09800052546254845,
       'std_error': 9.899521476442609e-05}
```

Note how much we've decreased 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:

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

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

6.2. Calculate an observable using the *Observable* module 23

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

6.2.2 TFIM Energy

Some observables cannot be computed directly from samples, but instead depend on the RBM 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 RBM 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 RBM parameters are from the first tutorial, where the example data was from the TFIM with 10 sites at its critical point ($\frac{h}{J} = 1$).

```
[12]: 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.

```
[13]: 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: -1.2347 +/- 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 RBM. 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.

```
[14]: 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")
```

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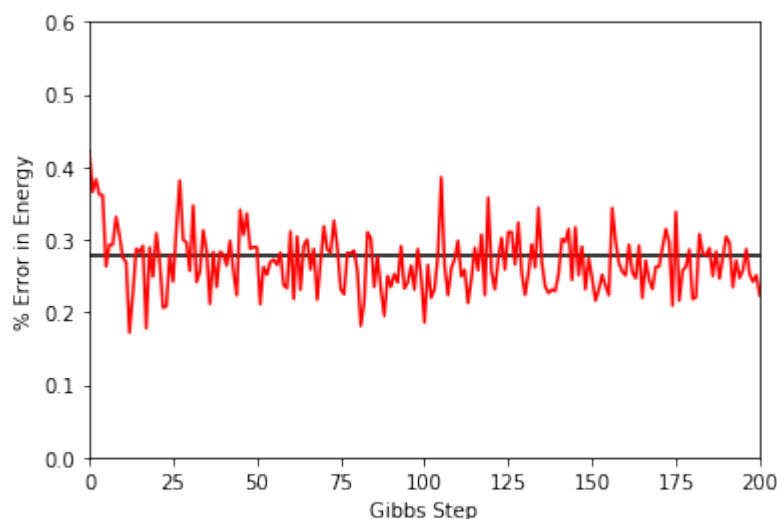

One can see a brief transient period in the magnetization observable, before the state of the machine "warms up" to equilibrium (this explains the `burn_in` argument we saw earlier). After that, the values fluctuate around the estimated mean (the horizontal black line).

6.2.3 Adding 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:

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

The TFIM chain we trained the RBM on did not have periodic boundary conditions, so `periodic_bcs=False`. Meanwhile, `c` specifies the 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.

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

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

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

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

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

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

```
[19]: 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: -1.2347 +/- 0.0005
Variance: 0.0022
```

The statistics above match with those computed earlier.

6.2.4 Renyi Entropy and the Swap operator

We can estimate the second Renyi Entropy using the Swap operator as shown by Hastings et al. (2010). The 2nd Renyi 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 Renyi entropy.

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

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

```
[21]: 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: 0.7937 +/- 0.0083
Variance: 0.3484
```

The 2nd Renyi 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.

```
[22]: 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: 0.2310 +/- 0.0105
```

6.2.5 Custom observable

QuCumber has a built-in module called `Observable` which makes it easy for the user to compute any arbitrary observable from the RBM. To see the the `Observable` module in action, an example 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.

```
[23]: 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):
              samples = to_pm1(samples)
              interaction_ = 0.0
              for i in range(samples.shape[-1]):
                  if (i + 3) > samples.shape[-1]:
                      continue
                  else:
                      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 RBM (`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:

```
[24]: help(ObservableBase.apply)

      Help on function apply in module qucumber.observables.observable:

      apply(self, nn_state, samples)
          Computes the value of the observable, row-wise, on a batch of
          samples.

          If we think of the samples given as a set of projective measurements
          in a given computational basis, this method must return the expectation
          of the operator with respect to each basis state in `samples`.
          It 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.

          :param nn_state: The WaveFunction that drew the samples.
          :type nn_state: qucumber.nn_states.WaveFunctionBase
          :param samples: A batch of sample states to calculate the observable on.
          :type samples: torch.Tensor
          :returns: The value of the observable of each given basis state.
          :rtype: torch.Tensor
```

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 RBM 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: 0.1754 +/- 0.0015
Variance: 0.0239
```

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
```

6.3 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.

```
[28]: 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:

```
[29]: piquil
```

```
[29]: PIQuIL
```

```
[30]: tfim
```

```
[30]: ((-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:

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

```
[31]: 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.

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

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

```
{'PIQuIL': {'mean': 0.1754200000000003,
            'std_error': 0.0015460340818165665,
            'variance': 0.02390221382138394},
 'SWAP': {'mean': 0.7937077159650355,
          'std_error': 0.008347777852987842,
          'variance': 0.3484269754141716},
 'SigmaZ': {'mean': 0.5524600000000002,
            'std_error': 0.0031183597061156376,
            'variance': 0.09724167256725606},
 'TFIM': {'mean': -1.234660072325191,
          'std_error': 0.0004716346213696688,
          'variance': 0.0022243921607451086}}
```

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

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

```
{'PIQuIL': {'mean': 0.17418,
            'std_error': 0.0015624906072324945,
            'variance': 0.02441376897689769},
 'SWAP': {'mean': 0.7977556285445141,
          'std_error': 0.006037075970673986,
          'variance': 0.3644628627568925},
 'SigmaZ': {'mean': 0.55228,
            'std_error': 0.0031312075316178864,
            'variance': 0.09804460606060576},
 'TFIM': {'mean': -1.235210919302773,
          'std_error': 0.000462796110346464,
          'variance': 0.0021418023975181646}}
CPU times: user 913 ms, sys: 0 ns, total: 913 ms
Wall time: 232 ms
```

Compare this to computing these statistics on each observable individually:

```
[35]: %%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': 0.17683000000000001,
            'std_error': 0.0015656501001256947,
            'variance': 0.024512602360235978},
 'SWAP': {'mean': 0.7894748061589013,
```

(continues on next page)

```
            'std_error': 0.005957763454588121,
            'variance': 0.3549494538082578},
 'SigmaZ': {'mean': 0.557,
            'std_error': 0.003132424174360939,
            'variance': 0.09812081208120808},
 'TFIM': {'mean': -1.2349736426229931,
          'std_error': 0.00047315568939076296,
          'variance': 0.0022387630640284817}}
CPU times: user 1.96 s, sys: 3.97 ms, total: 1.96 s
Wall time: 493 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.

6.3.1 Template for your custom observable

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

```python
[36]: import torch
      from qucumber.observables import ObservableBase


      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)

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

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.

```python
[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.utils.data as data

from quantum_ising_chain import TFIMChainEnergy
```

```python
[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)

epochs = 1000
pbs = 100   # pos_batch_size
nbs = 200   # neg_batch_size
lr = 0.01
k = 10

period = 100
```

(continues on next page)

```
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: -1.193770     variance: 0.024622     std_error: 0.001569
Epoch: 200
  TFIMChainEnergy:
```

```
       mean: -1.215802      variance: 0.013568      std_error: 0.001165
Epoch: 300
  TFIMChainEnergy:
    mean: -1.221930      variance: 0.009081      std_error: 0.000953
Epoch: 400
  TFIMChainEnergy:
    mean: -1.227180      variance: 0.006347      std_error: 0.000797
Epoch: 500
  TFIMChainEnergy:
    mean: -1.230074      variance: 0.004502      std_error: 0.000671
Epoch: 600
  TFIMChainEnergy:
    mean: -1.232001      variance: 0.003641      std_error: 0.000603
Epoch: 700
  TFIMChainEnergy:
    mean: -1.233434      variance: 0.002839      std_error: 0.000533
Epoch: 800
  TFIMChainEnergy:
    mean: -1.235324      variance: 0.002306      std_error: 0.000480
Epoch: 900
  TFIMChainEnergy:
    mean: -1.235313      variance: 0.001936      std_error: 0.000440
Epoch: 1000
  TFIMChainEnergy:
    mean: -1.235257      variance: 0.001590      std_error: 0.000399
```

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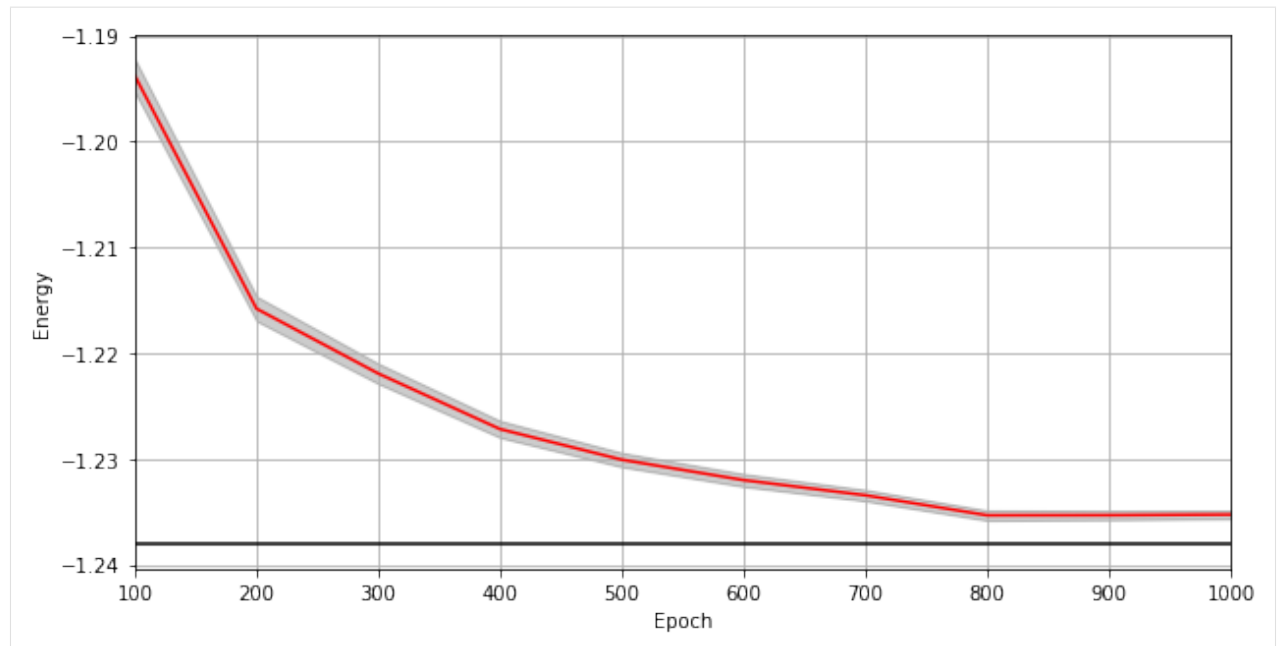
RBM

class `qucumber.rbm.`**BinaryRBM**(*num_visible*, *num_hidden*, *zero_weights=False*, *gpu=True*)
   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*)
      The gradients of the effective energies for the given visible states.

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

      Returns 1d vector 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 Chain. If given, *num_samples* will be ignored.

         • **overwrite** (`bool`) – Whether to overwrite the initial_state tensor, if it is provided.

   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

QUANTUM STATES

9.1 Positive WaveFunction

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

    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.

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=False*)

    Initializes a WaveFunction 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 WaveFunction initialized from the given parameters. The returned WaveFunction will be of whichever type this function was called on.

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

    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.

Returns List containing the gradients of the parameters.

Return type list

compute_normalization (*space*)
   Compute the normalization constant of the wavefunction.

$$Z_\lambda = \sqrt{\sum_\sigma |\psi_\lambda|^2} = \sqrt{\sum_\sigma p_\lambda(\sigma)}$$

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

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*, ***kwargs*)
   Train the WaveFunction.

Parameters

- **data** (`np.array`) – 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

- **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.

- **kwargs** – Keyword arguments to pass to the optimizer

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*)

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
>
> **Returns** A single tensor containing all of the parameter gradients.
>
> **Return type** torch.Tensor

load(*location*)

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

Note: The WaveFunction 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 WaveFunction parameters from.

max_size

Maximum size of the Hilbert space for full enumeration

networks

A list of the names of the internal RBMs.

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

probability(*v, Z*)

Evaluates the probability of the given vector(s) of visible states.

> **Parameters**
>
> - **v** (`torch.Tensor`) – The visible states.
>
> - **Z** (`float`) – The partition function.
>
> **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

---

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 Chain. 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 WaveFunction 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 WaveFunction parameters.

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

9.2 Complex WaveFunction

class qucumber.nn_states.**ComplexWaveFunction**(*num_visible*, *num_hidden=None*, *unitary_dict=None*, *gpu=True*)
    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.

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 WaveFunction 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 WaveFunction initialized from the given parameters. The returned WaveFunction will be of whichever type this function was called on.

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** (`np.array`) – Batch of the input bases corresponding to the samples in *samples_batch*.

Returns List containing the gradients of the parameters.

Return type list

compute_normalization(*space*)

Compute the normalization constant of the wavefunction.

$$Z_\lambda = \sqrt{\sum_\sigma |\psi_{\lambda\mu}|^2} = \sqrt{\sum_\sigma p_\lambda(\sigma)}$$

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

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*, ***kwargs*)
    Train the WaveFunction.

    Parameters

- **data** (`np.array`) – 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** (`np.array`) – The measurement bases for each sample. Must be provided if training a ComplexWaveFunction.

- **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.

- **kwargs** – Keyword arguments to pass to the optimizer

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 (*basis*, *sample*)
    Compute the gradient of a sample, measured in different bases.

    Parameters

- **basis** (`np.array`) – A set of bases.

- **sample** (`np.array`) – A sample to compute the gradient of.

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

    Return type list[torch.Tensor]

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

Note: The WaveFunction 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 WaveFunction parameters from.

max_size
   Maximum size of the Hilbert space for full enumeration

networks
   A list of the names of the internal RBMs.

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

probability(*v*, *Z*)
   Evaluates the probability of the given vector(s) of visible states.

   Parameters

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

   - **Z** (*float*) – The partition function.

   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

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

rbm_ph
   RBM used to learn the wavefunction phase.

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 Chain. 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 WaveFunction 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 WaveFunction parameters.

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

9.3 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: `abc.ABC`

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

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

Initializes a WaveFunction 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 WaveFunction initialized from the given parameters. The returned WaveFunction will be of whichever type this function was called on.

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** (`np.array`) – Batch of the input bases corresponding to the samples in *samples_batch*.

Returns List containing the gradients of the parameters.

Return type list

compute_normalization(*space*)

Compute the normalization constant of the wavefunction.

$$Z_\lambda = \sqrt{\sum_\sigma |\psi_{\lambda\mu}|^2} = \sqrt{\sum_\sigma p_\lambda(\sigma)}$$

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

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*, ***kwargs*)

Train the WaveFunction.

Parameters

- **data** (`np.array`) – 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** (*np.array*) – The measurement bases for each sample. Must be provided if training a ComplexWaveFunction.

- **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.

- **kwargs** – Keyword arguments to pass to the optimizer

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 ()
  Compute the gradient of a set of samples.

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

  Note: The WaveFunction 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 WaveFunction parameters from.

max_size
  Maximum size of the Hilbert space for full enumeration

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

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

probability(*v*, *Z*)
> Evaluates the probability of the given vector(s) of visible states.

> **Parameters**
>
> * **v** (`torch.Tensor`) – The visible states.
>
> * **Z** (`float`) – The partition function.
>
> **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.

$$\psi(\sigma)$$

> **Parameters** **v** (`torch.Tensor`) – visible states σ
>
> **Returns** Complex object containing the value of the wavefunction for each visible state
>
> **Return type** torch.Tensor

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 Chain. 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 WaveFunction 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 WaveFunction parameters.

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

CALLBACKS

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 Wave-
> > Function being trained.

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

> > **Parameters nn_state** (`qucumber.nn_states.WaveFunctionBase`) – The Wave-
> > Function being trained.

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`.

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.

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*.

class qucumber.callbacks.**EarlyStopping**(*period*, *tolerance*, *patience*, *evaluator_callback*, *quantity_name*)

Stop training once the model stops improving. The specific criterion for stopping is:

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

where M_t is the metric value at the current evaluation (time t), p is the "patience" parameter, 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*.

class qucumber.callbacks.**VarianceBasedEarlyStopping**(*period*, *tolerance*, *patience*, *evaluator_callback*, *quantity_name*, *variance_name*)

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 variance 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/obserable stored in *evaluator_callback*.
- **variance_name** (*str*) – The name of the variance stored in *evaluator_callback*.

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 WaveFunction 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** np.array

__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.

epochs

Return a list of all epochs that have been recorded.

> **Return type** np.array

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.

names
> The names of the tracked metrics.
>
> > **Return type** list[str]

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 WaveFunction. 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.

> **epochs**
> Return a list of all epochs that have been recorded.
>
> > **Return type** np.array

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.

names
> The names of the tracked observables.

> **Return type** list[str]

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*.

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.

ELEVEN

OBSERVABLES

11.1 Pauli Operators

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

Bases: *qucumber.observables.ObservableBase*

The σ_z observable.

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

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

Parameters

- **nn_state** (*qucumber.nn_states.WaveFunctionBase*) – The WaveFunction 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.

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 WaveFunction.

Parameters

- **nn_state** (*qucumber.nn_states.WaveFunctionBase*) – The WaveFunction 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*)
    Estimates the expected value, variance, and the standard error of the observable over the distribution defined by the WaveFunction.

    Parameters

- **nn_state** (`qucumber.nn_states.WaveFunctionBase`) – The WaveFunction 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.

- **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.

    Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable.

    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.WaveFunctionBase`) – The WaveFunction 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.

    Return type [dict](str, float)

symbol
    The algebraic symbol representing the Observable.

class qucumber.observables.**SigmaX**(*absolute=False*)
    Bases: `qucumber.observables.ObservableBase`

    The σ_x observable

    Computes the 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 magnetization along X of each sample in the given batch of samples.

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

    Parameters

- **nn_state** (`qucumber.nn_states.WaveFunctionBase`) – The WaveFunction 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.

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 WaveFunction.

    Parameters

    • **nn_state** (`qucumber.nn_states.WaveFunctionBase`) – The WaveFunction
      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*)
    Estimates the expected value, variance, and the standard error of the observable over the distribution de-
    fined by the WaveFunction.

    Parameters

    • **nn_state** (`qucumber.nn_states.WaveFunctionBase`) – The WaveFunction
      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 rec-
      ommended in the case where a *num_samples* is large, as this may use up all the available
      memory.

    • **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.

    Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key:
    "variance"), and standard error (key: "std_error") of the observable.

    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.WaveFunctionBase`) – The WaveFunction
      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.

    Return type dict(str, float)

> **symbol**
>> The algebraic symbol representing the Observable.

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

> Bases: `qucumber.observables.ObservableBase`

> The σ_y observable

> Computes the 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 magnetization along Y of each sample in the given batch of samples.

>> Assumes that the computational basis that the WaveFunction was trained on was the Z basis.

>> **Parameters**

>>> • **nn_state** ([`qucumber.nn_states.WaveFunctionBase`](#)) – The WaveFunction 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.

> **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 WaveFunction.

>> **Parameters**

>>> • **nn_state** ([`qucumber.nn_states.WaveFunctionBase`](#)) – The WaveFunction 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*)

>> Estimates the expected value, variance, and the standard error of the observable over the distribution defined by the WaveFunction.

>> **Parameters**

>>> • **nn_state** ([`qucumber.nn_states.WaveFunctionBase`](#)) – The WaveFunction 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.

- **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.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable.

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.WaveFunctionBase*) – The WaveFunction 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.

Return type dict(str, float)

symbol
The algebraic symbol representing the Observable.

11.2 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.WaveFunctionBase*) – The WaveFunction 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.

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 WaveFunction.

Parameters

- **nn_state** (*qucumber.nn_states.WaveFunctionBase*) – The WaveFunction 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*)

Estimates the expected value, variance, and the standard error of the observable over the distribution defined by the WaveFunction.

Parameters

- **nn_state** (*qucumber.nn_states.WaveFunctionBase*) – The WaveFunction 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.

- **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.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable.

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.WaveFunctionBase*) – The WaveFunction 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.

Return type dict(str, float)

symbol

The algebraic symbol representing the Observable.

11.3 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.

> **apply**(*nn_state*, *samples*)
>> Computes the value of the observable, row-wise, on a batch of samples.

>> If we think of the samples given as a set of projective measurements in a given computational basis, this method must return the expectation of the operator with respect to each basis state in *samples*. It 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.

>>> **Parameters**

>>> - **nn_state** (`qucumber.nn_states.WaveFunctionBase`) – The WaveFunction 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](#)

> **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 WaveFunction.

>>> **Parameters**

>>> - **nn_state** (`qucumber.nn_states.WaveFunctionBase`) – The WaveFunction 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*)
>> Estimates the expected value, variance, and the standard error of the observable over the distribution defined by the WaveFunction.

>>> **Parameters**

>>> - **nn_state** (`qucumber.nn_states.WaveFunctionBase`) – The WaveFunction 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.

- **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.

Returns A dictionary containing the (estimated) expected value (key: "mean"), variance (key: "variance"), and standard error (key: "std_error") of the observable.

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.WaveFunctionBase`) – The WaveFunction 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.

Return type dict(str, float)

symbol
    The algebraic symbol representing the Observable.

COMPLEX ALGEBRA

qucumber.utils.cplx.**absolute_value**(*x*)
> Computes the complex absolute value elementwise.
>
> > **Parameters** **x** (*torch.Tensor*) – A complex tensor.
> >
> > **Returns** A real tensor.
> >
> > **Return type** torch.Tensor

qucumber.utils.cplx.**conjugate**(*x*)
> A function that takes the conjugate transpose of the argument.
>
> > **Parameters** **x** (*torch.Tensor*) – A complex vector or matrix.
> >
> > **Returns** The conjugate of x.
> >
> > **Return type** torch.Tensor

qucumber.utils.cplx.**elementwise_division**(*x*, *y*)
> Elementwise 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.**kronecker_prod**(*x*, *y*)

A function that returns the tensor / kronecker product of 2 complex tensors, 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 tensorproduct of x and y, $x \otimes y$.
>
> **Return type** torch.Tensor

qucumber.utils.cplx.**make_complex**(*x*, *y=None*)

A function that combines the real (x) and imaginary (y) parts of a vector or a matrix.

Note: x and y must have the same shape. Also, this will not work for rank zero tensors.

> **Parameters**
>
> > - **x** (`torch.Tensor`) – The real part
> >
> > - **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 + yi.
>
> **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*)

A function that 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*)

    A function that returns the squared norm of the argument.

        Parameters **x** (`torch.Tensor`) – A complex scalar.

        Returns $|x|^2$.

        Return type torch.Tensor

`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 the function will not execute.

        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*)

    A function that computes the division of x by y.

        Parameters

            • **x** (`torch.Tensor`) – The numerator (a complex scalar, vector or matrix).

            • **y** (`torch.Tensor`) – The denominator (a complex scalar).

        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.

            • **z** (`torch.Tensor`) – The complex tensor to write the output to.

            • **z** – A complex scalar, vector or matrix. Can be None, in which case, a new tensor is created and returned. Otherwise, the method overwrites z.

        Returns The product between x and y. Either overwrites z, or returns a new tensor.

        Return type torch.Tensor

DATA HANDLING

qucumber.utils.data.**extract_refbasis_samples**(*train_samples*, *train_bases*)
    Extract the reference basis samples from the data.

> **Parameters**
>
> - **train_samples** (*numpy.array*) – The training samples.
>
> - **train_bases** (*numpy.array*) – 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

INDICES AND TABLES

- genindex

- search

PYTHON MODULE INDEX

q