
QuCumber Documentation

Release v0.3.1.post4

PIQuIL

2018-09-06

1	Installation	1
1.1	Github	1
1.2	Windows	1
1.3	Linux / MacOS	1
2	Theory	3
3	Download the tutorials	5
4	Reconstruction of a positive-real wavefunction	7
4.1	Transverse-field Ising model	7
4.2	Using qucumber to reconstruct the wavefunction	7
4.2.1	Imports	7
4.2.2	Training	8
5	Reconstruction of a complex wavefunction	13
5.1	The wavefunction to be reconstructed	13
5.2	Using qucumber to reconstruct the wavefunction	13
5.2.1	Imports	13
5.2.2	Training	14
6	Sampling and calculating observables	21
6.1	Generate new samples	21
6.2	Calculate an observable using the <i>Observable</i> module	22
6.2.1	Custom observable	22
6.2.2	TFIM Energy	24
6.2.3	Adding observables	25
6.2.4	Template for your custom observable	25
7	Training while monitoring observables	27
8	RBM	31
9	Quantum States	33
9.1	Positive Wavefunction	33
9.2	Complex Wavefunction	37
9.3	Abstract Wavefunction	40

10 Callbacks	45
11 Observables	51
11.1 Pauli Operators	51
11.2 Neighbour Interactions	55
11.3 Abstract Observable	56
12 Complex Algebra	59
13 Data Handling	63
14 Indices and tables	65
Python Module Index	67

CHAPTER 1

Installation

QuCumber only supports Python 3, not Python 2. If you are using Python 2, please update! You will also want to install the following packages, if you have not already.

1. Pytorch v0.4.1 (<https://pytorch.org/>)
2. tqdm (<https://github.com/tqdm/tqdm>)

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
```

CHAPTER 2

Theory

For a basic introduction to Restricted Boltzmann Machines, click [here](#).


Download the tutorials

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

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

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

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

Alternatively, you can view interactive notebooks online at: , though they may be slow.

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

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

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.

```
In [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.

```
In [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.

```
In [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.

```
In [4]: epochs = 1000
        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. **log_every**: the frequency of the training evaluators being calculated is controlled by the *log_every* argument (e.g. *log_every* = 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*.

```
In [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 “*log_every*” epochs can be given to the *MetricEvaluator*.

```
In [6]: log_every = 100
        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*).

```
In [7]: callbacks = [
        MetricEvaluator(
            log_every,
            {"Fidelity": ts.fidelity, "KL": ts.KL, "A_Ψrbm_5": psi_coefficient},
            target_psi=true_psi,
            verbose=True,
            space=space,
            A=3.,
        )
    ]

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

```
)
# nn_state.fit(train_data, callbacks=callbacks)

Epoch: 100      Fidelity = 0.916228      KL = 0.171583      A_rbm_5 = 0.222936
Epoch: 200      Fidelity = 0.964221      KL = 0.071276      A_rbm_5 = 0.210849
Epoch: 300      Fidelity = 0.979963      KL = 0.039937      A_rbm_5 = 0.221388
Epoch: 400      Fidelity = 0.987497      KL = 0.024977      A_rbm_5 = 0.223976
Epoch: 500      Fidelity = 0.989811      KL = 0.020543      A_rbm_5 = 0.235250
Epoch: 600      Fidelity = 0.991764      KL = 0.016631      A_rbm_5 = 0.232943
Epoch: 700      Fidelity = 0.993143      KL = 0.013830      A_rbm_5 = 0.234583
Epoch: 800      Fidelity = 0.993379      KL = 0.013242      A_rbm_5 = 0.241191
Epoch: 900      Fidelity = 0.994647      KL = 0.010728      A_rbm_5 = 0.237508
Epoch: 1000     Fidelity = 0.995182      KL = 0.009666      A_rbm_5 = 0.238725
```

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).

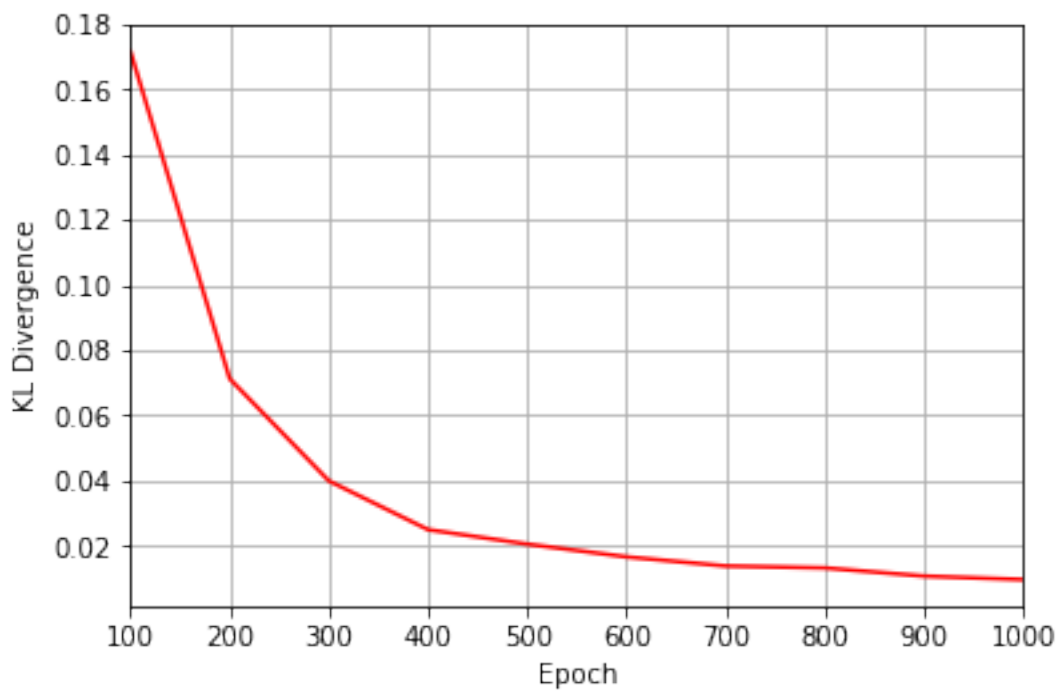
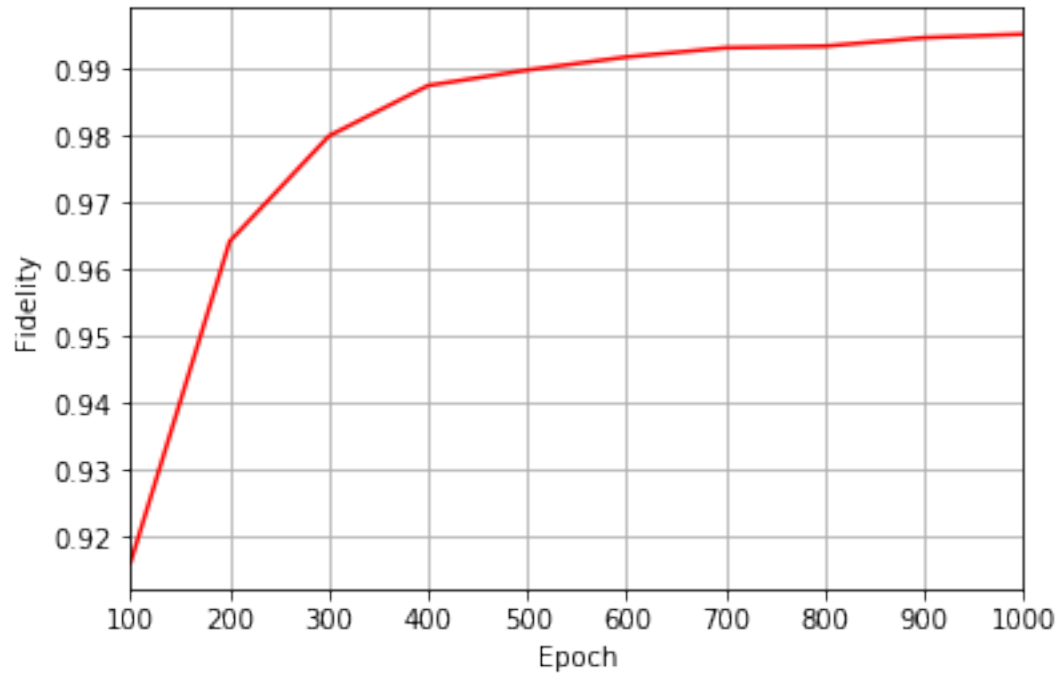
```
In [8]: fidelities = callbacks[0].Fidelity
        Kls = callbacks[0].KL
        coeffs = callbacks[0].A_Ψrbm_5
        # Please note that the key given to the *MetricEvaluator* must be what comes after callbacks
        epoch = np.arange(log_every, epochs + 1, log_every)

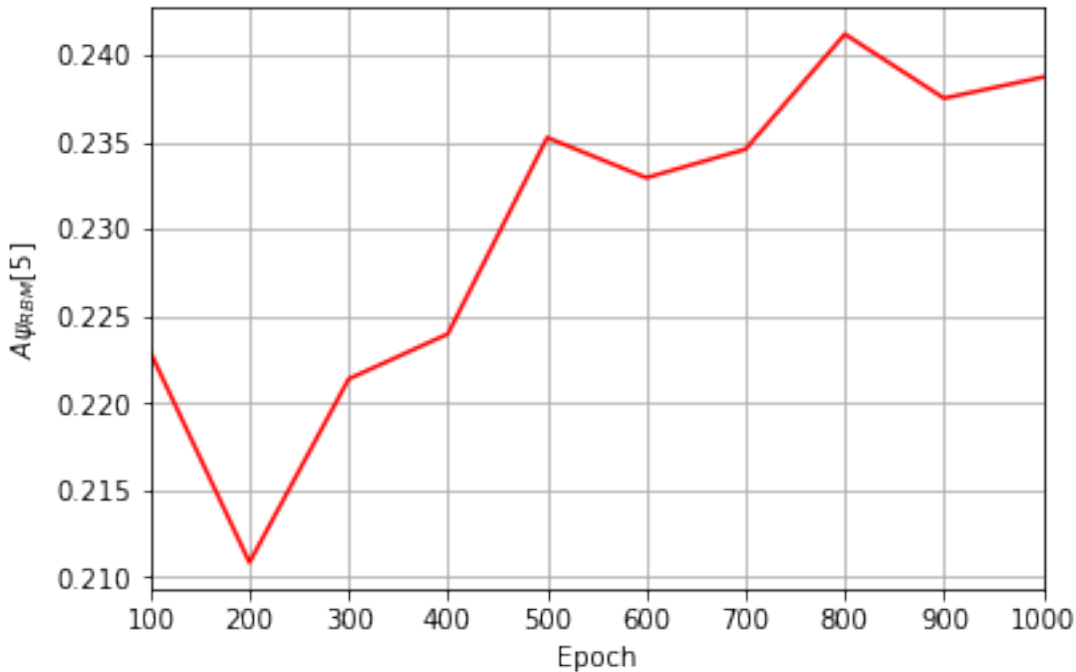
        plt.figure(1)
        ax1 = plt.axes()
        ax1.grid()
        ax1.set_xlim(log_every, epochs)
        ax1.set_xlabel("Epoch")
        ax1.set_ylabel("Fidelity")
        ax1.plot(epoch, fidelities, color="r")

        plt.figure(2)
        ax2 = plt.axes()
        ax2.grid()
        ax2.set_xlim(log_every, epochs)
        ax2.set_xlabel("Epoch")
        ax2.set_ylabel("KL Divergence")
        ax2.plot(epoch, Kls, color="r")

        plt.figure(3)
        ax3 = plt.axes()
        ax3.grid()
        ax3.set_xlim(log_every, epochs)
        ax3.set_xlabel("Epoch")
        ax3.set_ylabel(r"$A\backslash\psi_{\{RBM\}}[5]$")
        ax3.plot(epoch, coeffs, color="r")

Out [8]: [<matplotlib.lines.Line2D at 0x7f0ce75d00b8>]
```





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.

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.

```
In [9]: 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.

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle \quad (5.1)$$

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

$$\alpha = 0.2861 + 0.0539i \quad (5.2)$$

$$\beta = 0.3687 - 0.3023i \quad (5.3)$$

$$\gamma = -0.1672 - 0.3529i \quad (5.4)$$

$$\delta = -0.5659 - 0.4639i. \quad (5.5)$$

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.

```
In [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

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.

```
In [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.

```
In [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 with 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.

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

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

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 *ComplexWavefunction* object instantiation. Uncomment 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.

```
In [5]: epochs = 80
        pbs = 50 # pos_batch_size
        nbs = 10 # 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. **log_every**: the frequency of the training evaluators being calculated is controlled by the *log_every* argument (e.g. *log_every* = 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. 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.

```
In [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 “*log_every*” epochs must be given to the *MetricEvaluator*.

```
In [7]: log_every = 10
        space = nn_state.generate_hilbert_space(nv)

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

Now the training can begin. The *ComplexWavefunction* object has a property called *fit* which takes care of this.

```
In [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: 10	Fidelity = 0.584157	KL = 0.239528	norm = 0.203017	norm = 0.333242
Epoch: 20	Fidelity = 0.900338	KL = 0.046861	norm = 0.284082	norm = 0.468427
Epoch: 30	Fidelity = 0.964009	KL = 0.022833	norm = 0.284898	norm = 0.489092
Epoch: 40	Fidelity = 0.979635	KL = 0.015013	norm = 0.286033	norm = 0.462681
Epoch: 50	Fidelity = 0.978498	KL = 0.017450	norm = 0.264300	norm = 0.413593
Epoch: 60	Fidelity = 0.983162	KL = 0.013628	norm = 0.284534	norm = 0.433017
Epoch: 70	Fidelity = 0.989934	KL = 0.009804	norm = 0.265790	norm = 0.484255
Epoch: 80	Fidelity = 0.990302	KL = 0.009240	norm = 0.255228	norm = 0.458123

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).

```
In [9]: fidelities = callbacks[0].Fidelity
        KLs = callbacks[0].KL
        coeffs = callbacks[0].norm $\alpha$ 
        # Please note that the key given to the *MetricEvaluator* must be what comes after callbacks
        epoch = np.arange(log_every, epochs + 1, log_every)

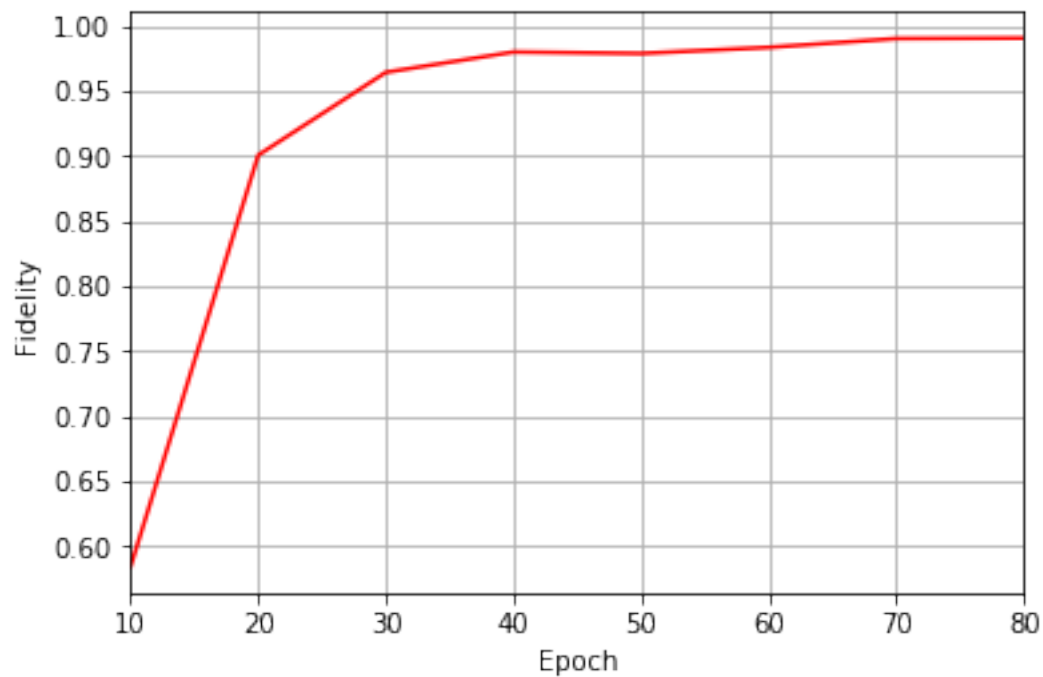
        plt.figure(1)
        ax1 = plt.axes()
        ax1.grid()
        ax1.set_xlim(log_every, epochs)
        ax1.set_xlabel("Epoch")
        ax1.set_ylabel("Fidelity")
        ax1.plot(epoch, fidelities, color="r")

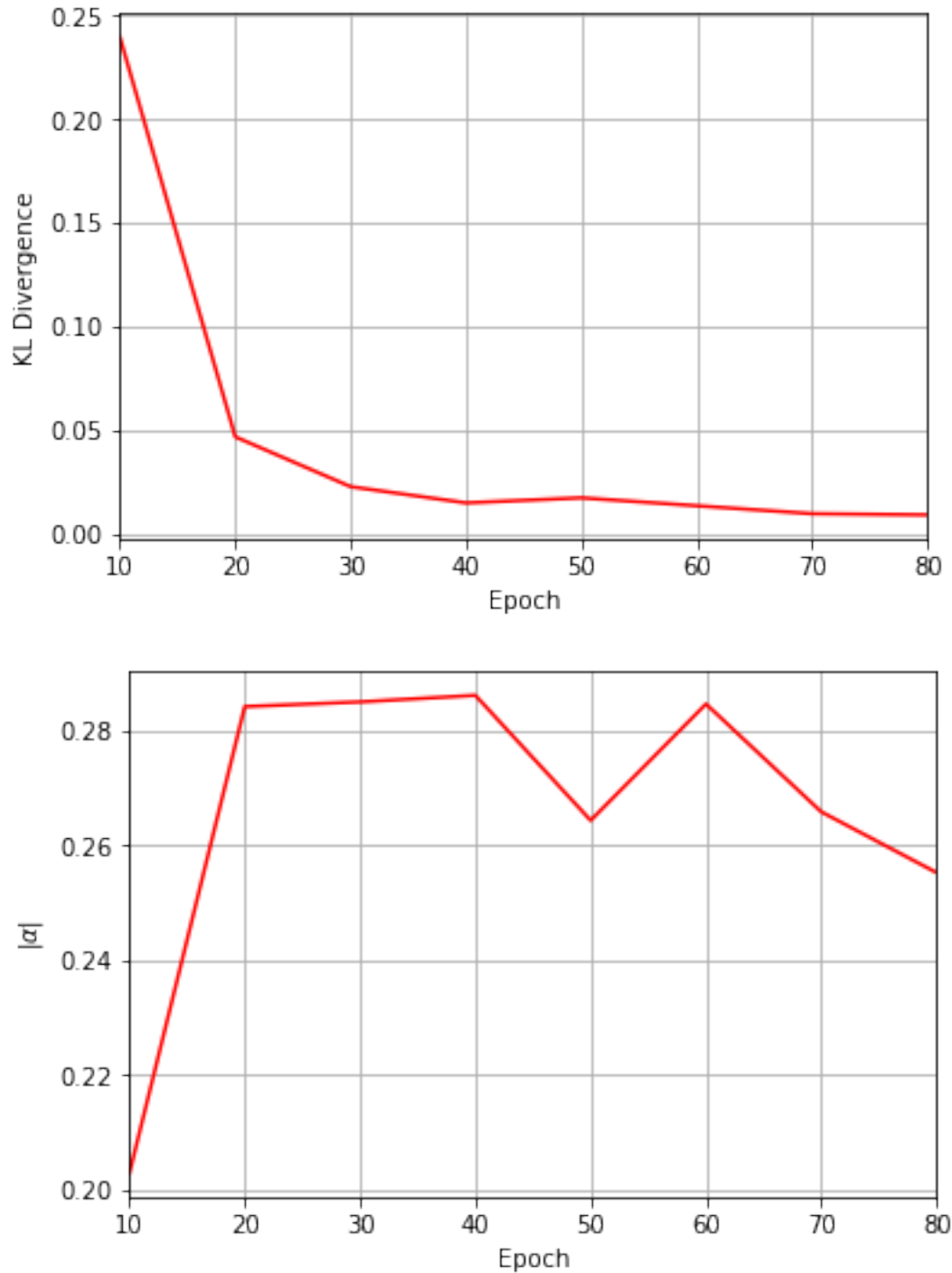
        plt.figure(2)
```

```
ax2 = plt.axes()
ax2.grid()
ax2.set_xlim(log_every, epochs)
ax2.set_xlabel("Epoch")
ax2.set_ylabel("KL Divergence")
ax2.plot(epoch, KLs, color="r")

plt.figure(3)
ax3 = plt.axes()
ax3.grid()
ax3.set_xlim(log_every, epochs)
ax3.set_xlabel("Epoch")
ax3.set_ylabel(r"$\vert\alpha\vert$")
ax3.plot(epoch, coeffs, color="r")
```

Out[9]: [





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.

```
In [10]: nn_state.save("saved_params.pt")
```

This saves the weights, visible biases and hidden biases as torch tensors with the following keys: “weights”, “visi-

ble_bias”, “hidden_bias”.

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.

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

from qucumber.nn_states import PositiveWavefunction

from qucumber.observables import Observable

import quantum_ising_chain
from quantum_ising_chain import TFIMChainEnergy

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

A *PositiveWavefunction* object has a property called *sample* that takes in the following arguments.

1. **k**: the number of Gibbs steps to perform to generate the new samples
2. **num_samples**: the number of new data points to be generated

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

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

With the newly generated samples, the user can now easily 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.

```
In [3]: def to_pml(samples):
        return samples.mul(2.).sub(1.)
```

Now, the magnetization is calculated as follows.

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

magnetization = Magnetization(new_samples).item()

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

```
Magnetization = 0.55620
```

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.

```
In [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 on top of the RBM parameters.

6.2 Calculate an observable using the *Observable* module

6.2.1 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 *Observable* module in action, an example observable called *PIQuIL*, which inherits properties from the *Observable* module, is shown below.

The *PIQuIL* observable takes an σ^z measurement at a site and multiplies it by the measurement two sites 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.

```
In [6]: class PIQuIL(Observable):
        def __init__(self, P):
            super(PIQuIL, self).__init__()
            self.P = P

        # Required : function that calculates the PIQuIL. Must be named "apply"
        def apply(self, nn_state, samples):
            to_pml(samples)
            interaction_ = 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 *Observables* that the user would like to define must contain a function called *apply* that calculates this new observable.

Although the *PIQuIL* observable could technically be computed without the use of the *Observable* module since it does not ever use the RBM (*nn_state*), it is still nonetheless a constructive example.

The real power in the *Observable* module is in the ability for the user to easily compute statistics of the observable from the generated sample. 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 give the RBM and the samples as arguments.

```
In [7]: 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.

```
In [8]: print(
        "Mean PIQuIL: %.4f" % piquil_stats1["mean"], "+/- %.4f" % piquil_stats1["std_error"]
    )
    print("Variance: %.4f" % piquil_stats1["variance"])
```

```
Mean PIQuIL: 0.1421 +/- 0.0014
Variance: 0.0192
```

However, if the user did not have samples generated already, that is no problem. The *statistics* function in the *Observable* module will generate new samples internally and compute the mean, standard error and variance on those samples. Since the samples are not an argument in the *statistics* function, the user must now give the following additional arguments to the *statistics* function to generate the new samples.

- **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 *statistics* function will also return a dictionary containing the mean, standard error and the variance with the keys “mean”, “std_error” and “variance”, respectively.

```
In [9]: num_samples = 10000
        burn_in = 100
        steps = 100

        piquil_stats2 = piquil.statistics(nn_state, num_samples, burn_in=burn_in, steps=steps)
        print(
            "Mean PIQuIL: %.4f" % piquil_stats2["mean"], "+/- %.4f" % piquil_stats2["std_error"]
        )
        print("Variance: %.4f" % piquil_stats2["variance"])
```

```
Mean PIQuIL: 0.1423 +/- 0.0014
Variance: 0.0188
```

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. Whereas the TFIM energy is much more complicated. 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 ($\frac{h}{J} = 1$).

```
In [10]: 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.

```
In [11]: energy_stats = tfim_energy.statistics_from_samples(nn_state, new_samples)
print("Mean: %.4f" % energy_stats["mean"], "+/- %.4f" % energy_stats["std_error"])
print("Variance: %.4f" % energy_stats["variance"])
```

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

```
Variance: 0.0022
```

The exact value for the energy is -1.2381.

To illustrate how quickly the energy converges as a function of the sampling step (i.e. the number of Gibbs steps to perform to generate a new batch of samples), *steps*, the *Convergence* function in *quantum_ising_chain.py* will do the trick. *Convergence* creates a batch of random samples initially, which is then used to generate a new batch of samples from the RBM. The TFIM energy will be calculated at every Gibbs step. Please note that the samples generated previously (*new_samples*) are not used here; different samples are generated.

```
In [12]: steps = 200
num_samples = 10000

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

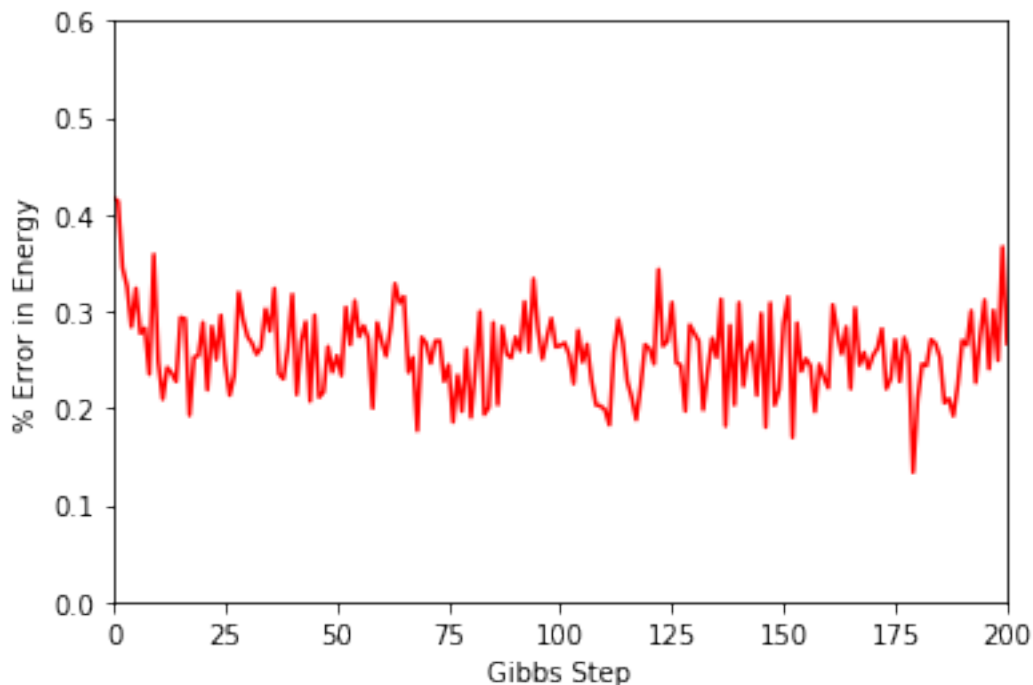
energy = dict_observables["energies"]
err_energy = dict_observables["error"]

step = np.arange(steps + 1)

E0 = -1.2381

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

```
Out[12]: 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. After that, the values fluctuate around the calculated mean.

6.2.3 Adding observables

One may also add / subtract observables with the new observable also retaining the same properties in the *Observables* module. For instance, a new observable can be defined by adding the TFIM energy observable multiplied by an arbitrary constant to the PIQuIL observable.

```
In [13]: new_obs = 0.01 * tfim_energy + piquil
```

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

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

```
Mean: 0.1297 +/- 0.0014
```

```
Variance: 0.0192
```

6.2.4 Template for your custom observable

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

```
In [15]: import torch
from qucumber.observables import Observable

class YourObservable(Observable):
    def __init__(self, your_constants):
        super(YourObservable, self).__init__()
        self.your_constants = your_constants
```

```
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*. The following implements the *ObservableEvaluator* to calculate the energy during the training on the TFIM data in the first tutorial. We will use the same 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.

```
In [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

In [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

log_every = 100

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

tfim_energy = TFIMChainEnergy(h)
```

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

1. **log_every**: the frequency of the training evaluators being calculated is controlled by the *log_every* argument (e.g. *log_every* = 200 means that the *MetricEvaluator* will update the user 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 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*).

- **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 via the *verbose=True* statement.

```
In [3]: callbacks = [
    ObservableEvaluator(
        log_every,
        [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: -1.194464    variance: 0.024380    std_error: 0.001561
Epoch: 200
  TFIMChainEnergy:
    mean: -1.217816    variance: 0.012093    std_error: 0.001100
Epoch: 300
```

```

    TFIMChainEnergy:
      mean: -1.226431      variance: 0.007242      std_error: 0.000851
Epoch: 400
    TFIMChainEnergy:
      mean: -1.229766      variance: 0.005400      std_error: 0.000735
Epoch: 500
    TFIMChainEnergy:
      mean: -1.231543      variance: 0.004386      std_error: 0.000662
Epoch: 600
    TFIMChainEnergy:
      mean: -1.232977      variance: 0.003655      std_error: 0.000605
Epoch: 700
    TFIMChainEnergy:
      mean: -1.232943      variance: 0.003286      std_error: 0.000573
Epoch: 800
    TFIMChainEnergy:
      mean: -1.234483      variance: 0.002825      std_error: 0.000532
Epoch: 900
    TFIMChainEnergy:
      mean: -1.235027      variance: 0.002327      std_error: 0.000482
Epoch: 1000
    TFIMChainEnergy:
      mean: -1.235285      variance: 0.002064      std_error: 0.000454

```

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

```

In [4]: energies = callbacks[0].TFIMChainEnergy.mean
        errors = callbacks[0].TFIMChainEnergy.std_error
        variance = callbacks[0].TFIMChainEnergy.variance
        # Please note that the name of the observable class that the user makes must be what comes a

```

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

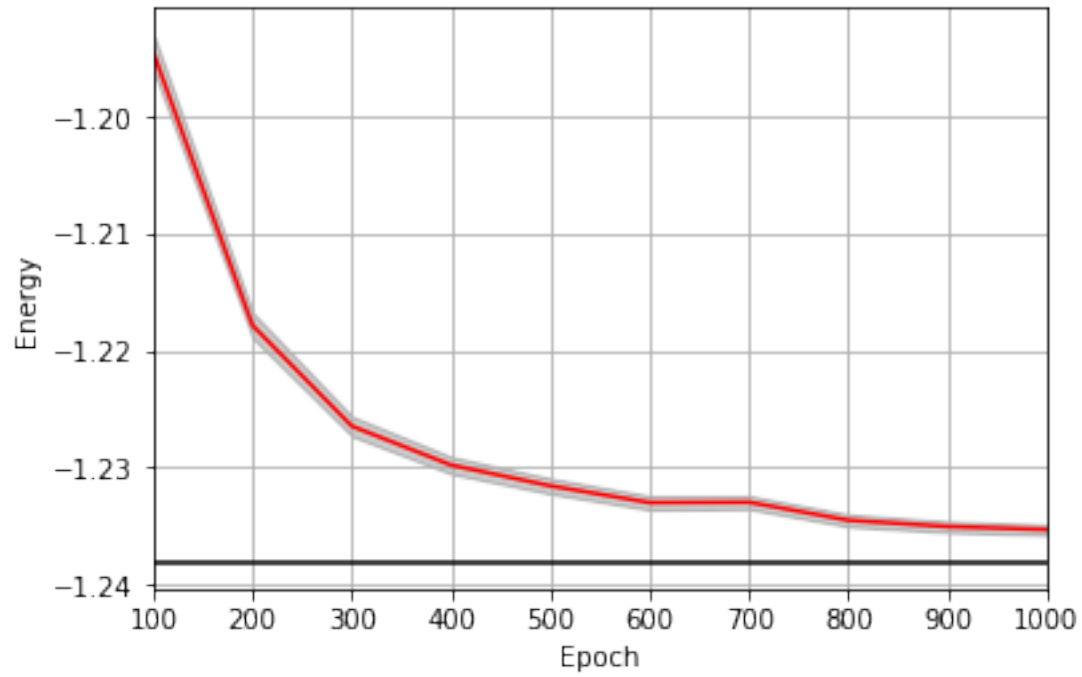
```

In [5]: epoch = np.arange(log_every, epochs + 1, log_every)

        E0 = -1.2381

        ax = plt.axes()
        ax.plot(epoch, energies, color="red")
        ax.set_xlim(log_every, 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()

```



```
class qucumber.rbm.BinaryRBM(num_visible, num_hidden, zero_weights=False, gpu=True)
```

```
    Bases: torch.nn.modules.module.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 $p(h | v)$, and sampling the visible state v from the conditional distribution $p(v | h)$.

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 Tuple containing *prob_h_given_v(v)* and the sampled hidden state.

Return type *tuple(torch.Tensor, 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 Tuple containing *prob_v_given_h(h)* and the sampled visible state.

Return type *tuple(torch.Tensor, torch.Tensor)*

9.1 Positive Wavefunction

```
class qucumber.nn_states.PositiveWavefunction(num_visible, num_hidden=None,
                                              gpu=True)
```

Bases: *qucumber.nn_states.Wavefunction*

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 \mathbf{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*, *progressbar=False*, *starting_epoch=1*, *time=False*, *callbacks=None*, *optimizer=<class 'torch.optim.sgd.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
- **progressbar** (*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.Callback]*) – 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 Positive Wavefunction, 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.Wavefunction*

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=<class 'torch.optim.sgd.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.Callback]*) – 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 whose 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.Wavefunction

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*=<class 'torch.optim.sgd.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.Callback*]) – 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`

class `qucumber.callbacks.Callback`

Base class for callbacks.

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

Called at the end of each batch.

Parameters

- **nn_state** (*Wavefunction*) – 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** (*Wavefunction*) – 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** (*Wavefunction*) – 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** (*Wavefunction*) – 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** (*Wavefunction*) – The Wavefunction being trained.

on_train_start (*nn_state*)

Called at the start of the training cycle.

Parameters **nn_state** (*Wavefunction*) – The Wavefunction 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 *Callback.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 *Callback.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 *Callback.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 *Callback.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 *Callback.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 *Callback.on_batch_end*.

```
class qucumber.callbacks.ModelSaver (period, folder_path, file_name, save_initial=True, meta-
                                     data=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/observable 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

__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 Callbacks 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.Observable`)) – 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 right-most observable argument.
- **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*

__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.

11.1 Pauli Operators

class qucumber.observables.SigmaZ

Bases: *qucumber.observables.Observable*

The σ_z observable.

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

apply (*nn_state*, *samples*)

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

Parameters

- **nn_state** (*qucumber.nn_states.Wavefunction*) – 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.Wavefunction*) – 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 it is provided, with the updated state of the Markov chain.

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.Wavefunction`) – 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, will use a number of chains equal to `num_samples`.
- **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.Wavefunction`) – The Wavefunction that drew the samples.
- **samples** (`torch.Tensor`) – A batch of sample states to calculate the observable on.

symbol

The algebraic symbol representing the Observable.

class `qucumber.observables.SigmaX`

Bases: `qucumber.observables.Observable`

The σ_x observable

Computes the magnetization in the X direction of a spin chain.

apply (*nn_state*, *samples*)

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

Parameters

- **nn_state** (`qucumber.nn_states.Wavefunction`) – 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.Wavefunction`) – 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 it is provided, with the updated state of the Markov chain.

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.Wavefunction`) – 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, will use a number of chains equal to `num_samples`.
- **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.Wavefunction`) – The Wavefunction that drew the samples.
- **samples** (`torch.Tensor`) – A batch of sample states to calculate the observable on.

symbol

The algebraic symbol representing the Observable.

class `qucumber.observables.SigmaY`

Bases: `qucumber.observables.Observable`

The σ_y observable

Computes the magnetization in the Y direction of a spin chain.

apply (`nn_state, samples`)

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

Parameters

- **nn_state** (`qucumber.nn_states.Wavefunction`) – 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.Wavefunction*) – 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 it is provided, with the updated state of the Markov chain.

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.Wavefunction*) – 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, will use a number of chains equal to *num_samples*.
- **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.Wavefunction*) – The Wavefunction that drew the samples.
- **samples** (*torch.Tensor*) – A batch of sample states to calculate the observable on.

symbol

The algebraic symbol representing the Observable.

11.2 Neighbour Interactions

class qucumber.observables.NeighbourInteraction (*periodic_bcs=False, c=1*)

Bases: `qucumber.observables.Observable`

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.Wavefunction`) – 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.Wavefunction`) – 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 it is provided, with the updated state of the Markov chain.

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.Wavefunction`) – 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, will use a number of chains equal to *num_samples*.
- **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.Wavefunction`) – The Wavefunction that drew the samples.
- **samples** (`torch.Tensor`) – A batch of sample states to calculate the observable on.

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.Observable`

Bases: `abc.ABC`

Base class for observables.

apply (*nn_state, samples*)

Computes the value of the observable, row-wise, on a batch of samples. Must be implemented by any subclasses.

Parameters

- **nn_state** (`qucumber.nn_states.Wavefunction`) – The Wavefunction that drew the samples.
- **samples** (`torch.Tensor`) – A batch of sample states to calculate the observable on.

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.Wavefunction`) – 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 it is provided, with the updated state of the Markov chain.

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.Wavefunction`) – 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, will use a number of chains equal to *num_samples*.
- **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.Wavefunction`) – The Wavefunction that drew the samples.
- **samples** (`torch.Tensor`) – A batch of sample states to calculate the observable on.

symbol

The algebraic symbol representing the Observable.

`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.inner_prod(x, y)`

A function that returns the inner product of two complex vectors, `x` and `y` ($\langle x|y \rangle$).

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`

`gucumber.utils.cplx.kronecker_prod(x, y)`

A function that returns the tensor / kronecker product of 2 complex tensors, x and y .

Parameters

- \mathbf{x} (`torch.Tensor`) – A complex matrix.
- \mathbf{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`

`gucumber.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

- \mathbf{x} (`torch.Tensor`) – The real part
- \mathbf{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`

`gucumber.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

- \mathbf{x} (`torch.Tensor`) – A complex matrix.
- \mathbf{y} (`torch.Tensor`) – A complex vector or matrix.

Returns The product between x and y .

Return type `torch.Tensor`

`gucumber.utils.cplx.norm(x)`

A function that returns the norm of the argument.

Parameters \mathbf{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.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`

`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`

CHAPTER 14

Indices and tables

- `genindex`
- `modindex`
- `search`

q

`qucumber.utils.cplx`, [59](#)

`qucumber.utils.data`, [63](#)

Symbols

`__getattr__()` (qucumber.callbacks.MetricEvaluator method), 48
`__getattr__()` (qucumber.callbacks.ObservableEvaluator method), 49
`__len__()` (qucumber.callbacks.MetricEvaluator method), 48
`__len__()` (qucumber.callbacks.ObservableEvaluator method), 49

A

`absolute_value()` (in module qucumber.utils.cplx), 59
`amplitude()` (qucumber.nn_states.ComplexWavefunction method), 37
`amplitude()` (qucumber.nn_states.PositiveWavefunction method), 33
`amplitude()` (qucumber.nn_states.Wavefunction method), 41
`apply()` (qucumber.observables.NeighbourInteraction method), 55
`apply()` (qucumber.observables.Observable method), 56
`apply()` (qucumber.observables.SigmaX method), 52
`apply()` (qucumber.observables.SigmaY method), 53
`apply()` (qucumber.observables.SigmaZ method), 51
`autoload()` (qucumber.nn_states.ComplexWavefunction static method), 37
`autoload()` (qucumber.nn_states.PositiveWavefunction static method), 33
`autoload()` (qucumber.nn_states.Wavefunction static method), 41

B

BinaryRBM (class in qucumber.rbm), 31

C

Callback (class in qucumber.callbacks), 45
`clear_history()` (qucumber.callbacks.MetricEvaluator method), 48

`clear_history()` (qucumber.callbacks.ObservableEvaluator method), 49

ComplexWavefunction (class in qucumber.nn_states), 37

`compute_batch_gradients()` (qucumber.nn_states.ComplexWavefunction method), 37

`compute_batch_gradients()` (qucumber.nn_states.PositiveWavefunction method), 34

`compute_batch_gradients()` (qucumber.nn_states.Wavefunction method), 41

`compute_normalization()` (qucumber.nn_states.ComplexWavefunction method), 37

`compute_normalization()` (qucumber.nn_states.PositiveWavefunction method), 34

`compute_normalization()` (qucumber.nn_states.Wavefunction method), 41

`conjugate()` (in module qucumber.utils.cplx), 59

D

device (qucumber.nn_states.ComplexWavefunction attribute), 38

device (qucumber.nn_states.PositiveWavefunction attribute), 34

device (qucumber.nn_states.Wavefunction attribute), 41

E

EarlyStopping (class in qucumber.callbacks), 47

`effective_energy()` (qucumber.rbm.BinaryRBM method), 31

`effective_energy_gradient()` (qucumber.rbm.BinaryRBM method), 31

`elementwise_division()` (in module qucumber.utils.cplx), 59

`elementwise_mult()` (in module qucumber.utils.cplx), 59

epochs (qucumber.callbacks.MetricEvaluator attribute), 48

epochs (qucumber.callbacks.ObservableEvaluator attribute), 49
 extract_refbasis_samples() (in module qucumber.utils.data), 63

F

fit() (qucumber.nn_states.ComplexWavefunction method), 38
 fit() (qucumber.nn_states.PositiveWavefunction method), 34
 fit() (qucumber.nn_states.Wavefunction method), 41

G

generate_hilbert_space() (qucumber.nn_states.ComplexWavefunction method), 38
 generate_hilbert_space() (qucumber.nn_states.PositiveWavefunction method), 34
 generate_hilbert_space() (qucumber.nn_states.Wavefunction method), 42
 get_value() (qucumber.callbacks.MetricEvaluator method), 49
 get_value() (qucumber.callbacks.ObservableEvaluator method), 50
 gibbs_steps() (qucumber.rbm.BinaryRBM method), 31
 gradient() (qucumber.nn_states.ComplexWavefunction method), 38
 gradient() (qucumber.nn_states.PositiveWavefunction method), 35
 gradient() (qucumber.nn_states.Wavefunction method), 42

I

initialize_parameters() (qucumber.rbm.BinaryRBM method), 31
 inner_prod() (in module qucumber.utils.cplx), 59

K

kronecker_prod() (in module qucumber.utils.cplx), 60

L

LambdaCallback (class in qucumber.callbacks), 46
 LivePlotting (class in qucumber.callbacks), 50
 load() (qucumber.nn_states.ComplexWavefunction method), 39
 load() (qucumber.nn_states.PositiveWavefunction method), 35
 load() (qucumber.nn_states.Wavefunction method), 42
 load_data() (in module qucumber.utils.data), 63
 Logger (class in qucumber.callbacks), 46

M

make_complex() (in module qucumber.utils.cplx), 60

matmul() (in module qucumber.utils.cplx), 60
 max_size (qucumber.nn_states.ComplexWavefunction attribute), 39
 max_size (qucumber.nn_states.PositiveWavefunction attribute), 35
 max_size (qucumber.nn_states.Wavefunction attribute), 43
 MetricEvaluator (class in qucumber.callbacks), 48
 ModelSaver (class in qucumber.callbacks), 46

N

name (qucumber.observables.NeighbourInteraction attribute), 55
 name (qucumber.observables.Observable attribute), 56
 name (qucumber.observables.SigmaX attribute), 52
 name (qucumber.observables.SigmaY attribute), 54
 name (qucumber.observables.SigmaZ attribute), 51
 names (qucumber.callbacks.MetricEvaluator attribute), 49
 names (qucumber.callbacks.ObservableEvaluator attribute), 50
 NeighbourInteraction (class in qucumber.observables), 55
 networks (qucumber.nn_states.ComplexWavefunction attribute), 39
 networks (qucumber.nn_states.PositiveWavefunction attribute), 35
 networks (qucumber.nn_states.Wavefunction attribute), 43
 norm() (in module qucumber.utils.cplx), 60
 norm_sqr() (in module qucumber.utils.cplx), 61

O

Observable (class in qucumber.observables), 56
 ObservableEvaluator (class in qucumber.callbacks), 49
 on_batch_end() (qucumber.callbacks.Callback method), 45
 on_batch_start() (qucumber.callbacks.Callback method), 45
 on_epoch_end() (qucumber.callbacks.Callback method), 45
 on_epoch_start() (qucumber.callbacks.Callback method), 45
 on_train_end() (qucumber.callbacks.Callback method), 46
 on_train_start() (qucumber.callbacks.Callback method), 46
 outer_prod() (in module qucumber.utils.cplx), 61

P

partition() (qucumber.rbm.BinaryRBM method), 32
 phase() (qucumber.nn_states.ComplexWavefunction method), 39
 phase() (qucumber.nn_states.PositiveWavefunction method), 35

phase() (qucumber.nn_states.Wavefunction method), 43
 PositiveWavefunction (class in qucumber.nn_states), 33
 prob_h_given_v() (qucumber.rbm.BinaryRBM method), 32
 prob_v_given_h() (qucumber.rbm.BinaryRBM method), 32
 probability() (qucumber.nn_states.ComplexWavefunction method), 39
 probability() (qucumber.nn_states.PositiveWavefunction method), 35
 probability() (qucumber.nn_states.Wavefunction method), 43
 psi() (qucumber.nn_states.ComplexWavefunction method), 39
 psi() (qucumber.nn_states.PositiveWavefunction method), 36
 psi() (qucumber.nn_states.Wavefunction method), 43

Q

qucumber.utils.cplx (module), 59
 qucumber.utils.data (module), 63

R

rbm_am (qucumber.nn_states.ComplexWavefunction attribute), 40
 rbm_am (qucumber.nn_states.PositiveWavefunction attribute), 36
 rbm_am (qucumber.nn_states.Wavefunction attribute), 43
 rbm_ph (qucumber.nn_states.ComplexWavefunction attribute), 40
 reinitialize_parameters() (qucumber.nn_states.ComplexWavefunction method), 40
 reinitialize_parameters() (qucumber.nn_states.PositiveWavefunction method), 36
 reinitialize_parameters() (qucumber.nn_states.Wavefunction method), 43

S

sample() (qucumber.nn_states.ComplexWavefunction method), 40
 sample() (qucumber.nn_states.PositiveWavefunction method), 36
 sample() (qucumber.nn_states.Wavefunction method), 43
 sample() (qucumber.observables.NeighbourInteraction method), 55
 sample() (qucumber.observables.Observable method), 56
 sample() (qucumber.observables.SigmaX method), 52
 sample() (qucumber.observables.SigmaY method), 54
 sample() (qucumber.observables.SigmaZ method), 51
 sample_h_given_v() (qucumber.rbm.BinaryRBM method), 32

sample_v_given_h() (qucumber.rbm.BinaryRBM method), 32
 save() (qucumber.nn_states.ComplexWavefunction method), 40
 save() (qucumber.nn_states.PositiveWavefunction method), 36
 save() (qucumber.nn_states.Wavefunction method), 43
 scalar_divide() (in module qucumber.utils.cplx), 61
 scalar_mult() (in module qucumber.utils.cplx), 61
 SigmaX (class in qucumber.observables), 52
 SigmaY (class in qucumber.observables), 53
 SigmaZ (class in qucumber.observables), 51
 statistics() (qucumber.observables.NeighbourInteraction method), 55
 statistics() (qucumber.observables.Observable method), 56
 statistics() (qucumber.observables.SigmaX method), 53
 statistics() (qucumber.observables.SigmaY method), 54
 statistics() (qucumber.observables.SigmaZ method), 52
 statistics_from_samples() (qucumber.observables.NeighbourInteraction method), 56
 statistics_from_samples() (qucumber.observables.Observable method), 57
 statistics_from_samples() (qucumber.observables.SigmaX method), 53
 statistics_from_samples() (qucumber.observables.SigmaY method), 54
 statistics_from_samples() (qucumber.observables.SigmaZ method), 52
 stop_training (qucumber.nn_states.ComplexWavefunction attribute), 40
 stop_training (qucumber.nn_states.PositiveWavefunction attribute), 36
 stop_training (qucumber.nn_states.Wavefunction attribute), 44
 subspace_vector() (qucumber.nn_states.ComplexWavefunction method), 40
 subspace_vector() (qucumber.nn_states.PositiveWavefunction method), 36
 subspace_vector() (qucumber.nn_states.Wavefunction method), 44
 symbol (qucumber.observables.NeighbourInteraction attribute), 56
 symbol (qucumber.observables.Observable attribute), 57
 symbol (qucumber.observables.SigmaX attribute), 53
 symbol (qucumber.observables.SigmaY attribute), 54
 symbol (qucumber.observables.SigmaZ attribute), 52

T

Timer (class in qucumber.callbacks), 50

V

VarianceBasedEarlyStopping (class in qucumber.callbacks), [47](#)

W

Wavefunction (class in qucumber.nn_states), [41](#)