QuCumber: neural network-based generative modelling for quantum wavefunction reconstruction

Matthew J. S. Beach, Isaac De Vlugt, Anna Golubeva, Patrick Huembeli[†], Roger G. Melko[*], Ejaaz Merali, Giacomo Torlai,

Department of Physics and Astronomy, University of Waterloo,
Ontario N2L 3G, Canada
Perimeter Institute for Theoretical Physics, Waterloo,
Ontario N2L 2Y5, Canada
ICFO-Institut de Ciencies Fotoniques, Barcelona Institute of Science and Technology,
08860 Castelldefels (Barcelona), Spain[†]
* rgmelko@uwaterloo.ca

August 14, 2018

Abstract

In this post we present QuCumber, an open-source Python package which uses Restricted Boltzmann Machines for the reconstruction of quantum states from measurement data. The use of modern machine learning techniques makes it possible to efficiently learn quantum states and access traditionally challenging many-body quantities which are not directly accessible to experiments. We give examples of how to use QuCumber on positive-real and complex wavefunctions and how to extract meaningful observables such as energy, magnetization and fidelity.

Contents

1 Introduction

Scientific progress in controlled quantum systems has allowed for increasingly large-scale devices which can accurately prepare pure quantum states. As the size of these devices grows to hundreds of qubits, we face the challenge of analyzing highly complex many-body states with a finite set of experimental measurements. Since this computational power grows exponentially with the number of qubits, there is a growing interest in using machine learning techniques which can successfully distill meaningful statistics from incredible amounts of data. In this post, we present a software package for representing large many-body state with machine learning models which can be used to improve characterization and analysis of wave functions produced from near-term highly-controlled quantum devices.

QuCumber is based on Restricted Boltzmann Machines (RBMs), one of the most successful generative models from machine learning [1]. RBMs can represent many complex objects including handwritten digit, natural images, and many-body wave functions [2–5]. An RBM is physically described as a classical Ising model with tunable couplings which can be optimized during a training procedure. As such, it can be interpreted as a graphical representation of a probability distribution. With further modifications, one can represent complex wave functions with an amplitude and phase [6].

The need for QuCumber arises from the growing availability of high quality data are large-scale quantum devices. QuCumber provides a freely accessible open-source software package that implements wave function reconstruction with RBMs directly from experimental data representing measurements of qubit eigenvalues, spin states, orbital occupation numbers, etc. QuCumber can be used to optimize the parameters of an RBM to produce model the most likely quantum state given series of measurements. This is analogous to the maximum likelihood technique, Bayesian inference or quantum state tomography, however, our method does not scale exponentially with system size. Once trained, the RBM parameters encodes a compressed version of the underlying wave function New quantum state measurements can then be generated by the RBM, and observables that are otherwise not easily accessible from the original data can be calculated. Examples of which include off-diagonal correlation functions and entanglement entropies [2, 6]

In this paper, we describe how QuCumber can be used for these and other tasks in characterizing quantum many-body phenomenon on hundreds or thousands of qubits. In the following, we provide code snippets written in Python 3, using PyTorch with CPU and GPU support [7]. We refer the reader to the full code documentation (`https://piquil.github.io/QuCumber/`) for further information on the code structure and operation.

In the simplest scenario the expansion coefficients of the wavefunction are all positive in the chosen basis. Then the target distribution a faithful representation of the wavefunction, and it can be simply encoded in the parameters of an RBM. This case is discussed in the next section of this post. However, RBMs are not limited to positive real wavefunctions, and QuCumber may also be used to reconstruct a wavefunction's full amplitude and

phase, provided measurements are available in bases other than the computational basis. We discuss this case with a simple example in Section 3.

Below we provide a starting point for writing code employing QuCumber. For further background, the reader is urged to consult the relevant scientific literature, in particular Refs. [2, 6]. A glossary of useful terms and equations appears at the end of the post in Section A. In addition, extensive theoretical documentation and code tutorials are provided in the documentation for QuCumber at `https://piquil.github.io/QuCumber/`.

2 Positive Real Wavefunctions

In this section, we discuss the application of QuCumber to wave functions that are positive and real-valued in the computational basis. That is, the expansion of the wave function has all positive and real coefficients in the basis in which experiments are performed. In this case, QuCumber has a highly efficient parallelizable method to determine the most likely state consistent with the given measurements. As an example, we demonstrate how to train and sample a QuCumber RBM model with measurements from the one-dimensional transverse-field Ising model (TFIM).

2.1 Transverse field Ising model

The Hamiltonian for the transverse-field Ising model (TFIM) is given by

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

where σ_i^α is a spin-1/2 Pauli operator on site i, with $\alpha = x, y, z$. We consider the critical point where $J = h = 1$, which is the most difficult state to reconstuct. For training data, we use a synthetic data set consisting of $N = 10,000$ measurements in the σ^z-basis for 10 spins, generated with standard numerical techniques [8]. The example dataset is provided in `https://github.com/PIQuIL/QuCumber/blob/master/examples/01_Ising/tfim1d_train_samples.txt`.

2.2 Example measurements

To begin with wavefunction reconstruction, the input data needs to be in a numpy array or a torch tensor. For a spin system with 10 physical spins and measurements of every spin, one input data point will be an array of the form `np.array([1,0,1,1,0,1,0,0,0,1])` , with shape `(10,)`. Here, we take the usual representationw ith 0 denoting spin-down and 1, spin-up (in the z-basis). All the input data together has to be an array of these arrays, which will have the shape `(N,10)`, where N is the number of data elements in the training set.

For the purpose of this tutorial, we will also compare the state reconstuction from QuCumber with the exact ground state. This is also provided in `https://github.com/PIQuIL/QuCumber/blob/master/examples/01_Ising/tfim1d_psi.txt`.

Loading the tutorial data can be done as follows,

```
import numpy as np
import torch
import qucumber.utils.data as data

train_path = "fim1d_train_samples.txt"
psi_path   = "tfim1d_psi.txt"
train, psi = data.load_data(train_path, psi_path)
```

The central object of QuCumber is the representation of the wavefunction, in the case of a positive real wavefunction, this is simply an RBM. This Python object `PositiveWavefunction` serves this purpose. It posseses weights and biases which determine its properties. To instanciate a positive wavefunction, 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. In contrast, the number of hidden units (`num_hidden`) can be varied to change the expressiveness of the neural network. Errors in the representation can be systematically improved by increasing the number of hidden units and consequently the number of parameters (weights and biases) in the network. The quality of the reconstruction will depend on the specific wavefunction and the ratio $\alpha =$ `num_hidden/num_visible`. Typically, in our studies $\alpha = 1$ leads to good approximations of positive real wavefunctions [2]. In the general case, however, the value of α required for a given wavefunction reconstruction should be explored and adjusted by the user.

The following code snippet instanciates a `PositiveWavefunction` object with 10 visible units and a neuron density $\alpha = 1$. The biases are initialized to zero, and the weights are initalized randomly according to a normal distrubution with zero mean and a variance of $1/$`num_visible`.

```
from qucumber.positive_wavefunction import PositiveWavefunction

nn_state = PositiveWavefunction(num_visible=10, num_hidden=10)
```

2.3 Training an RBM

The neural network wavefunction object can be trained with the constrastive divergence algorithm using various optimizers (see `https://pytorch.org/docs/stable/optim.html`). This is implimented with the function `PositiveWavefunction.fit`, which takes a number of hyperparameters. The choice of hyperparameter depends on the complexity of the system and also the training set size. In this example, we take a learning rate of 10^{-2} using stochastic (batch) gradient descent with a batch size of 100, trained for 1,000 epochs. The number of Gibbs sampling steps `k`, which dictates the number of steps in the contrastive divergence algorithm. influences the speed of the training. The smaller we choose k, the faster is the training. High values on the other hand give a better approximation for the model distribution. It has been shown that even for $k_{cd} = 1$ the RBM trains well for many typical cases [9].

```
nn_state.fit(train, epochs=1000, batch_size=100, lr=1e-2, k=10)
```

The success of training can be tracked by different measures, like the convergence of the energy or other observables. In the example of the one-dimensional TIFM, we can compare fidelity of the true ground state with the trained model. Note that this is only possible for very small system sizes, as the computation of the fidelity scale exponetially.

To monitor the fidelity with a target state `psi`, we use the `Callbacks` class which computes desired quantities every n epochs. We set $n = 50$ in this example.

```
from qucumber.callbacks import MetricEvaluator
import qucumber.utils.training_statistics as ts

nn_state.space = nn_state.generate_Hilbert_space(10)
callbacks = [MetricEvaluator(50, {"Fidelity": ts.fidelity, "KL": ts.KL},
                             target_psi=psi)]
```

which outputs

```
Epoch = 50  Fidelity = 0.785383 KL = 0.441264
Epoch = 100 Fidelity = 0.894514 KL = 0.208556
Epoch = 150 Fidelity = 0.935645 KL = 0.125355
Epoch = 200 Fidelity = 0.958977 KL = 0.080373
Epoch = 250 Fidelity = 0.970116 KL = 0.058963
Epoch = 300 Fidelity = 0.977806 KL = 0.043968
Epoch = 350 Fidelity = 0.979086 KL = 0.041768
Epoch = 400 Fidelity = 0.981559 KL = 0.037034
Epoch = 450 Fidelity = 0.985201 KL = 0.029750
Epoch = 500 Fidelity = 0.987439 KL = 0.025278
Epoch = 550 Fidelity = 0.988868 KL = 0.022384
Epoch = 600 Fidelity = 0.991312 KL = 0.017393
Epoch = 650 Fidelity = 0.991913 KL = 0.016160
Epoch = 700 Fidelity = 0.992833 KL = 0.014334
Epoch = 750 Fidelity = 0.992946 KL = 0.014146
Epoch = 800 Fidelity = 0.993429 KL = 0.013104
Epoch = 850 Fidelity = 0.993416 KL = 0.013208
Epoch = 900 Fidelity = 0.994565 KL = 0.010874
Epoch = 950 Fidelity = 0.994806 KL = 0.010380
Epoch = 1000  Fidelity = 0.995174 KL = 0.009641
```

After training is complete, the target distribution is encoded in the parameters of the RBM. The quality of the approximate wavefunction reconstruction depends on many factors, such as the number of hidden units, size of the training set, or errors in the training procedure. Evaluating and quantifying it is the task of the user for his or her given application; we outline some strategies in Section **??**.

2.3.1 Callbacks and monitoring training

So far there is no certainty that the RBM performs well and indeed learns the probability distribution it is supposed to learn. Therefore we introduce the concept of callbacks and the learning rate. The learning rate is crucial for the success of the training and has to be adjusted if the the RBM performs poorly. To tract a certain observable like the energy or the fidelity during the training, one can implement it as a class in the **This should be outdated!!** `observable.py` file and monitor it via callbacks. [I think it's weird to introduce the learning rate here]

```
Show example here
```

The user can track the training progress via these callbacks, for example by checking whether the energy converges to a reasonable value. If the system size is not too big, one can even check whether the approximate RBM probability distribution has high fidelity with the theoretical state that should be learned. **We could add plots with different learning behaviour to give a idea of bad learning rates** The learning rate is a hyperparamter that can be adjusted if the training performance is poor. that's also stated above The lower the learning rate, the slower the training, but the more likely it is to find a good minimum of the objective function. Large learning rates speed up the training and make it more likely to jump out of a local minimum, but they tend not to converge nicely. Good practical values for the learning rates lie typically in the interval $[0.001, 0.1]$.

2.3.2 Monitoring fidelity for small systems

[AN INTRO SENTENCE ABOUT FIDELITY?] If one has access to the full quantum state $|\phi\rangle = \sum_{\boldsymbol{\sigma}} \phi(\boldsymbol{\sigma}) |\boldsymbol{\sigma}\rangle$ and knows the coefficients $\phi(\boldsymbol{\sigma})$ that are approximated by the RBM $\psi_{\boldsymbol{\lambda}}(\boldsymbol{\sigma}) = \sqrt{p_{\boldsymbol{\lambda}}(\boldsymbol{\sigma})}$, the fidelity can be computed via $\sum_{\boldsymbol{\sigma}} \phi(\boldsymbol{\sigma}) \psi_{\boldsymbol{\lambda}}(\boldsymbol{\sigma})$. This can be used to compare the reconstructed wavefunction with the state underlying the measurement

data, for example. Because of the exponential growth of the wavefunction in the number of qubits, this is of course only feasible for small physical systems.

```
import utils.training_statistics as ts
train_stats = ts.TrainingStatistics(train_samples.shape[-1],log_every)
```

To effectively calculate the fidelity with a theoretical (what's a theoretical state?) state $|\phi\rangle$, one needs to load the coefficients $\phi(\boldsymbol{\sigma})$. In this example we create a random state with 2^N coefficients, where $N = 10$ is the system size.

fidelity seems not to work, because partition function is 0.

```
N = 10 # Define system size
phi = np.textload(...) # load wavefunction of TFIM
train_stats.load_target_psi(phi)
train_stats.fidelity(PositiveWavefunction)
fidelity = train_stats.F
```

[Let's load wavefunction of TFIM and calculate fidelity with the trained RBM from TFIM]

2.4 Saving and loading a model

These parameters can be saved to file with

```
location = "some_folder/filename"
nn_state.save(location)
nn_state.load(location)
```

2.5 Sampling from a RBM

RBMs are generative models, which means that they can produce new configurations of visible and hidden units drawn according to the learned joint distribution. These configurations are generated using Block Gibbs Sampling and represent a Markov chain. The number of samples to generate (Gibbs steps) is specified by the variable k:

```
samples = nn_state.sample(samples=1000, k=10)
```

Drawing these samples is the main purpose of generative modelling. They can be used for a variety of tasks defined by the user. For example, they can be used used to calculate estimators for some physical observable.

2.5.1 Observables

Observables like energy, magnetization, or correlation functions can be calculated directly on the sampled data from the RBM. For a general wavefunction Ψ and a general observable (or operator) $\mathcal{O}$ the expectation value of this operator is given by

$$\langle \mathcal{O} \rangle = \frac{\langle \Psi | \mathcal{O} | \Psi \rangle}{\langle \Psi | \Psi \rangle} = \frac{\sum_{\boldsymbol{\sigma}} \langle \Psi | \boldsymbol{\sigma} \rangle \langle \boldsymbol{\sigma} | \mathcal{O} | \Psi \rangle}{\sum_{\boldsymbol{\sigma}} \langle \Psi | \boldsymbol{\sigma} \rangle \langle \boldsymbol{\sigma} | \Psi \rangle} = \frac{\sum_{\boldsymbol{\sigma}} |\langle \Psi | \boldsymbol{\sigma} \rangle|^2 \frac{\langle \boldsymbol{\sigma} | \mathcal{O} | \Psi \rangle}{\langle \Psi | \boldsymbol{\sigma} \rangle}}{\sum_{\boldsymbol{\sigma}} |\langle \Psi | \boldsymbol{\sigma} \rangle|^2} \tag{2}$$

$$= \frac{\sum_{\boldsymbol{\sigma}} |\langle \Psi | \boldsymbol{\sigma} \rangle|^2 \mathcal{O}_L(\boldsymbol{\sigma})}{\sum_{\boldsymbol{\sigma}} |\langle \Psi | \boldsymbol{\sigma} \rangle|^2} \tag{3}$$

Here we have defined the local estimator,

$$\mathcal{O}_L(\boldsymbol{\sigma}) = \frac{\langle \boldsymbol{\sigma} | \mathcal{O} | \Psi \rangle}{\langle \Psi | \boldsymbol{\sigma} \rangle}. \tag{4}$$

Considering that

$$\mathcal{P}(\boldsymbol{\sigma}) = \frac{|\langle\Psi|\boldsymbol{\sigma}\rangle|^2}{\sum_{\boldsymbol{\sigma}}|\langle\Psi|\boldsymbol{\sigma}\rangle|^2} \tag{5}$$

is a probability distribution, the calculation of an expectation value of an operator $\mathcal{O}$ can be simplified into the calculation of the average of the local random variable $\mathcal{O}_L(\boldsymbol{\sigma})$ over the distribution $\mathcal{P}(\boldsymbol{\sigma})$. Thus, if we sample from the RBM using Gibbs sampling, where the probability distribution $\mathcal{P}(\boldsymbol{\sigma})$ is the learned joint distribution, we can evaluate $\langle\mathcal{O}\rangle$ by computing the mean of the random variable $\mathcal{O}_L(\boldsymbol{\sigma})$ over the sampled configurations $\boldsymbol{\sigma}^{(k)}$:

$$\langle\mathcal{O}\rangle \approx \frac{1}{N}\sum_{k=1}^{N}\mathcal{O}_L(\boldsymbol{\sigma}^{(k)}). \tag{6}$$

[CHECK definitions of k, N, and M below... Also need to crosscheck with code. What it really returns.] Most observables depend on the physical system under study, e.g. the Hamiltonian that underlies the original model, the lattice connectivity, etc. Therefore, we now turn to a discussion of a specific model example. In Section 2.1 we trained an RBM on the TFIM data. Now we will show how to calculate several quantities of interest from the learned distribution.

2.5.2 Magnetization

The magnetization of an Ising chain can be calculated simply by averaging over all σ^z measurements of the chain. Since the RBM can only sample in the σ^z direction, we have to average over the sample $\boldsymbol{\sigma}$. The only additional step that has to be performed is the convergence from binary values $\{0,1\}$ to actual spin measurements $\{-1,1\}$. [I don't understand this paragraph]

```
magnetization = samples.mean()
```

2.5.3 Energy using Observables module

For the transverse field Ising model a standard observable is the energy, obtained as the expectation value of the Hamiltonian operator 1. Calculating the energy for a sample $|\boldsymbol{\sigma}\rangle = |\sigma_1 \ldots \sigma_n\rangle$ is not as straightforward as for the magnetization, because the Hamiltonian operator σ_i^x is off-diagonal in the computational basis. Just calculating $\langle\boldsymbol{\sigma}|H|\boldsymbol{\sigma}\rangle$ ignores the second part of the Hamiltonian completely. If we write the wavefunction in its expanded form, $|\psi\rangle = \sum_{\boldsymbol{\sigma}}\psi(\boldsymbol{\sigma})|\boldsymbol{\sigma}\rangle$, we calculate the expectation value of the Hamiltonian using the local observable,

$$\langle H\rangle_L(\boldsymbol{\sigma}) = \sum_{\boldsymbol{\sigma'}} H_{\boldsymbol{\sigma},\boldsymbol{\sigma'}}\cdot\psi(\boldsymbol{\sigma'})/\psi(\boldsymbol{\sigma}) \tag{7}$$

$$= \left[-\sum_i \sigma_i\sigma_{i+1} - h\sum_i \frac{\psi(\sigma_1,\sigma_2,\ldots,-\sigma_i,\sigma_{i+1}\ldots)}{\psi(\boldsymbol{\sigma})}\right] \tag{8}$$

giving an estimator

$$\langle H\rangle \approx \frac{1}{M}\sum_{k}^{M}\left[-\sum_i \sigma_i\sigma_{i+1} - h\sum_i \frac{\psi(\boldsymbol{\sigma}_{-i})}{\psi(\boldsymbol{\sigma})}\right]. \tag{9}$$

Here we used that $H_{\boldsymbol{\sigma},\boldsymbol{\sigma'}} = \langle\boldsymbol{\sigma}|H|\boldsymbol{\sigma'}\rangle = -\sum_i \delta_{\boldsymbol{\sigma},\boldsymbol{\sigma'}} - h\sum_i \delta_{\sigma_1,\sigma_1'}\ldots\delta_{\sigma_i,-\sigma_i'}\ldots$, and defined $\boldsymbol{\sigma}_{-i} = \sigma_1,\sigma_2,\ldots,-\sigma_i,\sigma_{i+1}\ldots$, which is the sample $\boldsymbol{\sigma}$ with the i-th spin flipped.

The sum $\sum_i \sigma_i \sigma_{i+1}$ simply multiplies the neighbouring elements of the sample $\boldsymbol{\sigma}$. For the second sum one actually has to calculate the probabilities $\psi(\boldsymbol{\sigma})$ of the sample $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma}_{-i}$ with Eq. **??**, with $p_\lambda(\boldsymbol{\sigma})$ from Eq. 27. The partition functions Z cancel out if we divide the probabilities and the calculation is tractable also for large system sizes. The `observable.py` package contains a few examples like this.

```
sys.path.append('../examples/')
from observables import quantum_ising_chain as TFIM
# TFIM stands for transverse field ising model
n_measurements= 50 # number of measurements
hx=1.0        # Magnetic field
tfim = TFIM.TransverseFieldIsingChain(hx,n_measurements)
simulation = tfim.Run(nn_state,n_eq=200) # run

Energy = simulation['energy']
```

3 Complex wavefunctions

For the positive-real wavefunctions there is no additional information contained in the wavefunction $\psi(\boldsymbol{\sigma})$ that could be lost by Borns rule $q(\boldsymbol{\sigma}) = |\psi(\boldsymbol{\sigma})|^2$. For negative-real and complex functions the phase information contained in these coefficients will be lost if the measurement is made in only one basis. Therefore, in experiments one has to apply local unitary transformations before the measurements to capture the phase information. Given this additional information one can successfully perform quantum state reconstruction. QuCumber also provides the possibility to do quantum state reconstruction for complex wavefunctions.

For complex wavefunctions, additionally to the marginal distribution (Eq. 27) we require a phase factor that has to be learned by an additional set of hidden units and corresponding network parameters $\boldsymbol{\mu}$ of the RBM:

$$\psi_{\boldsymbol{\lambda\mu}}(\boldsymbol{\sigma}) = \sqrt{p_{\boldsymbol{\lambda}}(\boldsymbol{\sigma})} e^{i\phi_{\boldsymbol{\mu}}(\boldsymbol{\sigma})/2}, \tag{10}$$

where $\phi_{\boldsymbol{\mu}}(\boldsymbol{\sigma}) = \log(p_{\boldsymbol{\mu}}(\boldsymbol{\sigma}))$. The complex wavefunction can be rewritten in a generic basis $\{\boldsymbol{\sigma}^b\}$as :

$$\psi_{\boldsymbol{\lambda\mu}}(\boldsymbol{\sigma}^b) = \sum_{\boldsymbol{\sigma}} U(\boldsymbol{\sigma}^b, \boldsymbol{\sigma}) \psi_{\boldsymbol{\lambda\mu}}(\boldsymbol{\sigma}), \tag{11}$$

with the unitary $U(\boldsymbol{\sigma}^b, \boldsymbol{\sigma})$ that rotates the state $|\boldsymbol{\sigma}^b\rangle$ to $|\boldsymbol{\sigma}\rangle$.

Our objective is to learn the full complex wavefunction $\psi_{\boldsymbol{\lambda\mu}}(\boldsymbol{\sigma})$ in the reference basis (in this case, the computational basis) from the measurements of the rotated wavefunction $\psi_{\boldsymbol{\lambda\mu}}(\boldsymbol{\sigma}^b)$. The RBM again is trained by minimizing the negative log-likelihood (NLL). Following [] **ADD REFERENCE TO GIACOMOS THESIS OR SOMETHING THAT SHOWS THE MATH OF THE COMPLEX WAVEFUNCTION**, the gradients of the NLL will contain the unitaries $U(\boldsymbol{\sigma}^b, \boldsymbol{\sigma})$, and therefore we have to apply a slightly different learning algorithm.

3.1 A two-qubit example

In this section we work through a simple example for full quantum state tomography of a complex wavefunction with two qubits. The data set comprises the measurements

'qubits_train_samples.txt', the unitaries that have been applied before the measurement 'qubits_train_bases.txt' and the actual wavefunction $|\psi\rangle$ the measurements have been sampled from 'qubits_psi.txt'.

Load the files the following way:

```
from complex_wavefunction import ComplexWavefunction
from quantum_reconstruction import QuantumReconstruction
import unitaries
import utils.training_statistics as ts
train_samples = np.loadtxt('qubits_train_samples.txt', dtype= 'float32')
target_psi = torch.tensor(np.loadtxt('qubits_psi.txt', dtype= 'float32'),
                                    dtype=torch.double, device = torch.
                                   device('cpu'))
train_bases = np.loadtxt('qubits_train_bases.txt',dtype=str)
unitary_dict = unitaries.create_dict() # creates dictionary with standard
                                    unitaries I, H and K for basis Z, X,
                                    Y
```

We want to do tomography of the two-qubit state $|\psi\rangle = 1/2\{|00\rangle - |01\rangle + |10\rangle - i\,|11\rangle\}$, which is the state in target_psi. [Say that this is not the state in the file or generate test example for this state] If we measure this state in the computational basis Z [previously the comp basis was denoted differently], we obtain all states ($|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$) with the same probability $p = 1/4$, but we do not learn anything about the phase i or (-1). Therefore, we apply the local unitaries

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix},\ K = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & -i\\ 1 & i\end{bmatrix} \tag{12}$$

[I think H was used for sth else before] on the state before we measure in the computational basis. Measuring the state in the computational basis is equivalent to applying the identity $\mathbb{1}$ on the qubits, measuring in the X basis is equivalent to applying a unitary H on the qubit and then measure in Z basis, measuring in Y basis is equivalent to applying a K and then measure in Z basis. Because of the application of local unitaries the probability amplitudes of certain measurement outcomes start mixing and therefore we can extract more information about them. If one applies for example a Hadamard gate H on the second qubit of the state $|\psi\rangle$, we obtain the state:

$$\mathbb{1}\otimes H\,|\psi\rangle = \frac{1}{2}\{|0+\rangle - |0-\rangle + |1+\rangle - i\,|1-\rangle\} \tag{13}$$

$$= \frac{1}{2\sqrt{2}}\{2\,|01\rangle + (1-i)\,|10\rangle + (1+i)\,|11\rangle\}. \tag{14}$$

If we measure this state in the Z basis, we obtain the state $|01\rangle$ with probability $p(|01\rangle) = 1/2$ and the other two states with $p(|10\rangle) = p(|11\rangle) = 1/4$. The state $|00\rangle$ will never be measured. If we repeat the last step with a Hadamard gate on the first qubit, we obtain:

$$H\otimes\mathbb{1}\,|\psi\rangle = \frac{1}{2}\{|+0\rangle - |+1\rangle + |-0\rangle - i\,|-1\rangle\} \tag{15}$$

$$= \frac{1}{2\sqrt{2}}\{2\,|00\rangle + (1-i)\,|01\rangle - (1-i)\,|11\rangle\}. \tag{16}$$

Now $p(|00\rangle) = 1/2$, $p(|01\rangle) = p(|11\rangle) = 1/4$ and $p(|10\rangle) = 0$. We can compare these results to an arbitrary two-qubit state $|\psi\rangle_c = c_1\,|00\rangle + c_2\,|01\rangle + c_3\,|10\rangle + c_4\,|11\rangle$, with all

the unitary transformations

$$\mathbb{1} \otimes H \left|\psi\right\rangle_c = \frac{1}{2\sqrt{2}}\{(c_1 + c_2)\left|00\right\rangle + (c_1 - c_2)\left|01\right\rangle + (c_3 + c_4)\left|10\right\rangle + (c_3 - c_4)\left|11\right\rangle\} \quad (17)$$

$$H \otimes \mathbb{1} \left|\psi\right\rangle_c = \frac{1}{2\sqrt{2}}\{(c_1 + c_3)\left|00\right\rangle + (c_1 - c_3)\left|01\right\rangle + (c_2 + c_4)\left|10\right\rangle + (c_2 - c_4)\left|11\right\rangle\}. \quad (18)$$

From the measurement without unitaries we know that all the probabilities are the same, which means $c_i \in \{\pm 1/2, \pm i/2\}$. From the application of H to the second qubit we know that $c_1 = -c_2$ and $c_3 \neq \pm c_4$. From the application of H to the first qubit we know that $c_1 = c_3$ and $c_2 \neq \pm c_4$. Already from this information we can do almost full tomography of the state $|\psi\rangle$. If we fix $c_1 = 1/2$, which just defines a global phase, we find that $c_3 = 1/2$ and $c_2 = -1/2$. We also find that $c_4 = \pm i/2$. The only piece that is missing is the sign of c_4. To find this sign we apply the unitary K on the second qubit:

$$\mathbb{1} \otimes K \left|\psi\right\rangle = \frac{1}{2}\{\left|0+\right\rangle + i\left|0-\right\rangle + \left|1+\right\rangle - \left|1-\right\rangle\} \quad (19)$$

$$= \frac{1}{2\sqrt{2}}\{2\left|11\right\rangle + (1 - i)\left|01\right\rangle + (1 + i)\left|00\right\rangle\}. \quad (20)$$

and compare it to the arbitrary state

$$\mathbb{1} \otimes K \left|\psi\right\rangle_c = \frac{1}{2\sqrt{2}}\{(c_1 - ic_2)\left|00\right\rangle + (c_1 + ic_2)\left|01\right\rangle + (c_3 - ic_4)\left|10\right\rangle + (c_3 + ic_4)\left|11\right\rangle\}, \quad (21)$$

where we find that $c_3 = ic_4$. Therefore, $c_4 = -i$. Finding the full state $|\psi\rangle$ with this set of unitaries shows that it is a complete set of unitaries. It might very well be that the complete set also contains the unitary $K \otimes \mathbb{1}$, if the amplitudes of the coefficients were not all the same ($|c_i|^2 \neq 1/4$).

The training set was measured in the overcomplete basis $\{ZZ, ZX, XZ, ZY, YZ\}$, and therefore `train_bases` contains the unitaries applied to the respective measurement and has the form `np.array([['Z','Z'], ['X','Z'], ['Z','X'], ...])`. This means that the first measurement has been done in the Z basis for both qubits, and therefore no unitary has been applied. The second measurement has been done in the XZ basis, which means that the first qubit has been measured in the X basis. The training of the RBM is analogous to the case of positive-real wavefunction. We define the training parameters,

```
epochs    = 200
num_chains = 10
batch_size = 10
k      = 10
lr      = 0.1
log_every = 10
```

and start the training:

```
nn_state = ComplexWavefunction(num_visible=nv,num_hidden=nh)
QR = QuantumReconstruction(nn_state)
train_stats = ts.TrainingStatistics(train_samples.shape[-1],frequency=1)
train_stats.load(bases = bases,target_psi_dict = target_psi_dict)
QR.fit(train_samples, epochs, batch_size, num_chains, k,
lr,train_bases, progbar=False,observer=train_stats)
```

After the training we can calculate state fidelity, observables or sample from the complex wavefunction the same way we did from the real-positive wavefunction. However, one has to keep in mind that the sampling only works in the Z basis.

3.2 Defining general unitaries

QuCumber contains by default the unitaries $\mathbb{1}$, H and K, with:

$$\mathbb{1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \; H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \; K = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -i \\ 1 & i \end{bmatrix} \tag{22}$$

New basis transformations can be added in the following way:

```
from qucumber import unitary
import numpy as np
new_unitary
unitary_dict = unitaries.create_dict(name = 'new_unitary', unitary =
                                     new_unitary)
```

This creates a new unitary instance which is called `'new_unitary'` and connects this name with the `numpy` array `new_unitary` that has the form `A = np.array([a,b],[c,d]])` for a matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}. \tag{23}$$

To call this basis transformation during the training, an element of the `train_bases` array from above has to have the form `np.array(['Z','new_unitary'])` for a two-qubit example, which means before the measurement the `new_unitary` was applied to the second qubit.

So far we only used maximally one local unitary that is not the identity, such that maximally one qubit was not measured in the `'Z'` basis. QuCumber provides also the possibility for an arbitrary number of non-trivial unitaries per measurement. An element of `train_bases`, for example, for a six-qubit state could have the following form: `np.array(['X','Z','new_unitary','Z','Y','X'])`

Apart from the training there is no difference in the positiv-real and the complex wavefunctions. The observables and callbacks are caculated as described in Section 2.5 and 2.3.1. The sampling from the RBM is also exactly the same for both cases. In the complex case, sampling from the RBM can only be performed in the original basis, which in our case was the Z basis. [what's correct? Double check this with code]

4 Conclusion

We introduced the open-source package QuCumber for quantum state tomography with Restricted Boltzmann Machines and demonstrated on examples that the package is applicable on positive-real and complex wavefunctions for any possible physical system with binary measurement outputs. The class provided for the case of positive-real wavefunctions is highly parallelizable on GPUs and exhibits very high performance. For complex wavefunctions QuCumber provides standard local unitaries for basis transformations and can easily be equipped with customized unitaries, and therefore applied on any system with binary measurement outcomes. After successful training of the RBM, one has full access to the wavefunction, respectively to the probability distribution of the measurements of the system. One can sample new measurements, calculate observables or the fidelity to any other state. We provide example code for any of these tasks to give the user a first impression of the implementation of

In the future QuCumber will be extended to the application of quantum state tomography on mixed states, multinomial quantum systems (like boson systems) and eventually on continuous variable systems.

Acknowledgements

We thank G. Carleo, J. Carrasquilla and L. Hayward Sierens for stimulating discussions. PIQuIL is supported by the Perimeter Institute for Theoretical Physics.

Author contributions Authors are listed alphabetically. For an updated record of individual contributions, consult the repository at `https://github.com/PIQuIL/QuCumber/graphs/contributors`.

Funding information P.H. acknowledges support from ICFOstepstone - PhD Programme for Early-Stage Researchers in Photonics, funded by the Marie Sklodowska-Curie Co-funding of regional, national and international programmes (GA665884) of the European Commission, as well as by the Severo Ochoa 2016-2019' program at ICFO (SEV-2015-0522), funded by the Spanish Ministry of Economy, Industry, and Competitiveness (MINECO). R.G.M. is supported in part by funding from the Natural Sciences and Engineering Research Council of Canada (NSERC) and a Canada Research Chair. Research at Perimeter Institute is supported by the Government of Canada through Industry Canada and by the Province of Ontario through the Ministry of Research & Innovation.

A Glossary

We list an overview of terms discussed in the document and relevant for RBMs. For more detail we refer to the code documentation on `https://piquil.github.io/QuCumber/`, and References [9, 10].

- *Batch*: The subset of data selected for one iteration of training in stochastic gradient descent.
- *Biases*: For a visible unit v_j and a hidden unit h_i, the respective biases in the RBM energy are b_j and c_i. They act like a magnetic field term in the energy Eq (24).
- *Contrastive Divergence*: An approximate maximum-likelihood learning algorithm for RBMs [9].
- *Energy*: In analogy to statistical physics the energy of an RBM is defined given the joint configuration (v, h) of visible and hidden units:

$$E_{\boldsymbol{\lambda}}(v,h) = -\sum_{j=1}^{V} b_j v_j - \sum_{i=1}^{H} c_i h_i - \sum_{ij} h_i W_{ij} v_j, \tag{24}$$

- *Effective energy*: Energy traced over the hidden units h:

$$\mathcal{E}_{\boldsymbol{\lambda}}(v) = -\sum_{j=1}^{V} b_j v_j - \sum_{i=1}^{H} \log\left\{1 + \exp\left(\sum_j W_{ij} v_j + c_i\right)\right\}, \tag{25}$$

- *Epoch*: A single pass through an entire training set of data.
- *Hidden Units*: There are H units in the second layer of the RBM, denoted by the vector $h = (h_1, ..., h_H)$, representing latent variables and are referred to as "hidden". The number of hidden units H can be adjusted to tune the representational capacity of the RBM.
- *Joint distribution*: The RBM assigns a probability to each joint configuration (v, h) according to the Boltzmann distribution of the energy,

$$p_{\boldsymbol{\lambda}}(v, h) = \frac{1}{Z_{\boldsymbol{\lambda}}} e^{-E_{\boldsymbol{\lambda}}(v,h)}, \tag{26}$$

- *Marginal distribution*: Obtained by marginalizing the joint distribution, e.g.

$$p_{\boldsymbol{\lambda}}(v) = \frac{1}{Z_{\boldsymbol{\lambda}}} \sum_{h \in \mathcal{H}} e^{-E_{\boldsymbol{\lambda}}(v,h)} = \frac{1}{Z_{\boldsymbol{\lambda}}} e^{-\mathcal{E}_{\boldsymbol{\lambda}}(v)}. \tag{27}$$

- *QuCumber*: A quantum calculator used for many-body eigenstate reconstruction.
- *Parameters*: An RBM's energy is defined via a set of neural network parameters $\boldsymbol{\lambda} = \{b, c, W\}$, consisting of weights and biases.
- *Partition function*: The normalizing constant of the Boltzmann distribution. It is obtained by summing over all possible pairs of visible and hidden vectors,

$$Z_{\boldsymbol{\lambda}} = \sum_{v \in \mathcal{V}} \sum_{h \in \mathcal{H}} e^{-E_{\boldsymbol{\lambda}}(v,h)}. \tag{28}$$

- *Restricted Boltzmann Machine*: A two-layer network with bidirectionally connected stochastic processing units. "Restricted" refers to the connections (or weights) between the visible and hidden units. Each visible unit is connected with each hidden unit, but there are no intra-layer connections.
- *Visible Units*: There are V units in the first layer of the RBM, denoted by the vector $v = (v_1, ..., v_V)$, which correspond to the experimental data and are therefore called "visible". The number of visible units V is fixed to the number of physical qubits.
- *Weights*: W_{ij} is the symmetric connection or interaction between the visible unit v_j and the hidden unit h_i.

B Algorithm for a real positive wave function

The training algorithm of the RBM has the following structure.

Algorithm 1: Training Algorithm of QuantumReconstruction. **QR.train**()

for *batch in training set* **do**
  Load batch from training set, batch = $(\hat{\boldsymbol{\sigma}}_1\ \hat{\boldsymbol{\sigma}}_2 \dots)$;
  compute the gradients from the batch $\Delta\boldsymbol{\Theta} = (\Delta W, \Delta b, \Delta c)$
  QR.compute_batch_gradients(k, batch, basis) ; ▷ `Algorithm 2`
  update weights and biases
  $\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \Delta\boldsymbol{\Theta}$;
end

The gradients are calculated according to the contrastive divergence algorithm, which allows us to approximate the probability distribution of the model with k Gibbs sampling steps from the actual training data.

Algorithm 2: Compute Gradient from Batch. **QR.compute_batch_gradients**(k, batch, basis)

if *basis = None* **then**
  Reset gradients ΔW, Δh_b, $\Delta v_b = 0$;
  for $\hat{\boldsymbol{\sigma}}_i$ *in batch* **do**
    sample $\boldsymbol{h}_0$, $\boldsymbol{v}_k$ and $\boldsymbol{p}_{h_k}$ from $\hat{\boldsymbol{\sigma}}_i$
    RBM.gibbs_sampling(k, $\hat{\boldsymbol{\sigma}}_i$) ; ▷ Algorithm 3
    calculate gradients
    $\Delta W+ = \boldsymbol{v}_0\boldsymbol{h}_0^T - \boldsymbol{v}_k\boldsymbol{p}_{h_k}^T$
    $\Delta c+ = \boldsymbol{h}_0 - \boldsymbol{p}_{h_k}$
    $\Delta b+ = \boldsymbol{v}_0 - \boldsymbol{v}_k$;
  end
  $M = |batch|$;
  return $\Delta W/M$, $\Delta c/M$, $\Delta b/M$;
else
  Do complex gradient ; ▷ Algorithm 6
end

The Gibbs sampling is done k times back and forth. The contrastive divergence algorithm already shows good results for $k = 1$. But for better results this value can be increased.

Algorithm 3: Gibbs sampling. **RBM.gibbs_sampling**(k, $\hat{\boldsymbol{\sigma}}_i$)

calculate $\boldsymbol{p}_h$ and sample $\boldsymbol{h}_0$ from $\hat{\boldsymbol{\sigma}}_i$
RBM.sample_h_given_v($\hat{\boldsymbol{\sigma}}$) ; ▷ Algorithm 4
$\boldsymbol{h} = \boldsymbol{h}_0$;
i = 0;
while $i \leq k$ **do**
  calculate $\boldsymbol{p}(\boldsymbol{v}|\boldsymbol{h}_i)$ and sample $\boldsymbol{v}$ from $\boldsymbol{h}$
  RBM.sample_v_given_h($\boldsymbol{h}$) ; ▷ Algorithm 5
  calculate $\boldsymbol{p}(\boldsymbol{h}|\boldsymbol{v})$ and sample $\boldsymbol{h}$ from $\boldsymbol{v}$
  RBM.sample_h_given_v($\boldsymbol{v}$) ; ▷ Algorithm 4
  $i+ = 1$
end
return $\boldsymbol{p}_{h_k} = \boldsymbol{p}(\boldsymbol{h}|\boldsymbol{v})$, $\boldsymbol{v}_k = \boldsymbol{v}$ and $\boldsymbol{h}_0$;

Algorithm 4: calculate $\boldsymbol{p}(\boldsymbol{v}|\boldsymbol{h})$ and sample $\boldsymbol{v}$ **RBM.v_given_h**()

calculate probability $\boldsymbol{p}(\boldsymbol{v} = 1|\boldsymbol{h}) = \sigma(W\boldsymbol{h} + v_b)$;
Bernoulli sample $\boldsymbol{v}$ from this probability;
return $\boldsymbol{v}$ and $\boldsymbol{p}(\boldsymbol{v} = 1|\boldsymbol{h})$;

Algorithm 5: calculate $\boldsymbol{p}(\boldsymbol{h}|\boldsymbol{v})$ and sample $\boldsymbol{h}$ **RBM.h_given_v**()

calculate probability $\boldsymbol{p}(\boldsymbol{h} = 1|\boldsymbol{v}) = \sigma(\boldsymbol{v}^T W + h_b)$;
Bernoulli sample $\boldsymbol{h}$ from this probability;
return $\boldsymbol{h}$ and $\boldsymbol{p}(\boldsymbol{h} = 1|\boldsymbol{v})$;

References

[1] P. Smolensky, *Information processing in dynamical systems: Foundations of harmony theory*, In D. E. Rumelhart, J. L. McClelland and C. PDP Research Group, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, chap. 6, p. 194–281. MIT Press, Cambridge, MA, USA, ISBN 0-262-68053-X (1986).

[2] G. Torlai and R. G. Melko, *Learning thermodynamics with boltzmann machines*, Phys. Rev. B **94**, 165134 (2016), doi:10.1103/PhysRevB.94.165134.

[3] G. Carleo and M. Troyer, *Solving the quantum many-body problem with artificial neural networks*, Science **355**(6325), 602 (2017), doi:10.1126/science.aag2302, `http://science.sciencemag.org/content/355/6325/602.full.pdf`.

[4] J. Chen, S. Cheng, H. Xie, L. Wang and T. Xiang, *Equivalence of restricted boltzmann machines and tensor network states*, Phys. Rev. B **97**, 085104 (2018), doi:10.1103/PhysRevB.97.085104.

[5] I. Glasser, N. Pancotti, M. August, I. D. Rodriguez and J. I. Cirac, *Neural-network quantum states, string-bond states, and chiral topological states*, Phys. Rev. X **8**, 011006 (2018), doi:10.1103/PhysRevX.8.011006.

[6] G. Torlai, G. Mazzola, J. Carrasquilla, M. Troyer, R. Melko and G. Carleo, *Neural-network quantum state tomography*, Nature Physics **14**(5), 447 (2018), doi:10.1038/s41567-018-0048-5.

[7] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga and A. Lerer, *Automatic differentiation in pytorch*, In *NIPS-W* (2017).

[8] *http://itensor.org/* Calculations were performed using the ITensor Library (version 0.2.4).

[9] G. E. Hinton, *Training products of experts by minimizing contrastive divergence*, Neural computation **14**(8), 1771 (2002), doi:10.1162/089976602760128018.

[10] G. E. Hinton, *A practical guide to training restricted boltzmann machines*, In *Neural networks: Tricks of the trade*, pp. 599–619. Springer, doi:10.1007/978-3-642-35289-8_32 (2012).