# 1 What can I use this RBM Quantum State Tomography Library for?

# 2 Restricted Boltzmann Machine for Quantum State Tomography

This tomography package allows to approximate a wave function $\psi(\boldsymbol{\sigma}) = \langle \Psi | \boldsymbol{\sigma} \rangle$ in the reference basis $\{|\boldsymbol{\sigma}\rangle\}$ from experimental data. A so called Restricted Boltzmann Machine (RBM) can learn the distribution of the experimental data (e.g. qubit measurements $|\widehat{\boldsymbol{\sigma}}\rangle = |\hat{\sigma}_1 \ \hat{\sigma}_2 \dots \rangle$) and allows after training to sample from it.

This file is a walk through for our code. For more theoretical background we refer to the following tutorials: **ADD some refrences or Anna's File**

# 3 How to use NNQuST

We will guide the user through short snippets of our code. First we load all the packages needed. **We might have to rename rbm to something like NNQuST**

## 3.1 Training the NNQuST

```
from rbm import RBM #import NNQuST package
import numpy as np #generic math function
import torch #ML package
```

Prepare the input data. The input data needs to be in a numpy array or a torch tensor. E.g. 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,)`. all the input data together needs to be an array of these arrays, which will have the shape `(N,10)`. Where N is the number of data points in the training set.

For a dummy test we can define a training set:

```
train_set = np.array([[1]*10]*1000) #define a dummy training set with the correct
                                    input shape
```

Define model parameters:

```
num_visible = 10 #number of visible units (has to be equal to input data dimension)
num_hidden = 10 #number of hidden units
epochs = 1000 #number of epochs for the training
batch_size = 32 #batch size for the training
k = 1 #Number of Gibbs sampling steps
learning_rate = 0.01 #Learning Rate
```

**Get rid of SEED!!! it is already in ToDo list on Github**

```
rbm = RBM(num_visible=num_visible, num_hidden=num_hidden, seed=seed)

rbm.train(train_set, epochs, batch_size, k=k, lr=learning_rate)
```

This will already train the RBM.

## 3.2 Saving, Loading the NNQuST

```
location = 'some_folder/filename'
rbm.save(location) #save the weights and biases to some location
rbm.load(location) #load the weights and biases from some location
```

# 4 What can I do with a trained RBM?

## 4.1 Sampling

```
num_samples = 100 #number of samples drawn from the RBM
k = 100 #number of Gibbs steps for each sample
rbm.sample(num_samples, k)
```

## 4.2 Observables

## 4.3 Fidelity

If one has access to the wave function $\psi(\boldsymbol{\sigma})$ that is approximated by the RBM $\psi_\Theta(\boldsymbol{\sigma})$ the fidelity can be computed via $\sum_{\boldsymbol{\sigma}} \psi(\boldsymbol{\sigma})\psi_\Theta(\boldsymbol{\sigma})$. This can be used to verify for example that experimental data actually comes from the state that should be prepared in an experiment. Because of the exponential growth of the wave function this is only feasible for small physical systems.

# 5 Algorithm

The training algorithm of the RBM has the following structure.

**Algorithm 1:** Training Algorithm of RBM. **RBM.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 $\Delta\boldsymbol{\Theta} = (\Delta W, \Delta b, \Delta c)$
  **RBM.compute_batch_gradients**(k, batch) ; ▷ 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. **RBM.compute_batch_gradients**(k, batch)

Reset gradients $\Delta W$, $\Delta h_b$, $\Delta v_b = 0$;
**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 $\Delta W/M$, $\Delta c/M$, $\Delta b/M$

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 $\boldsymbol{p}(\boldsymbol{h} = 1|\boldsymbol{v})$;

# 6 Algorithm example

**Algorithm 6:** Training Algorithm of RBM. **RBM.train**()

**Data:** this text
**Result:** how to write algorithm with LaTeX2e
initialization;
**while** *not at end of this document* **do**
  read current;
  **if** *understand* **then**
    go to next section ; ▷ `Some comment`
    current section becomes this one;
  **else**
    go back to the beginning of current section;
  **end**
**end**