Quantum Neural Network States and Their Applications

Lu Wei, Tengyue Wang, Luofan Chen

School of the Gifted Young, School of Physics,
University of Science and Technology of China

May 8, 2021

Overview

Main challenge of quantum many-body system

- N particle system: Hilbert space $\mathcal{H} = \otimes_{i=1}^{N} \mathcal{H}_i$ and Hamiltonian $H = \sum_i H_i + H_{int}$, where H_{int} is the interaction term.
- The wave function $\Psi(x_1, \cdots, x_N; t)$ is an N variable function.
- Schrödinger equation $i\partial_t \Psi = H\Psi$. For stationary case $\Psi(x_1, \cdots, x_N; t) = \sum_n \psi_n(x_1, \cdots, x_N) e^{-i\frac{E_n t}{\hbar}}$, stationary Schödinger equation

$$\boxed{H\psi(x_1, \cdots, x_N) = E_n \psi(x_1, \cdots, x_N)} \tag{1}$$

  N is usually very large ($\sim 10^{23}$).
- Complexity comes from too many of degrees of freedom.

Quantum ansatz states

Mean-field method

To reduce the complexity of the problem, physical consideration: symmetries, identical particles, etc. $\Longrightarrow$ simplified equation, e.g., single particle approximation, $H\phi = \varepsilon_n \phi \Longrightarrow$ approximative quantum states $\psi(x_1, \cdots, x_N) = \phi_1(x_1) \cdots \phi_N(x_N)$.

Variational method

To obtain the ground state ψ_0 of Hamiltonian H, choose a function $\psi(x_1, \cdots, x_N; \alpha_1, \cdots, \alpha_k)$ with undetermined parameters $\alpha_1, \cdots, \alpha_k$, calculate the functional

$$E(\alpha_1, \cdots, \alpha_k) = \frac{\langle \psi | H | \psi \rangle}{\langle \psi | \psi \rangle}. \tag{2}$$

find the corresponding values of $\alpha_1, \cdots, \alpha_k$ which minimize the E.

Quantum ansatz states

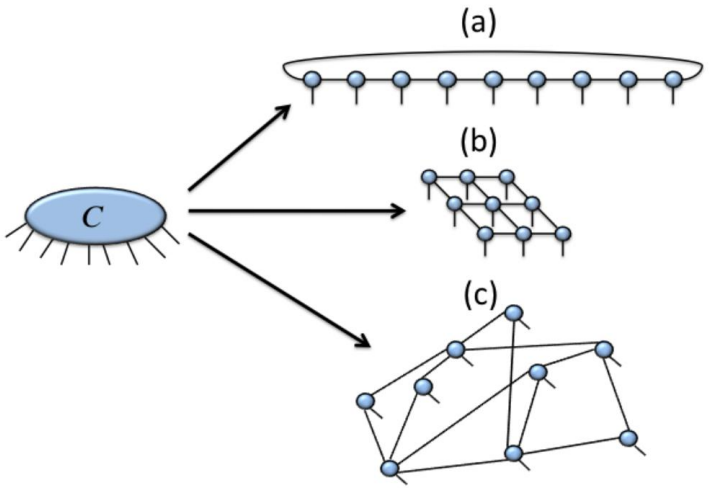
Different method share a common point: pre-assume the wave functions have a special form. (a guess-work depends on experience?)

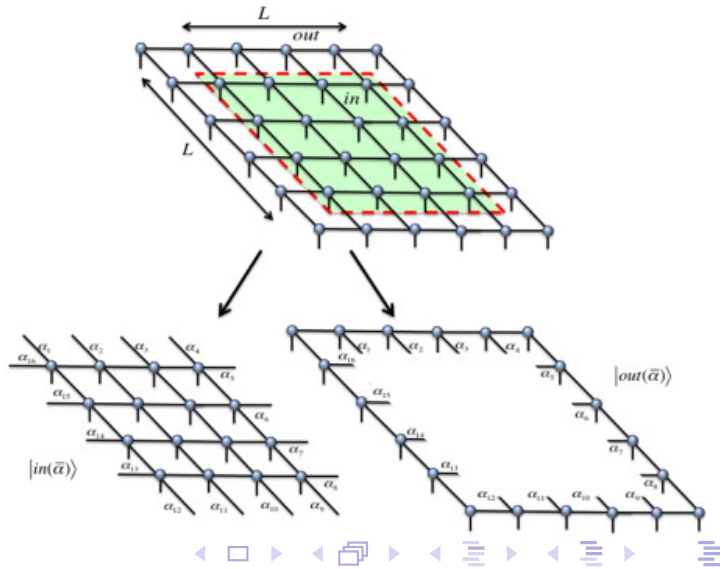
- Mean-field approach (for weakly interacting system), ansatz states are product state $\psi(x_1, \cdots, x_N) = \phi_1(x_1) \cdots \phi_N(x)_N$, also known as Hartree-Fock approximation, the validity is guaranteed by the quantum de Finetti theorem;
- local correlator product state (e.g., $\psi(x_1, \cdots, x_N) = \phi_1(x_1, x_2)\phi_2(x_2, x_3) \cdots$);
- Laughlin states, Moore-Read states for quantum Hall system;
- Jastrow state; String-bond state; RVB state; AKLT state; etc.
- Tensor network states, including MPS, PEPS, etc.
- **Neural network states**. (we are here!)

Tensor network ansatz states

Tensor network states is extensively exploited, the advantage is that entanglement is easy to read out in this representation.

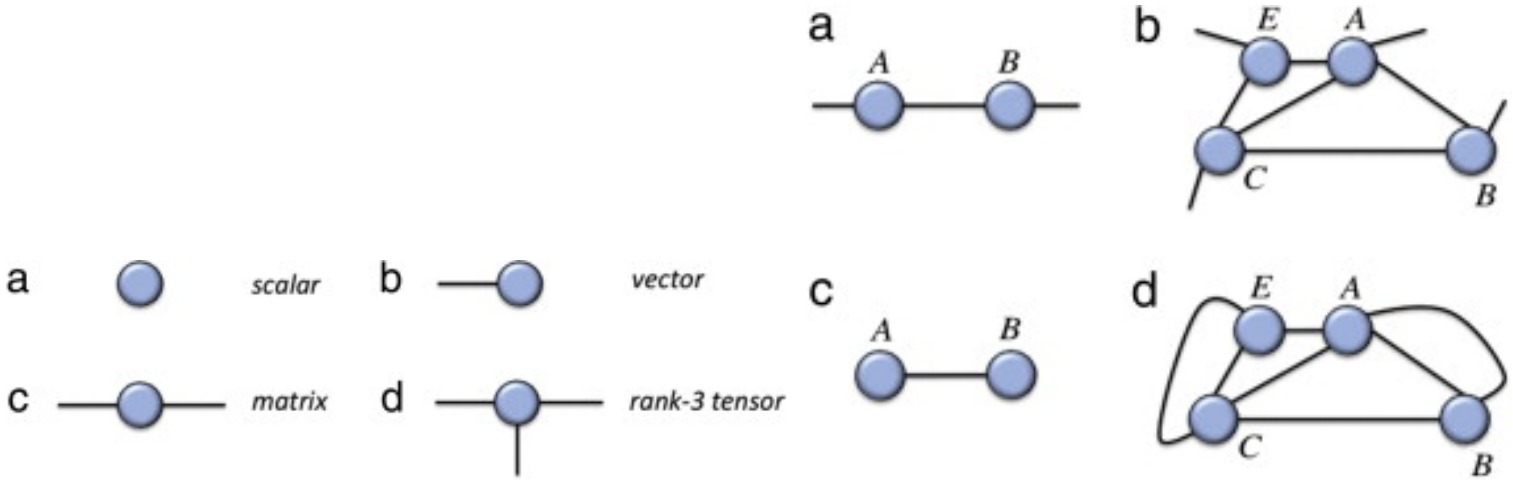
- For a state $|\psi\rangle = \sum_{i_1,\cdots,i_N} c_{i_1\cdots i_N} |i_1\rangle \otimes \cdots \otimes |i_N\rangle$, a tensor network represetation of a state is to divide a coefficient tensor $c_{i_1\cdots i_N}$ into some small pieces, e.g. $c_{i_1\cdots i_N} = \sum_{\alpha_1\cdots\alpha_N} a_{i_1\alpha_1} a_{i_2\alpha_1\alpha_2} \cdots a_{i_N\alpha_N\alpha_1}$.





Tensor network ansatz states

- in a graphical representation, each vertex represents a tensor and the edge represents the index of the tensor.

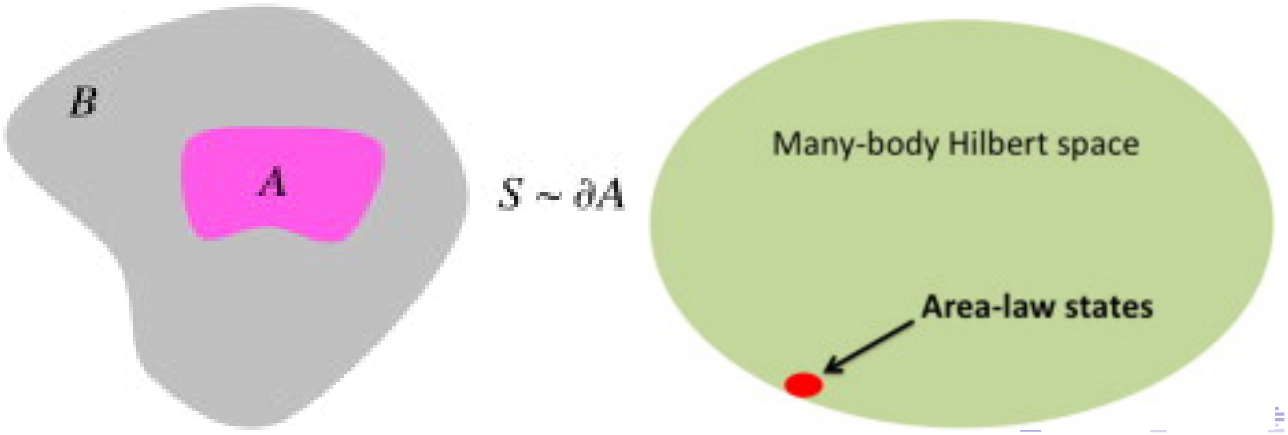


- The different connection pattern can be used to encode different entanglement pattern.

Physics behind different ansatz state

The validity of different ansatz states is a result of the fact that the mathematical Hilbert space is too large (larger than the set of states that appears in our universe).

- From relativity we know that the interaction of particles in our universe is all local interaction, the Hamiltonian is thus a local Hamiltonian
- The ground state of local gap Hamiltonian often satisfy entanglement area law, thus, when we calculate the ground state, we choose ansatz state as ones which satisfy area law.



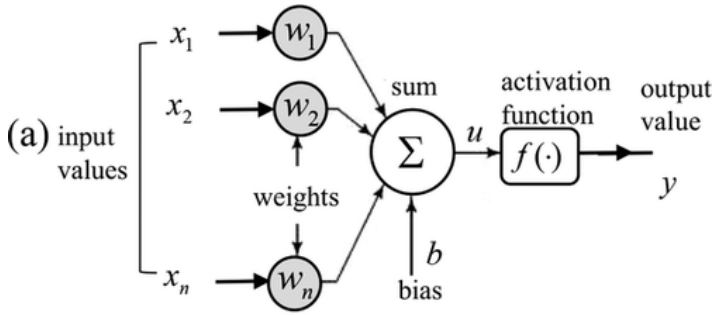
Why neural network states?

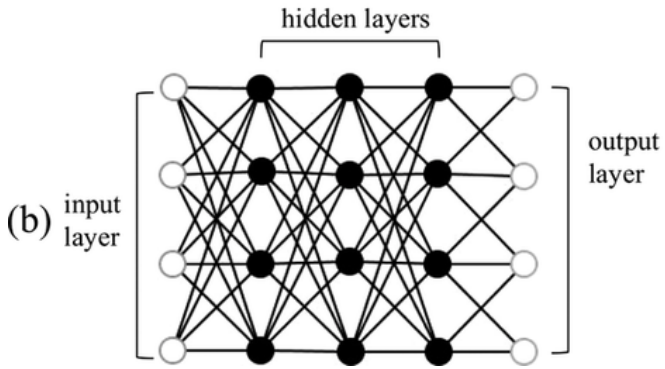
Motivation for using neural network as an ansatz state:

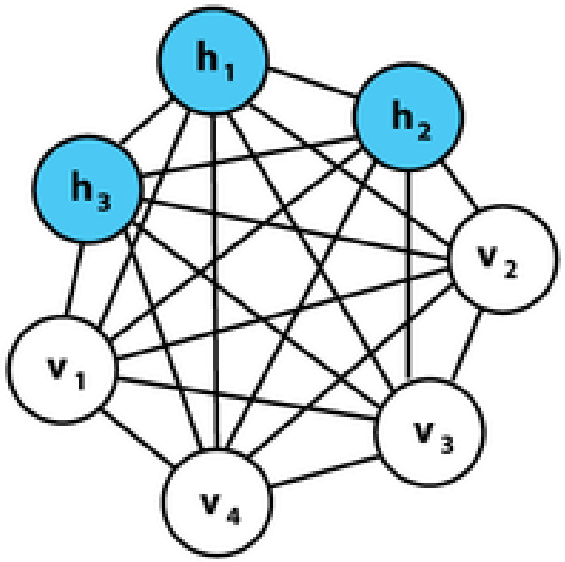
- Many physical systems have a **simplified internal structure** that typically makes the parameters needed to characterize their ground states exponentially smaller.
- Sometimes we **need not to known the exact wave functions** but only the expectation value of observables, the correlation function, the phase diagram, etc.
- The neural network has great power in **approximating functions** and **extracting features of big data**. This matches well with our goal for solving some physical problems.
- **Efficiency consideration**, neural network approach is much more efficient than many other approaches.

Why neural network states?

- We can regard neural network as a block which for some input $v_i, \cdots, v_N$, it output a result[1] $F(v_1, \cdots, v_N)$, this fits well with picture of wave function.



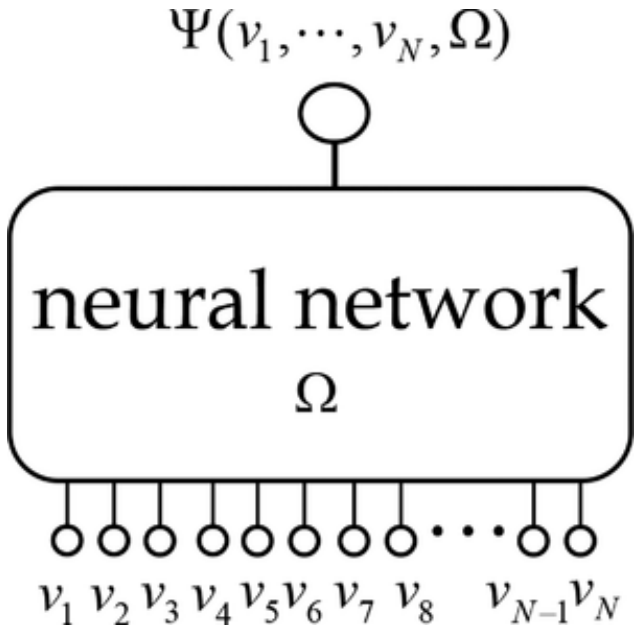




[1] it can also be a vector function, here for our purpose, we assume it as a scalar function

What is neural network state

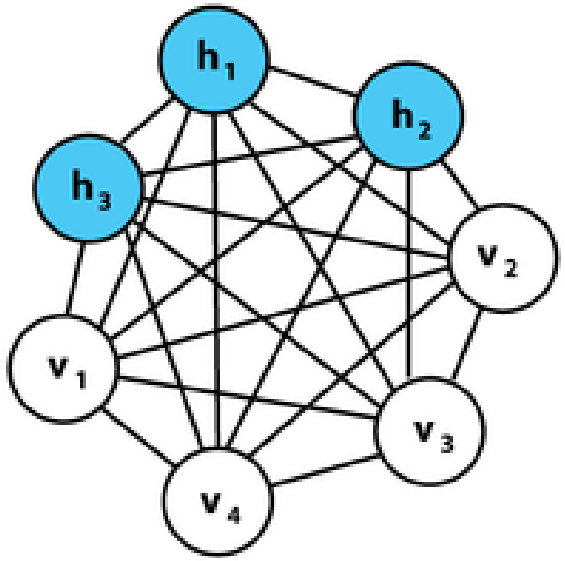
A wave function is a map, for some input $v_1, \cdots, v_N$, output a complex number $\Psi(v_1, \cdots, v_N)$, this can be represented using a neural network.



Here we using Ω to denote the set of parameters of the neural network. Different neural networks give different neural network states.

What is neural network state: Boltzmann machine state

- visible neuron v_i with bias a_i;
- hidden neuron h_j with bias b_j;
- connection: W_{ij}, $W_{ii'}$, $W_{jj'}$
- v_i and h_j real; the weights and biases are complex



BM state

The energy function is given as $E(\mathbf{h},\mathbf{v}) = -(\sum_i v_i a_i + \sum_j h_j b_j + \sum_{\langle ij \rangle} W_{ij} v_i h_j + \sum_{\langle jj' \rangle} W_{jj'} h_j h_{j'} + \sum_{\langle ii' \rangle} W_{ii'} v_i v_{i'})$ The wave function is (up to a normalization factor): $\Psi_{BM}(\mathbf{v},\Omega) = \sum_{h_1} \cdots \sum_{h_l} \frac{e^{-E(\mathbf{h},\mathbf{v})}}{Z}$, where $Z = \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{h},\mathbf{v})}$.

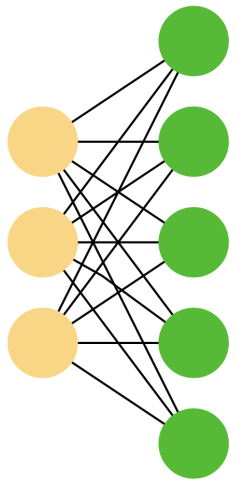
What is neural network state: RMB states

RBM states

- Energy function is $E(\mathbf{h}, \mathbf{v}) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_{ij} v_i W_{ij} h_j$
- wave function is $\Psi(\mathbf{v}, \Omega) = \sum_{h_1} \cdots \sum_{h_l} e^{\sum_i a_i v_i + \sum_j h_j \left(b_j + \sum_i v_i W_{ij}\right)} = \prod_i e^{a_i v_i} \prod_j \Gamma_j (\mathbf{v}; b_j, W_{ij})$
  overall normalization factor and the partition function $Z(\Omega)$ are omitted,
- $\Gamma_j = \sum_{h_j} e^{h_j \left(b_j + \sum_i v_i W_{ij}\right)}$ is $2\cosh\left(b_j + \sum_i v_i W_{ij}\right)$ or $1 + e^{b_j + \sum_i v_i W_{ij}}$ for h_j takes values in $\{\pm 1\}$ and $\{0, 1\}$ respectively.

RBM neural network

- visible: v_i with biases a_i;
- hidden: h_j with biases b_j;
- connection: W_{ij}



What is neural network state: DBM states

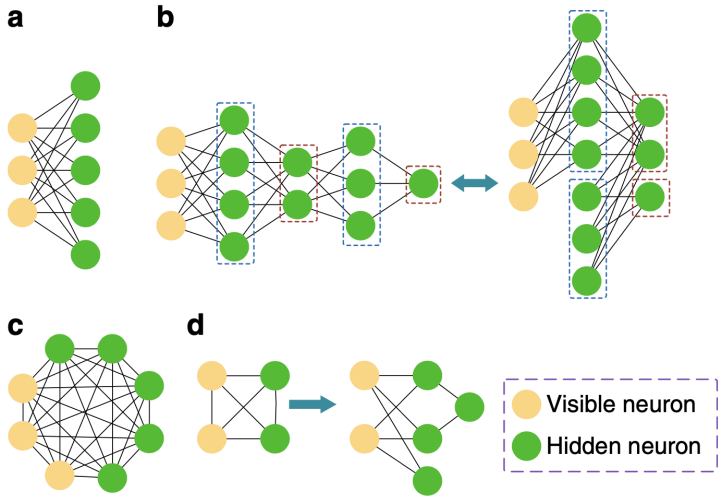
DBM

All Boltzmann machines can be transformed into a DBM with two hidden layers

DBM states

The wave function is given as (up to a normalization factor):

$$\Psi(\mathbf{v}, \Omega) = \sum_{h_1} \cdots \sum_{h_l} \sum_{g_1} \cdots \sum_{g_q} \frac{\exp^{-E(\mathbf{v},\mathbf{h},\mathbf{g})}}{Z}$$



RBM states in calculating physical properties

- transverse-field Ising (TFI) model:
  $H_{\mathrm{TFI}} = -h\sum_i \sigma_i^x - \sum_{ij}\sigma_i^z\sigma_j^z$
- antiferromagnetic Heisenberg (AFH) model:
  $\mathcal{H}_{\mathrm{AFH}} = \sum_{ij}\sigma_i^x\sigma_j^x + \sigma_i^y\sigma_j^y + \sigma_i^z\sigma_j^z$

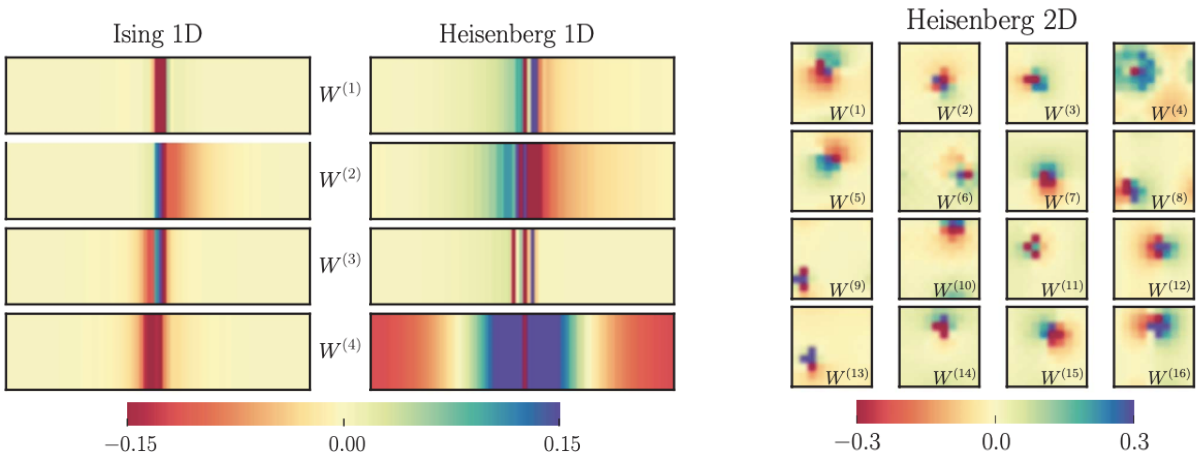


Figure: weights for typical states.

RBM states in calculating physical properties

- transverse-field Ising (TFI) model:
  $H_{\mathrm{TFI}} = -h\sum_i \sigma_i^x - \sum_{ij}\sigma_i^z\sigma_j^z$
- antiferromagnetic Heisenberg (AFH) model:
  $\mathcal{H}_{\mathrm{AFH}} = \sum_{ij}\sigma_i^x\sigma_j^x + \sigma_i^y\sigma_j^y + \sigma_i^z\sigma_j^z$

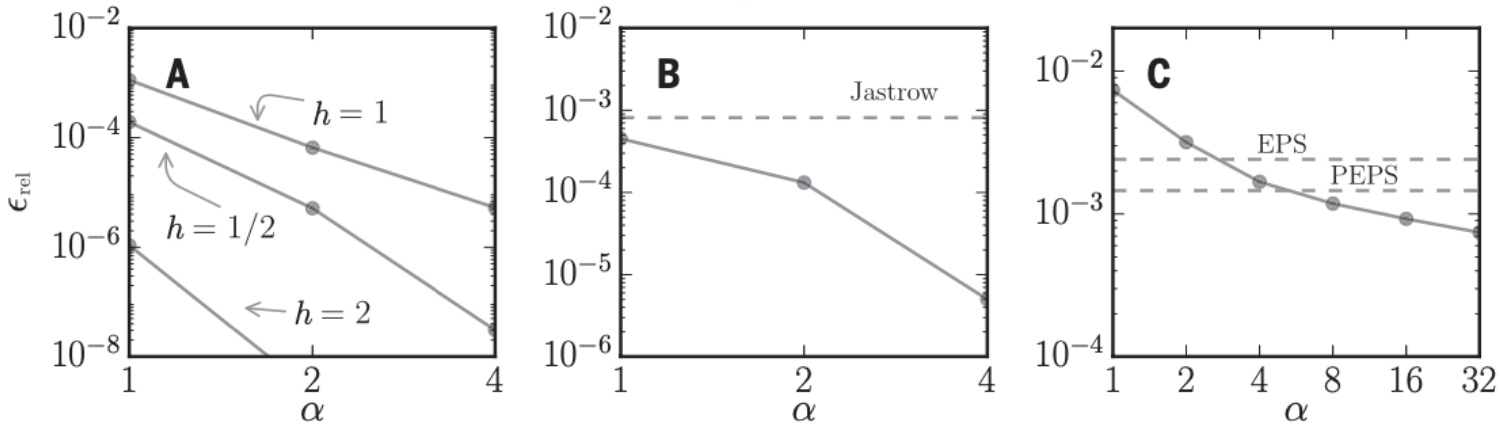


Figure: energy calculation.

RBM states in calculating physical properties

- transverse-field Ising (TFI) model:
  $H_{\mathrm{TFI}} = -h\sum_i \sigma_i^x - \sum_{ij}\sigma_i^z\sigma_j^z$
- antiferromagnetic Heisenberg (AFH) model:
  $\mathcal{H}_{\mathrm{AFH}} = \sum_{ij}\sigma_i^x\sigma_j^x + \sigma_i^y\sigma_j^y + \sigma_i^z\sigma_j^z$

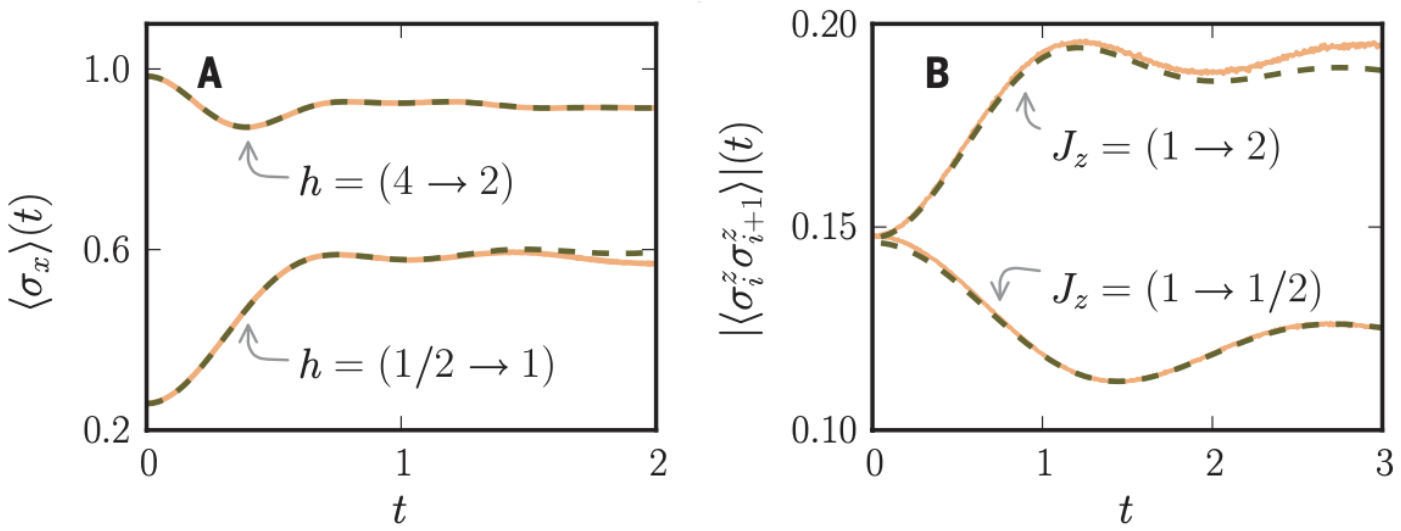


Figure: evolution calculation.

Disgonal gates

Diagonal Gates

Disgonal gates can be applied to an RBM quamtum state by solving a set of linear equations.

Considering an RBM machine with weights $\{\alpha, \beta, W\}$

- Single-Qubit Z rotation: $R_l^Z = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$ modifies the bias of the visible neuron l: $a_j' = a_j + \delta_{jl} i\theta$

Disgonal gates

- Controll Z rotation: $CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{pmatrix}$: new weights are given by:

$$\begin{aligned} W_{lc} &= -2A(\theta) \\ W_{mc} &= 2A(\theta) \\ \Delta a_l &= i\frac{\theta}{2} + A(\theta) \\ \Delta a_m &= i\frac{\theta}{2} - A(\theta) \end{aligned} \tag{3}$$

Where $A(\theta) = \text{arccosh}\ (e^{-i\frac{\theta}{2}})$

Non-diagonal Gates

For any non-diagonal gates: $G = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$

1. Draw samples $\mathbf{d}_i = (c_i, t_i)$ from Gibbs Sampling and transition probilities given by

$$||\phi(v_{v_l=0})\rangle|^2 = |a \cdot |\psi(v_{v_l=0})\rangle + c \cdot |\psi(v_{v_l=1})\rangle|^2 \tag{4}$$

$$||\phi(v_{v_l=1})\rangle|^2 = |b \cdot |\psi(v_{v_l=0})\rangle + d \cdot |\psi(v_{v_l=1})\rangle|^2 \tag{5}$$

2. Calculate gradients $\frac{\partial L}{\partial \theta_i}$ of the loss function defined by:

$$\partial_{p_i} L(W) = \langle O_k^*(\mathbf{v})\rangle_\psi - \frac{\langle \frac{\phi(\mathbf{v})}{\psi(\mathbf{v})} O_k^*(\mathbf{v})\rangle_\psi}{\langle \frac{\phi(\mathbf{v})}{\psi(\mathbf{v})}\rangle_\psi} \tag{6}$$

3. Use a gradient descent method like AdaMax or SR to update the parameters $\theta_i \in W$

Per-gate error: 10^{-3}

Implementation for using RBM calculate ground state

RBM for calculating ground state

Generate random weights and biases at start, then use gradient decent to approach minimum and find the corresponding weights and biases, from the weights and biases to give the ground state.

We will take 1D Heisenberg model and 1D Ising model as examples.

- transverse-field Ising (TFI) model: $H_{\mathrm{TFI}} = -h\sum_i \sigma_i^x - \sum_{ij}\sigma_i^z\sigma_j^z$
- antiferromagnetic Heisenberg (AFH) model: $\mathcal{H}_{\mathrm{AFH}} = \sum_{ij}\sigma_i^x\sigma_j^x + \sigma_i^y\sigma_j^y + \sigma_i^z\sigma_j^z$

Notice that boundary condition (open or closed) need to be added.

RBM network setting code

```python
from machine.machine import Machine

class RBM(Machine):

    def __init__(self, num_visible, density=2):
        Machine.__init__(self)
        self.num_visible = num_visible
        self.density = density
        self.num_hidden = self.num_visible * self.density
        self.W = None
        self.bv = None  # visible layer bias
        self.bh = None
        self.connection = None

    def is_complex(self):
        return False
```

```python
class Machine(object):

    def __init__(self):
        self.num_visible = 0

    def log_val(self, v):
        pass

    def log_val_diff(self, v1, v2):
        pass

    def derlog(self, v, size):
        pass

    def get_new_visible(self, v):
        pass

    def get_parameters(self):
        pass
```

Figure: define the RBM class and machine class

RBM network setting code

The following are network transfering code:

```
import tensorflow as tf
import copy
import sys
import numpy as np
import os
import pickle
import itertools

class RBMTransfer(object):

    def __init__ (self, machine_target, graph_target, base_model_path,
     base_model_number=None):
        self.machine_target = machine_target
        self.graph_target = graph_target
        self.base_model_path = base_model_path
        self.base_model_number = base_model_number
        self.initialize()

    def update_machine(self):
        self.machine_target.W_array = self.W_transfer
        self.machine_target.bv_array = self.bv_transfer
        self.machine_target.bh_array = self.bh_transfer

    def initialize(self):
        # Get the base model from the path
        self.learner_base = self.get_base_model()
        self.machine_base = self.learner_base.machine
        self.graph_base = self.learner_base.graph

        # Initialize the transferred weight and biases from the target
         machine
        if self.learner_base.machine.__class__.__name__ == 'rbm_real_symm':
            self.W_transfer = self.machine_target.W_symm_array
            self.bv_transfer = self.machine_target.bv_symm_array
            self.bh_transfer = self.machine_target.bh_symm_array

            self.W_base = self.machine_base.W_symm
            self.bv_base = self.machine_base.bv_symm
            self.bh_base = self.machine_base.bh_symm

        else:
            self.W_transfer = self.machine_target.W_array
            self.bv_transfer = self.machine_target.bv_array
            self.bh_transfer = self.machine_target.bh_array

            self.W_base = self.machine_base.W
            self.bv_base = self.machine_base.bv
            self.bh_base = self.machine_base.bh


    def get_base_model(self):
        if self.base_model_number is None:
            dir_names = [int(f) for f in os.listdir(self.base_model_path)
             if os.path.isdir(self.base_model_path + f)]
            self.base_model_number = max(dir_names)
```

RBM network setting code

```
        self.base_model_path = '%s/%d/model.p' % (self.base_model_path,
         self.base_model_number)
        base_model = pickle.load(open(self.base_model_path))
        return base_model


    def tiling (self, k_val):
        assert self.machine_target.num_visible >=
         self.machine_base.num_visible and self.machine_target.num_visible
         % self.machine_base.num_visible == 0, "Number of visible node in
         the machine must be larger than or equal to and divisible by the
         number of visible node in the base machine!"
        assert self.graph_base.length % k_val == 0, "k must be divisible by
         the number of visible node in base machine!"

        p_val = self.graph_target.length / self.graph_base.length

        base_coor = []
        for point in range(self.graph_base.num_points):
            ##### Map old coordinate to the new coordinate which is the
             old_coor * the k_size
            ## For instance:
            ## 1D from 4 to 8 particles
            ## o-o-o-o  to o-o-o-o-o-o-o-o
            ## 0 1 2 3  to 0 1 2 3 4 5 6 7
            ## 0 will be transferred to 0
            ## 1 will be transferred to 2 and so on
            ##
            ## 2D from 2x2 to 4x4
            ## 0,0    0,1        0,0     0,1    0,2    0,3
            ##  o------o          o-------o------o------o
            ##  |      |          |      1|1    1|2     |
            ##  o------o     1,0  o-------o------o------o 1,3
            ## 1,0    1,1         |      2|1    2|2     |
            ##                2,0  o-------o------o------o 2,3
            ##                     |      3|1    3|2     |
            ##                3,0  o-------o------o------o 3,3
            ## 0,0 will be transfered to 0,0
            ## 1,0 will be tranferred to 2,0
            ## and so on.
            ## Similar for 3D
            old_coor = np.array(self.graph_base._point_to_coordinate(point))

            ## map the first position of the old coordinate in the base
             network to the new coordinate in the target network
            new_coor = (old_coor / k_val) * (k_val * p_val) + (old_coor %
             k_val)


            ##### Generate all possible combinations for the product
            ## We want to transfer 0 to 0 and 1 for 1D
            ## and 0,0 to 0,0; 0,1; 1,0; 1,1 for 2D
            ## We generate all possible combinations for the product
            ## For instance:
            ## 1D from 4 to 8 particles
```

RBM network setting code

```
## old_coor 0 -> new_coor 0 -> to_iter = [[0,1]]
## old_coor 2 -> new coor 4 -> to_iter = [[4,5]]
## 1D from 4 to 16 particles
## old_coor 0 -> new_coor 0 -> to_iter = [[0, 1, 2, 4]
## old_coor 2 -> new_coor 8 -> to_iter = [[8, 9, 10, 11]]
## 2D from 2x2 to 4x4 particles
## old_coor 0,0 -> new_coor 0,0 -> to_iter = [[0,1],[0,1]]
## old_coor 1,0 -> new_coor 2,0 -> to_iter = [[2,3],[0,1]]
## 3D from 2x2x2 to 4x4x4
## old_coor (0,0,0) -> new_coor 0,0,0 -> to_iter=[[0,1], [0,1],
 [0,1]]
## old_coor (1,0,1) -> new_coor 2,0,2 -> to_iter=[[2,3], [0,1],
 [2,3]]
##
## because later we will do a product multiply on the to_iter
 to generate all possible combinations except for 1D
## 2D from 2x2 to 4x4
## new_coor 0,0 -> to_iter = [[0,1],[0,1]] do a product multiply
## [0,1] x [0,1] = [[0,0], [0,1], [1,0], [1,1]
## new_coor 2,0 -> to_iter = [[2,3],[0,1]] do a product multiply
## [2,3] x [0,1] = [[2,0], [2,1], [3,0], [3,1]]
## so we get the mapping for transfer
## 3D from 2x2x2 to 4x4x4
## [2,3] x [0,1] x [2,3] = [2,0,2], [2,0,3], [2,1,2], [2,1,3],
 ....

to_iter = []
for dd in range(self.graph_target.dimension):
    temp = []
    for pp in range(p_val):
        temp.append(new_coor[dd] + pp * k_val)
    to_iter.append(temp)

### List all combinations to be replaced which is the product
 that has been explained before
## For example in 3d from 2 to 4
## old_coor (0,0,0), new coordinates = (0,0,0), (0,0,1),
 (0,1,0), (0,1,1) ....
## old_coor (1,1,1), new_coordinates = (2,2,2), (2,2,3),
 (2,3,2), ....

new_coordinates = []
if self.graph_target.dimension == 1:
    new_coordinates = [[a] for a in to_iter[0]]
else:
    for kk in to_iter:
        if len(new_coordinates) == 0:
            new_coordinates = kk
        else:
            new_coordinates = [list(cc[0] + [cc[1]]) if
             isinstance(cc[0], list) else list(cc)  for cc in
             list(itertools.product(new_coordinates, kk))]
```

RBM network setting code

```
            ## Connect to new hidden
            for coord in new_coordinates:
                quadrant = [c / self.graph_base.length for c in coord]
                hid_pos = 0
                for ddd in range(self.graph_base.dimension):
                    hid_pos += quadrant[ddd] * (p_val  ** ddd)

                target_point = self.graph_target._coordinate_to_point(coord)

                self.W_transfer[target_point, hid_pos *
                 self.W_base.shape[1] :(hid_pos + 1) *
                 self.W_base.shape[1]] = self.W_base[point, :]


        self.update_machine()


    def cutpaste(self):
        assert self.machine_target.num_visible >=
         self.machine_base.num_visible, "Number of visible node in the
         machine must be larger than or equal to the numbero f visible node
         in the base machine!"

        for ii in range(self.graph_base.num_points):
            new_coor =
             self.graph_target._coordinate_to_point(self.graph_base
             ._point_to_coordinate(ii))
            self.W_transfer[new_coor,:self.W_base.shape[1]] =
             self.W_base[ii]

        self.bv_transfer[:self.bv_base.shape[1]] = self.bv_base
        self.bh_transfer[:self.bh_base.shape[1]]  = self.bh_base

        self.update_machine()
```

RBM network setting code

The following are the real RBM network setting

```
from machine.rbm import RBM
import tensorflow as tf
import copy
from functools import partial
import numpy as np


class RBMReal(RBM):
    def __init__(
        self, num_visible, density=2, initializer=None, num_expe=None,
         use_bias=True
    ):
        RBM.__init__(self, num_visible, density)
        self.initializer = initializer
        self.use_bias = use_bias
        self.num_expe = num_expe
        self.build_model()

    def build_model(self):
        self.random_initialize()

    def random_initialize(self):
        if self.num_expe is not None:
            np.random.seed(self.num_expe)
        self.W_array = self.initializer(size=(self.num_visible,
         self.num_hidden))
        self.bv_array = np.zeros((1, self.num_visible))
        self.bh_array = np.zeros((1, self.num_hidden))

        np.random.seed()

    def create_variable(self):
        self.W = tf.Variable(
            tf.convert_to_tensor(value=self.W_array.astype(np.float32)),
            name="weights",
            trainable=True,
        )
        self.bv = tf.Variable(
            tf.convert_to_tensor(value=self.bv_array.astype(np.float32)),
            name="visible_bias",
            trainable=True,
        )
        self.bh = tf.Variable(
            tf.convert_to_tensor(value=self.bh_array.astype(np.float32)),
            name="hidden_bias",
            trainable=True,
        )

    # Calculate log of p_RBM with configuration v
    def log_val(self, v):
        theta = tf.matmul(v, self.W) + self.bh
        sum_ln_thetas = tf.reduce_sum(
            input_tensor=tf.math.log(2 * tf.cosh(theta)), axis=1,
             keepdims=True
        )
```

RBM network setting code

```
    ln_bias = tf.matmul(v, tf.transpose(a=self.bv))
    return sum_ln_thetas + ln_bias

def log_val_diff(self, v1, v2):
    log_val_1 = self.log_val(v1)
    log_val_2 = self.log_val(v2)
    return log_val_1 - log_val_2

def derlog(self, v, sample_size):
    theta = tf.matmul(v, self.W) + self.bh
    if self.use_bias:
        D_bv = v * 0.5
        D_bh = tf.tanh(theta) * 0.5
    else:
        D_bv = v * 0.0
        D_bh = tf.tanh(theta) * 0.0
    D_w_temp = (
        tf.reshape(tf.tanh(theta), (sample_size, 1, self.num_hidden))
        * tf.reshape(v, (sample_size, self.num_visible, 1))
        * 0.5
    )

    D_w = tf.reshape(D_w_temp, (sample_size, self.num_visible *
     self.num_hidden))

    derlog_dict = {"w": D_w, "v": D_bv, "h": D_bh}
    return derlog_dict

def reshape_grads(self, grad_dict):
    grad_bv = tf.reshape(grad_dict["v"], (1, self.num_visible))
    grad_bh = tf.reshape(grad_dict["h"], (1, self.num_hidden))
    grad_w = tf.reshape(grad_dict["w"], (self.num_visible,
     self.num_hidden))
    return [grad_w, grad_bv, grad_bh]

# helpers for sampling
def get_new_visible(self, v):
    hprob = self.get_hidden_prob_given_visible(v)
    hstate = self.convert_from_prob_to_state(hprob)
    vprob = self.get_visible_prob_given_hidden(hstate)
    vstate = self.convert_from_prob_to_state(vprob)
    return vstate

def get_hidden_prob_given_visible(self, v):
    return tf.sigmoid(2.0 * (tf.matmul(v, self.W) + self.bh))

def get_visible_prob_given_hidden(self, h):
    return tf.sigmoid(2.0 * (tf.matmul(h, tf.transpose(a=self.W)) +
     self.bv))

def convert_from_prob_to_state(self, prob):
    v = prob - tf.random.uniform(tf.shape(input=prob), 0, 1)
    return tf.where(
        tf.greater_equal(v, tf.zeros_like(v)), tf.ones_like(v), -1 *
         tf.ones_like(v)
```

RBM network setting code

```
    )

def get_parameters(self):
    return [self.W, self.bv, self.bh]

def set_connection(self, connection):
    self.connection = connection

def make_pickle_object(self, sess):
    temp_rbm = copy.copy(self)
    temp_rbm.W, temp_rbm.bv, temp_rbm.bh = sess.run((self.W, self.bv,
     self.bh))
    return temp_rbm

def __str__(self):
    return "RBM %d-%d" % (self.num_visible, self.num_hidden)

def to_xml(self):
    stri = ""
    stri += "<machine>\n"
    stri += "\t<type>rbm_real</type>\n"
    stri += "\t<params>\n"
    stri += "\t\t<num_visible>%d</num_visible>\n" % self.num_visible
    stri += "\t\t<num_hidden>%d</num_hidden>\n" % self.num_hidden
    stri += "\t\t<density>%d</density>\n" % self.density
    stri += "\t\t<use_bias>%s</use_bias>\n" % str(self.use_bias)
    stri += "\t</params>\n"
    stri += "</machine>\n"
    return stri
```

Heisenberg 1D Model (code)

The code is as follows

```
import tensorflow as tf
from hamiltonian import Hamiltonian
import itertools
import numpy as np

class Heisenberg(Hamiltonian):

    def __init__(self, graph, jx=1.0, jy=1.0, jz=1.0):

        Hamiltonian.__init__(self, graph)
        self.jx = jx
        self.jy = jy
        self.jz = jz

    def calculate_hamiltonian_matrix(self, samples, num_samples):

        num_spins = self.graph.num_points

        diagonal_element = None
        off_diagonal_element = None
        spins = tf.split(samples, num_spins, axis=1)
        for (s, s_2) in self.graph.bonds:
            if diagonal_element is None:
                diagonal_element = self.jz * spins[s] * spins[s_2]
            else:
                diagonal_element = tf.concat((diagonal_element, self.jz *
                 spins[s] * spins[s_2]), axis=1)

            if off_diagonal_element is None:
                off_diagonal_element = -(self.jx - self.jy * spins[s] *
                 spins[s_2])
            else:
                off_diagonal_element = tf.concat((off_diagonal_element,
                 -(self.jx - self.jy * spins[s] * spins[s_2])),
                                                 axis=1)

        diagonal_element = tf.reduce_sum(input_tensor=diagonal_element,
         axis=1, keepdims=True)

        hamiltonian = tf.concat((diagonal_element, off_diagonal_element),
         axis=1)

        return hamiltonian
```

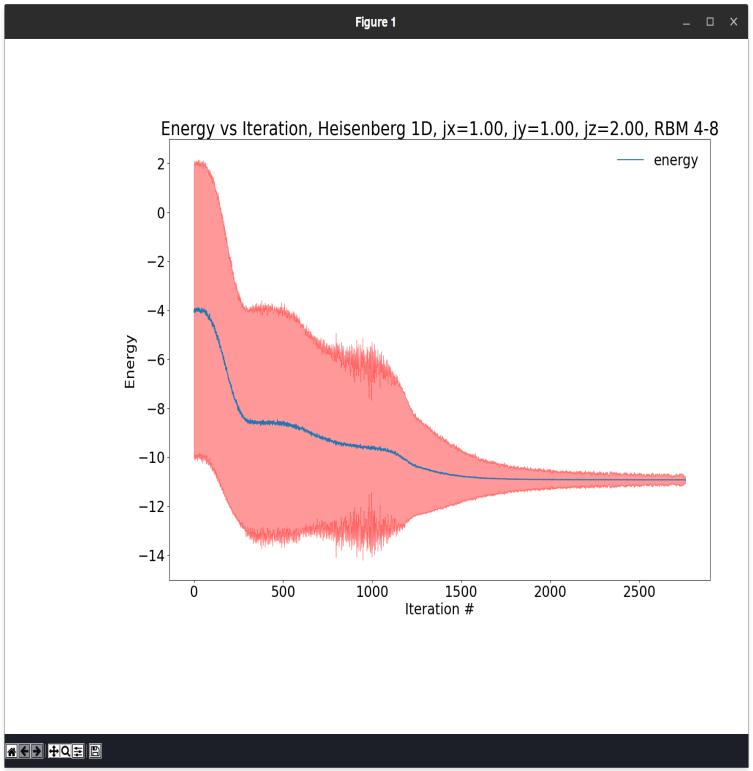
Heisenberg 1D Model (code)

```
def flip(self, x, p1, p2, num_samples):
    num_spins = self.graph.num_points
    y = np.eye(num_spins, dtype=np.float32)
    y[p1][p1] = -1
    y[p2][p2] = -1
    return tf.matmul(x, tf.convert_to_tensor(value=y))

def calculate_lvd(self, samples, machine, num_samples):

    lvd = machine.log_val_diff(samples, samples)
    for (s, s_2) in self.graph.bonds:


        new_config = self.flip(samples, s, s_2, num_samples)
        lvd = tf.concat((lvd, machine.log_val_diff(new_config,
         samples)), axis=1)
    return lvd

def __str__(self):
    return "Heisenberg %dD, jx=%.2f, jy=%.2f, jz=%.2f" %
     (self.graph.dimension, self.jx, self.jy, self.jz)

def to_xml(self):
    str = ""
    str += "<hamiltonian>\n"
    str += "\t<type>heisenberg</type>\n"
    str += "\t<params>\n"
    str += "\t\t<jx>%d</jx>\n" % self.jx
    str += "\t\t<jy>%d</jy>\n" % self.jy
    str += "\t\t<jz>%d</jz>\n" % self.jz
    str += "\t</params>\n"
    str += "</hamiltonian>\n"
    return str
```

Heisenberg 1D Model (result)

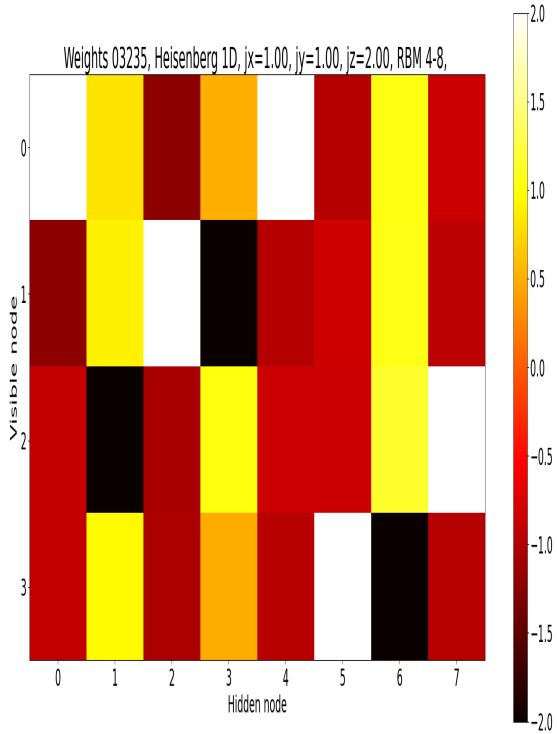
Find the ground enery of Heisenberg 1D model

1D.png



(a) Energy vs Iteration, Heisenberg 1D, 4 qubits

1D.png



(b) Weights after 3000 iterations, Heisenberg 1D, 4 qubits

Ising 1D Model (code)

The code is as follows

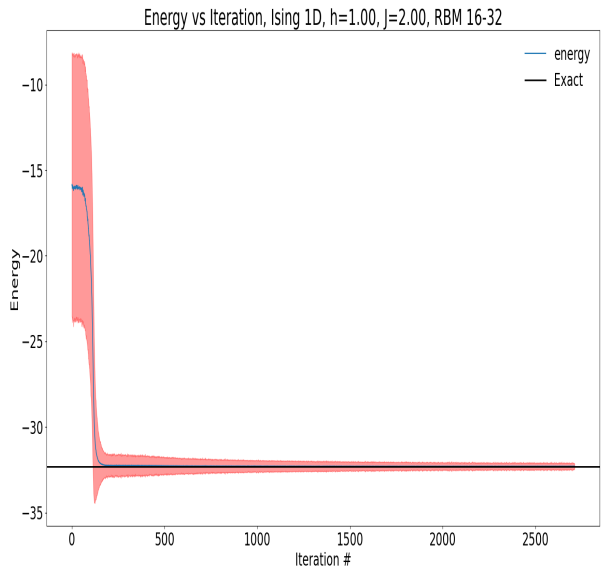
```
import tensorflow as tf
from hamiltonian import Hamiltonian
import itertools
import numpy as np


class Ising (Hamiltonian):

    def __init__(self, graph, j=1.0, h=1.0):

        Hamiltonian.__init__(self, graph)
        self.j = j
        self.h = h

    def calculate_hamiltonian_matrix(self, samples, num_samples):

        num_spins = self.graph.num_points

        interact_energy = None
        spins = tf.split(samples, num_spins, axis=1)
        for (s, s_2) in self.graph.bonds:
            if interact_energy is None:
                interact_energy = -self.j * spins[s] * spins[s_2]
            else:
                interact_energy = tf.concat((interact_energy, -self.j *
                 spins[s] * spins[s_2]), axis=1)

        interact_energy = tf.reduce_sum(input_tensor=interact_energy,
         axis=1, keepdims=True)
        external_energy = tf.fill((num_samples, num_spins), -self.h)
        hamiltonian = tf.concat((interact_energy, external_energy), axis=1)

        return hamiltonian
```

Ising 1D Model (code)

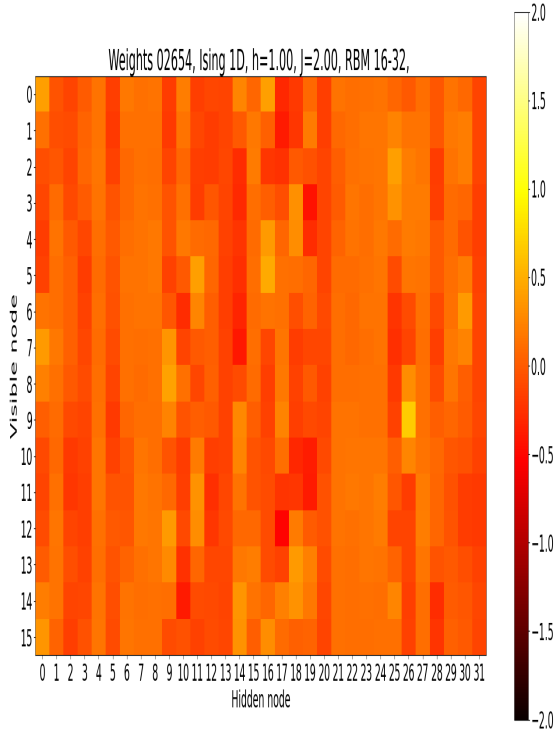
```
def flip(self, x, p, num_samples):
    num_spins = self.graph.num_points

    y = np.eye(num_spins, dtype=np.float32)
    y[p][p] = -1
    return tf.matmul(x, tf.convert_to_tensor(value=y))

def calculate_lvd(self, samples, machine, num_samples):
    lvd = machine.log_val_diff(samples, samples)
    for s in range(self.graph.num_points):
        new_config = self.flip(samples, s, num_samples)
        lvd = tf.concat((lvd, machine.log_val_diff(new_config,
         samples)), axis=1)
    return lvd

def __str__(self):
    return "Ising %dD, h=%.2f, J=%.2f" % (self.graph.dimension, self.h,
     self.j)

def to_xml(self):
    str = ""

                    str += "<hamiltonian>\n"
                    str += "\t<type>ising</type>\n"
                    str += "\t<params>\n"
                    str += "\t\t<j>%.2f</j>\n" % self.j
                    str += "\t\t<h>%.2f</h>\n" % self.h
                    str += "\t</params>\n"
                    str += "</hamiltonian>\n"
                    return str
```

Ising 1D Model (result)

Find the ground energy of Ising 1D model



(a) Energy vs Iteration, Ising 1D, 16 qubits



(b) Weights after 3000 iterations, Ising 1D, 16 qubits

Calculation detail code

Here are details of the calculation setting

```
import os
import tensorflow.compat.v1 as tf
from functools import partial
import pickle
import numpy as np
from machine.rbm.real import RBMReal
from hamiltonian import Ising, Heisenberg
from graph import Hypercube
from sampler import Gibbs
from learner import Learner
from logger import Logger
from parse_qasm import parser

import sys

tf.disable_v2_behavior()

tf.get_logger().setLevel('ERROR')

if len(sys.argv) > 1:
    in_qasm_file = sys.argv[1]
else:
    in_qasm_file = input("Please define the input qasm file: ")

qasm_seq = parser.Parser(in_qasm_file)
```

Calculation detail code

```
# Number of visible nodes
num_visible = qasm_seq.num_quibit
# dimension
dimension = 1
# periodic boundary condition (True) or open boundary condition (False)
pbc = True

### Parameters for the Hamiltonian
# Type of the Hamiltonian
hamiltonian_type = "ISING"
# hamiltonian_type = "HEISENBERG"
# Parameters of the Hamiltonian
h = 1.0
jx = 1.0
jy = 1.0
jz = 2.0

### Parameters for the Sampler
# Number of samples
num_samples = 10000
# Number of steps in the sampling process
num_steps = 1

### Parameters for the RBM
# Density (ratio between the number of hidden and visible nodes)
density = 2
# Function to initialise the weight
# np.random.normal: Draw random samples from a normal (Gaussian)
 distribution.
```

Calculation detail code

```
initializer = partial(np.random.normal, loc=0.0, scale=0.01)

### Parameters for the Learner
# Initialise tensorflow session
sess = tf.Session()
# Optimiser for the gradient descent
trainer = tf.train.RMSPropOptimizer
# Initial learning of the optimiser
learning_rate = 0.001
# The number of iterations/epochs for the training
num_epochs = 10000
window_period = 200
# Size of the minibatch
minibatch_size = 0
# Threshold value for the stopping criterion
stopping_threshold = 0.005

### Parameters for the Logger
log = True
# The path for the result folder
result_path = "./results/"
# The name of the subpath for your experiment, by default if it is empty it
 will be named 'cold-start' for cold start
subpath = ""
# Indicate whether you want to visualise the weight or visible layer and
 how frequent
visualize_weight = False
visualize_visible = False
visualize_freq = 10

# Indicate whether you want to see the weight different after and before
 training
weight_diff = True

#### Create instances from all of the parameters
graph = Hypercube(num_visible, dimension, pbc)

hamiltonian = None
if hamiltonian_type == "ISING":
    hamiltonian = Ising(graph, jz, h)
elif hamiltonian_type == "HEISENBERG":
    hamiltonian = Heisenberg(graph, jx, jy, jz)
```

Calculation detail code

```
## Choose the type of sampler here
sampler = Gibbs(num_samples, num_steps)
machine = RBMReal(
    graph.num_points, density, initializer, num_expe=1, use_bias=False
)
machine.create_variable()

learner = Learner(
    sess,
    graph,
    hamiltonian,
    machine,
            sampler,
            trainer,
            learning_rate,
            num_epochs,
            minibatch_size,
            window_period,
            stopping_threshold,
            visualize_weight,
            visualize_visible,
            visualize_freq,
            qasm_seq.diagonal_gates,
            qasm_seq.non_diagonal_gates,
        )

        logger = Logger(
            log,
            result_path,
            subpath,
            visualize_weight,
            visualize_visible,
            visualize_freq,
            weight_diff,
        )

        learner.learn()
        learner.apply_gates()
        logger.log(learner)

        # clear previous graph for multiple runs of learner
        tf.reset_default_graph()

        sess.close()
```

Neural network simulation of quantum computation

Notes

For this part, we haven't done much simulation work, because of the time limitation and the difficulties in representing non-diagonal gates in RBM.
But given the representation of gates in RBM, we propose several ways to simulate the whole quantum circuit.

Neural network simulation of quantum computation

RBM simulation of quantum computation

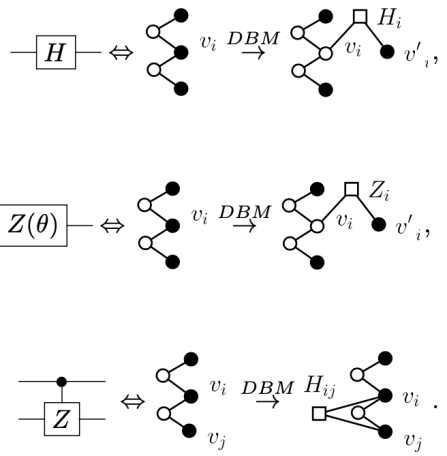
- constructing the RBM network for the input state with the number of visible neurons equal the the number of qubits of the quantum circuit;
- applying quantum gates by adding hidden neurons or changing weights of the network;
- do measurements of the final states and obtain the probability distribution in each computational bases

Notice: there are some gates cannot be applied exactly, thus we need to change the weights of connections in a local region of the qubits for which the gate is applied.

Neural network simulation of quantum computation

DBM simulation of quantum computation

- constructing the DBM network for the input state with the number of visible neurons equal the the number of qubits of the quantum circuit;
- applying quantum gates by adding hidden neurons or changing weights of the network;
- do measurements of the final states and obtain the probability distribution in each computational bases



TODO:

1. Try different approaches to approximate non-diagonal gates and optimize the fidelity of the gates;
2. Try to use a more expressive methods like DBM or tensor network approach
3. Give explicit examples for the simulation of a given algorithm, like Grover search, quantum Fourier transformation.
4. Generalize to fault-tolerant case, like stabilizer code calculation.

Some final comments:

For neural network ansatz state of physical system

- Our code can be easily generalized to many 1D spin chains, like Kitaev chain, J_1-J_2model, etc.
- It also has potential applications in calculating unitary evolution, adiabatic quantum computation, many-body localization, etc.

For classical simulation of quantum computation

- Existing realization of RBM representation of surface code and color code suffering from the problem that they cannot do universal quantum computation, our method can be directly generalized to the fault-tolerant situation and can do computation universally;
- RBM representations of topological codes is more powerful in error detection and error correction, but not in simulation of quantum computation;

Contributions of each author:

- Lu Wei: design the project, make the physical analyzation, collecting the related references, design the framework for the code realization, and write the ppt;
- Tengyue Wang: make the part of physical analyzation and help to do the code debug
- Luofan Chen: run and debug the code and collect the references about the code realization

References:

1. G. Carleo and M. Troyer, Solving the quantum many-body problem with artificial neural networks, Science 355, 602 (2017).
2. D.-L. Deng, X. Li, and S. Das Sarma, Quantum entanglement in neural network states, Phys. Rev. X 7, 021021 (2017).
3. X. Gao and L.-M. Duan, Efficient representation of quantum many-body states with deep neural networks, Nature Communications 8, 662 (2017).
4. Jia, Z. A., Wei, L., Wu, Y. C., Guo, G. C., and Guo, G. P. Entanglement area law for shallow and deep quantum neural network states. New Journal of Physics, 22(5), 053022 (2020).
5. Orus, Roman. "A practical introduction to tensor networks: Matrix product states and projected entangled pair states." Annals of Physics 349 : 117-158 (2014).

THANK YOU FOR YOUR ATTENTIONS