Restricted Boltzmann Machine applied to Quantum Mechanical Problems

Vilde Flugsrud[1] Morten Hjorth-Jensen[1,2]

Department of Physics, University of Oslo[1]

Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University[2]

Apr 12, 2018

What is Machine Learning?

Machine learning is the science of giving computers the ability to learn without being explicitly programmed. The idea is that there exist generic algorithms which can be used to find patterns in a broad class of data sets without having to write code specifically for each problem. The algorithm will build its own logic based on the data.

Machine learning is a subfield of computer science, and is closely related to computational statistics. It evolved from the study of pattern recognition in artificial intelligence (AI) research, and has made contributions to AI tasks like computer vision, natural language processing and speech recognition. It has also, especially in later years, found applications in a wide variety of other areas, including bioinformatics, economy, physics, finance and marketing.

Types of Machine Learning

The approaches to machine learning are many, but are often split into two main categories. In *supervised learning* we know the answer to a problem, and let the computer deduce the logic behind it. On the other hand, *unsupervised learning* is a method for finding patterns and relationship in data sets without any prior knowledge of the system. Some authours also operate with a third category, namely *reinforcement learning*. This is a paradigm of learning inspired by behavioural psychology, where learning is achieved by trial-and-error, solely from rewards and punishment.

Another way to categorize machine learning tasks is to consider the desired output of a system. Some of the most common tasks are:

- Classification: Outputs are divided into two or more classes. The goal is to produce a model that assigns inputs into one of these classes. An example is to identify digits based on pictures of hand-written ones. Classification is typically supervised learning.
- Regression: Finding a functional relationship between an input

Artificial neurons

The field of artificial neural networks has a long history of development, and is closely connected with the advancement of computer science and computers in general. A model of artificial neurons was first developed by McCulloch and Pitts in 1943 to study signal processing in the brain and has later been refined by others. The general idea is to mimic neural networks in the human brain, which is composed of billions of neurons that communicate with each other by sending electrical signals. Each neuron accumulates its incoming signals, which must exceed an activation threshold to yield an output. If the threshold is not overcome, the neuron remains inactive, i.e. has zero output.

This behaviour has inspired a simple mathematical model for an artificial neuron.

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \qquad (1)$$

Here, the output y of the neuron is the value of its activation

Neural network types

An artificial neural network (NN), is a computational model that consists of layers of connected neurons, or *nodes*. It is supposed to mimic a biological nervous system by letting each neuron interact with other neurons by sending signals in the form of mathematical functions between layers. A wide variety of different NNs have been developed, but most of them consist of an input layer, an output layer and eventual layers in-between, called *hidden layers*. All layers can contain an arbitrary number of nodes, and each connection between two nodes is associated with a weight variable.

Feed-forward neural networks

The feed-forward neural network (FFNN) was the first and simplest type of NN devised. In this network, the information moves in only one direction: forward through the layers.

Nodes are represented by circles, while the arrows display the connections between the nodes, including the direction of information flow. Additionally, each arrow corresponds to a weight variable, not displayed here. We observe that each node in a layer is connected to *all* nodes in the subsequent layer, making this a so-called *fully-connected* FFNN.

A different variant of FFNNs are *convolutional neural networks* (CNNs), which have a connectivity pattern inspired by the animal visual cortex. Individual neurons in the visual cortex only respond to stimuli from small sub-regions of the visual field, called a receptive field. This makes the neurons well-suited to exploit the strong spatially local correlation present in natural images. The response of each neuron can be approximated mathematically as a convolution operation.

Recurrent neural networks

So far we have only mentioned NNs where information flows in one direction: forward. *Recurrent neural networks* on the other hand, have connections between nodes that form directed *cycles*. This creates a form of internal memory which are able to capture information on what has been calculated before; the output is dependent on the previous computations. Recurrent NNs make use of sequential information by performing the same task for every element in a sequence, where each element depends on previous elements. An example of such information is sentences, making recurrent NNs especially well-suited for handwriting and speech recognition.

Other types of networks

There are many other kinds of NNs that have been developed. One type that is specifically designed for interpolation in multidimensional space is the radial basis function (RBF) network. RBFs are typically made up of three layers: an input layer, a hidden layer with non-linear radial symmetric activation functions and a linear output layer (''linear'' here means that each node in the output layer has a linear activation function). The layers are normally fully-connected and there are no cycles, thus RBFs can be viewed as a type of fully-connected FFNN. They are however usually treated as a separate type of NN due the unusual activation functions.

Other types of NNs could also be mentioned, but are outside the scope of this work. We will now move on to a detailed description of how a fully-connected FFNN works, and how it can be used to interpolate data sets.

Multilayer perceptrons

One use often so-called fully-connected feed-forward neural networks with three or more layers (an input layer, one or more hidden layers and an output layer) consisting of neurons that have non-linear activation functions.

Such networks are often called *multilayer perceptrons* (MLPs)

Why multilayer perceptrons?

According to the *Universal approximation theorem*, a feed-forward neural network with just a single hidden layer containing a finite number of neurons can approximate a continuous multidimensional function to arbitrary accuracy, assuming the activation function for the hidden layer is a **non-constant, bounded and monotonically-increasing continuous function**. Note that the requirements on the activation function only applies to the hidden layer, the output nodes are always assumed to be linear, so as to not restrict the range of output values.

We note that this theorem is only applicable to a NN with *one* hidden layer. Therefore, we can easily construct an NN that employs activation functions which do not satisfy the above requirements, as long as we have at least one layer with activation functions that *do*. Furthermore, although the universal approximation theorem lays the theoretical foundation for regression with neural networks, it does not say anything about how things work in practice: A neural network can still be able to approximate a given function reasonably well without having the

Mathematical model

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(u) \tag{2}$$

In an FFNN of such neurons, the *inputs* x_i are the *outputs* of the neurons in the preceding layer. Furthermore, an MLP is fully-connected, which means that each neuron receives a weighted sum of the outputs of *all* neurons in the previous layer.

Mathematical model

First, for each node i in the first hidden layer, we calculate a weighted sum u_i^1 of the input coordinates x_j,

$$u_i^1 = \sum_{j=1}^{2} w_{ij}^1 x_j + b_i^1 \tag{3}$$

This value is the argument to the activation function f_1 of each neuron i, producing the output y_i^1 of all neurons in layer 1,

$$y_i^1 = f_1(u_i^1) = f_1\left(\sum_{j=1}^{2} w_{ij}^1 x_j + b_i^1\right) \tag{4}$$

where we assume that all nodes in the same layer have identical activation functions, hence the notation f_l

$$y_i^l = f_l(u_i^l) = f_l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right) \tag{5}$$

Mathematical model

The output of neuron i in layer 2 is thus,

$$y_i^2 = f_2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right) \tag{6}$$

$$= f_2\left[\sum_{j=1}^{3} w_{ij}^2 f_1\left(\sum_{k=1}^{2} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \tag{7}$$

where we have substituted y_m^1 with. Finally, the NN output yields,

$$y_1^3 = f_3\left(\sum_{j=1}^{3} w_{1m}^3 y_j^2 + b_1^3\right) \tag{8}$$

$$= f_3\left[\sum_{j=1}^{3} w_{1j}^3 f_2\left(\sum_{k=1}^{3} w_{jk}^2 f_1\left(\sum_{m=1}^{2} w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_1^3\right] \tag{9}$$

Mathematical model

We can generalize this expression to an MLP with l hidden layers. The complete functional form is,

$$y_1^{l+1} = f_{l+1}\left[\sum_{j=1}^{N_l} w_{1j}^3 f_l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^2 f_{l-1}\left(\ldots f_1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\ldots\right) + b_k^2\right) + b_1^3\right] \tag{10}$$

which illustrates a basic property of MLPs: The only independent variables are the input values x_n.

Mathematical model

This confirms that an MLP, despite its quite convoluted mathematical form, is nothing more than an analytic function, specifically a mapping of real-valued vectors $\vec{x} \in \mathbb{R}^n \rightarrow \vec{y} \in \mathbb{R}^m$. In our example, $n = 2$ and $m = 1$. Consequentially, the number of input and output values of the function we want to fit must be equal to the number of inputs and outputs of our MLP.

Furthermore, the flexibility and universality of a MLP can be illustrated by realizing that the expression is essentially a nested sum of scaled activation functions of the form

$$h(x) = c_1 f(c_2 x + c_3) + c_4 \tag{11}$$

where the parameters c_i are weights and biases. By adjusting these parameters, the activation functions can be shifted up and down or left and right, change slope or be rescaled which is the key to the flexibility of a neural network.

Matrix-vector notation. We can introduce a more convenient notation for the activations in a NN.

Additionally, we can represent the biases and activations as layer-wise column vectors $\vec{b}_l$ and $\vec{y}_l$, so that the i-th element of each vector is the bias b_i^l and activation y_i^l of node i in layer l respectively.

We have that W_l is a $N_{l-1} \times N_l$ matrix, while $\vec{b}_l$ and $\vec{y}_l$ are $N_l \times 1$ column vectors. With this notation, the sum in becomes a matrix-vector multiplication, and we can write the equation for the activations of hidden layer 2 in

$$\vec{y}_2 = f_2(\mathrm{W}_2\vec{y}_1 + \vec{b}_2) = f_2\left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix}\right). \tag{12}$$

Matrix-vector notation and activation. The activation of node i in layer 2 is

$$y_i^2 = f_2\Bigl(w_{i1}^2 y_1^1 + w_{i2}^2 y_2^1 + w_{i3}^2 y_3^1 + b_i^2\Bigr) = f_2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right). \tag{13}$$

This is not just a convenient and compact notation, but also a useful and intuitive way to think about MLPs: The output is calculated by a series of matrix-vector multiplications and vector additions that are used as input to the activation functions. For each operation $\mathrm{W}_l \vec{y}_{l-1}$ we move forward one layer.

Activation functions. A property that characterizes a neural network, other than its connectivity, is the choice of activation function(s). As described in, the following restrictions are imposed on an activation function for a FFNN to fulfill the universal approximation theorem

- Non-constant
- Bounded
- Monotonically-increasing
- Continuous

Activation functions, Logistic and Hyperbolic ones. The second requirement excludes all linear functions. Furthermore, in a MLP with only linear activation functions, each layer simply performs a linear transformation of its inputs.

Regardless of the number of layers, the output of the NN will be nothing but a linear function of the inputs. Thus we need to introduce some kind of non-linearity to the NN to be able to fit non-linear functions Typical examples are the logistic *Sigmoid*

$$f(x) = \frac{1}{1+e^{-x}}, \tag{14}$$

and the *hyperbolic tangent* function

$$f(x) = \tanh(x) \tag{15}$$

Boltzmann Machines

Why use a generative ("dreaming") model rather than the more well known discriminative deep neural networks (DNN)?

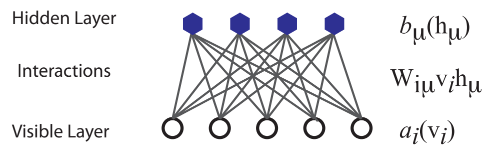
- Discriminitave methods have several limitations: They are supervised learning methods, thus requiring labeled data. And there are tasks they cannot accomplish, like drawing new examples from an unknown probability distribution.
- A generative model can learn to represent and sample from a probability distribution. The core idea is to learn a parametric model of the probability distribution from which the training data was drawn. As an example
  1. A model for images could learn to draw new examples of cats and dogs, given a training dataset of images of cats and dogs.
  2. Generate a sample of an ordered or disordered Ising model phase, having been given samples of such phases.
  3. Model the trial wave function for Monte Carlo calculations

Some similarities and differences from DNNs

1. Both use gradient-descent based learning procedures for minimizing cost functions
2. Energy based models don't use backpropagation and automatic differentiation for computing gradients, instead turning to Markov Chain Monte Carlo methods.
3. DNNs often have several hidden layers. A restricted Boltzmann machine has only one hidden layer, however several RBMs can be stacked to make up Deep Belief Networks, of which they constitute the building blocks.

History: The RBM was developed by amongst others Geoffrey Hinton, called by some the "Godfather of Deep Learning", working with the University of Toronto and Google.

The structure of the RBM network



The network

The network layers:

1. The function **x** represents the visible layer, a vector of M elements (nodes). This layer represents both what the RBM might be given as training input, *and* what we want it to be able to reconstruct. This might for example be the pixels of an image, the spin values of the Ising model, or coefficients representing speech.
2. The function **h** represents the hidden, or latent, layer. A vector of N elements (nodes). Also called "feature detectors".

Goals

The goal of the hidden layer is to increase the model's expressive power. We encode complex interactions between visible variables by introducing additional, hidden variables that interact with visible degrees of freedom in a simple manner, yet still reproduce the complex correlations between visible degrees in the data once marginalized over (integrated out).

Examples of this trick being employed in physics:

1. The Hubbard-Stratonovich transformation
2. The introduction of ghost fields in gauge theory

The network parameters, to be optimized/learned:

1. **a** represents the visible bias, a vector of same lenght as **x**.
2. **b** represents the hidden bias, a vector of same lenght as **h**.
3. W represents the interaction weights, a matrix of size $M \times N$.

Joint distribution and the Energy function

The restricted Boltzmann machine is described by a Bolztmann distribution

$$P_{rbm}(\mathbf{x},\mathbf{h}) = \frac{1}{Z} e^{-\frac{1}{T_0}E(\mathbf{x},\mathbf{h})}, \tag{16}$$

where Z is the normalization constant or partition function, defined as

$$Z = \int \int e^{-\frac{1}{T_0}E(\mathbf{x},\mathbf{h})} d\mathbf{x} d\mathbf{h}. \tag{17}$$

It is common to ignore T_0 by setting it to one.

Network Elements

The function $E(\mathbf{x},\mathbf{h})$ gives the **energy** of a configuration (pair of vectors) $(\mathbf{x},\mathbf{h})$. The lower the energy of a configuration, the higher the probability of it. This function also depends on the parameters $\mathbf{a}$, $\mathbf{b}$ and W. Thus, when we adjust them during the learning procedure, we are adjusting the energy function to best fit our problem.

Defining different types of RBMs

There are different variants of RBMs, and the differences lie in the types of visible and hidden units we choose as well as in the implementation of the energy function $E(\mathbf{x},\mathbf{h})$.

Binary-Binary RBM:

RBMs were first developed using binary units in both the visible and hidden layer. The corresponding energy function is defined as follows:

$$E(\mathbf{x},\mathbf{h}) = -\sum_i^M x_i a_i - \sum_j^N b_j h_j - \sum_{i,j}^{M,N} x_i w_{ij} h_j, \tag{18}$$

where the binary values taken on by the nodes are most commonly 0 and 1.

Gaussian-Binary RBM:

Another varient is the RBM where the visible units are Gaussian while the hidden units remain binary:

More about RBMs

1. Useful when we model continuous data (i.e., we wish $\mathbf{x}$ to be continuous)
2. Requires a smaller learning rate, since there's no upper bound to the value a component might take in the reconstruction

Other types of units include:

1. Softmax and multinomial units
2. Gaussian visible and hidden units
3. Binomial units
4. Rectified linear units

Sampling: Metropolis sampling

In order to sample from the RBM probability distribution it is common to use Markov Chain Monte Carlo (MCMC) algorithms such as Metropolis-Hastings or Gibbs sampling.

Metropolis sampling starts by suggesting a new configuration $\boldsymbol{x}^{k+1}$. In the brute force method this is done by some random change of the visible units. The new configuration is then accepted with the acceptance probability

$$A(\boldsymbol{x}^k \rightarrow \boldsymbol{x}^{k+1}) = \min(1, \frac{P(\boldsymbol{x}^{k+1})}{P(\boldsymbol{x}^k)}), \tag{20}$$

where we need the marginalized probability

$$\begin{aligned} P(\boldsymbol{x}) &= \sum_{\mathbf{h}} P_{rbm}(\mathbf{x},\mathbf{h}) && (21) \\ &= \frac{1}{Z}\sum_{\mathbf{h}} e^{-E(\mathbf{x},\mathbf{h})}. && (22) \end{aligned}$$

Sampling: Gibbs sampling

In this method we sample from the joint probability $P_{rbm}(\mathbf{x},\mathbf{h})$ by way of a two step sampling process. We alternately update the visible and hidden units. New samples are generated according to the conditional probabilities $P(x_i|\mathbf{h})$ and $P(h_j|\mathbf{x})$ respectively and accepted with the probability of 1. While the the visible nodes are dependent on the hidden nodes and vice versa, the nodes are independent of other nodes within the same layer. This is due to there being no intra layer interactions in the restricted Boltzmann machine.

The conditional probabilities are often referred to as the activitation functions in the neural networks context due to their role in determining the node outputs. For the binary-binary RBM they are

$$P(h_j = 1|\boldsymbol{x}) = \frac{1}{1 + e^{-b_j - \sum_i x_i w_{ij}}} \tag{23}$$

$$P(x_i = 1|\boldsymbol{h}) = \frac{1}{1 + e^{-a_j - \sum_j h_j w_{ij}}}, \tag{24}$$

where we recognize the logistic sigmoid function $\sigma(x) = 1/(1 + \exp(-x))$.

Gaussian RBM

For the Gaussian-Binary RBM the conditional probabilities are

$$P(x_i|\mathbf{h}) = \mathcal{N}(x_i; a_i + \sum_j h_j w_{ij}, \sigma^2) \tag{25}$$

$$P(h_j = 1|\mathbf{x}) = \frac{1}{1 + e^{-b_j - \frac{1}{\sigma^2}\sum_i x_i w_{ij}}}, \tag{26}$$

while the visible units now follow a normal distribution, we see the hidden units again follow the logistic sigmoid function.

Cost function

When working with a training dataset, the most common training approach is maximizing the log-likelihood of the training data. The log likelihood characterizes the log-probability of generating the observed data using our generative model. Using this method our cost function is chosen as the negative log-likelihood. The learning then consists of trying to find parameters that maximize the probability of the dataset, and is known as Maximum Likelihood Estimation (MLE). Denoting the parameters as $\boldsymbol{\theta} = a_1, ..., a_M, b_1, ..., b_N, w_{11}, ..., w_{MN}$, the log-likelihood is given by

$$\begin{aligned} \mathcal{L}(\{\theta_i\}) &= \langle \log P_\theta(\boldsymbol{x}) \rangle_{data} \qquad (27) \\ &= -\langle E(\boldsymbol{x}; \{\theta_i\}) \rangle_{data} - \log Z(\{\theta_i\}), \qquad (28) \end{aligned}$$

where we used that the normalization constant does not depend on the data, $\langle \log Z(\{\theta_i\}) \rangle = \log Z(\{\theta_i\})$ Our cost function is the negative log-likelihood, $\mathcal{C}(\{\theta_i\}) = -\mathcal{L}(\{\theta_i\})$

Optimization / Training

The training procedure of choice often is Stochastic Gradient Descent (SGD). It consists of a series of iterations where we update the parameters according to the equation

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla \mathcal{C}(\boldsymbol{\theta}_k) \tag{29}$$

at each k-th iteration. There are a range of variants of the algorithm which aim at making the learning rate η more adaptive so the method might be more efficient while remaining stable.

We now need the gradient of the cost function in order to minimize it. We find that

$$\begin{aligned} \frac{\partial \mathcal{C}(\{\theta_i\})}{\partial \theta_i} &= \langle \frac{\partial E(\boldsymbol{x}; \theta_i)}{\partial \theta_i} \rangle_{data} + \frac{\partial \log Z(\{\theta_i\})}{\partial \theta_i} \qquad (30) \\ &= \langle O_i(\boldsymbol{x}) \rangle_{data} - \langle O_i(\boldsymbol{x}) \rangle_{model}, \qquad (31) \end{aligned}$$

where in order to simplify notation we defined the "operator"

$$O_i(\boldsymbol{x}) = \frac{\partial E(\boldsymbol{x}; \theta_i)}{\partial \theta_i}, \tag{32}$$

and used the statistical mechanics relationship between expectation

More on RBMs

The data-dependent term in the gradient is known as the positive phase of the gradient, while the model-dependent term is known as the negative phase of the gradient. The aim of the training is to lower the energy of configurations that are near observed data points (increasing their probability), and raising the energy of configurations that are far from observed data points (decreasing their probability).

The gradient of the negative log-likelihood cost function of a Binary-Binary RBM is then

$$\begin{aligned} \frac{\partial \mathcal{C}(w_{ij}, a_i, b_j)}{\partial w_{ij}} &= \langle x_i h_j \rangle_{data} - \langle x_i h_j \rangle_{model} \qquad (34) \\ \frac{\partial \mathcal{C}(w_{ij}, a_i, b_j)}{\partial a_{ij}} &= \langle x_i \rangle_{data} - \langle x_i \rangle_{model} \qquad (35) \\ \frac{\partial \mathcal{C}(w_{ij}, a_i, b_j)}{\partial b_{ij}} &= \langle h_i \rangle_{data} - \langle h_i \rangle_{model}. \qquad (36) \\ & \qquad (37) \end{aligned}$$

To get the expecation values with respect to the *data*, we set the

Which sampling to use

To get the expectation values with respect to the *model*, we use Gibbs sampling. We can either initialize the $\boldsymbol{x}$ randomly or with a training sample. While we ideally want a large number of Gibbs iterations $n \rightarrow n$, one might decide to truncate it earlier for efficiency. Doing this while having intialized $\boldsymbol{x}$ with a training data vector is referred to as contrastive divergence (CD), because one is then closer to approximating the gradient of this function than the negative log-likelihood. The contrastive divergence function is the difference between two Kullback-Leibler divergences (also called relative entropy), which measure how one probability distribution diverges from a second, expected probability distribution (in this case the estimated one from the ground truth one).

RBMs for the quantum many body problem

The idea of applying RBMs to quantum many body problems was presented by G. Carleo and M. Troyer, working with ETH Zurich and Microsoft Research.

Some of their motivation included

- "The wave function Ψ is a monolithic mathematical quantity that contains all the information on a quantum state, be it a single particle or a complex molecule. In principle, an exponential amount of information is needed to fully encode a generic many-body quantum state."
- There are still interesting open problems, including fundamental questions ranging from the dynamical properties of high-dimensional systems to the exact ground-state properties of strongly interacting fermions.
- The difficulty lies in finding a general strategy to reduce the exponential complexity of the full many-body wave function down to its most essential features. That is
  1. $\rightarrow$ Dimensional reduction
  2. $\rightarrow$ Feature extraction

Choose the right RBM

Carleo and Troyer applied the RBM to the quantum mechanical spin lattice systems of the Ising model and Heisenberg model, with encouraging results. Our goal is to test the method on systems of moving particles. For the spin lattice systems it was natural to use a binary-binary RBM, with the nodes taking values of 1 and -1. For moving particles, on the other hand, we want the visible nodes to be continuous, representing position coordinates. Thus, we start by choosing a Gaussian-binary RBM, where the visible nodes are continuous and hidden nodes take on values of 0 or 1. If eventually we would like the hidden nodes to be continuous as well the rectified linear units seem like the most relevant choice.

Representing the wave function

The wavefunction should be a probability amplitude depending on $\boldsymbol{x}$. The RBM model is given by the joint distribution of $\boldsymbol{x}$ and $\boldsymbol{h}$

$$F_{rbm}(\mathbf{x},\mathbf{h}) = \frac{1}{Z} e^{-\frac{1}{T_0}E(\mathbf{x},\mathbf{h})}. \tag{38}$$

To find the marginal distribution of $\boldsymbol{x}$ we set:

$$F_{rbm}(\mathbf{x}) = \sum_{\mathbf{h}} F_{rbm}(\mathbf{x},\mathbf{h}) \tag{39}$$

$$= \frac{1}{Z}\sum_{\mathbf{h}} e^{-E(\mathbf{x},\mathbf{h})}. \tag{40}$$

Now this is what we use to represent the wave function, calling it a neural-network quantum state (NQS)

$$\Psi(\mathbf{X}) = F_{rbm}(\mathbf{x}) \tag{41}$$

$$= \frac{1}{Z}\sum_{\boldsymbol{h}} e^{-E(\mathbf{x},\mathbf{h})} \tag{42}$$

$$= \frac{1}{Z}\sum_{\{h_j\}} e^{-\sum_i^M \frac{(x_i-a_i)^2}{2\sigma^2} + \sum_j^N b_j h_j + \sum_{i,j}^{M,N} \frac{x_i w_{ij} h_j}{\sigma^2}} \tag{43}$$

Choose the cost function

Now we don't necessarily have training data (unless we generate it by using some other method). However, what we do have is the variational principle which allows us to obtain the ground state wave function by minimizing the expectation value of the energy of a trial wavefunction (corresponding to the untrained NQS). Similarly to the traditional variational Monte Carlo method then, it is the local energy we wish to minimize. The gradient to use for the stochastic gradient descent procedure is

$$G_i = \frac{\partial \langle E_L \rangle}{\partial \theta_i} = 2(\langle E_L \frac{1}{\Psi}\frac{\partial \Psi}{\partial \theta_i}\rangle - \langle E_L \rangle \langle \frac{1}{\Psi}\frac{\partial \Psi}{\partial \theta_i}\rangle), \tag{46}$$

where the local energy is given by

$$E_L = \frac{1}{\Psi}\hat{\mathbf{H}}\Psi. \tag{47}$$

The physical system

As described in more detail in the project, we start by investigating a harmonic oscillator system, with the Hamiltonian given by

$$\hat{\mathbf{H}} = \sum_{p}^{P}(-\frac{1}{2}\nabla_p^2 + \frac{1}{2}\omega^2 r_p^2) + \sum_{p<q}\frac{1}{r_{pq}}. \tag{48}$$