Boltzmann machines and deep learning

Morten Hjorth-Jensen Email morten.hjorth-jensen@fys.uio.no

Department of Physics and Center fo Computing in Science Education, University of Oslo, Oslo, Norway

April 11, 2025

Plans for the week of April 7-11, 2025

1. Neural Networks and Boltzmann Machines, examples of generative deep learning
2. Discussions of project 2

- Video of lecture at URL":"https://youtu.be/cXyKzlMjApI"
- Whiteboard notes at https://github.com/CompPhysics/ComputationalPhysics2/blob/gh-pages/doc/HandWrittenNotes/2025/NotesApril11.pdf

Energy models

Last week we defined a domain $\boldsymbol{X}$ of stochastic variables $\boldsymbol{X} = \{x_0, x_1, \dots, x_{n-1}\}$ with a pertinent probability distribution

$$p(\boldsymbol{X}) = \prod_{x_i \in \boldsymbol{X}} p(x_i),$$

where we have assumed that the random varaibles x_i are all independent and identically distributed (iid).

We will now assume that we can defined this function in terms of optimization parameters $\boldsymbol{\Theta}$, which could be the biases and weights of deep network, and a set of hidden variables we also assume to be random variables which also are iid. The domain of these variables is $\boldsymbol{H} = \{h_0, h_1, \dots, h_{m-1}\}$.

Probability model

We define a probability

$$p(x_i, h_j; \boldsymbol{\Theta}) = \frac{f(x_i, h_j; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})},$$

where $f(x_i, h_j; \boldsymbol{\Theta})$ is a function which we assume is larger or equal than zero and obeys all properties required for a probability distribution and $Z(\boldsymbol{\Theta})$ is a normalization constant. Inspired by statistical mechanics, we call it often for the partition function. It is defined as (assuming that we have discrete probability distributions)

$$Z(\boldsymbol{\Theta}) = \sum_{x_i \in \boldsymbol{X}} \sum_{h_j \in \boldsymbol{H}} f(x_i, h_j; \boldsymbol{\Theta}).$$

Marginal and conditional probabilities

We can in turn define the marginal probabilities

$$p(x_i; \boldsymbol{\Theta}) = \frac{\sum_{h_j \in \boldsymbol{H}} f(x_i, h_j; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})},$$

and

$$p(h_i; \boldsymbol{\Theta}) = \frac{\sum_{x_i \in \boldsymbol{X}} f(x_i, h_j; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})}.$$

Change of notation

Note the change to a vector notation. A variable like $\boldsymbol{x}$ represents now a specific **configuration**. We can generate an infinity of such configurations. The final partition function is then the sum over all such possible configurations, that is

$$Z(\boldsymbol{\Theta}) = \sum_{x_i \in \boldsymbol{X}} \sum_{h_j \in \boldsymbol{H}} f(x_i, h_j; \boldsymbol{\Theta}),$$

changes to

$$Z(\boldsymbol{\Theta}) = \sum_{\boldsymbol{x}} \sum_{\boldsymbol{h}} f(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta}).$$

If we have a binary set of variable x_i and h_j and M values of x_i and N values of h_j we have in total 2^M and 2^N possible $\boldsymbol{x}$ and $\boldsymbol{h}$ configurations, respectively.

We see that even for the modest binary case, we can easily approach a number of configuration which is not possible to deal with.

Optimization problem

At the end, we are not interested in the probabilities of the hidden variables. The probability we thus want to optimize is

$$p(\boldsymbol{X};\boldsymbol{\Theta}) = \prod_{x_i \in \boldsymbol{X}} p(x_i;\boldsymbol{\Theta}) = \prod_{x_i \in \boldsymbol{X}} \left(\frac{\sum_{h_j \in \boldsymbol{H}} f(x_i, h_j;\boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})} \right),$$

which we rewrite as

$$p(\boldsymbol{X};\boldsymbol{\Theta}) = \frac{1}{Z(\boldsymbol{\Theta})} \prod_{x_i \in \boldsymbol{X}} \left(\sum_{h_j \in \boldsymbol{H}} f(x_i, h_j;\boldsymbol{\Theta}) \right).$$

Further simplifications

We simplify further by rewriting it as

$$p(\boldsymbol{X};\boldsymbol{\Theta}) = \frac{1}{Z(\boldsymbol{\Theta})} \prod_{x_i \in \boldsymbol{X}} f(x_i;\boldsymbol{\Theta}),$$

where we used $p(x_i;\boldsymbol{\Theta}) = \sum_{h_j \in \boldsymbol{H}} f(x_i, h_j;\boldsymbol{\Theta})$. The optimization problem is then

$$\arg \max_{\boldsymbol{\Theta} \in \mathbb{R}^p} p(\boldsymbol{X};\boldsymbol{\Theta}).$$

Optimizing the logarithm instead

Computing the derivatives with respect to the parameters $\boldsymbol{\Theta}$ is easier (and equivalent) if we compute the logarithm of the probability. We will thus optimize

$$\arg \max_{\boldsymbol{\Theta} \in \mathbb{R}^p} \log p(\boldsymbol{X};\boldsymbol{\Theta}),$$

which leads to

$$\nabla_{\boldsymbol{\Theta}} \log p(\boldsymbol{X};\boldsymbol{\Theta}) = 0.$$

Expression for the gradients

This leads to the following equation

$$\nabla_{\boldsymbol{\Theta}} \log p(\boldsymbol{X};\boldsymbol{\Theta}) = \nabla_{\boldsymbol{\Theta}} \left(\sum_{x_i \in \boldsymbol{X}} \log f(x_i;\boldsymbol{\Theta}) \right) - \nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = 0.$$

The first term is called the positive phase and we assume that we have a model for the function f from which we can sample values. Below we will develop an explicit model for this. The second term is called the negative phase and is the one which leads to more difficulties.

The derivative of the partition function

The partition function, defined above as

$$Z(\boldsymbol{\Theta}) = \sum_{x_i \in \boldsymbol{X}} \sum_{h_j \in \boldsymbol{H}} f(x_i, h_j; \boldsymbol{\Theta}),$$

is in general the most problematic term. In principle both x and h can span large degrees of freedom, if not even infinitely many ones, and computing the partition function itself is often not desirable or even feasible. The above derivative of the partition function can however be written in terms of an expectation value which is in turn evaluated using Monte Carlo sampling and the theory of Markov chains, popularly shortened to MCMC (or just MC^2).

Explicit expression for the derivative

We can rewrite

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \frac{\nabla_{\boldsymbol{\Theta}} Z(\boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})},$$

which reads in more detail

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \frac{\nabla_{\boldsymbol{\Theta}} \sum_{x_i \in \boldsymbol{X}} f(x_i; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})}.$$

We can rewrite the function f (we have assumed that is larger or equal than zero) as $f = \exp \log f$. We can then reqrite the last equation as

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \frac{\sum_{x_i \in \boldsymbol{X}} \nabla_{\boldsymbol{\Theta}} \exp \log f(x_i; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})}.$$

Final expression

Taking the derivative gives us

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \frac{\sum_{x_i \in \boldsymbol{X}} f(x_i; \boldsymbol{\Theta}) \nabla_{\boldsymbol{\Theta}} \log f(x_i; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})},$$

which is the expectation value of $\log f$

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \sum_{x_i \in \boldsymbol{X}} p(x_i; \boldsymbol{\Theta}) \nabla_{\boldsymbol{\Theta}} \log f(x_i; \boldsymbol{\Theta}),$$

that is

$$\nabla_{\boldsymbol{\Theta}} \log Z(\boldsymbol{\Theta}) = \mathbb{E}(\log f(x_i; \boldsymbol{\Theta})).$$

This quantity is evaluated using Monte Carlo sampling, with Gibbs sampling as the standard sampling rule. Before we discuss the explicit algorithms, we need to remind ourselves about Markov chains and sampling rules like the Metropolis-Hastings algorithm and Gibbs sampling.

Introducing the energy model

As we will see below, a typical Boltzmann machines employs a probability distribution

$$p(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta}) = \frac{f(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta})}{Z(\boldsymbol{\Theta})},$$

where $f(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta})$ is given by a so-called energy model. If we assume that the random variables x_i and h_j take binary values only, for example $x_i, h_j = \{0, 1\}$, we have a so-called binary-binary model where

$$f(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta}) = -E(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta}) = \sum_{x_i \in \boldsymbol{X}} x_i a_i + \sum_{h_j \in \boldsymbol{H}} b_j h_j + \sum_{x_i \in \boldsymbol{X}, h_j \in \boldsymbol{H}} x_i w_{ij} h_j,$$

where the set of parameters are given by the biases and weights $\boldsymbol{\Theta} = \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{W}\}$. **Note the vector notation** instead of x_i and h_j for f. The vectors $\boldsymbol{x}$ and $\boldsymbol{h}$ represent a specific instance of stochastic variables x_i and h_j. These arrangements of $\boldsymbol{x}$ and $\boldsymbol{h}$ lead to a specific energy configuration.

More compact notation

With the above definition we can write the probability as

$$p(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta}) = \frac{\exp\left(\boldsymbol{a}^T \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{h} + \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{h}\right)}{Z(\boldsymbol{\Theta})},$$

where the biases $\boldsymbol{a}$ and $\boldsymbol{h}$ and the weights defined by the matrix $\boldsymbol{W}$ are the parameters we need to optimize.

Anticipating results to be derived

Since the binary-binary energy model is linear in the parameters a_i, b_j and w_{ij}, it is easy to see that the derivatives with respect to the various optimization parameters yield expressions used in the evaluation of gradients like

$$\frac{\partial E(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta})}{\partial w_{ij}} = -x_i h_j,$$

and

$$\frac{\partial E(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta})}{\partial a_i} = -x_i,$$

and

$$\frac{\partial E(\boldsymbol{x}, \boldsymbol{h}; \boldsymbol{\Theta})}{\partial b_j} = -h_j.$$

Boltzmann Machines, marginal and conditional probabilities

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 function for Monte Carlo calculations

Generative and discriminative models

Generative and discriminative models use both gradient-descent based learning procedures for minimizing cost functions

However, in energy based models we don't use backpropagation and automatic differentiation for computing gradients, instead we turn to Markov Chain Monte Carlo methods.

A typical deep neural network has several hidden layers. A restricted Boltzmann machine has normally one hidden layer, however several RBMs can be stacked to make up deep Belief Networks, of which they constitute the building blocks.

Basics of the Boltzmann machine

A BM is what we would call an undirected probabilistic graphical model with stochastic continuous or discrete units.

It is interpreted as a stochastic recurrent neural network where the state of each unit(neurons/nodes) depends on the units it is connected to. The weights in the network represent thus the strength of the interaction between various units/nodes.

More about the basics

A standard BM network is divided into a set of observable and visible units $\boldsymbol{x}$ and a set of unknown hidden units/nodes $\boldsymbol{h}$.

Additionally there can be bias nodes for the hidden and visible layers. These biases are normally set to 1.

BMs are stackable, meaning they cwe can train a BM which serves as input to another BM. We can construct deep networks for learning complex PDFs. The layers can be trained one after another, a feature which makes them popular in deep learning

Difficult to train

However, they are often hard to train. This leads to the introduction of so-called restricted BMs, or RBMS. Here we take away all lateral connections between nodes in the visible layer as well as connections between nodes in the hidden layer.

The network layers

1. A function $\boldsymbol{x}$ that 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 $\boldsymbol{h}$ represents the hidden, or latent, layer. A vector of N elements (nodes). Also called "feature detectors".

Goal of hidden layer

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).

The parameters

The network parameters, to be optimized/learned:

1. $\boldsymbol{a}$ represents the visible bias, a vector of same length M as $\boldsymbol{x}$.

2. $\boldsymbol{b}$ represents the hidden bias, a vector of same length N as $\boldsymbol{h}$.

3. $\boldsymbol{W}$ represents the interaction weights, a matrix of size $M \times N$.

Note that we have specified the lengths of bmx and $\boldsymbol{h}$. These lengths define the number of visible and hidden units, respectively.

Joint distribution

The restricted Boltzmann machine is described by a Boltzmann distribution

$$P_{rbm}(\boldsymbol{x}, \boldsymbol{h}, \boldsymbol{\Theta}) = \frac{1}{Z(\boldsymbol{\Theta})} \exp -(E(\boldsymbol{x}, \boldsymbol{h}, \boldsymbol{\Theta})),$$

where Z is the normalization constant or partition function discussed earlier and defined as

$$Z(\boldsymbol{\Theta}) = \int \int \exp -E(\boldsymbol{x}, \boldsymbol{h}, \boldsymbol{\Theta}) d\boldsymbol{x} d\boldsymbol{h}.$$

It is common to set the temperature T to one. It is omitted in the equations above. The energy is thus a dimensionless function.

Network Elements, the energy function

The function $E(\boldsymbol{x}, \boldsymbol{h}, \boldsymbol{\Theta})$ gives the **energy** of a configuration (pair of vectors) $(\boldsymbol{x}, \boldsymbol{h})$. The lower the energy of a configuration, the higher the probability of it. This function also depends on the parameters $\boldsymbol{a}$, $\boldsymbol{b}$ 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 $E(\boldsymbol{x}, \boldsymbol{h}, \boldsymbol{\Theta})$. The connection between the nodes in the two layers is given by the weights w_{ij}.

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:

$$E(\boldsymbol{x}, \boldsymbol{h}, \boldsymbol{\Theta}) = -\sum_{i}^{M} x_i a_i - \sum_{j}^{N} b_j h_j - \sum_{i,j}^{M,N} x_i w_{ij} h_j,$$

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:

$$E(\boldsymbol{x}, \boldsymbol{h}, \boldsymbol{\Theta}) = \sum_{i}^{M} \frac{(x_i - a_i)^2}{2\sigma_i^2} - \sum_{j}^{N} b_j h_j - \sum_{i,j}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2}.$$

This type of RBMs are useful when we model continuous data (i.e., we wish $\boldsymbol{x}$ to be continuous). The paramater σ_i^2 is meant to represent a variance and is foten just set to one.

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

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

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

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

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.

Gradients

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

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

Simplifications

In order to simplify notation we defined the "operator"

$$O_i(\boldsymbol{x}) = \frac{\partial E(\boldsymbol{x}; \Theta_i)}{\partial \Theta_i},$$

and used the statistical mechanics relationship between expectation values and the log-partition function:

$$\langle O_i(\boldsymbol{x})\rangle_{model} = \mathrm{Tr} P_\Theta(\boldsymbol{x}) O_i(\boldsymbol{x}) = -\frac{\partial \log Z(\{\Theta_i\})}{\partial \Theta_i}.$$

Positive and negative phases

As discussed earlier, 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).

Gradient examples

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

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

To get the expectation values with respect to the *data*, we set the visible units to each of the observed samples in the training data, then update the hidden units according to the conditional probability found before. We then average over all samples in the training data to calculate expectation values with respect to the data.

Kullback-Leibler relative entropy

When the goal of the training is to approximate a probability distribution, as it is in generative modeling, another relevant measure is the **Kullback-Leibler divergence**, also known as the relative entropy or Shannon entropy. It is a non-symmetric measure of the dissimilarity between two probability density functions p and q. If p is the unkown probability which we approximate with q, we can measure the difference by

$$\mathrm{KL}(p||q) = \int_{-\infty}^{\infty} p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} d\boldsymbol{x}.$$

Kullback-Leibler divergence

Thus, the Kullback-Leibler divergence between the distribution of the training data $f(\boldsymbol{x})$ and the model distribution $p(\boldsymbol{x}|\boldsymbol{\Theta})$ is

$$\begin{aligned}
\mathrm{KL}(f(\boldsymbol{x})||p(\boldsymbol{x}|\boldsymbol{\Theta})) &= \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log \frac{f(\boldsymbol{x})}{p(\boldsymbol{x}|\boldsymbol{\Theta})} d\boldsymbol{x} \\
&= \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log f(\boldsymbol{x}) d\boldsymbol{x} - \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log p(\boldsymbol{x}|\boldsymbol{\Theta}) d\boldsymbol{x} \\
&= \langle \log f(\boldsymbol{x}) \rangle_{f(\boldsymbol{x})} - \langle \log p(\boldsymbol{x}|\boldsymbol{\Theta}) \rangle_{f(\boldsymbol{x})} \\
&= \langle \log f(\boldsymbol{x}) \rangle_{data} + \langle E(\boldsymbol{x}) \rangle_{data} + \log Z \\
&= \langle \log f(\boldsymbol{x}) \rangle_{data} + \mathcal{C}_{LL}.
\end{aligned}$$

Maximizing log-likelihood

The first term is constant with respect to $\boldsymbol{\Theta}$ since $f(\boldsymbol{x})$ is independent of $\boldsymbol{\Theta}$. Thus the Kullback-Leibler Divergence is minimal when the second term is minimal. The second term is the log-likelihood cost function, hence minimizing the Kullback-Leibler divergence is equivalent to maximizing the log-likelihood.

To further understand generative models it is useful to study the gradient of the cost function which is needed in order to minimize it using methods like stochastic gradient descent.

More on the partition function

The partition function is the generating function of expectation values, in particular there are mathematical relationships between expectation values and the log-partition function. In this case we have

$$\langle \frac{\partial E(\boldsymbol{x}; \Theta_i)}{\partial \Theta_i} \rangle_{model} = \int p(\boldsymbol{x}|\boldsymbol{\Theta}) \frac{\partial E(\boldsymbol{x}; \Theta_i)}{\partial \Theta_i} d\boldsymbol{x} = -\frac{\partial \log Z(\Theta_i)}{\partial \Theta_i}.$$

Here $\langle \cdot \rangle_{model}$ is the expectation value over the model probability distribution $p(\boldsymbol{x}|\boldsymbol{\Theta})$.

Setting up for gradient descent calculations

Using the previous relationship we can express the gradient of the cost function as

$$\begin{aligned}
\frac{\partial \mathcal{C}_{LL}}{\partial \Theta_i} &= \langle \frac{\partial E(\boldsymbol{x}; \Theta_i)}{\partial \Theta_i} \rangle_{data} + \frac{\partial \log Z(\Theta_i)}{\partial \Theta_i} \\
&= \langle \frac{\partial E(\boldsymbol{x}; \Theta_i)}{\partial \Theta_i} \rangle_{data} - \langle \frac{\partial E(\boldsymbol{x}; \Theta_i)}{\partial \Theta_i} \rangle_{model}
\end{aligned}$$

Difference of moments

This expression shows that the gradient of the log-likelihood cost function is a **difference of moments**, with one calculated from the data and one calculated from the model. The data-dependent term is called the **positive phase** and the model-dependent term is called the **negative phase** of the gradient. We see now that minimizing the cost function results in lowering the energy of configurations $\boldsymbol{x}$ near points in the training data and increasing the energy of configurations not observed in the training data. That means we increase the model's probability of configurations similar to those in the training data.

More observations

The gradient of the cost function also demonstrates why gradients of unsupervised, generative models must be computed differently from for those of for example FNNs. While the data-dependent expectation value is easily calculated based on the samples $\boldsymbol{x}_i$ in the training data, we must sample from the model in order to generate samples from which to caclulate the model-dependent term. We sample from the model by using MCMC-based methods. We can not sample from the model directly because the partition function Z is generally intractable.

Adding hyperparameters

As in supervised machine learning problems, the goal is also here to perform well on **unseen** data, that is to have good generalization from the training data. The distribution $f(x)$ we approximate is not the **true** distribution we wish to estimate, it is limited to the training data. Hence, in unsupervised training as well it is important to prevent overfitting to the training data. Thus it is common to add regularizers to the cost function in the same manner as we discussed for say linear regression.

Mathematical details

Because we are restricted to potential functions which are positive it is convenient to express them as exponentials.

The original RBM had binary visible and hidden nodes. They were showned to be universal approximators of discrete distributions. It was also shown that adding hidden units yields strictly improved modelling power.

Binary-binary (BB) RBMs

The common choice of binary values are 0 and 1. However, in some physics applications, -1 and 1 might be a more natural choice. We will here use 0 and 1. We habe the energy function

$$E_{BB}(\boldsymbol{x}, \boldsymbol{h}, \boldsymbol{\Theta}) = -\sum_{i}^{M} x_i a_i - \sum_{j}^{N} b_j h_j - \sum_{i,j}^{M,N} x_i w_{ij} h_j.$$

Marginal probability

We have the binary-binary marginal probability defined as

$$\begin{aligned} p_{BB}(\boldsymbol{x}, \boldsymbol{h}, \boldsymbol{\Theta}) =& \frac{1}{Z_{BB}(\boldsymbol{\Theta})} e^{\sum_i^M a_i x_i + \sum_j^N b_j h_j + \sum_{ij}^{M,N} x_i w_{ij} h_j} \\ =& \frac{1}{Z_{BB}(\boldsymbol{\Theta})} e^{\boldsymbol{a}^T \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{h} + \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{h}} \end{aligned}$$

with the partition function

$$Z_{BB}(\boldsymbol{\Theta}) = \sum_{\boldsymbol{x}, \boldsymbol{h}} e^{\boldsymbol{a}^T \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{h} + \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{h}}.$$

Marginal Probability Density Function for the visible units

In order to find the probability of any configuration of the visible units we derive the marginal probability density function.

$$\begin{aligned} p_{BB}(\boldsymbol{x}, \boldsymbol{\Theta}) =& \sum_{\boldsymbol{h}} p_{BB}(\boldsymbol{x}, \boldsymbol{h}, \boldsymbol{\Theta}) \\ =& \frac{1}{Z_{BB}} \sum_{\boldsymbol{h}} e^{\boldsymbol{a}^T \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{h} + \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{h}} \\ =& \frac{1}{Z_{BB}} e^{\boldsymbol{a}^T \boldsymbol{x}} \sum_{\boldsymbol{h}} e^{\sum_j^N (b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j} \\ =& \frac{1}{Z_{BB}} e^{\boldsymbol{a}^T \boldsymbol{x}} \sum_{\boldsymbol{h}} \prod_j^N e^{(b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j} \\ =& \frac{1}{Z_{BB}} e^{\boldsymbol{a}^T \boldsymbol{x}} \Bigg(\sum_{h_1} e^{(b_1 + \boldsymbol{x}^T \boldsymbol{w}_{*1}) h_1} \times \sum_{h_2} e^{(b_2 + \boldsymbol{x}^T \boldsymbol{w}_{*2}) h_2} \times \\ & ... \times \sum_{h_2} e^{(b_N + \boldsymbol{x}^T \boldsymbol{w}_{*N}) h_N} \Bigg) \\ =& \frac{1}{Z_{BB}} e^{\boldsymbol{a}^T \boldsymbol{x}} \prod_j^N \sum_{h_j} e^{(b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j} \\ =& \frac{1}{Z_{BB}} e^{\boldsymbol{a}^T \boldsymbol{x}} \prod_j^N (1 + e^{b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}}). \end{aligned}$$

Marginal probability for hidden units

A similar derivation yields the marginal probability of the hidden units

$$p_{BB}(\boldsymbol{h}, \boldsymbol{\Theta}) = \frac{1}{Z_{BB}(\boldsymbol{\Theta})} e^{\boldsymbol{b}^T \boldsymbol{h}} \prod_i^M (1 + e^{a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}}).$$

Conditional Probability Density Functions

We derive the probability of the hidden units given the visible units using Bayes' rule (we drop the explicit $\boldsymbol{\Theta}$ dependence)

$$\begin{aligned}
p_{BB}(\boldsymbol{h}|\boldsymbol{x}) =& \frac{p_{BB}(\boldsymbol{x}, \boldsymbol{h})}{p_{BB}(\boldsymbol{x})} \\
=& \frac{\frac{1}{Z_{BB}} e^{\boldsymbol{a}^T \boldsymbol{x} + \boldsymbol{b}^T \boldsymbol{h} + \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{h}}}{\frac{1}{Z_{BB}} e^{\boldsymbol{a}^T \boldsymbol{x}} \prod_j^N (1 + e^{b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}})} \\
=& \frac{e^{\boldsymbol{a}^T \boldsymbol{x}} e^{\sum_j^N (b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j}}{e^{\boldsymbol{a}^T \boldsymbol{x}} \prod_j^N (1 + e^{b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}})} \\
=& \prod_j^N \frac{e^{(b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j}}{1 + e^{b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}}} \\
=& \prod_j^N p_{BB}(h_j|\boldsymbol{x}).
\end{aligned}$$

On and off probabilities

From this we find the probability of a hidden unit being "on" or "off":

$$\begin{aligned}
p_{BB}(h_j = 1|\boldsymbol{x}) =& \frac{e^{(b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j}}{1 + e^{b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}}} \\
=& \frac{e^{(b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j})}}{1 + e^{b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}}} \\
=& \frac{1}{1 + e^{-(b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j})}},
\end{aligned}$$

and

$$p_{BB}(h_j = 0|\boldsymbol{x}) = \frac{1}{1 + e^{b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}}}.$$

Conditional probability for visible units

Similarly we have that the conditional probability of the visible units given the hidden are

$$\begin{aligned} p_{BB}(\boldsymbol{x}|\boldsymbol{h}) =& \prod_i^M \frac{e^{(a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h})x_i}}{1+e^{a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h}}} \\ =& \prod_i^M p_{BB}(x_i|\boldsymbol{h}). \end{aligned}$$

We have

$$\begin{aligned} p_{BB}(x_i=1|\boldsymbol{h}) =& \frac{1}{1+e^{-(a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h})}} \\ p_{BB}(x_i=0|\boldsymbol{h}) =& \frac{1}{1+e^{a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h}}}. \end{aligned}$$

Gaussian-Binary Restricted Boltzmann Machines

Inserting into the expression for $E_{RBM}(\boldsymbol{x},\boldsymbol{h},\boldsymbol{\Theta})$ in equation results in the energy

$$\begin{aligned} E_{GB}(\boldsymbol{x},\boldsymbol{h},\boldsymbol{\Theta}) =& \sum_i^M \frac{(x_i-a_i)^2}{2\sigma_i^2} - \sum_j^N b_j h_j - \sum_{ij}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2} \\ =& ||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 - \boldsymbol{b}^T\boldsymbol{h} - (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T\boldsymbol{W}\boldsymbol{h}. \end{aligned}$$

Joint Probability Density Function

$$\begin{aligned} p_{GB}(\boldsymbol{x},\boldsymbol{h},\boldsymbol{\Theta}) =& \frac{1}{Z_{GB}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2+\boldsymbol{b}^T\boldsymbol{h}+(\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T\boldsymbol{W}\boldsymbol{h}} \\ =& \frac{1}{Z_{GB}} e^{-\sum_i^M \frac{(x_i-a_i)^2}{2\sigma_i^2}+\sum_j^N b_j h_j+\sum_{ij}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2}} \\ =& \frac{1}{Z_{GB}} \prod_{ij}^{M,N} e^{-\frac{(x_i-a_i)^2}{2\sigma_i^2}+b_j h_j+\frac{x_i w_{ij} h_j}{\sigma_i^2}}. \end{aligned}$$

Partition function

The partition function is given by

$$Z_{GB} = \int \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} e^{-||\frac{\tilde{\boldsymbol{x}}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2+\boldsymbol{b}^T\tilde{\boldsymbol{h}}+(\frac{\tilde{\boldsymbol{x}}}{\boldsymbol{\sigma}^2})^T\boldsymbol{W}\tilde{\boldsymbol{h}}} d\tilde{\boldsymbol{x}}.$$

Marginal Probability Density Functions

We proceed to find the marginal probability densitites of the Gaussian-binary RBM. We first marginalize over the binary hidden units to find $p_{GB}(\boldsymbol{x})$

$$\begin{aligned} p_{GB}(\boldsymbol{x}) =& \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} p_{GB}(\boldsymbol{x}, \tilde{\boldsymbol{h}}) \\ =& \frac{1}{Z_{GB}} \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 + \boldsymbol{b}^T \tilde{\boldsymbol{h}} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W} \tilde{\boldsymbol{h}}} \\ =& \frac{1}{Z_{GB}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2} \prod_{j}^{N} (1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}). \end{aligned}$$

Then the visible units

We next marginalize over the visible units. This is the first time we marginalize over continuous values. We rewrite the exponential factor dependent on $\boldsymbol{x}$ as a Gaussian function before we integrate in the last step.

$$
\begin{aligned}
p_{GB}(\boldsymbol{h}) &= \int p_{GB}(\tilde{\boldsymbol{x}}, \boldsymbol{h}) d\tilde{\boldsymbol{x}} \\
&= \frac{1}{Z_{GB}} \int e^{-||\frac{\tilde{\boldsymbol{x}}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 + \boldsymbol{b}^T\boldsymbol{h} + (\frac{\tilde{\boldsymbol{x}}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W}\boldsymbol{h}} d\tilde{\boldsymbol{x}} \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \int \prod_i^M e^{-\frac{(\tilde{x}_i - a_i)^2}{2\sigma_i^2} + \frac{\tilde{x}_i \boldsymbol{w}_{i*}^T \boldsymbol{h}}{\sigma_i^2}} d\tilde{\boldsymbol{x}} \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \Bigg(\int e^{-\frac{(\tilde{x}_1 - a_1)^2}{2\sigma_1^2} + \frac{\tilde{x}_1 \boldsymbol{w}_{1*}^T \boldsymbol{h}}{\sigma_1^2}} d\tilde{x}_1 \\
&\quad \times \int e^{-\frac{(\tilde{x}_2 - a_2)^2}{2\sigma_2^2} + \frac{\tilde{x}_2 \boldsymbol{w}_{2*}^T \boldsymbol{h}}{\sigma_2^2}} d\tilde{x}_2 \\
&\quad \times ... \\
&\quad \times \int e^{-\frac{(\tilde{x}_M - a_M)^2}{2\sigma_M^2} + \frac{\tilde{x}_M \boldsymbol{w}_{M*}^T \boldsymbol{h}}{\sigma_M^2}} d\tilde{x}_M \Bigg) \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{(\tilde{x}_i - a_i)^2 - 2\tilde{x}_i \boldsymbol{w}_{i*}^T \boldsymbol{h}}{2\sigma_i^2}} d\tilde{x}_i \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{\tilde{x}_i^2 - 2\tilde{x}_i(a_i + \tilde{x}_i \boldsymbol{w}_{i*}^T \boldsymbol{h}) + a_i^2}{2\sigma_i^2}} d\tilde{x}_i \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{\tilde{x}_i^2 - 2\tilde{x}_i(a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}) + (a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h})^2 - (a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h})^2 + a_i^2}{2\sigma_i^2}} d\tilde{x}_i \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{(\tilde{x}_i - (a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}))^2 - a_i^2 - 2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} - (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2 + a_i^2}{2\sigma_i^2}} d\tilde{x}_i \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}} \int e^{-\frac{(\tilde{x}_i - a_i - \boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}} d\tilde{x}_i \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \sqrt{2\pi\sigma_i^2} e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}}.
\end{aligned}
$$

Conditional Probability Density Functions

We finish by deriving the conditional probabilities.

$$\begin{aligned}
p_{GB}(\boldsymbol{h}|\boldsymbol{x}) =& \frac{p_{GB}(\boldsymbol{x},\boldsymbol{h})}{p_{GB}(\boldsymbol{x})} \\
=& \frac{\frac{1}{Z_{GB}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 + \boldsymbol{b}^T\boldsymbol{h} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W}\boldsymbol{h}}}{\frac{1}{Z_{GB}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2} \prod_j^N (1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}})} \\
=& \prod_j^N \frac{e^{(b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}) h_j}}{1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}} \\
=& \prod_j^N p_{GB}(h_j|\boldsymbol{x}).
\end{aligned}$$

Hidden units

The conditional probability of a binary hidden unit h_j being on or off again takes the form of a sigmoid function

$$\begin{aligned}
p_{GB}(h_j = 1|\boldsymbol{x}) =& \frac{e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}}{1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}} \\
=& \frac{1}{1 + e^{-b_j - (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}} \\
p_{GB}(h_j = 0|\boldsymbol{x}) =& \frac{1}{1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}}.
\end{aligned}$$

Visible units

The conditional probability of the continuous $\boldsymbol{x}$ now has another form, however.

$$
\begin{aligned}
p_{GB}(\boldsymbol{x}|\boldsymbol{h}) =& \frac{p_{GB}(\boldsymbol{x},\boldsymbol{h})}{p_{GB}(\boldsymbol{h})} \\
=& \frac{\frac{1}{Z_{GB}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 + \boldsymbol{b}^T\boldsymbol{h} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W}\boldsymbol{h}}}{\frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \sqrt{2\pi\sigma_i^2} e^{\frac{2a_i \boldsymbol{w}_{i*}^T\boldsymbol{h} + (\boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} \frac{e^{-\frac{(x_i-a_i)^2}{2\sigma_i^2} + \frac{x_i \boldsymbol{w}_{i*}^T\boldsymbol{h}}{2\sigma_i^2}}}{e^{\frac{2a_i \boldsymbol{w}_{i*}^T\boldsymbol{h} + (\boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} \frac{e^{-\frac{x_i^2 - 2a_i x_i + a_i^2 - 2x_i \boldsymbol{w}_{i*}^T\boldsymbol{h}}{2\sigma_i^2}}}{e^{\frac{2a_i \boldsymbol{w}_{i*}^T\boldsymbol{h} + (\boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{x_i^2 - 2a_i x_i + a_i^2 - 2x_i \boldsymbol{w}_{i*}^T\boldsymbol{h} + 2a_i \boldsymbol{w}_{i*}^T\boldsymbol{h} + (\boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(x_i - b_i - \boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}} \\
=& \prod_i^M \mathcal{N}(x_i|b_i + \boldsymbol{w}_{i*}^T\boldsymbol{h}, \sigma_i^2) \\
\Rightarrow p_{GB}(x_i|\boldsymbol{h}) =& \mathcal{N}(x_i|b_i + \boldsymbol{w}_{i*}^T\boldsymbol{h}, \sigma_i^2).
\end{aligned}
$$

Comments

The form of these conditional probabilities explains the name "Gaussian" and the form of the Gaussian-binary energy function. We see that the conditional probability of x_i given $\boldsymbol{h}$ is a normal distribution with mean $b_i + \boldsymbol{w}_{i*}^T\boldsymbol{h}$ and variance σ_i^2.

For the quantum mechanical calculations however, there are several ingredients which simplify our calculations.

Neural Quantum States

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

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

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

$$
\begin{aligned}
F_{rbm}(\mathbf{x}) &= \sum_{\mathbf{h}} F_{rbm}(\mathbf{x}, \mathbf{h}) \\
&= \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{x},\mathbf{h})}
\end{aligned}
$$

Model for the trial wave function

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

$$
\begin{aligned}
\Psi(\mathbf{X}) &= F_{rbm}(\mathbf{x}) \\
&= \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\mathbf{x},\mathbf{h})} \\
&= \frac{1}{Z} \sum_{\{h_j\}} e^{-\sum_i^M \frac{(x_i - a_i)^2}{2\sigma^2} + \sum_j^N b_j h_j + \sum_{i,j}^{M,N} \frac{x_i w_{ij} h_j}{\sigma^2}} \\
&= \frac{1}{Z} e^{-\sum_i^M \frac{(x_i - a_i)^2}{2\sigma^2}} \prod_j^N (1 + e^{b_j + \sum_i^M \frac{x_i w_{ij}}{\sigma^2}})
\end{aligned}
$$

Allowing for complex valued functions

The above wavefunction is the most general one because it allows for complex valued wavefunctions. However it fundamentally changes the probabilistic foundation of the RBM, because what is usually a probability in the RBM framework is now a an amplitude. This means that a lot of the theoretical framework usually used to interpret the model, i.e. graphical models, conditional probabilities, and Markov random fields, breaks down.

Squared wave function

If we assume the wavefunction to be positive definite, however, we can use the RBM to represent the squared wavefunction, and thereby a probability. This also makes it possible to sample from the model using Gibbs sampling, because we can obtain the conditional probabilities.

$$
\begin{aligned}
|\Psi(\mathbf{X})|^2 &= F_{rbm}(\mathbf{X}) \\
\Rightarrow \Psi(\mathbf{X}) &= \sqrt{F_{rbm}(\mathbf{X})} \\
&= \frac{1}{\sqrt{Z}}\sqrt{\sum_{\{h_j\}} e^{-E(\mathbf{X},\mathbf{h})}} \\
&= \frac{1}{\sqrt{Z}}\sqrt{\sum_{\{h_j\}} e^{-\sum_i^M \frac{(X_i-a_i)^2}{2\sigma^2}+\sum_j^N b_j h_j+\sum_{i,j}^{M,N}\frac{X_i w_{ij} h_j}{\sigma^2}}} \\
&= \frac{1}{\sqrt{Z}} e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\sqrt{\sum_{\{h_j\}}\prod_j^N e^{b_j h_j+\sum_i^M \frac{X_i w_{ij} h_j}{\sigma^2}}} \\
&= \frac{1}{\sqrt{Z}} e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\sqrt{\prod_j^N\sum_{h_j} e^{b_j h_j+\sum_i^M \frac{X_i w_{ij} h_j}{\sigma^2}}} \\
&= \frac{1}{\sqrt{Z}} e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\prod_j^N\sqrt{e^0+e^{b_j+\sum_i^M \frac{X_i w_{ij}}{\sigma^2}}} \\
&= \frac{1}{\sqrt{Z}} e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\prod_j^N\sqrt{1+e^{b_j+\sum_i^M \frac{X_i w_{ij}}{\sigma^2}}}
\end{aligned}
$$

Cost function

This is where we deviate from what is common in machine learning. Rather than defining a cost function based on some dataset, our cost function is the energy of the quantum mechanical system. From the variational principle we know that minimizing this energy should lead to the ground state wavefunction. As stated previously the local energy is given by

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

And the gradient

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

where $\alpha_i = a_1, ..., a_M, b_1, ..., b_N, w_{11}, ..., w_{MN}$.

Additional details

We use that $\frac{1}{\Psi}\frac{\partial\Psi}{\partial\alpha_i} = \frac{\partial \ln\Psi}{\partial\alpha_i}$, and find

$$\ln \Psi(\mathbf{X}) = -\ln Z - \sum_m^M \frac{(X_m - a_m)^2}{2\sigma^2} + \sum_n^N \ln(1 + e^{b_n + \sum_i^M \frac{X_i w_{in}}{\sigma^2}}).$$

Final equation

This gives

$$\begin{aligned}
\frac{\partial}{\partial a_m} \ln \Psi &= \frac{1}{\sigma^2}(X_m - a_m) \\
\frac{\partial}{\partial b_n} \ln \Psi &= \frac{1}{e^{-b_n - \frac{1}{\sigma^2}\sum_i^M X_i w_{in}} + 1} \\
\frac{\partial}{\partial w_{mn}} \ln \Psi &= \frac{X_m}{\sigma^2(e^{-b_n - \frac{1}{\sigma^2}\sum_i^M X_i w_{in}} + 1)}.
\end{aligned}$$

Code example

This part is best seen using the jupyter-notebook

```
# 2-electron VMC code for 2dim quantum dot with importance sampling
# Using gaussian rng for new positions and Metropolis- Hastings
# Added restricted boltzmann machine method for dealing with the wavefunction
# RBM code based heavily off of:
# https://github.com/CompPhysics/ComputationalPhysics2/tree/gh-pages/doc/Programs/BoltzmannMachin
from math import exp, sqrt
from random import random, seed, normalvariate
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import sys



# Trial wave function for the 2-electron quantum dot in two dims
def WaveFunction(r,a,b,w):
    sigma=1.0
    sig2 = sigma**2
    Psi1 = 0.0
    Psi2 = 1.0
    Q = Qfac(r,b,w)

    for iq in range(NumberParticles):
        for ix in range(Dimension):
            Psi1 += (r[iq,ix]-a[iq,ix])**2

    for ih in range(NumberHidden):
        Psi2 *= (1.0 + np.exp(Q[ih]))

    Psi1 = np.exp(-Psi1/(2*sig2))
```

```
    return Psi1*Psi2

# Local energy  for the 2-electron quantum dot in two dims, using analytical local energy
def LocalEnergy(r,a,b,w):
    sigma=1.0
    sig2 = sigma**2
    locenergy = 0.0

    Q = Qfac(r,b,w)

    for iq in range(NumberParticles):
        for ix in range(Dimension):
            sum1 = 0.0
            sum2 = 0.0
            for ih in range(NumberHidden):
                sum1 += w[iq,ix,ih]/(1+np.exp(-Q[ih]))
                sum2 += w[iq,ix,ih]**2 * np.exp(Q[ih]) / (1.0 + np.exp(Q[ih]))**2

            dlnpsi1 = -(r[iq,ix] - a[iq,ix]) /sig2 + sum1/sig2
            dlnpsi2 = -1/sig2 + sum2/sig2**2
            locenergy += 0.5*(-dlnpsi1*dlnpsi1 - dlnpsi2 + r[iq,ix]**2)

    if(interaction==True):
        for iq1 in range(NumberParticles):
            for iq2 in range(iq1):
                distance = 0.0
                for ix in range(Dimension):
                    distance += (r[iq1,ix] - r[iq2,ix])**2

                locenergy += 1/sqrt(distance)

    return locenergy

# Derivate of wave function ansatz as function of variational parameters
def DerivativeWFansatz(r,a,b,w):

    sigma=1.0
    sig2 = sigma**2

    Q = Qfac(r,b,w)

    WfDer = np.empty((3,),dtype=object)
    WfDer = [np.copy(a),np.copy(b),np.copy(w)]

    WfDer[0] = (r-a)/sig2
    WfDer[1] = 1 / (1 + np.exp(-Q))

    for ih in range(NumberHidden):
        WfDer[2][:,:,ih] = w[:,:,ih] / (sig2*(1+np.exp(-Q[ih])))

    return  WfDer

# Setting up the quantum force for the two-electron quantum dot, recall that it is a vector
def QuantumForce(r,a,b,w):

    sigma=1.0
    sig2 = sigma**2

    qforce = np.zeros((NumberParticles,Dimension), np.double)
    sum1 = np.zeros((NumberParticles,Dimension), np.double)
```

```
    Q = Qfac(r,b,w)

    for ih in range(NumberHidden):
        sum1 += w[:,:,ih]/(1+np.exp(-Q[ih]))

    qforce = 2*(-(r-a)/sig2 + sum1/sig2)

    return qforce

def Qfac(r,b,w):
    Q = np.zeros((NumberHidden), np.double)
    temp = np.zeros((NumberHidden), np.double)

    for ih in range(NumberHidden):
        temp[ih] = (r*w[:,:,ih]).sum()

    Q = b + temp

    return Q

# Computing the derivative of the energy and the energy
def EnergyMinimization(a,b,w):

    NumberMCcycles= 10000
    # Parameters in the Fokker-Planck simulation of the quantum force
    D = 0.5
    TimeStep = 0.05
    # positions
    PositionOld = np.zeros((NumberParticles,Dimension), np.double)
    PositionNew = np.zeros((NumberParticles,Dimension), np.double)
    # Quantum force
    QuantumForceOld = np.zeros((NumberParticles,Dimension), np.double)
    QuantumForceNew = np.zeros((NumberParticles,Dimension), np.double)

    # seed for rng generator
    seed()
    energy = 0.0
    DeltaE = 0.0

    EnergyDer = np.empty((3,),dtype=object)
    DeltaPsi = np.empty((3,),dtype=object)
    DerivativePsiE = np.empty((3,),dtype=object)
    EnergyDer = [np.copy(a),np.copy(b),np.copy(w)]
    DeltaPsi = [np.copy(a),np.copy(b),np.copy(w)]
    DerivativePsiE = [np.copy(a),np.copy(b),np.copy(w)]
    for i in range(3): EnergyDer[i].fill(0.0)
    for i in range(3): DeltaPsi[i].fill(0.0)
    for i in range(3): DerivativePsiE[i].fill(0.0)


    #Initial position
    for i in range(NumberParticles):
        for j in range(Dimension):
            PositionOld[i,j] = normalvariate(0.0,1.0)*sqrt(TimeStep)
    wfold = WaveFunction(PositionOld,a,b,w)
    QuantumForceOld = QuantumForce(PositionOld,a,b,w)

    #Loop over MC MCcycles
    for MCcycle in range(NumberMCcycles):
        #Trial position moving one particle at the time
        for i in range(NumberParticles):
```

```
            for j in range(Dimension):
                PositionNew[i,j] = PositionOld[i,j]+normalvariate(0.0,1.0)*sqrt(TimeStep)+\
                                   QuantumForceOld[i,j]*TimeStep*D
            wfnew = WaveFunction(PositionNew,a,b,w)
            QuantumForceNew = QuantumForce(PositionNew,a,b,w)

            GreensFunction = 0.0
            for j in range(Dimension):
                GreensFunction += 0.5*(QuantumForceOld[i,j]+QuantumForceNew[i,j])*\
                                  (D*TimeStep*0.5*(QuantumForceOld[i,j]-QuantumForceNew[i,j])-
                                  PositionNew[i,j]+PositionOld[i,j])

            GreensFunction = exp(GreensFunction)
            ProbabilityRatio = GreensFunction*wfnew**2/wfold**2
            #Metropolis-Hastings test to see whether we accept the move
            if random() <= ProbabilityRatio:
                for j in range(Dimension):
                    PositionOld[i,j] = PositionNew[i,j]
                    QuantumForceOld[i,j] = QuantumForceNew[i,j]
                wfold = wfnew
        #print("wf new:         ", wfnew)
        #print("force on 1 new:", QuantumForceNew[0,:])
        #print("pos of 1 new:  ", PositionNew[0,:])
        #print("force on 2 new:", QuantumForceNew[1,:])
        #print("pos of 2 new:  ", PositionNew[1,:])
        DeltaE = LocalEnergy(PositionOld,a,b,w)
        DerPsi = DerivativeWFansatz(PositionOld,a,b,w)

        DeltaPsi[0] += DerPsi[0]
        DeltaPsi[1] += DerPsi[1]
        DeltaPsi[2] += DerPsi[2]

        energy += DeltaE

        DerivativePsiE[0] += DerPsi[0]*DeltaE
        DerivativePsiE[1] += DerPsi[1]*DeltaE
        DerivativePsiE[2] += DerPsi[2]*DeltaE

    # We calculate mean values
    energy /= NumberMCcycles
    DerivativePsiE[0] /= NumberMCcycles
    DerivativePsiE[1] /= NumberMCcycles
    DerivativePsiE[2] /= NumberMCcycles
    DeltaPsi[0] /= NumberMCcycles
    DeltaPsi[1] /= NumberMCcycles
    DeltaPsi[2] /= NumberMCcycles
    EnergyDer[0]  = 2*(DerivativePsiE[0]-DeltaPsi[0]*energy)
    EnergyDer[1]  = 2*(DerivativePsiE[1]-DeltaPsi[1]*energy)
    EnergyDer[2]  = 2*(DerivativePsiE[2]-DeltaPsi[2]*energy)
    return energy, EnergyDer


#Here starts the main program with variable declarations
NumberParticles = 2
Dimension = 2
NumberHidden = 2

interaction=False

# guess for parameters
a=np.random.normal(loc=0.0, scale=0.001, size=(NumberParticles,Dimension))
```

```
b=np.random.normal(loc=0.0, scale=0.001, size=(NumberHidden))
w=np.random.normal(loc=0.0, scale=0.001, size=(NumberParticles,Dimension,NumberHidden))
# Set up iteration using stochastic gradient method
Energy = 0
EDerivative = np.empty((3,),dtype=object)
EDerivative = [np.copy(a),np.copy(b),np.copy(w)]
# Learning rate eta, max iterations, need to change to adaptive learning rate
eta = 0.001
MaxIterations = 50
iter = 0
np.seterr(invalid='raise')
Energies = np.zeros(MaxIterations)
EnergyDerivatives1 = np.zeros(MaxIterations)
EnergyDerivatives2 = np.zeros(MaxIterations)

while iter < MaxIterations:
    Energy, EDerivative = EnergyMinimization(a,b,w)
    agradient = EDerivative[0]
    bgradient = EDerivative[1]
    wgradient = EDerivative[2]
    a -= eta*agradient
    b -= eta*bgradient
    w -= eta*wgradient
    Energies[iter] = Energy
    print("Energy:",Energy)
    #EnergyDerivatives1[iter] = EDerivative[0]
    #EnergyDerivatives2[iter] = EDerivative[1]
    #EnergyDerivatives3[iter] = EDerivative[2]


    iter += 1

#nice printout with Pandas
import pandas as pd
from pandas import DataFrame
pd.set_option('max_columns', 6)
data ={'Energy':Energies}#,'A Derivative':EnergyDerivatives1,'B Derivative':EnergyDerivatives2,'W

frame = pd.DataFrame(data)
print(frame)
```

Project 2, VMC for fermions: Efficient calculation of Slater determinants

The potentially most time-consuming part is the evaluation of the gradient and the Laplacian of an N-particle Slater determinant.

We have to differentiate the determinant with respect to all spatial coordinates of all particles. A brute force differentiation would involve $N \cdot d$ evaluations of the entire determinant which would even worsen the already undesirable time scaling, making it $Nd \cdot O(N^3) \sim O(d \cdot N^4)$.

This poses serious hindrances to the overall efficiency of our code.

Matrix elements of Slater determinants

The efficiency can be improved however if we move only one electron at the time. The Slater determinant matrix $\hat{D}$ is defined by the matrix elements

$$d_{ij} = \phi_j(x_i)$$

where $\phi_j(\mathbf{r}_i)$ is a single particle wave function. The columns correspond to the position of a given particle while the rows stand for the various quantum numbers.

Efficient calculation of Slater determinants

What we need to realize is that when differentiating a Slater determinant with respect to some given coordinate, only one row of the corresponding Slater matrix is changed.

Therefore, by recalculating the whole determinant we risk producing redundant information. The solution turns out to be an algorithm that requires to keep track of the *inverse* of the Slater matrix.

Efficient calculation of Slater determinants

Let the current position in phase space be represented by the $(N \cdot d)$-element vector $\mathbf{r}^{\text{old}}$ and the new suggested position by the vector $\mathbf{r}^{\text{new}}$.

The inverse of $\hat{D}$ can be expressed in terms of its cofactors C_{ij} and its determinant (this our notation for a determinant) $|\hat{D}|$:

$$d_{ij}^{-1} = \frac{C_{ji}}{|\hat{D}|} \tag{1}$$

Notice that the interchanged indices indicate that the matrix of cofactors is to be transposed.

Efficient calculation of Slater determinants

If $\hat{D}$ is invertible, then we must obviously have $\hat{D}^{-1}\hat{D} = \mathbf{1}$, or explicitly in terms of the individual elements of $\hat{D}$ and $\hat{D}^{-1}$:

$$\sum_{k=1}^{N} d_{ik} d_{kj}^{-1} = \delta_{ij} \tag{2}$$

Efficient calculation of Slater determinants

Consider the ratio, which we shall call R, between $|\hat{D}(\mathbf{r}^{\text{new}})|$ and $|\hat{D}(\mathbf{r}^{\text{old}})|$. By definition, each of these determinants can individually be expressed in terms of the i-th row of its cofactor matrix

$$R \equiv \frac{|\hat{D}(\mathbf{r}^{\text{new}})|}{|\hat{D}(\mathbf{r}^{\text{old}})|} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, C_{ij}(\mathbf{r}^{\text{new}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, C_{ij}(\mathbf{r}^{\text{old}})} \tag{3}$$

Efficient calculation of Slater determinants

Suppose now that we move only one particle at a time, meaning that $\mathbf{r}^{\text{new}}$ differs from $\mathbf{r}^{\text{old}}$ by the position of only one, say the i-th, particle . This means that $\hat{D}(\mathbf{r}^{\text{new}})$ and $\hat{D}(\mathbf{r}^{\text{old}})$ differ only by the entries of the i-th row. Recall also that the i-th row of a cofactor matrix $\hat{C}$ is independent of the entries of the i-th row of its corresponding matrix $\hat{D}$. In this particular case we therefore get that the i-th row of $\hat{C}(\mathbf{r}^{\text{new}})$ and $\hat{C}(\mathbf{r}^{\text{old}})$ must be equal. Explicitly, we have:

$$C_{ij}(\mathbf{r}^{\text{new}}) = C_{ij}(\mathbf{r}^{\text{old}}) \quad \forall\, j \in \{1, \ldots, N\} \tag{4}$$

Efficient calculation of Slater determinants

Inserting this into the numerator of eq. (3) and using eq. (1) to substitute the cofactors with the elements of the inverse matrix, we get:

$$R = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, C_{ij}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, C_{ij}(\mathbf{r}^{\text{old}})} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}})} \tag{5}$$

Efficient calculation of Slater determinants

Now by eq. (2) the denominator of the rightmost expression must be unity, so that we finally arrive at:

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}}) = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}}) \tag{6}$$

What this means is that in order to get the ratio when only the i-th particle has been moved, we only need to calculate the dot product of the vector $(\phi_1(\mathbf{r}_i^{\text{new}}), \ldots, \phi_N(\mathbf{r}_i^{\text{new}}))$ of single particle wave functions evaluated at this new position with the i-th column of the inverse matrix $\hat{D}^{-1}$ evaluated at the original position. Such an operation has a time scaling of $O(N)$. The only extra thing we need to do is to maintain the inverse matrix $\hat{D}^{-1}(\mathbf{x}^{\text{old}})$.

Efficient calculation of Slater determinants

If the new position $\mathbf{r}^{\text{new}}$ is accepted, then the inverse matrix can by suitably updated by an algorithm having a time scaling of $O(N^2)$. This algorithm goes as follows. First we update all but the i-th column of $\hat{D}^{-1}$. For each column $j \neq i$, we first calculate the quantity:

$$S_j = (\hat{D}(\mathbf{r}^{\text{new}}) \times \hat{D}^{-1}(\mathbf{r}^{\text{old}}))_{ij} = \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\text{new}})\, d_{lj}^{-1}(\mathbf{r}^{\text{old}}) \tag{7}$$

Efficient calculation of Slater determinants

The new elements of the j-th column of $\hat{D}^{-1}$ are then given by:

$$d_{kj}^{-1}(\mathbf{r}^{\rm new}) = d_{kj}^{-1}(\mathbf{r}^{\rm old}) - \frac{S_j}{R}\, d_{ki}^{-1}(\mathbf{r}^{\rm old}) \qquad \begin{array}{l} \forall \;\; k \in \{1,\ldots,N\} \\ j \neq i \end{array} \tag{8}$$

Efficient calculation of Slater determinants

Finally the elements of the i-th column of $\hat{D}^{-1}$ are updated simply as follows:

$$d_{ki}^{-1}(\mathbf{r}^{\rm new}) = \frac{1}{R}\, d_{ki}^{-1}(\mathbf{r}^{\rm old}) \quad \forall \;\; k \in \{1,\ldots,N\} \tag{9}$$

We see from these formulas that the time scaling of an update of $\hat{D}^{-1}$ after changing one row of $\hat{D}$ is $O(N^2)$.

The scheme is also applicable for the calculation of the ratios involving derivatives. It turns out that differentiating the Slater determinant with respect to the coordinates of a single particle $\mathbf{r}_i$ changes only the i-th row of the corresponding Slater matrix.

The gradient and the Laplacian

The gradient and the Laplacian can therefore be calculated as follows:

$$\frac{\vec{\nabla}_i |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \vec{\nabla}_i d_{ij}(\mathbf{r}) d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \vec{\nabla}_i \phi_j(\mathbf{r}_i) d_{ji}^{-1}(\mathbf{r})$$

and

$$\frac{\nabla_i^2 |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \nabla_i^2 d_{ij}(\mathbf{r}) d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \nabla_i^2 \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

How to compute the derivates of the Slater determinant

Thus, to calculate all the derivatives of the Slater determinant, we only need the derivatives of the single particle wave functions ($\vec{\nabla}_i \phi_j(\mathbf{r}_i)$ and $\nabla_i^2 \phi_j(\mathbf{r}_i)$) and the elements of the corresponding inverse Slater matrix ($\hat{D}^{-1}(\mathbf{r}_i)$). A calculation of a single derivative is by the above result an $O(N)$ operation. Since there are $d \cdot N$ derivatives, the time scaling of the total evaluation becomes $O(d \cdot N^2)$. With an $O(N^2)$ updating algorithm for the inverse matrix, the total scaling is no worse, which is far better than the brute force approach yielding $O(d \cdot N^4)$.

Important note: In most cases you end with closed form expressions for the single-particle wave functions. It is then useful to calculate the various derivatives and make separate functions for them.

The Slater determinant

The Slater determinant takes the form

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta)=\frac{1}{\sqrt{4!}}\begin{vmatrix}\psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) & \psi_{100\uparrow}(\mathbf{r}_3) & \psi_{100\uparrow}(\mathbf{r}_4)\\ \psi_{100\downarrow}(\mathbf{r}_1) & \psi_{100\downarrow}(\mathbf{r}_2) & \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4)\\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) & \psi_{200\uparrow}(\mathbf{r}_3) & \psi_{200\uparrow}(\mathbf{r}_4)\\ \psi_{200\downarrow}(\mathbf{r}_1) & \psi_{200\downarrow}(\mathbf{r}_2) & \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4)\end{vmatrix}.$$

The Slater determinant as written is zero since the spatial wave functions for the spin up and spin down states are equal. But we can rewrite it as the product of two Slater determinants, one for spin up and one for spin down.

Rewriting the Slater determinant

We can rewrite it as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta)=\det\uparrow(1,2)\det\downarrow(3,4)-\det\uparrow(1,3)\det\downarrow(2,4)$$
$$-\det\uparrow(1,4)\det\downarrow(3,2)+\det\uparrow(2,3)\det\downarrow(1,4)-\det\uparrow(2,4)\det\downarrow(1,3)$$
$$+\det\uparrow(3,4)\det\downarrow(1,2),$$

where we have defined

$$\det\uparrow(1,2)=\frac{1}{\sqrt{2}}\begin{vmatrix}\psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2)\\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2)\end{vmatrix},$$

and

$$\det\downarrow(3,4)=\frac{1}{\sqrt{2}}\begin{vmatrix}\psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4)\\ \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4)\end{vmatrix}.$$

The total determinant is still zero!

Splitting the Slater determinant

We want to avoid to sum over spin variables, in particular when the interaction does not depend on spin.

It can be shown, see for example Moskowitz and Kalos, Int. J. Quantum Chem. **20** 1107 (1981), that for the variational energy we can approximate the Slater determinant as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta)\propto\det\uparrow(1,2)\det\downarrow(3,4),$$

or more generally as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\dots\mathbf{r}_N)\propto\det\uparrow\det\downarrow,$$

where we have the Slater determinant as the product of a spin up part involving the number of electrons with spin up only (2 for beryllium and 5 for neon) and a spin down part involving the electrons with spin down.

This ansatz is not antisymmetric under the exchange of electrons with opposite spins but it can be shown (show this) that it gives the same expectation value for the energy as the full Slater determinant.

As long as the Hamiltonian is spin independent, the above is correct. It is rather straightforward to see this if you go back to the equations for the energy discussed earlier this semester.

Spin up and spin down parts

We will thus factorize the full determinant $|\hat{D}|$ into two smaller ones, where each can be identified with $\uparrow$ and $\downarrow$ respectively:

$$|\hat{D}| = |\hat{D}|_{\uparrow} \cdot |\hat{D}|_{\downarrow}$$

Factorization

The combined dimensionality of the two smaller determinants equals the dimensionality of the full determinant. Such a factorization is advantageous in that it makes it possible to perform the calculation of the ratio R and the updating of the inverse matrix separately for $|\hat{D}|_{\uparrow}$ and $|\hat{D}|_{\downarrow}$:

$$\frac{|\hat{D}|^{\text{new}}}{|\hat{D}|^{\text{old}}} = \frac{|\hat{D}|^{\text{new}}_{\uparrow}}{|\hat{D}|^{\text{old}}_{\uparrow}} \cdot \frac{|\hat{D}|^{\text{new}}_{\downarrow}}{|\hat{D}|^{\text{old}}_{\downarrow}}$$

This reduces the calculation time by a constant factor. The maximal time reduction happens in a system of equal numbers of $\uparrow$ and $\downarrow$ particles, so that the two factorized determinants are half the size of the original one.

Number of operations

Consider the case of moving only one particle at a time which originally had the following time scaling for one transition:

$$O_R(N) + O_{\text{inverse}}(N^2)$$

For the factorized determinants one of the two determinants is obviously unaffected by the change so that it cancels from the ratio R.

Counting the number of FLOPS

Therefore, only one determinant of size $N/2$ is involved in each calculation of R and update of the inverse matrix. The scaling of each transition then becomes:

$$O_R(N/2) + O_{\text{inverse}}(N^2/4)$$

and the time scaling when the transitions for all N particles are put together:

$$O_R(N^2/2) + O_{\text{inverse}}(N^3/4)$$

which gives the same reduction as in the case of moving all particles at once.

Computation of ratios

Computing the ratios discussed above requires that we maintain the inverse of the Slater matrix evaluated at the current position. Each time a trial position is accepted, the row number i of the Slater matrix changes and updating its inverse has to be carried out. Getting the inverse of an $N \times N$ matrix by Gaussian elimination has a complexity of order of $\mathcal{O}(N^3)$ operations, a luxury that we cannot afford for each time a particle move is accepted. We will use the expression

$$d_{kj}^{-1}(\mathbf{x^{new}}) = \begin{cases} d_{kj}^{-1}(\mathbf{x^{old}}) - \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R} \sum_{l=1}^{N} d_{il}(\mathbf{x^{new}}) d_{lj}^{-1}(\mathbf{x^{old}}) & \text{if } j \neq i \\ \\ \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R} \sum_{l=1}^{N} d_{il}(\mathbf{x^{old}}) d_{lj}^{-1}(\mathbf{x^{old}}) & \text{if } j = i \end{cases}$$

Scaling properties

This equation scales as $O(N^2)$. The evaluation of the determinant of an $N \times N$ matrix by standard Gaussian elimination requires $\mathbf{O}(N^3)$ calculations. As there are Nd independent coordinates we need to evaluate Nd Slater determinants for the gradient (quantum force) and Nd for the Laplacian (kinetic energy). With the updating algorithm we need only to invert the Slater determinant matrix once. This can be done by standard LU decomposition methods.

How to get the determinant

Determining a determinant of an $N \times N$ matrix by standard Gaussian elimination is of the order of $\mathbf{O}(N^3)$ calculations. As there are $N \cdot d$ independent coordinates we need to evaluate Nd Slater determinants for the gradient (quantum force) and $N \cdot d$ for the Laplacian (kinetic energy)

With the updating algorithm we need only to invert the Slater determinant matrix once. This is done by calling standard LU decomposition methods.

Expectation value of the kinetic energy

The expectation value of the kinetic energy expressed in atomic units for electron i is

$$\langle \hat{K}_i \rangle = -\frac{1}{2} \frac{\langle \Psi | \nabla_i^2 | \Psi \rangle}{\langle \Psi | \Psi \rangle},$$

$$K_i = -\frac{1}{2} \frac{\nabla_i^2 \Psi}{\Psi}. \tag{10}$$

$$\begin{aligned} \frac{\nabla^2 \Psi}{\Psi} &= \frac{\nabla^2 (\Psi_D \, \Psi_C)}{\Psi_D \, \Psi_C} = \frac{\nabla \cdot [\nabla (\Psi_D \, \Psi_C)]}{\Psi_D \, \Psi_C} = \frac{\nabla \cdot [\Psi_C \nabla \Psi_D + \Psi_D \nabla \Psi_C]}{\Psi_D \, \Psi_C} \\ &= \frac{\nabla \Psi_C \cdot \nabla \Psi_D + \Psi_C \nabla^2 \Psi_D + \nabla \Psi_D \cdot \nabla \Psi_C + \Psi_D \nabla^2 \Psi_C}{\Psi_D \, \Psi_C} \end{aligned} \tag{11}$$

$$\frac{\nabla^2\Psi}{\Psi} = \frac{\nabla^2\Psi_D}{\Psi_D} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\frac{\nabla\Psi_D}{\Psi_D}\cdot\frac{\nabla\Psi_C}{\Psi_C} \tag{12}$$

Second derivative of the Jastrow factor

The second derivative of the Jastrow factor divided by the Jastrow factor (the way it enters the kinetic energy) is

$$\left[\frac{\nabla^2\Psi_C}{\Psi_C}\right]_x = 2\sum_{k=1}^{N}\sum_{i=1}^{k-1}\frac{\partial^2 g_{ik}}{\partial x_k^2} + \sum_{k=1}^{N}\left(\sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}\right)^2$$

Functional form

But we have a simple form for the function, namely

$$\Psi_C = \prod_{i<j}\exp f(r_{ij}) = \exp\left\{\sum_{i<j}\frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

and it is easy to see that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}f'(r_{ki})f'(r_{kj}) + \sum_{j\neq k}\left(f''(r_{kj}) + \frac{2}{r_{kj}}f'(r_{kj})\right)$$

Second derivative of the Jastrow factor

Using

$$f(r_{ij}) = \frac{ar_{ij}}{1+\beta r_{ij}},$$

and $g'(r_{kj}) = dg(r_{kj})/dr_{kj}$ and $g''(r_{kj}) = d^2g(r_{kj})/dr_{kj}^2$ we find that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}\frac{a}{(1+\beta r_{ki})^2}\frac{a}{(1+\beta r_{kj})^2} + \sum_{j\neq k}\left(\frac{2a}{r_{kj}(1+\beta r_{kj})^2} - \frac{2a\beta}{(1+\beta r_{kj})^3}\right)$$

Gradient and Laplacian

The gradient and Laplacian can be calculated as follows:

$$\frac{\nabla_i|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N}\vec{\nabla}_i d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N}\vec{\nabla}_i\phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

and

$$\frac{\nabla_i^2|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N}\nabla_i^2 d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N}\nabla_i^2\phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

The gradient for the determinant

The gradient for the determinant is

$$\frac{\nabla_i|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \nabla_i d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \nabla_i \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r}).$$

Jastrow gradient in quantum force

We have

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j} \frac{a r_{ij}}{1+\beta r_{ij}}\right\},$$

the gradient needed for the quantum force and local energy is easy to compute. We get for particle k

$$\frac{\nabla_k \Psi_C}{\Psi_C} = \sum_{j\neq k} \frac{\mathbf{r}_{kj}}{r_{kj}} \frac{a}{(1+\beta r_{kj})^2},$$

which is rather easy to code. Remember to sum over all particles when you compute the local energy.

Metropolis Hastings part

We need to compute the ratio between wave functions, in particular for the Slater determinants.

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\mathrm{new}})\, d_{ji}^{-1}(\mathbf{r}^{\mathrm{old}}) = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\mathrm{new}})\, d_{ji}^{-1}(\mathbf{r}^{\mathrm{old}})$$

What this means is that in order to get the ratio when only the i-th particle has been moved, we only need to calculate the dot product of the vector $(\phi_1(\mathbf{r}_i^{\mathrm{new}}), \ldots, \phi_N(\mathbf{r}_i^{\mathrm{new}}))$ of single particle wave functions evaluated at this new position with the i-th column of the inverse matrix $\hat{D}^{-1}$ evaluated at the original position. Such an operation has a time scaling of $O(N)$. The only extra thing we need to do is to maintain the inverse matrix $\hat{D}^{-1}(\mathbf{x}^{\mathrm{old}})$.

Proof for updating algorithm for Slater determinant

As a starting point we may consider that each time a new position is suggested in the Metropolis algorithm, a row of the current Slater matrix experiences some kind of perturbation. Hence, the Slater matrix with its orbitals evaluated at the new position equals the old Slater matrix plus a perturbation matrix,

$$d_{jk}(\mathbf{x}^{\mathbf{new}}) = d_{jk}(\mathbf{x}^{\mathbf{old}}) + \Delta_{jk}, \tag{13}$$

where

$$\Delta_{jk} = \delta_{ik}[\phi_j(\mathbf{x}_{\mathbf{i}}^{\mathbf{new}}) - \phi_j(\mathbf{x}_{\mathbf{i}}^{\mathbf{old}})] = \delta_{ik}(\Delta\phi)_j. \tag{14}$$

Proof for updating algorithm for Slater determinant

Computing the inverse of the transposed matrix we arrive at

$$d_{kj}(\mathbf{x^{new}})^{-1} = [d_{kj}(\mathbf{x^{old}}) + \Delta_{kj}]^{-1}. \tag{15}$$

Proof for updating algorithm for Slater determinant

The evaluation of the right hand side (rhs) term above is carried out by applying the identity $(A+B)^{-1} = A^{-1} - (A+B)^{-1}BA^{-1}$. In compact notation it yields

$$\begin{aligned}
[\mathbf{D}^T(\mathbf{x^{new}})]^{-1} &= [\mathbf{D}^T(\mathbf{x^{old}}) + \Delta^T]^{-1} \\
&= [\mathbf{D}^T(\mathbf{x^{old}})]^{-1} - [\mathbf{D}^T(\mathbf{x^{old}}) + \Delta^T]^{-1}\Delta^T[\mathbf{D}^T(\mathbf{x^{old}})]^{-1} \\
&= [\mathbf{D}^T(\mathbf{x^{old}})]^{-1} - \underbrace{[\mathbf{D}^T(\mathbf{x^{new}})]^{-1}}_{\text{By Eq.15}}\Delta^T[\mathbf{D}^T(\mathbf{x^{old}})]^{-1}.
\end{aligned}$$

Proof for updating algorithm for Slater determinant

Using index notation, the last result may be expanded by

$$\begin{aligned}
d_{kj}^{-1}(\mathbf{x^{new}}) &= d_{kj}^{-1}(\mathbf{x^{old}}) - \sum_l \sum_m d_{km}^{-1}(\mathbf{x^{new}})\Delta_{ml}^T d_{lj}^{-1}(\mathbf{x^{old}}) \\
&= d_{kj}^{-1}(\mathbf{x^{old}}) - \sum_l \sum_m d_{km}^{-1}(\mathbf{x^{new}})\Delta_{lm} d_{lj}^{-1}(\mathbf{x^{cur}}) \\
&= d_{kj}^{-1}(\mathbf{x^{old}}) - \sum_l \sum_m d_{km}^{-1}(\mathbf{x^{new}})\delta_{im}(\Delta\phi)_l d_{lj}^{-1}(\mathbf{x^{old}}) \\
&= d_{kj}^{-1}(\mathbf{x^{old}}) - d_{ki}^{-1}(\mathbf{x^{new}}) \sum_{l=1}^{N} (\Delta\phi)_l d_{lj}^{-1}(\mathbf{x^{old}}) \\
&= d_{kj}^{-1}(\mathbf{x^{old}}) - d_{ki}^{-1}(\mathbf{x^{new}}) \sum_{l=1}^{N} [\phi_l(\mathbf{r_i^{new}}) - \phi_l(\mathbf{r_i^{old}})] D_{lj}^{-1}(\mathbf{x^{old}}).
\end{aligned}$$

Proof for updating algorithm for Slater determinant

Using

$$\mathbf{D}^{-1}(\mathbf{x^{old}}) = \frac{adj\mathbf{D}}{|\mathbf{D}(\mathbf{x^{old}})|} \quad \text{and} \quad \mathbf{D}^{-1}(\mathbf{x^{new}}) = \frac{adj\mathbf{D}}{|\mathbf{D}(\mathbf{x^{new}})|},$$

and dividing these two equations we get

$$\frac{\mathbf{D}^{-1}(\mathbf{x^{old}})}{\mathbf{D}^{-1}(\mathbf{x^{new}})} = \frac{|\mathbf{D}(\mathbf{x^{new}})|}{|\mathbf{D}(\mathbf{x^{old}})|} = R \Rightarrow d_{ki}^{-1}(\mathbf{x^{new}}) = \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}.$$

Proof for updating algorithm for Slater determinant

We have

$$d_{kj}^{-1}(\mathbf{x^{new}}) = d_{kj}^{-1}(\mathbf{x^{old}}) - \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N}[\phi_l(\mathbf{r_i^{new}}) - \phi_l(\mathbf{r_i^{old}})]d_{lj}^{-1}(\mathbf{x^{old}}),$$

or

$$\begin{aligned} d_{kj}^{-1}(\mathbf{x^{new}}) = d_{kj}^{-1}(\mathbf{x^{old}}) \quad &- \quad \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N}\phi_l(\mathbf{r_i^{new}})d_{lj}^{-1}(\mathbf{x^{old}}) \\ &+ \quad \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N}\phi_l(\mathbf{r_i^{old}})d_{lj}^{-1}(\mathbf{x^{old}}) \\ = d_{kj}^{-1}(\mathbf{x^{old}}) \quad &- \quad \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N}d_{il}(\mathbf{x^{new}})d_{lj}^{-1}(\mathbf{x^{old}}) \\ &+ \quad \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N}d_{il}(\mathbf{x^{old}})d_{lj}^{-1}(\mathbf{x^{old}}). \end{aligned}$$

Proof for updating algorithm for Slater determinant

In this equation, the first line becomes zero for $j = i$ and the second for $j \neq i$. Therefore, the update of the inverse for the new Slater matrix is given by

$$\boxed{d_{kj}^{-1}(\mathbf{x^{new}}) = \begin{cases} d_{kj}^{-1}(\mathbf{x^{old}}) - \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N}d_{il}(\mathbf{x^{new}})d_{lj}^{-1}(\mathbf{x^{old}}) & \text{if } j \neq i \\ \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N}d_{il}(\mathbf{x^{old}})d_{lj}^{-1}(\mathbf{x^{old}}) & \text{if } j = i \end{cases}}$$