COMPUTATIONAL PHYSICS 2
Project 2
Machine Learning (ML) and Markov Chain Monte Carlo (MCMC) for calculating the ground state energy of interacting electrons

Martin Krokan Hovden

June 16, 2020

Abstract

In the paper "Solving the quantum many-body problem with artificial neural networks", G. Carleo and M. Troyer proposed a method for using neural networks for representing the wave function of a system of particles. The method showed great results on a quantum mechanical spin lattice system of the Ising model and the Heisenberg model. In this article the methods are tested on a system of electrons confined to move in a harmonic oscillator trap. A restricted Boltzmann machine is used for representing the wave function. Markov Chain Monte Carlo methods are used for sampling and estimating the ground state energy of the system. The methods gave good results for the non-interacting case, and managed to estimate the ground state energy with good accuracy. For the interacting case the methods did not mange to get the same accuracy as for the non-interacting case, but the results were still good. Overall, Gibbs sampling was the sampling method that performed the best. In addition, using a higher number of hidden nodes in the Boltzmann machine seems to give higher accuracy and expressive power.

Contents

1 INTRODUCTION

Analytical solutions of quantum mechanical many-body systems are a rarity [2]. Methods for solving them efficiently using numerical methods are therefore of great interest. The traditional numerical methods often give good results, however, there are systems where the methods don't work well [1]. A new approach uses machine learning for representing the wave function [1]. The goal is that those new methods can fix some of the shortcomings of the more traditional methods.

The popularity of machine learning algorithms have increased over the last years. One of the reasons is the increase in computing power and the amount of data available. This has made it possible to train complex models applicable in both research and industry. Unsupervised learning is a subset of machine learning methods. They do not need labeled data. Unsupervised learning models can be used for exploratory analysis, data generation and dimension reduction [3]. In the article "Solving the quantum many-body problem with artificial neural networks", G. Carleo and M. Troyer presented a method for using a Boltzmann machine for representing the wave function of a system together with a variational approach, for finding the ground state energy of the system [1]. The methods achieved high accuracy, similar to the accuracy of the traditional methods [1] and estimating the energies and gradients.

In this article, the method is further explored and tested on a system of electrons confined to move in a harmonic oscillator trap. The algorithms are implemented from scratch in Julia. The choice of Julia was based on the goal of achieving high speed code while maintaining ease of use. The idea is to use the Boltzmann machine to represent the wave function. The Boltzmann machine is a generative network, and the goal is to learn the probability distribution [3]. The rest is similar to what is done in Variational Monte Carlo where the parameters of the model are tweaked to find the ground state energy which is the quantity of interest [4]. By using Markov Chain Monte Carlo (MCMC) methods, it is possible to calculate gradients that can be used to tune the parameters of the network until the minimum energy of the system is obtained. This can be used as an estimate for the ground state energy. Three different sampling methods are used; Metropolis Brute Force sampling, Metropolis Importance sampling, and Gibbs sampling.

The methods will be tested against known analytical values. For the non-interacting case, the analytical values are known for all sizes of the system. For the interacting case, there exist analytical values for a system of two particles in two dimensions.

The article is split into three main parts. The first part gives the theoretical background of the problem and the methods used for solving it. The second part is a description of how the methods are implemented in Julia and how the GitHub repository is structured. After that, the results are presented and discussed for both the interacting and the non-interacting system. The article is then wrapped up with a conclusion.

2 THEORY

The theory and methods part is based on the lecture notes written by Morten Hjorth-Jensen for the course FYS4411: "Computational Physics II", at the University of Oslo. For more in-depth derivations of the theory, see `http://compphysics.github.io/ComputationalPhysics2/doc/web/course`. The main goal of this part is to show how the methods are used to solve this specific problem. For more background on machine learning and Markov Chain Monte Carlo in general, see for example the link above, or [6] and [5]. The reader is assumed to have some prior knowledge about neural networks, machine learning, and Markov Chain Monte Carlo methods, so the basic theory will not be shown here.

2.1 Description of the System

In the article "Solving the quantum many-body problem with artificial neural networks", G. Carleo and M. Troyer applied machine learning algorithms for studying a quantum mechanical spin lattice system of the Ising model and the Heisenberg model [1]. The results were promising and they achieved good accuracy. It is interesting to test the method on other systems, to see if similar performance can be achieved. In this article, systems of electrons confined to move in a harmonic oscillator trap is studied. The system can be described by a Hamiltonian and a wave function. The main idea is to represent the wave function with a Boltzmann machine, and use Markov Chain Monte Carlo (MCMC) methods for sampling.

2.1.1 Hamiltonian

The Hamiltonian describing the system is given by

$$\hat{H} = \underbrace{\sum_{i=1}^{N}\left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2\right)}_{\text{harmonic oscillator}} + \underbrace{\sum_{i<j}\frac{1}{r_{ij}}}_{\text{repulsive interaction}} . \tag{1}$$

where N is the number of electrons, $r_i = \sqrt{\sum_{x=1}^{D} r_{i_x}^2}$, and the distance between two particles is given by

$$r_{ij} = |\boldsymbol{r_i} - \boldsymbol{r_j}| \tag{2}$$

[3]. Natural units are used. This means that $\hbar = c = e = m_e = 1$ and that all the energies are in units of a.u. Setting $\omega = 1$ makes it possible to find the analytical values, which makes testing of the methods easier [2].

The first part of equation 1 is a two-dimensional isotropic harmonic oscillator and the last part is the interaction part. Both the interacting and the non-interacting system is studied. For the non-interacting system, the repulsive interaction term is left out and the only terms remaining are the harmonic oscillator part. It can be shown that with the given simplifications, the energy of of a non-interacting system is given by

$$E = \frac{1}{2}P \cdot D \tag{3}$$

where P is the number of particles and D is the number of dimensions [3].

The study of the interacting system will be limited to two electrons in a quantum dot. For this system, the exact values can be found. The energy for a interacting system of two particles in three dimensions is 3 a.u. [2].

2.2 Restricted Boltzmann Machines

A Boltzmann machine is a generative and unsupervised machine learning model. For a general introduction to machine learning, see project 2 from FYS-STK4155 [5]. The idea behind using a generative model is that it can be used to sample from a probability distribution [3]. As for most machine learning models, gradient based optimization methods are used for finding the weights and biases that minimizes a cost function. However, in unsupervised learning methods, the training data only contains feature without labels.

While a standard deep neural network often have multiple layers, the Boltzmann machine only has two. Multiple Boltzmann machines can however be connected [3]. In this article a single Boltzmann machine with two layers are used. In general, the Boltzmann machine contains a hidden layer and a visible layer. Each node in the layers are connected to every other node. In addition, each node can have its own bias. The main problem with the Boltzmann machine is that they are hard to train [3]. As a way to fix this, connections between nodes in the same layer can be removed. The new network is then called a restricted Boltzmann machine [3].

A restricted Boltzmann machines is used to represent the wave function. It is a special case of a Boltzmann machine, where all the nodes are only connected to nodes in the opposite layer[3]. A restricted Boltzmann machine can be illustrated by the diagram in figure 1. Note that there are no connections between the nodes in the same layer. The illustration if for a network with 4 visible nodes and 3 hidden nodes. This can easily be extended to more nodes in each layer. In addition, there are biases that are added to each hidden and visible node. The hidden layer is vector denoted by $\boldsymbol{h}$ of length N. The visible layer $\boldsymbol{x}$ is of length M. In addition, each layer have a bias vector of the same length as length of the layer, as depicted in the figure. The bias vector of the hidden layer is called $\boldsymbol{b}$ and the bias vector for the visible layer is called $\boldsymbol{a}$. The weights between nodes in different layers will be represented by the matrix $\boldsymbol{W}$ of size $M \times N$. Element w_{ij} is the connection between node i in the visible layer to node j in the hidden layer. For this problem, the visible layer will represent the position of the particles in the system. Each component of the vector represents one particles coordinate. The length of the visible layer for a system with 2 particles in two dimensions is $2 \times 2 = 4$,

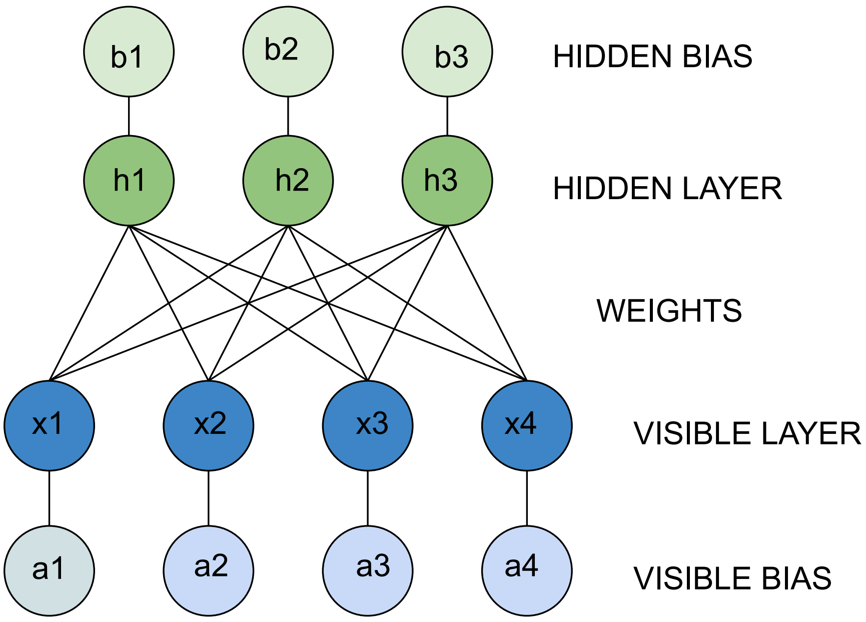


Figure 1: Illustration of a Restricted Boltzmann Machine with 4 visible nodes and 3 hidden nodes.

$$\boldsymbol{x} = [x_{11}, x_{12}, x_{21}, x_{22}] \tag{4}$$

where x_{ij} is the j'th coordinate of the i'th particle. The hidden layer is sometimes called the feature vector and the length can be chosen freely.

To be able to sample from a distribution, the network have to described by some probability function. The joint probability function of the hidden layer and the visible layer is given by the Boltzmann distribution

$$P_{rbm}(\mathbf{x},\mathbf{h}) = \frac{1}{Z} e^{-\frac{1}{T_0} E(\mathbf{x},\mathbf{h})}, \tag{5}$$

where T_0 is set to one, and

$$Z = \int\int e^{-\frac{1}{T_0} E(\boldsymbol{x},\boldsymbol{h})}. \tag{6}$$

where Z is the partition function [3]. E is the energy of the Boltzmann machines configuration of hidden and visible nodes. This is not the same as the energy of the quantum mechanical system.

The energy function depends on the type of Boltzmann machine being used. For this project a so called Gaussian-Binary RBM is used. In a Gaussian-Binary RBM, the visible units are Gaussian, while the hidden units are binary [3]. It is important to use continuous visible nodes, since they are supposed to represent the position of the particles. The energy of the Gaussian-Binary RBM is given by

$$E(\mathbf{x},\mathbf{h}) = \sum_i^M \frac{(x_i - a_i)^2}{2\sigma_i^2} - \sum_j^N b_j h_j - \sum_{i,j}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2}. \tag{7}$$

where $\boldsymbol{x}, \boldsymbol{h}, \boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{W}$ are as described above [3].

2.3 Neural-Network Quantum State

The network is used to learn a probability distribution by updating its weights and biases. In contrast to standard neural network, the restricted Boltzmann machine does not return some output. Instead, it produces a probability distribution from which we can sample from [3]. The wave function of the system can be derived from the joint distribution of the Boltzmann machine [3]. Since the wave function of the system should depend on the position of the particles, the marginal density of $\boldsymbol{x}$ is the distribution of interest. Integrating out the hidden variable from the joint distribution gives

$$F_{\text{rbm}}(\boldsymbol{x}) = \sum_{\boldsymbol{h}} F_{\text{rbm}}(\boldsymbol{x}, \boldsymbol{h}) = \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{x},\boldsymbol{h})}. \tag{8}$$

This is what will be used to represent the wave function. Inserting the energy-function for the Gaussian-Binary RBM and rearranging, gives

$$\psi(\boldsymbol{x}) = F_{\text{rbm}}(\boldsymbol{x}) = \frac{1}{Z} e^{-\sum_i^M \frac{(x_i - a_i)^2}{2\sigma^2}} \prod_j^N \left(1 + e^{b_j + \sum_i^M \frac{x_i w_{ij}}{\sigma^2}}\right) \tag{9}$$

This is called the neural quantum state [1].

2.4 Cost Function

Most machine learning algorithms use some kind of cost function. For supervised learning, the cost function is often a measure of difference between the output of the network and the desired output. This means that the training data consist of labeled data. For unsupervised learning, this is not the case. For this case the energy of the system is used as the cost function [3], and is the quantity we want to minimize.

2.4.1 Local Energy

The local energy of the system is defined as

$$E_L = \frac{1}{\psi} \hat{\boldsymbol{H}} \psi \tag{10}$$

where ψ is the NQS given by the Boltzmann machine [3]. It can be shown that

$$E_L = \frac{1}{2} \sum_p^P \sum_d^D \left(-\left(\frac{\partial}{\partial x_{pd}} \ln \psi \right)^2 - \frac{\partial^2}{\partial x_{pd}^2} \psi + \omega^2 x_{pd}^2 \right) + \sum_{p<q} \frac{1}{r_{pq}}. \tag{11}$$

See the appendix for a full derivation. For the non-interacting system, the last term is zero.

Note that

$$\frac{1}{\psi} \frac{\partial \psi}{\partial \alpha_i} = \frac{\partial \ln \psi}{\partial \alpha_i}. \tag{12}$$

Taking the logarithm of the wave function gives

$$\ln(\psi(\boldsymbol{X}) = -\ln(Z) - \sum_i^M \frac{(x_i - a_i)^2}{2\sigma^2} + \sum_j^N \ln(1 + e^{b_j + \sum_i^M \frac{X_i w_{ij}}{\sigma^2}}). \tag{13}$$

The derivatives with respect to the visible nodes are given by

$$\frac{\partial}{\partial x_m}\ln\psi = -\frac{1}{\sigma^2}(x_m - a_m) + \frac{1}{\sigma^2}\sum_n^N \frac{w_{mn}}{\exp\left[-b_n - \frac{1}{\sigma^2}\sum_i^M x_i w_{in}\right] + 1} \tag{14}$$

and

$$\frac{\partial^2}{\partial x_m^2}\ln\psi = -\frac{1}{\sigma^2} + \frac{1}{\sigma^4}\sum_n^N w_{mn}^2 \frac{\exp\left[b_n + \frac{1}{\sigma^2}\sum_i^M x_i w_{in}\right]}{\left(\exp\left[b_n + \frac{1}{\sigma^2}\sum_i^M x_i w_{in}\right] + 1\right)^2} \tag{15}$$

[3]

2.4.2 Minimizing the Cost Function

As mentioned, the energy of the quantum mechanical energy is used as a cost function. The goal is to tweak the weights and biases of the network to obtain the lowest possible energy. This is done by optimization methods. One method that is often used in machine learning is the gradient descent algorithm. For a thorough description of gradient based optimization methods and updating of weights in a neural network, see [5]. The main step of the gradient descent method is to move in the opposite direction of the gradient of the function, where the function f is dependent on some parameters θ. The idea is that since the gradient points in the direction where the function increase fastest, moving in the opposite direction will lead to the fastest decrease in function value. The algorithm goes as follows:

- Choose an initial set of parameters.
- Continue until stopping criteria:
    - Update the parameters according to

$$\theta^{(t+1)} = \theta^{(t)} - \lambda\nabla f(\theta^{(t)}) \tag{16}$$

where λ is the learning rate. To minimize the cost function (local energy), the gradient is needed. Again, using the logarithm of the wave function, the gradients with respect to the weights and biases are given by

$$G_\alpha = \frac{\partial\langle E_L\rangle}{\partial\alpha} = 2\left(\langle E_L\frac{1}{\psi}\frac{\partial\psi}{\partial\alpha}\rangle - \langle E_L\rangle\langle\frac{1}{\psi}\frac{\partial\psi}{\partial\alpha}\rangle\right) \tag{17}$$

[3] .The expected values are calculated with Monte Carlo methods using samples from the Markov Chain Monte Carlo methods presented later.

The derivatives of the wave function with respect to the biases and weights are needed in the gradient descent updates. For a full derivation, see the appendix. Those are given by

$$\frac{\partial}{\partial a_m}\ln\psi = \frac{1}{\sigma^2}(X_m - a_m) \tag{18}$$

$$\frac{\partial}{\partial b_n}\ln\psi = \frac{1}{\left(\exp\left[-b_n - \frac{1}{\sigma^2}\sum_i^M X_i w_{in}\right] + 1\right)} \tag{19}$$

and

$$\frac{\partial}{\partial w_{mn}}\ln\psi = \frac{X_m}{\sigma^2\left(\left(\exp\left[-b_n - \frac{1}{\sigma^2}\sum_i^M X_i w_{in}\right] + 1\right)} \tag{20}$$

[3] For Gibbs Sampling an alternative wave function is used, $\psi(\boldsymbol{X}) = \sqrt{F_{\text{rbm}}(\boldsymbol{X})}$. This gives slightly different derivatives:

$$\frac{\partial}{\partial a_m}\ln\psi = \frac{1}{2\sigma^2}(X_m - a_m) \tag{21}$$

$$\frac{\partial}{\partial b_n}\ln\psi = \frac{1}{2\left(\exp\left[-b_n - \frac{1}{\sigma^2}\sum_i^M X_i w_{in}\right] + 1\right)} \tag{22}$$

and

$$\frac{\partial}{\partial w_{mn}} \ln \psi = \frac{X_m}{2\sigma^2 \left((\exp\left[-b_n - \frac{1}{\sigma^2} \sum_i^M X_i w_{in} \right] + 1 \right)} \tag{23}$$

[3]

The ground state energy is found when the quantum mechanical energy is at its lowest. The weights are then tweaked according to the gradient descent algorithm to find this.

2.5 Markov Chain Monte Carlo

Markov Chain Monte Carlo methods is methods used for sampling from distributions f that are hard to sample from, but easily can be evaluated at each point. The idea is to create a sequence of samples that converge towards f. This is done by making an irreducible and aperiodic Markov chain with a limiting distribution equal to f [9]. The samples $\{X^{(t)}\}$ from the Markov Chain can then be used to calculate expectation values of functions. There are different ways to create such a chain. In this section 3 different methods are presented. For a more theoretical background on the methods, see the lecture notes from FYS3150 [7] or the report for the VMC project [6]. The 3 methods are the Metropolis Brute-Force sampling, Metropolis Importance sampling, and Gibbs sampling.

2.5.1 Metropolis Brute-Force

The Metropolis Hastings algorithm goes as follows [9],

- Sample an initial point $\boldsymbol{X^{(0)}}$ from an initial distribution.
- for t = 0,...,N:
  - Sample a proposal value $\boldsymbol{X^*}$ from a proposal distribution $p(\cdot|\boldsymbol{x}^{(t)})$ for updating one randomly chosen coordinate.
  - Evaluate the Metropolis acceptance ratio

$$R(\boldsymbol{x^{(t)}}, \boldsymbol{X}^*) = \frac{f(\boldsymbol{X}^*)p(\boldsymbol{x^{(t)}}|\boldsymbol{X}^*)}{f(\boldsymbol{x}^{(t)})p(\boldsymbol{X}^*|\boldsymbol{x}^{(t)})} \tag{24}$$

  - Sample U from Uniform$(0, 1)$
  - Keep the proposal value $\boldsymbol{X}^*$ if $U < R$.
  - Keep the previous value $\boldsymbol{x}^{(t)}$ if $U > R$.

For the brute-force version, a random walk chain is used. This means that the proposed value is samples from

$$\boldsymbol{X}^{(t+1)} = \boldsymbol{X}^{(t)} + \Delta x \cdot \epsilon \tag{25}$$

where Δx is the step length and ϵ is random variable [9]. The random variable can for example be drawn from the uniform distribution between -1 and 1 or from a standard normal distribution. This is a simple method for proposing new steps, but does not take into account information about the distribution. The initial value can for example be sampled from a Uniform or Gaussian distribution centered at zero.

The metropolis ratio is given by

$$R(\boldsymbol{x^{(t)}}, \boldsymbol{X}^*) = \frac{\psi(\boldsymbol{X}^*)}{\psi(\boldsymbol{x}^{(t)})} \tag{26}$$

where the p disappears because of the symmetry of the proposal distribution.

2.5.2 Metropolis Importance Sampling

The main problem with the brute force method, is that the proposal of a new step don't use information about the wave function. The difference between the Brute Force sampling and Importance sampling is how the algorithm proposes a new position, and how it calculates the Metropolis ratio. For a detailed background on the justification of the new transition distribution and ratio, see [7]. The idea is to use a proposal distribution that use information about the wave function to move into areas with higher probability, as well as changing the Metropolis ratio to the so-called greens function ratio. The Importance Sampling algorithm goes as follows [4],

- Sample an initial point $\boldsymbol{X^{(0)}}$ from an initial distribution.
- for t = 0,...,N:
  - Sample a proposal value $\boldsymbol{X^*} = \boldsymbol{x}^{(t)} + DF(\boldsymbol{x}^{(t)})\Delta t + \xi\sqrt{\Delta t}$ for one randomly chosen coordinate. D is the diffusion constant, Δt is the importance time-step, and ξ is a Gaussian random number. The drift force is given by

    $$\boldsymbol{F} = 2\frac{1}{\psi}\nabla\psi \tag{27}$$

  - Evaluate the Metropolis acceptance ratio

    $$q(\boldsymbol{X}^*, \boldsymbol{x}^{(t)}) = \frac{G(\boldsymbol{x}^{(t)}, \boldsymbol{X}^*, \Delta t)|\psi(\boldsymbol{X}^*)|^2}{G(\boldsymbol{X}^*, \boldsymbol{x}^{(t)}, \Delta t)|\psi(\boldsymbol{x}^{(t)})|^2} \tag{28}$$

  - Sample U from Uniform$(0, 1)$
  - Keep the proposal value $\boldsymbol{X}^*$ if $U < q$.
  - Keep the previous value $\boldsymbol{x}^{(t)}$ if $U > q$.

Here, G is greens function, given by

$$G(y, x, \Delta t) = \frac{1}{(4\pi D\Delta t)^{3N/2}} \exp\left(-(y - x - D\Delta t F(x))2/4D\Delta t\right) \tag{29}$$

2.5.3 Gibbs Sampling

An alternative sampling method is Gibbs sampling. The main difference is that the Gibbs sampler samples each dimension for every iteration instead of picking one randomly as in Metropolis. It is therefore well suited for multidimensional distributions [9]. However, the method requires finding conditional distributions for all components of interest.

As mentioned, the Gibbs sampler uses conditional distribution for sampling. For Gibbs sampling, an alternative wave function is used, $\psi(x) = \sqrt{F_{rbm}}$. The idea is to sample from a two step sample process where first the visible layers are updated, then the hidden layer is updated given the newly updated visible layer. The posterior distributions to sample from are given by

$$P(\boldsymbol{X}|h) = \sum_{i}^{M} \mathcal{N}(X_i; a_i + \boldsymbol{w_i h}, \sigma^2) = \mathcal{N}(\boldsymbol{X}; \boldsymbol{a} + \boldsymbol{Wh}, \sigma^2) \tag{30}$$

and

$$P(H_j|\boldsymbol{x}) = \frac{e^{\left(b_j + \frac{\boldsymbol{x^T w_{*j}}}{\sigma^2}\right)H_j}}{1 + e^{\left(b_j + \frac{\boldsymbol{x^T w_{*j}}}{\sigma^2}\right)}} \tag{31}$$

which gives that

$$P(\boldsymbol{H}|\boldsymbol{x}) = \prod_{j}^{N} \frac{e^{\left(b_j + \frac{\boldsymbol{x^T w_{*j}}}{\sigma^2}\right)H_j}}{1 + e^{\left(b_j + \frac{\boldsymbol{x^T w_{*j}}}{\sigma^2}\right)}} \tag{32}$$

[10]. The algorithm can be described as follows:

- Choose some initial values for the visible layer $\boldsymbol{x}$ and the hidden layer $\boldsymbol{h}$
- for t=0,1,2..,N
  - Update the visible layer given the values from the previous step

    $$\boldsymbol{X}^{(t+1)}|\boldsymbol{h}^{(t)} \sim P(\boldsymbol{x}|\boldsymbol{h}^{(t)}) \tag{33}$$

  - Update the hidden layer given the values from the newly updated visible values:

    $$\boldsymbol{H}^{(t+1)}|\boldsymbol{x}^{(t+1)} \sim P(\boldsymbol{h}|\boldsymbol{x}^{(t+1)}) \tag{34}$$

Note that there are no acceptance step as it was in the Metropolis sampling methods. For details on implementation, see the implementation part.

2.6 Putting it all together

The idea is now to do the sampling for each iteration in the optimization algorithm. The samples are then used to calculate the gradient. The first step is to set up the network with random weights and biases. The number of visible nodes have to be $P \times D$ where P is the number of particles and D is the number of dimensions. The number of optimization steps and number of sampling steps are then chosen. For each optimization step, samples are produced using one of the MCMC methods presented above. The number of samples produced is the number of sampling steps set before the iterations start. Those samples are used for calculating the current local energy, as well as the gradients used for that optimization step. This is done by estimating the expected values with Monte Carlo integration. When the gradient is found, the weights in the network is updated and one optimization step is done. This is repeated multiple times. The method can then be summarized as follows

- Set up the network and initialize values.
- For t in number of optimization steps:
  - For i in number of sampling steps
    - Sample $\boldsymbol{X}_i$
  - Calculate expected values of interest using the samples in optimization step t. Use the formula
    $$\langle f(Y) \rangle \approx \frac{1}{n} \sum_{i=1}^{n} f(Y_i) \tag{35}$$
    where n is the number of samples.
  - Calculate the gradient using the formula 35 and 17.
  - Update the weights according to the gradient descent update formula 16.

3 IMPLEMENTATION

All code can be found in the GitHub-repository at `https://github.com/MartinKHovden/ComputationalPhysics2/tree/master/Project2`. I have tried to make the code as readable as possible, with describing names and comments. The algorithms are implemented close to as described in the theory part, so it should be possible to follow the code. The code is written in Julia. Julia is a programming language that aim to be high-performing while maintaining ease of use. Since Julia not is a object oriented language with classes, the code is based on functions working on a struct representing the Boltzmann machine. The functions then acts upon the parameters of the struct. This is explained in more detail later. In this section the most important parts of the code is shown and explained. The main part of the code is located in the file `library.jl`. Another file called `runSimluations.jl` is used to run the different simulations. Part of this file can be commented out to run the simulations of interest. To run the code, an installation of Julia is needed. Julia files can then be run using `julia filename.jl`. The GitHub-repository also contains jupyter notebooks for the analysis of the results, as well as a python-script for calculating the variance using the blocking method presented by Marius Jonsson [8]. The blocking method is taken from the course-material in FYS4411.

3.1 Representing the network

The information about, and parameters of, the network are stored in the struct NQS. This struct contains arrays representing the layers of the network, as well as arrays for representing the biases and weights. It also contains information about the networks variance and if the system studied should include interactions or not. This is the struct that most of the methods acts on and some of the methods updates the parameters stored within the struct.

3.2 Implementation of sampling methods

The three sampling methods implemented for this project is Metropolis Brute Force Sampling, Metropolis Importance Sampling and Gibbs Sampling. The first two are very similar. The differences is the way they proposes and accepts new steps. In contrast, Gibbs Sampling don't have the acceptance step. Instead it samples from the posterior distributions. In the file `library.jl`, functions for sampling and optimizing are found for all the methods methods.

3.2.1 Metropolis Brute Force Sampling

The function for sampling new steps in Metropolis Brute Force Sampling is called `metropolisStepBruteForce(...)`. It is as described in the theory part and can be implemented in Julia as follows. The first step is the proposal of a new configuration for one randomly chosen coordinate.

```
nqs.x[coordinate] += (rand(Float64) - 0.5)*step_length
```

where coordinate is a randomly chosen coordinate, nps.x is the visible layer of the Boltzmann machine, and step_length is the brute force step length. The function updates the NQS structs visible layer. The acceptance step can be implemented as

```
U = rand(Float64)
if U < (new_wavefunction_value^2)/(old_wavefunction_value^2)
    return
else
    nqs.x[coordinate] = old_coordinate
    return
end
```

where U is a uniformly distributed random variable. Note that if the test do not pass, then the coordinate is changed back to the old coordinate.

3.2.2 Metropolis Importance Sampling

The same steps can be implemented in Julia the following way. The function is called `metropolisStepImportanceSampling(...)` First, the proposal of a new step:

```
nqs.x[coordinate] += D*current_drift_force*time_step + randn(Float64)*sqrt(time_step)
```

and then the acceptance of the step as

```
greens_function_argument = (old_coordinate - nqs.x[coordinate] -
    D*time_step*new_drift_force)^2
    - (nqs.x[coordinate] - old_coordinate - D*time_step*current_drift_force)^2

greens_function_argument /= (4.0*D*time_step)

greens_function = exp(-greens_function_argument)

U = rand(Float64)

if U < greens_function*(new_wavefunction_value^2)/(old_wavefunction_value^2)
    return
else
    nqs.x[coordinate] = old_coordinate
    return
end
```

where the greens function argument is used.

3.2.3 Gibbs Sampling

Gibbs sampling is different, since all of the visible and hidden nodes are updates each time. This can easily be done as described in the theory part. The implementation in Julia is as follows

```
#Update the visible nodes
for i = 1:num_visible
    nqs.x[i] = sqrt(variance)*randn(Float64) + mean[i]
end

U = rand(Float64)

#Update the hidden nodes
```

```
for j = 1:num_hidden
    nqs.h[j] = U < 1.0/(exp(-precalc[j]) + 1.0)
end
```

3.3 Gradient Descent

The gradient descent step can be implemented in Julia the following way

```
function optimizationStep(nqs::NQS, grad_a::Array{Float64, 2},
    grad_b::Array{Float64, 2}, grad_w::Array{Float64, 2}, learning_rate::Float64)
    nqs.a[:] = nqs.a - learning_rate*grad_a
    nqs.b[:] = nqs.b - learning_rate*grad_b
    nqs.w[:,:] = nqs.w - learning_rate*grad_w
end
```

where the gradients are returned from the sampling functions. where num_iterations are the number of optimization steps. Again, one function is made for each method, and are called runOptimizatoinMETHODNAME(...).

3.4 Connecting the different parts of the code

The idea is now to run the optimization step multiple times, like below

```
for k = 1:num_iterations
    local_energy,_grad_a, _grad_b, _grad_w = runMETHODNAME(...)
    optimizationStep(nqs, _grad_a, _grad_b, _grad_w, learning_rate)
end
```

where runMETHODNAME(...) is functions for sampling multiple samples using one of the methods. num_iterations is the number of optimization steps. The methods returns the monte carlo estimate of the local energy, as well as the gradients.

3.5 Testing of the code

Tests of the code are placed in the file `test.jl`. To run the tests, use

```
julia tests.jl
```

This file contains unit tests of some of the functions in `library.jl`. However, the sampling methods are tested by looking at the convergence properties in the plots and results from running the algorithms.

4 RESULTS AND DISCUSSION

In this section the results from running the algorithm on systems of electrons in harmonic oscillator traps are presented and discussed. The first part will focus on the non-interacting system, before the system is extended to include interactions. For the non-interacting case the exact values are known. The methods can therefore easily be tested and tuned to get the best estimate of the ground state energy. For the interacting system, the exact value is known for a system of two particles in two dimensions. As mentioned in the theory part, it does not make sense to test on any other system when the interactions are included. The study of interacting particles is therefore limited to a system of two particles in two dimensions.

For each method, the performance is compared for different parameters. The parameters are fine-tuned to find the optimal result. To best compare the different parameter configurations, all tests should be run from the same initial value. However, this was hard, due to the different dimensions of the randomly initialized weights and biases for different number of hidden nodes. This means that the initial value is the same only for the same number of hidden nodes. The learning rate can then be compared directly, but the effect of the hidden nodes have to be studied for each case and the results with regard to the number of hidden nodes are not necessarily 100% reliable. However, by running the gradient descent algorithm for enough step, the effects of initial values should be negligible.

All variances are found using blocking. Blocking requires that the number of samples are a factor of two. I forgot this when doing the initial sampling, so some samples had to be discarded to get to the highest factor of two below the number of total samples. This might make the results a bit less reliable, but due to the time it took to run some of the optimizations, I did not have time to do it all over.

4.1 Non-interacting system

For this case the analytical values are known, and the ground state energy is given by

$$E = \frac{1}{2} P \cdot D \tag{36}$$

where P is the number of particles and D is the number of dimensions. This will be used as a benchmark for the different methods.

The next parts presents the results from running the three different methods on a non-interacting system.

4.1.1 Metropolis Brute-Force

This section will start of with an analysis of the convergence properties of the Markov Chain produced by the brute-force method. After that, the optimal parameters from this part will be used to fine tune the parameters of the Boltzmann machine and the gradient descent algorithm to achieve the best possible accuracy.

The goal of analyzing the chain is to check the convergence properties and find the brute force step length that produces the best chain. In this part the focus will be on the convergence properties of the chain and not the accuracy of the approximation. This will be done at a later stage. A system with one particle in two dimensions is used to study the convergence properties. The gradient descent optimization algorithm is run for 100 steps for 100000 Metropolis steps in each optimization iteration with 2 hidden nodes. The Boltzmann machines σ is set to 1. The resulting weights and biases are used for one final run of the Metropolis sampling algorithm for 500000 sampling steps. The results from this last run is shown in figure 2 and 3, and table 1. The table shows that all step-length gives high acceptance ratios, but the lower the step-length, the higher the acceptance ratio. This is as expected, since moves into close regions of the last step will probably have a high Metropolis ratio and therefore a high chance of being accepted. From the trace-plots it looks like 0.1 produces the best chain. It accepts close to every proposal and stays close to the mean of the limiting distribution for each step. The trace-plots for 1.0 and 0.5 looks ok, and the algorithms explores the area of interest. All of the trace-plots seems to have good mixing properties. Mixing says something about how long it stays on the same point before a new step is accepted [9]. This can also be seen from the acceptance ratio, since all of tests gave high acceptance ratios.

For all step lengths, the chain seems to converge to the limiting distribution immediately. This was also the case for Metropolis runs where the weights and nodes in the network was not optimized. This indicates that the burn-in period can be low. It is sometimes advised to use a burn-in period of 0.5 when a lot of data is available [9]. However, a burn-in period seems unnecessary after looking at the trace-plots. I will discard the first 10% of the samples in case the initial values are worse for other sets of samples. The fast convergence also means that each optimization step needs fewer samples. This speeds up the computations drastically.

Looking at the histogram and comparing for the various step-lengths, the histogram for a step length of 1.0 is more skewed towards lower energies. For 0.5 and 0.1, the energies are approximately normally distributed. However, the mean for 0.5 seems to be slightly more skewed to the left compared to for 0.1. In addition, the variance in the samples is much higher for 0.5. This can also be seen from table 1. After looking at the two plots and the table, a step-length of 0.1 seems like a good choice. The rest of the analysis will use 0.1 as the step-length for Metropolis brute force sampling.

The results from running the gradient descent algorithm for different learning rate and number

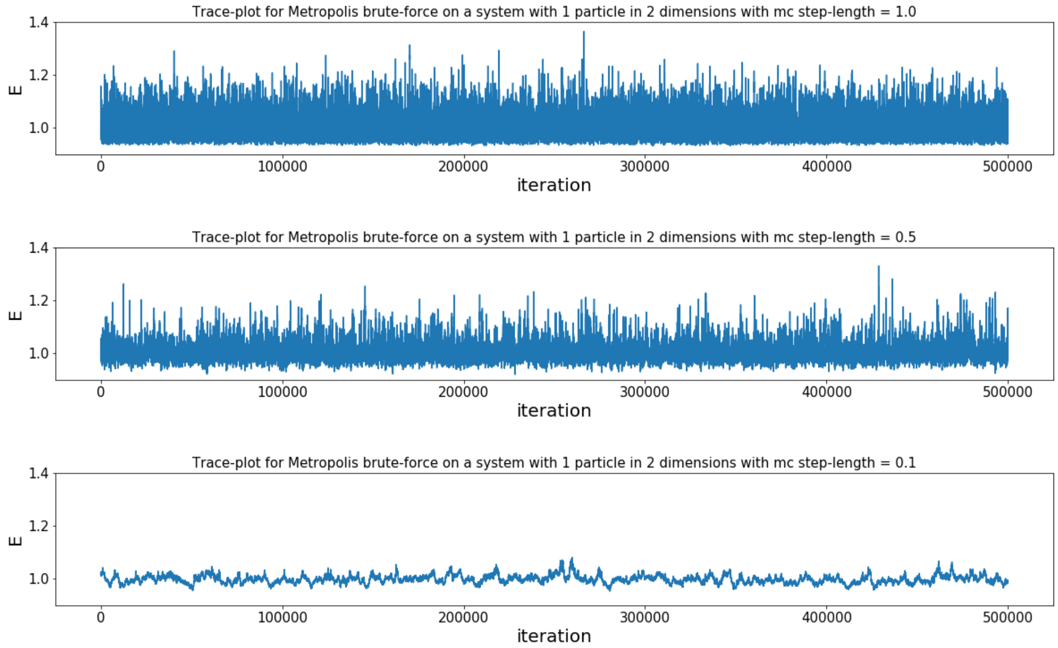


Figure 2: Trace-plots of energies for different step-lengths when using Brute force Metropolis. All energies in units of a.u.

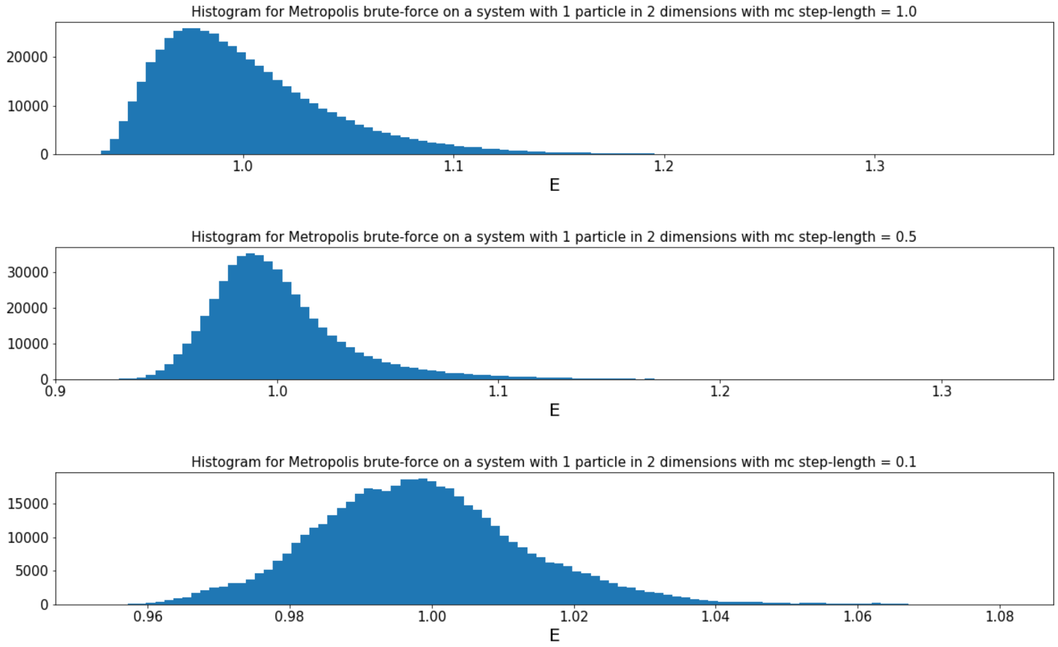


Figure 3: Histograms of energies for different step-lengths when using Brute force Metropolis on the non-interacting system. All energies in units of a.u.

Table 1: Table of acceptance ratio and variance for Metropolis brute-force for different mc step-lengths on a non-interacting system with 1 particle in 2 dimensions.

MC step-length	A	Variance
1.0	0.861200	4.767153e-06
0.5	0.929970	2.259132e-06
0.1	0.986736	9.971917e-08

of hidden nodes can be seen in figure 4 and table 2. The Boltzmann σ was set to 1, and changing this did not lead to better results. All combinations are run for 200 optimization steps with 100000 Monte-Carlo samples used for each optimization iteration. Note that the runs with the same number of hidden nodes have the same initial value. The energies converge to the exact

Table 2: Optimal energy, error and computational time for different network architectures using Metropolis Brute-force on a non-interacting system with 1 particles in 2 dimensions. Using σ^2 =1.0, MC-iterations = 100000 and MC step-length = 0.1. Number of optimization steps = 300

Network Architecture	Optimal Energy	Abs error	Rel error	Time [s]
lr = 0.01 , hidden nodes = 2	1.0109	0.0109	0.0109	33.872
lr = 0.01 , hidden nodes = 3	1.0045	0.0045	0.0045	37.153
lr = 0.01 , hidden nodes = 4	1.0044	0.0044	0.0044	39.467
lr = 0.1 , hidden nodes = 2	1.0013	0.0013	0.0013	33.390
lr = 0.1 , hidden nodes = 3	1.0004	0.0004	0.0004	36.616
lr = 0.1 , hidden nodes = 4	1.0011	0.0011	0.0011	38.478
lr = 1.0 , hidden nodes = 2	1.0009	0.0009	0.0009	32.917
lr = 1.0 , hidden nodes = 3	1.0002	0.0002	0.0002	35.811
lr = 1.0 , hidden nodes = 4	1.0001	0.0001	0.0001	38.051

values with varying convergence rates. The exact value for the system of 1 particle in 2 dimensions are 1. Note that the curves for lr = 0.01 decreases slower compared to the higher learning rates. In general, the plots shows that the higher the learning rate, the higher the convergence rate. However, after 200 iterations with the gradient descent algorithm, the results are good for most combinations. The only line with bad behaviour is for 2 hidden layers with learning rate= 0.01. The energy seems to have problems stabilizing. This might indicate that 2 hidden nodes is to few. We can see that even though the blue have the same learning rate as the orange line, the orange line converges much faster. The orange line have 3 hidden nodes. The effect of the number of hidden nodes can also be studied by looking at table 2. In the table the absolute errors are shown after 300 optimization iterations with 10000 samples each step. From this it seems like the best combination was a learning rate of 1.0 and 4 hidden nodes. This resulted in an absolute error of 0.0001. In general, the computational time increases with the number of hidden nodes. The time in the table is the time it took to run 200 optimization steps with 100000 MCMC samples for each step. This is as expected since more computations are needed when the dimensions of the vectors and matrices increase. However, since the difference is so small in computational time, 4 hidden nodes still looks like the best alternative.

The analysis shows that the optimal parameters when using Metropolis Brute Force sampling are a brute force step length of 0.1, 4 hidden nodes in the Boltzmann machine, and a learning rate of 1.0 for the gradient descent algorithm.

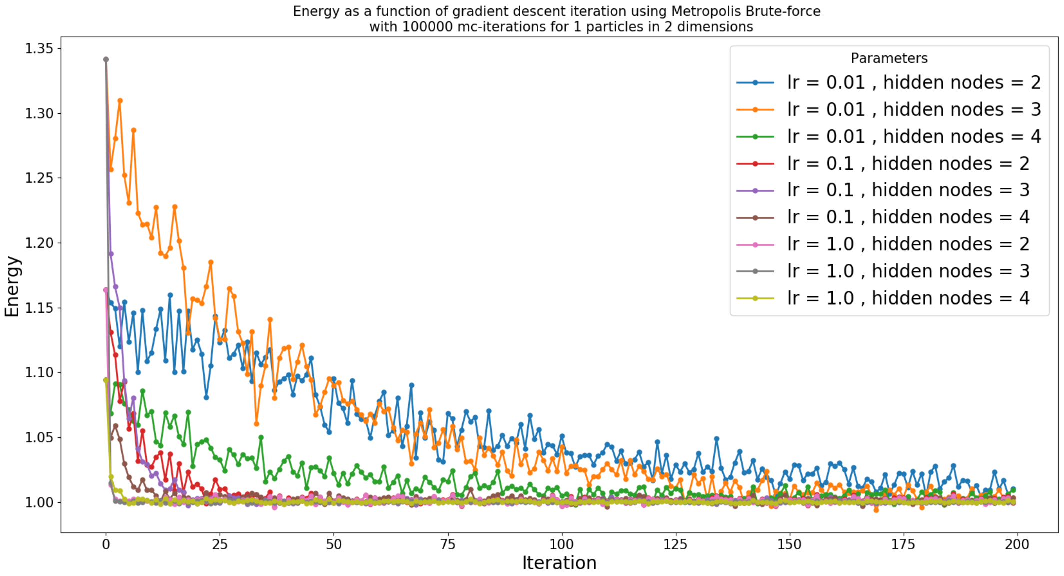


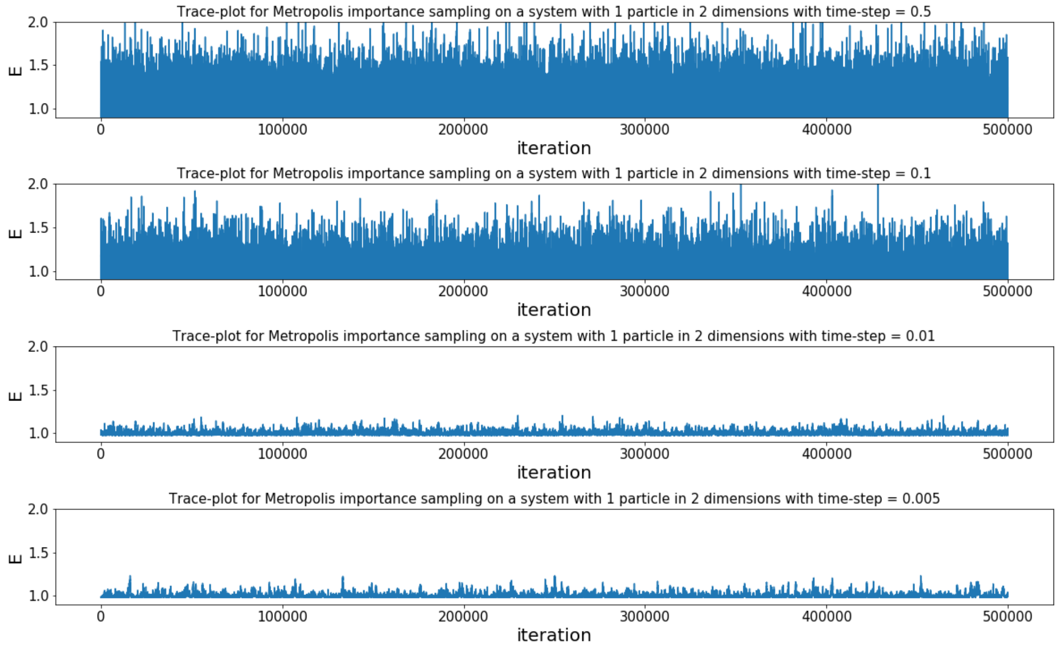
Figure 4: Non-interacting system. Energies in units of a.u.

4.1.2 Metropolis Importance-Sampling

As for the Brute force version, the first step is to analyse the convergence properties of the Markov chain produced by the algorithm. The same system with 1 particle in 2 dimensions is studied. First, 100 optimization steps with 100000 MCMC-samples each optimization iteration is run. Then the resulting weights and biases are used for one final run with 500000 samples.

In figure 5 and figure 6 the trace plots and histograms are shown for 4 different time steps. In table 3 the acceptance ratio and variance are shown for the same time steps. The σ in the Boltzmann machine is set to 1. As for the Brute Force method, all of the time-steps gives high acceptance.The acceptance ratio is over 90% for all of the time steps. However, the variance is lower. This indicates that the importance sampling method explores the important parts of the system more efficiently than the brute-force method. It proposes steps into, and stays in, the areas where the probability distribution is high. The table shows that a time-step of 0.005 gives a high acceptance ratio while the variance stays low. Looking at the histograms shows some strange behavior. For the time steps of 0.01 and 0.005, the histograms are highly skewed to the left. In addition, the energy is cut of at a minimum value. I'm not sure about the reason for this behaviour. As for the brute force algorithm, the chain seems to converge immediately to the limiting distribution, so a long burn-in period is not needed. However, a burn-in period of 0.1 is used to be on the safe side. This is because the distribution is only the same as the target distribution in the limit of the chain [?] and chain with worse convergence might occur in other sets of samples. It depends a on the initial value of the chain.

After studying the chain, a time step of 0.005 seems to be the best one. It gives high acceptance ratio while keeping the variance low. The highest bin in the histogram being lower than 1 can maybe be a result of to few optimization steps before running the final simulation. I think that by doing more optimization steps, the whole histogram would probably shift more towards the right. We will see that importance sampling gives good results later, so I think this might be a possible explanation.


Trace-plot for Metropolis importance sampling on a system with 1 particle in 2 dimensions with time-step = 0.5

Trace-plot for Metropolis importance sampling on a system with 1 particle in 2 dimensions with time-step = 0.1

Trace-plot for Metropolis importance sampling on a system with 1 particle in 2 dimensions with time-step = 0.01

Trace-plot for Metropolis importance sampling on a system with 1 particle in 2 dimensions with time-step = 0.005

Figure 5: Trace-plots of energies for different step-lengths when using Importance sampling Metropolis for a non interacting system. All energies in units of a.u.

The results from running the optimization algorithm with an importance sampling time step of 0.005 is shown in figure 7 and table 4. The results are presented for different number of hidden nodes and different learning rates. Again, the Boltzmann machine's σ is set to 1. All the values are found using 200 optimization steps with 100000 samples each optimization iteration. As for with Brute-force sampling, most of the combinations seems to converge towards the ground state energy of 1. However, the one with learning rate of 0.01 and 2 hidden nodes seems to converge very slow and not actually towards the optimal value. Since the learning rate is so low, the results might have been better if more optimization steps were used. However, for

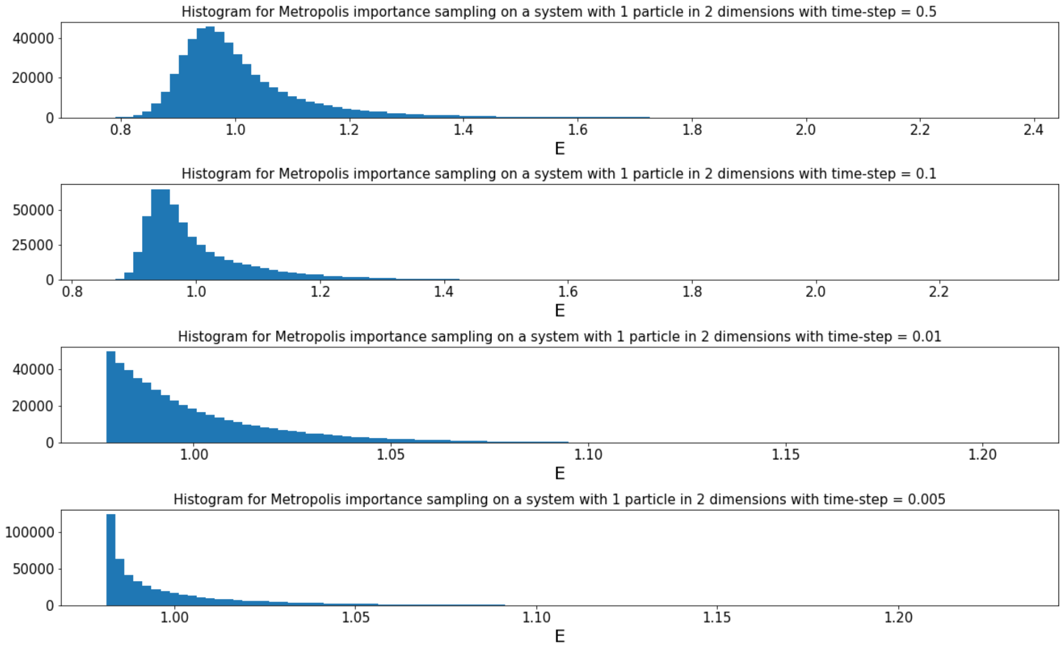


Figure 6: Histograms of energies for different step-lengths when using Importance sampling Metropolis on a non-interacting system. All energies in units of a.u.n

Table 3: Table of acceptance ratio and variance for Metropolis importance sampling for different time-steps on a non-interacting system with 1 particle in 2 dimensions.

time step	A	Variance
0.500	0.928258	1.560764e-04
0.100	0.993572	3.774283e-04
0.010	0.999794	2.088548e-07
0.005	0.999906	2.741694e-08

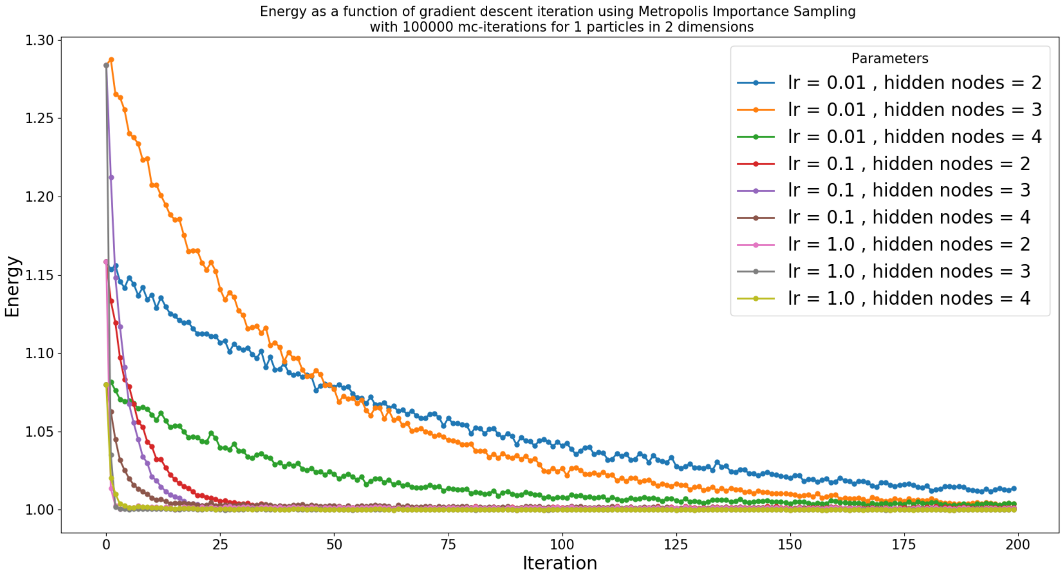


Figure 7: Non-interacting system. Energies in units of a.u.

lr=0.01 and 3 hidden nodes, the convergence is much better. This might indicate that 2 hidden nodes is two little, as was also the case when using brute-force sampling. Looking at the table, it seems like the trend is that the higher learning rate gives better results. When it comes to the number of hidden nodes, 3 and 4 seems to generally give better results compared to 2 hidden nodes. After looking at the table, the best combination seems to be a learning rate of 1 with 4 hidden nodes.

Table 4: Optimal energy, error and time for different network architectures using Metropolis Importance Sampling on a non-interacting system with 1 particles in 2 dimensions. Using σ^2 =1.0, MC-iterations = 100000 and MC step-length = 0.005

Network Architecture	Optimal Energy	Abs error	Rel error	Time [s]
lr = 0.01 , hidden nodes = 2	1.0127	0.0127	0.0127	33.259
lr = 0.01 , hidden nodes = 3	1.0041	0.0041	0.0041	37.169
lr = 0.01 , hidden nodes = 4	1.0037	0.0037	0.0037	38.950
lr = 0.1 , hidden nodes = 2	1.0013	0.0013	0.0013	33.054
lr = 0.1 , hidden nodes = 3	1.0004	0.0004	0.0004	36.812
lr = 0.1 , hidden nodes = 4	1.0009	0.0009	0.0009	38.813
lr = 1.0 , hidden nodes = 2	1.0009	0.0009	0.0009	33.374
lr = 1.0 , hidden nodes = 3	1.0001	0.0001	0.0001	36.606
lr = 1.0 , hidden nodes = 4	1.0001	0.0001	0.0001	38.481

4.1.3 Gibbs Sampling

For Gibbs sampling there is no step-length to find. The analysis can therefore focus on finding the optimal learning rate in the gradient descent algorithm and the number of hidden nodes. In addition, it is interesting to look at the effects of the choice of σ. The results from running Gibbs-sampling with $\sigma^2 = 0.5$ are shown in figure 8 and table 5. This seemed to be a good starting point for the analysis after some trial and error with the value of σ^2. Note that for 3 hidden nodes and learning-rate = 1.0, the optimization diverges. This seems to be a problem with Gibbs sampling sometimes, when the step-length is to large with a high number of hidden nodes. However, the best results are for 4 hidden nodes with learning rate = 1.0. This also gave consistently good results compared to 3 hidden nodes which often diverged. The figure also shows that the convergence rate is much higher with a high step-length, so for the comparison of the optimal results, the Boltzmann machine with 4 hidden nodes and using a learning rate of 1.0, will be used. The results from varying the value of σ^2 with 4 hidden nodes and a learning rate of 1.0 are shown in figure 9. From this it can be seen that $\sigma^2 = 0.48$ gives the fastest convergence as well as the most accurate result. This will be used when comparing the methods.

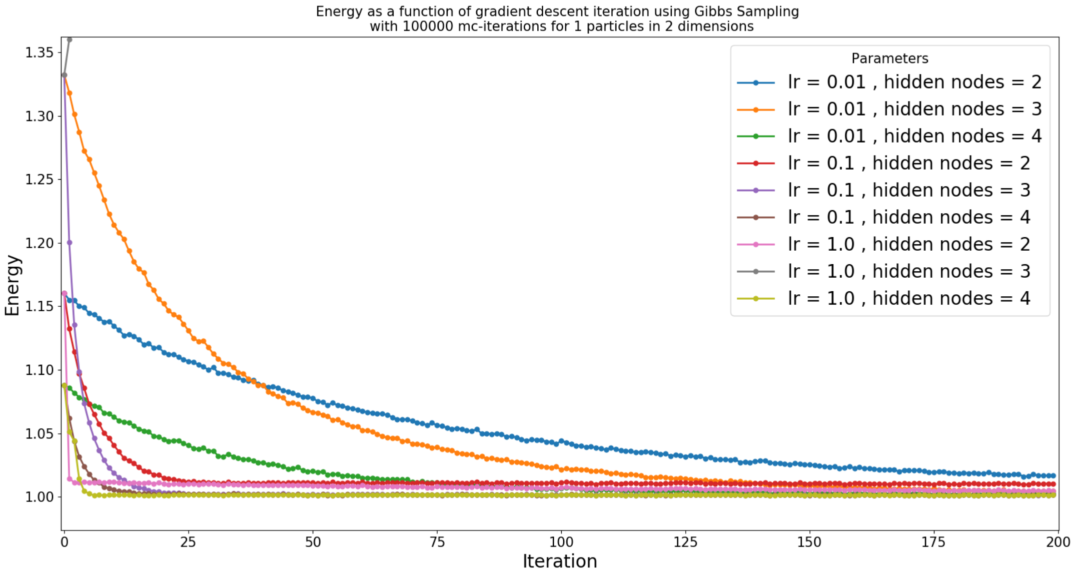


Figure 8: Non-interacting system. Energies in units of a.u.

4.1.4 Comparison of the methods

In this section the best parameter combinations are compared on two different sized systems. The convergence are also compared. For the brute force method, a step-length of 0.1 is used. For the importance sampling method, a time-step of 0.005 is used. For Gibbs sampling, the Boltzmanns machines σ^2 is set to 0.48. All methods use a learning rate of 1 with 4 hidden nodes. In figure 10 and table 7 the results from running with the best parameter combinations for each

Table 5: Optimal energy, error and time for different network architectures using Gibbs Sampling on a non-interacting system with 1 particles in 2 dimensions. Using σ^2 =0.5 and MC-iterations = 100000

Network Architecture	Optimal Energy [a.u.]	Abs error	Rel error	Time [s]
lr = 0.01 , hidden nodes = 2	1.0170	0.0170	0.0170	31.936
lr = 0.01 , hidden nodes = 3	1.0044	0.0044	0.0044	37.416
lr = 0.01 , hidden nodes = 4	1.0023	0.0023	0.0023	38.309
lr = 0.1 , hidden nodes = 2	1.0101	0.0101	0.0101	32.552
lr = 0.1 , hidden nodes = 3	1.0013	0.0013	0.0013	35.342
lr = 0.1 , hidden nodes = 4	1.0014	0.0014	0.0014	38.114
lr = 1.0 , hidden nodes = 2	1.0048	0.0048	0.0048	31.460
lr = 1.0 , hidden nodes = 3	NaN	NaN	NaN	31.465
lr = 1.0 , hidden nodes = 4	1.0013	0.0013	0.0013	36.479

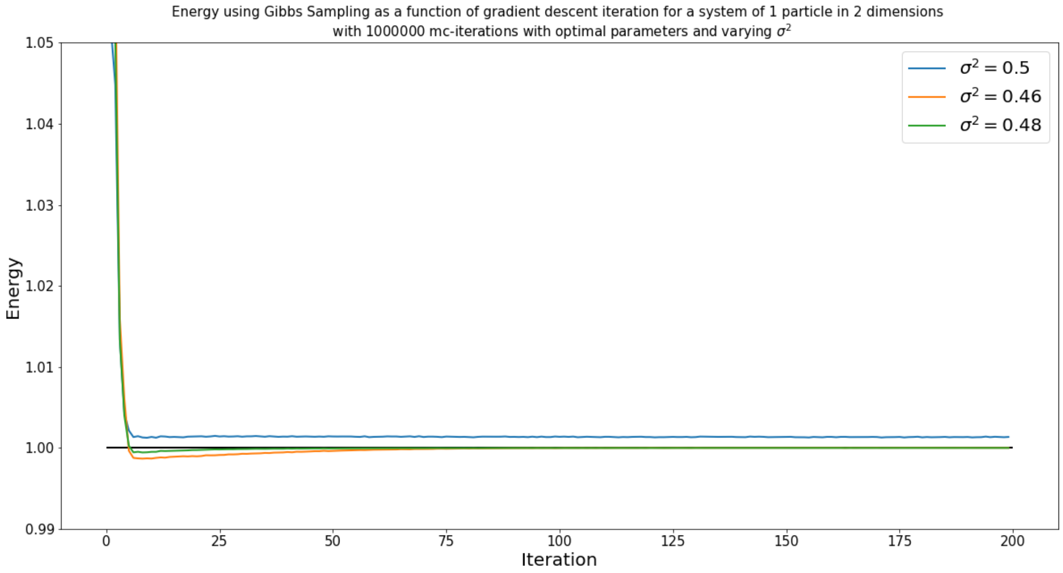


Figure 9: Non-interacting. Energies in units of a.u. Optimal parameters for Gibbs sampling: Learning rate = 1 and hidden nodes = 4.

Table 6: Table of ground state energy estimate and its variance using Gibbs sampling on a non-interacting system of 1 particle in 2 dimensions. The σ^2 is varied to find the optimal value.

Method	E [a.u.]	Variance
Gibbs $\sigma^2 = 0.46$	1.000029	5.178936e-11
Gibbs $\sigma^2 = 0.48$	1.000006	3.445578e-12
Gibbs $\sigma^2 = 0.50$	1.001319	1.018195e-09

method are shown. The main thing to note about the convergence of the different methods is the smoothness of the convergence. The convergence for Brute-force is clearly the worst, with the values jumping up and down from iteration to iteration. When using importance-sampling the convergence gets smoother. This is because the variance of the estimated energy and gradients are lower, as we saw in the analysis of the convergence properties of the markov chains. For brute-force, the variance for the step length of 0.1 was 9.97e-08. For importance-sampling, the variance for a time-step of 0.005 was 2.74e-08. The plot shows that using Gibbs sampling leads to faster and more stable convergence. The table also shows that the variance is the lowest for Gibbs-sampling, with an variance of 3.44e-12, as well as giving the best estimate for the ground state energy of 1.000006 a.u. The exact value is 1. It also shows that by adding the importance sampling to Metropolis leads to lower variance and in general a better estimate compared to the brute force method. In addition, the computational time of Gibbs-sampling is lower compared to the other two methods. To conclude, for a system with 1 particle in 2 dimensions, Gibbs sampling was clearly the best alternative with a final estimate of the ground state energy of 1.000006.

The system can be extended to 2 particles in 3 dimensions and the optimal configurations

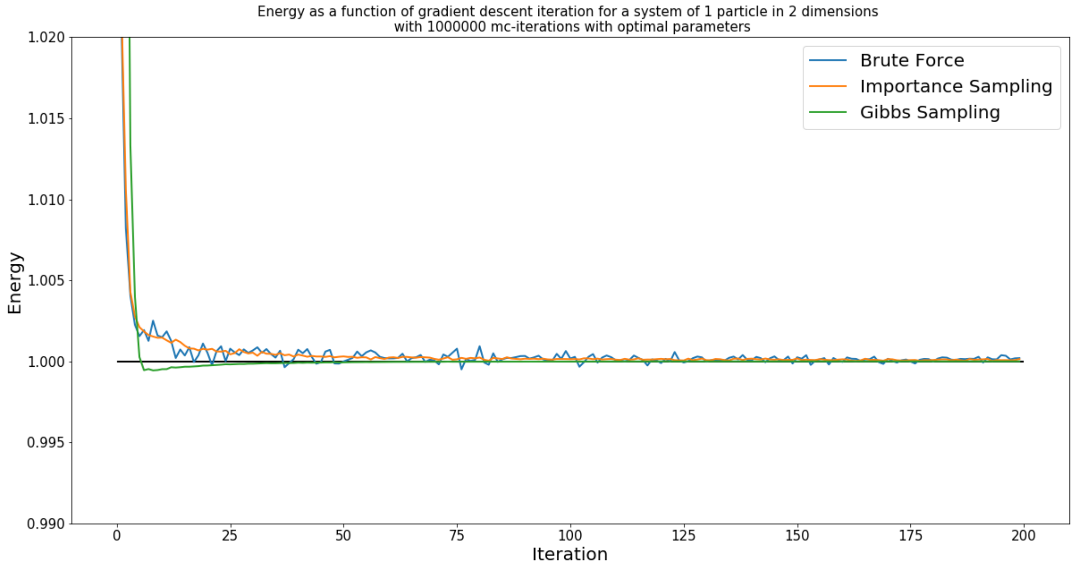


Figure 10: Non-interacting. Energy in units of a.u. Optimal parameters are: Brute force: learning rate = 1, hidden nodes = 4 and $\sigma^2 = 1$. Importance sampling: learning rate = 1, hidden nodes = 4 and $\sigma^2 = 1$. For Gibbs sampling: learning rate = 1, hidden nodes = 4 and $\sigma^2 = 0.48$.

Table 7: Table of optimal ground state estimates for each method for a non-interacting system of 1 particle in 2 dimensions. Optimal parameters are: Brute force: learning rate = 1, hidden nodes = 4 and $\sigma^2 = 1$. Importance sampling: learning rate = 1, hidden nodes = 4 and $\sigma^2 = 1$. For Gibbs sampling: learning rate = 1, hidden nodes = 4 and $\sigma^2 = 0.48$.

Method	E [a.u.]	Variance	Time [s]
Brute-Force	1.000146	3.474843e-08	1.839
Importance-Sampling	1.000034	1.430919e-09	1.734
Gibbs-Sampling	1.000006	3.445578e-12	1.703

Table 8: Table of ground state energies for Gibbs sampling with varying σ^2. The system is 2 particles in 3 dimensions and non-interacting.

Method	Energy [a.u.]	Variance
Gibbs, $\sigma^2 = 0.48$	3.005347	8.123279e-09
Gibbs, $\sigma^2 = 0.50$	3.006696	7.830133e-09
Gibbs, $\sigma^2 = 0.52$	3.011223	2.158457e-08

from before can be tested again. For this system, the optimal value is 3 a.u. The results are shown for brute force with step length 0.1, importance sampling with time-step 0.005. All methods used the optimal learning rate of 1.0 and 4 hidden nodes. All optimization steps uses 1000000 MCMC-samples to estimate the local energies and gradients. Again, I will test for different σ values to see if it is possible to obtain better results. Varying σ for the brute force and importance sampling methods gave no better results. However, for Gibbs sampling, the results are shown in figure 11. The figure shows that $\sigma^2 = 0.48$ gives the best results and it seems to be the minimum. In figure 12 the optimal results are shown for each method, and the ground state energy approximation for each method are shown in table 9. For this system, Gibbs sampling performs much worse compared to for the smaller system. In this case, both brute force sampling and importance sampling gives much better results when it comes to accuracy. Importance sampling gave the best result, with an energy estimate of 3.0053.

4.2 Interacting system

Now that the methods seems to work on the non-interacting system, it is time to test them on a more realistic system where interactions between the particles are accounted for. For the interacting system the exact values are only known for a system of two particles in two dimensions. The ground state energy for this system is given by 3 a.u. Again, the optimal

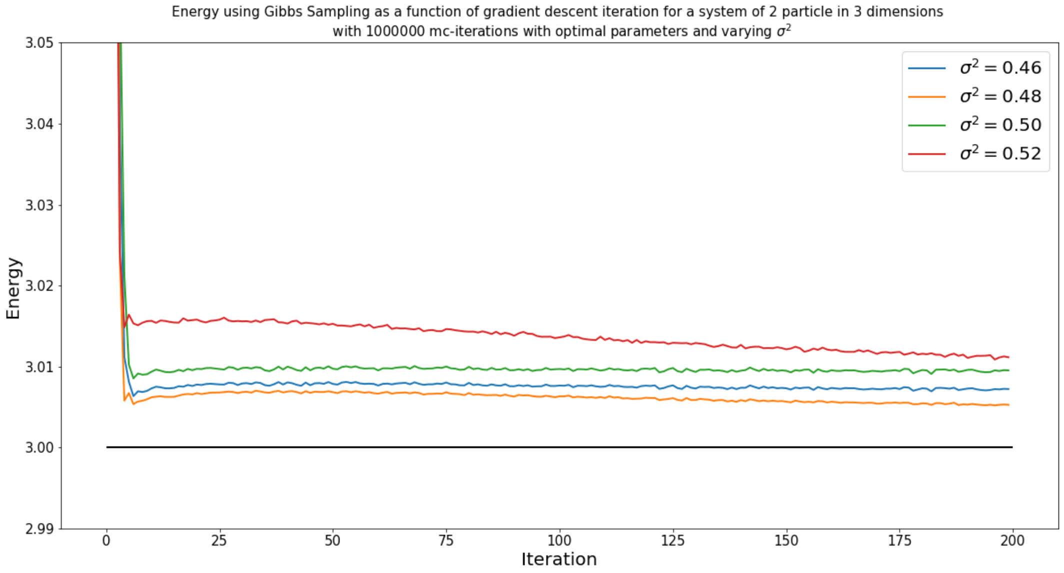


Figure 11: Non-interacting. Energy in units of a.u. Optimal parameters for Gibbs sampling: learning rate = 1, hidden nodes = 4.

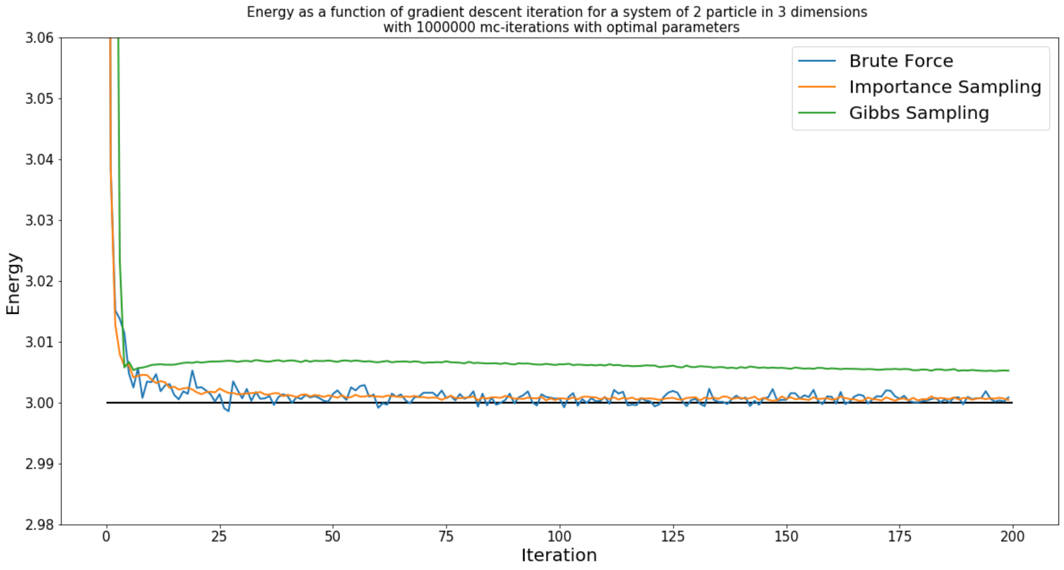


Figure 12: Non-interacting. Energy in units of a.u. Optimal parameters are: Brute force: learning rate = 1, hidden nodes = 4, $\sigma^2 = 1$. Importance sampling: learning rate = 1, hidden nodes = 4, $\sigma^2 = 1$. For Gibbs sampling: learning rate = 1, hidden nodes = 4 and $\sigma^2 = 0.48$.

Table 9: Table for optimal parameters on non-interacting system with 2 particles in 3 dimensions. Exact value is 3 a.u. Optimal parameters are: Brute force: learning rate = 1, hidden nodes = 4, $\sigma^2 = 1$. Importance sampling: learning rate = 1, hidden nodes = 4, $\sigma^2 = 1$. For Gibbs sampling: learning rate = 1, hidden nodes = 4 and $\sigma^2 = 0.48$.

Method	Energy [a.u.]	Variance
Brute-Force	3.001203	6.937688e-07
Importance-Sampling	3.000751	2.967982e-08
Gibbs-Sampling	3.005347	8.123279e-09

parameters will be found using a grid-search. For the interacting case, the effect of the sigma value will be studied for all methods. The results are presented in the sections below and at the end the results are compared. The methods using Metropolis brute force sampling and Metropolis importance sampling use the same step length of 0.1 and time step of 0.005 as was found in the analysis of the non-interacting system. Those seemed to give good convergence properties.

Table 10: Optimal energy, error and time for different network architectures using Metropolis Brute-force on a interacting system with 2 particles in 2 dimensions. Using σ^2 =1.0, MC-iterations = 1000000 and MC step-length = 0.1 and 300 optimization steps.

Network Architecture	Optimal Energy [a.u.]	Abs error	Rel error	Time [s]
lr = 0.01 , hidden nodes = 2	3.2354	0.2354	0.078467	574.306
lr = 0.01 , hidden nodes = 3	3.1987	0.1987	0.066233	640.024
lr = 0.01 , hidden nodes = 4	3.1686	0.1686	0.056200	681.869
lr = 0.1 , hidden nodes = 2	3.1468	0.1468	0.048933	583.606
lr = 0.1 , hidden nodes = 3	3.0851	0.0851	0.028367	639.169
lr = 0.1 , hidden nodes = 4	3.0860	0.0860	0.028667	702.625
lr = 1.0 , hidden nodes = 2	3.1172	0.1172	0.039067	588.109
lr = 1.0 , hidden nodes = 3	3.0830	0.0830	0.027667	653.079
lr = 1.0 , hidden nodes = 4	3.0840	0.0840	0.028000	715.149

4.2.1 Metropolis Brute-Force

In figure 13 and table 10 the results from running the grid search on a interacting system with 2 particles in 2 dimensions are shown. The simulations are run for 300 optimization steps with 1000000 MCMC-samples for each step in the optimization. The immediate thing to notice is the bad convergence. The values vary a lot in every direction from one optimization step to the next. This comes from the high variance in the brute force methods estimates. Looking at the different number of hidden nodes and learning rate, it shows that the high learning rates converge fast and gives the best result. Looking at the lines for a learning rate of 1, and 3 or 4 hidden nodes, it converges very fast. However, it overshoots the actual ground state energy of 3 a.u by quite a bit. Again, it can be seen that by using 2 hidden nodes, the accuracy gets even worse. Looking at the pink line with learning rate 1 and 2 hidden layers, the result are much worse compared to 3 and 4 hidden nodes. For the other combinations of lower learning rates, they do not converge to the limiting distribution within the 300 optimization steps. It is possible that those would give better results in the long run, but due to the high computational time, the number of optimization steps was limited to 300. When it comes to the computational time for the different methods they also vary a bit more than for the non-interacting system.

The table shows that the best combination of parameters are a learning rate of 1 with 3 hidden nodes, which gave a estimate of 3.0840 for the ground state energy. Testing for different σ values had seems to have a small effect 14. It looks like the line for σ^2 = 0.95 in general is the line with the lowest energy. This will be used for the comparison.

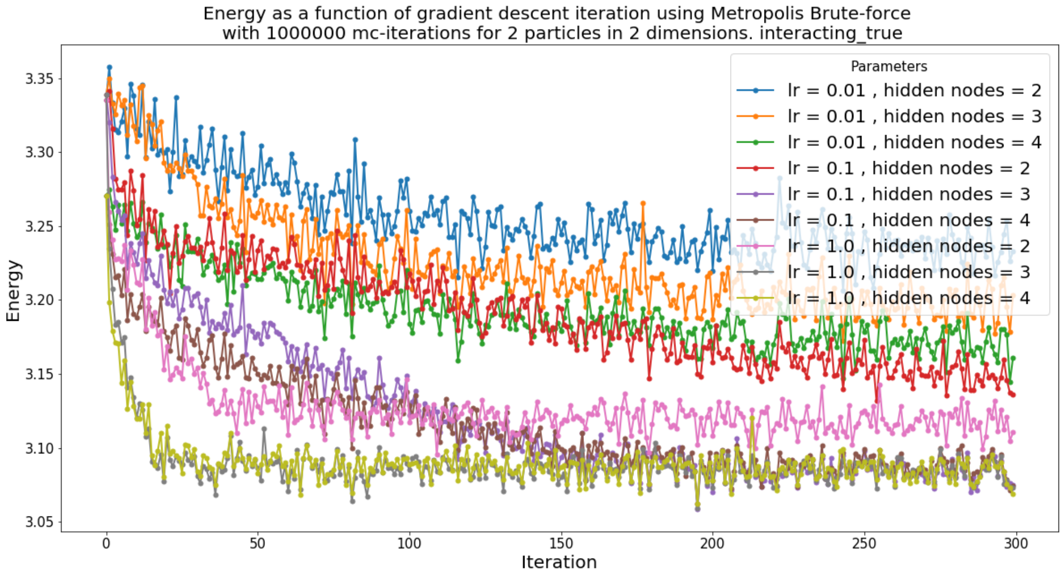


Figure 13: The energy is in units of a.u.

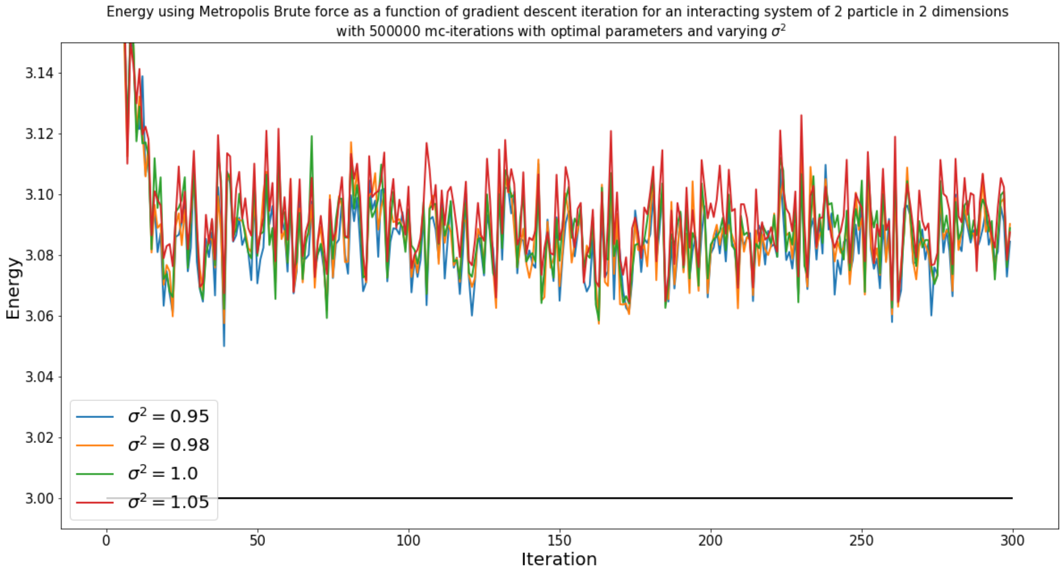


Figure 14: Energy in units of a.u. Optimal parameters for Brute force sampling: learning rate = 1, hidden nodes = 3.

4.2.2 Metropolis Importance-Sampling

The results from running importance sampling are found in figure 15 and 11. All are run for 300 optimization steps with 1000000 samples for each step. The general trend of the convergence of the different parameter combinations seems to be close to the same as for the brute force method. However, the convergence is much smoother, and the variations from one optimization step to the next is much smaller. From the figure and the table, it seems like a learning rate of 1 with 3 hidden nodes gave the best results. As for for brute force sampling, the effect of changing σ^2 was small. However, it looks like $\sigma^2 = 0.9$ gave slightly better results in figure 16.

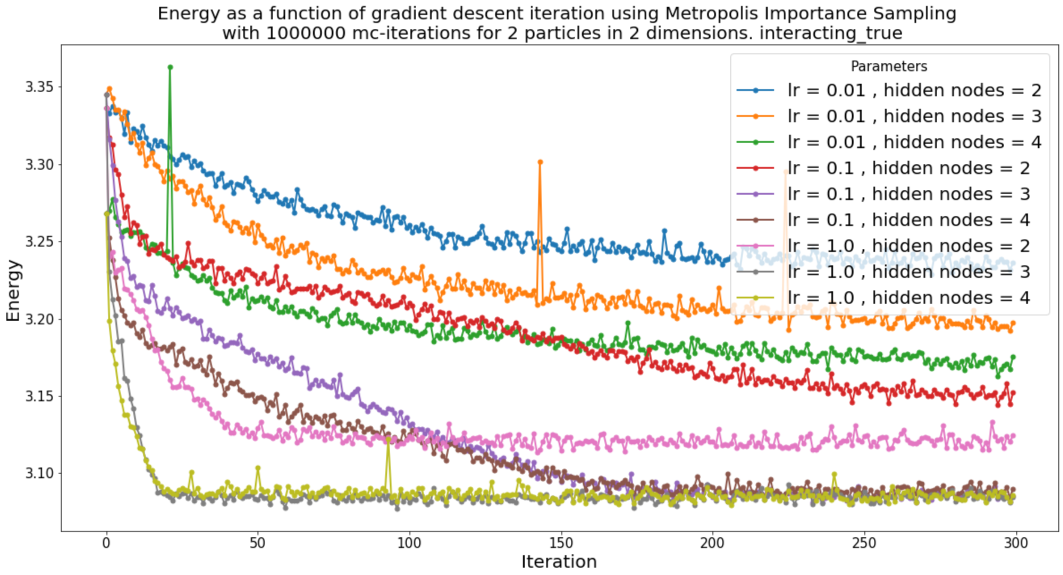


Figure 15: Energy in units of a.u.

4.2.3 Gibbs Sampling

The results from running Gibbs sampling are found in figure 17 and table 12. Again, the best combination is a learning rate of 1, with 3 hidden nodes. Varying σ have a greater effect for Gibbs sampling. The best convergence is obtained using $\sigma^2 = 0.45$ as can be seen in figure 18.

4.3 Comparison

Now that the best parameters are found for all the methods, we can compare the best to each other. For the brute force method, the optimal parameters was a learning rate of 1 with 3 hidden nodes and the Boltzmann machines σ^2 set to 0.95. For importance sampling, the best

Table 11: Optimal energy, error and time for different network architectures using Metropolis Importance Sampling on a interacting system with 2 particles in 2 dimensions. Using σ^2 =1.0, MC-iterations = 1000000 and MC step-length = 0.005 and 300 optimization steps.

Network Architecture	Optimal Energy [a.u]	Abs error	Rel error	Time [s]
lr = 0.01 , hidden nodes = 2	3.2360	0.2360	0.078667	530.966
lr = 0.01 , hidden nodes = 3	3.1962	0.1962	0.065400	594.376
lr = 0.01 , hidden nodes = 4	3.1706	0.1706	0.056867	653.603
lr = 0.1 , hidden nodes = 2	3.1503	0.1503	0.050100	549.037
lr = 0.1 , hidden nodes = 3	3.0854	0.0854	0.028467	629.133
lr = 0.1 , hidden nodes = 4	3.0890	0.0890	0.029667	686.497
lr = 1.0 , hidden nodes = 2	3.1220	0.1220	0.040667	554.398
lr = 1.0 , hidden nodes = 3	3.0843	0.0843	0.028100	616.330
lr = 1.0 , hidden nodes = 4	3.0855	0.0855	0.028500	680.806

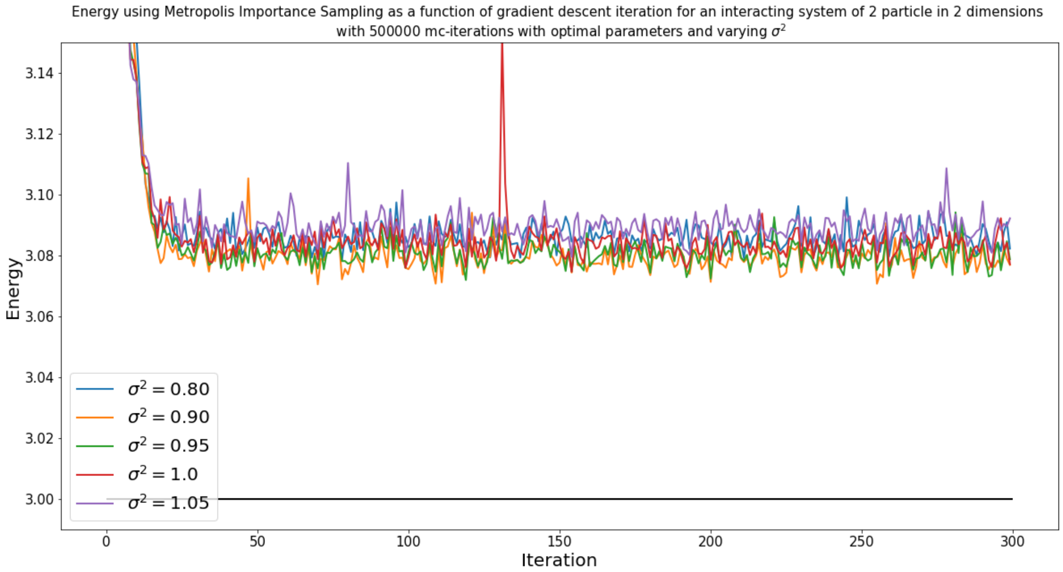


Figure 16: Energy in units of a.u. Optimal parameters for Importance sampling: learning rate = 1, hidden nodes = 3.

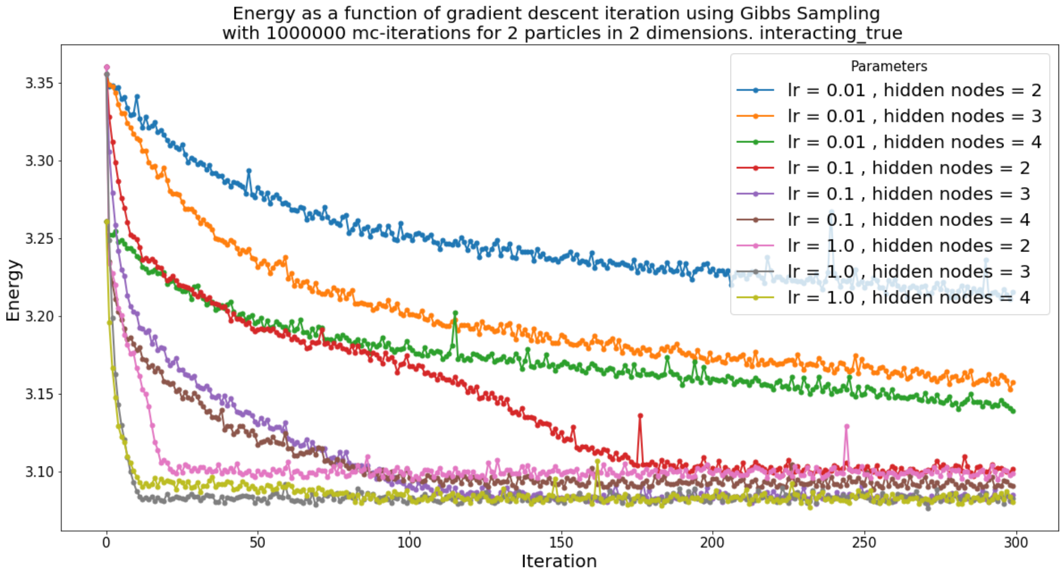


Figure 17: Energy in units of a.u.

combination was also a learning rate of 1 with 3 hidden nodes, but now the σ^2 is set to 0.9. The same learning rate and number of hidden nodes was the case for Gibbs sampling, but the optimal value of σ^2 was σ^2 = 0.45. The results can be seen in figure 19 and table 13. The figure shows that the convergence of Gibbs sampling is fastest and more stable. However, it seems to diverge away for one point, but it manages to get back to convergence. All methods

Table 12: Optimal energy, error and time for different network architectures using Gibbs Sampling on a interacting system with 2 particles in 2 dimensions. Using σ^2 =0.5 and MC-iterations = 1000000 and 300 optimization steps.

Network Architecture	Optimal Energy [a.u]	Abs error	Rel error	Time [s]
lr = 0.01 , hidden nodes = 2	3.2166	0.2166	0.072200	578.779
lr = 0.01 , hidden nodes = 3	3.1589	0.1589	0.052967	639.968
lr = 0.01 , hidden nodes = 4	3.1432	0.1432	0.047733	704.036
lr = 0.1 , hidden nodes = 2	3.1016	0.1016	0.033867	577.463
lr = 0.1 , hidden nodes = 3	3.0839	0.0839	0.027967	657.768
lr = 0.1 , hidden nodes = 4	3.0927	0.0927	0.030900	716.776
lr = 1.0 , hidden nodes = 2	3.0990	0.0990	0.033000	594.737
lr = 1.0 , hidden nodes = 3	3.0830	0.0830	0.027667	632.372
lr = 1.0 , hidden nodes = 4	3.0840	0.0840	0.028000	699.002

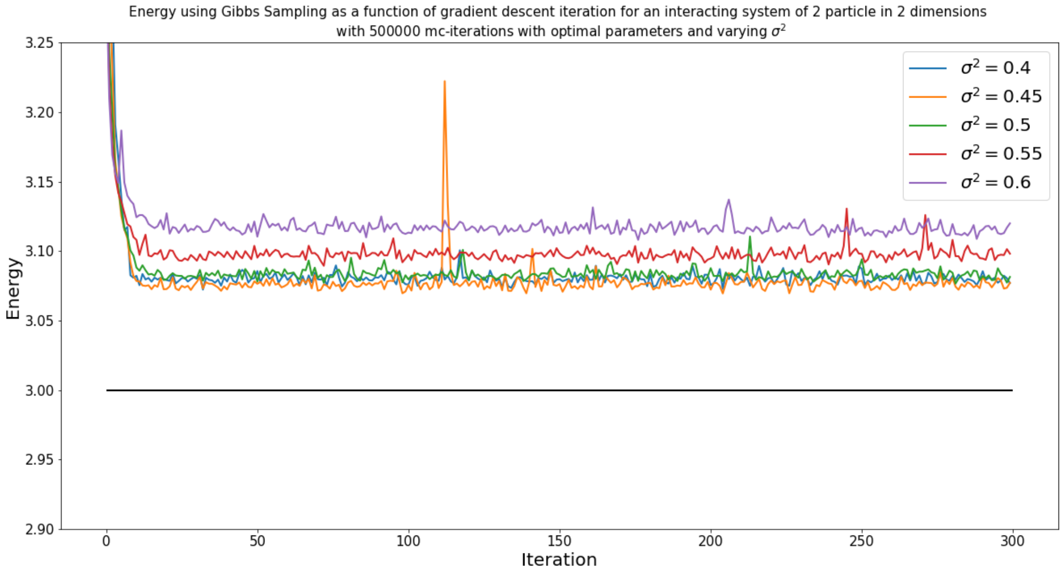


Figure 18: Energy in units of a.u. Optimal parameters for Gibbs sampling: learning rate = 1, hidden nodes = 3.

overshoots the actual solution by a bit. The table also shows that the final estimate of the ground state energy is most precise for Gibbs sampling with an error of 0.078. Those final estimates was found by first running the optimization algorithms for 300 iterations with 500000 MCMCM-samples for each step. Then, with the final weights found in the optimization process, one final run of the sampling algorithms with 10000000 MCMC-samples was used to calculate the ground state energy and the variances. Again, Gibbs sampling gives the best results. Both in terms of convergence speed but also in accuracy.

Table 13: Table of the optimal ground state energy estimates for the interacting system of two particles in two dimensions. The exact value is 3 a.u. Optimal parameters are: Brute force: learning rate = 1, hidden nodes = 3, σ^2 = 0.95. Importance sampling: learning rate = 1, hidden nodes = 3, σ^2 = 0.90. For Gibbs sampling: learning rate = 1, hidden nodes = 3 and $\sigma^2 = 0.45$.

Method	Energy [a.u]	Abs Error	Variance
Brute force sampling	3.078907	0.078907	5.559442e-06
Importance sampling	3.079198	0.079198	7.250546e-07
Gibbs sampling	3.077630	0.077630	8.872653e-07

5 CONCLUSION

In this article the method presented by G. Carleo and M. Troyer in their article "Solving the quantum many-body problem with artificial neural networks" was tested on a system of elec-

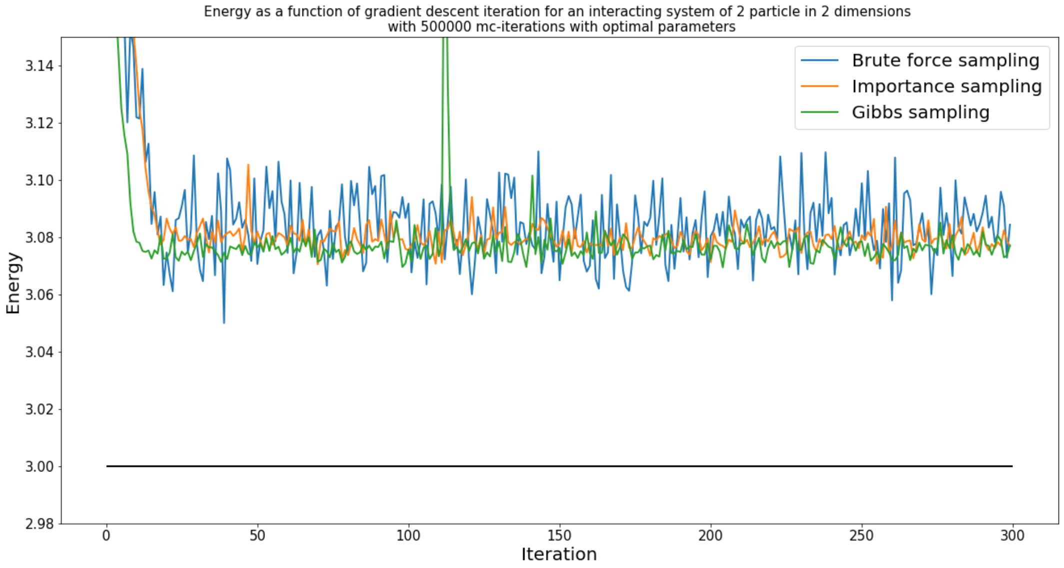


Figure 19: Energy in units of a.u. Optimal parameters are: Brute force: learning rate = 1, hidden nodes = 3, $\sigma^2 = 0.95$. Importance sampling: learning rate = 1, hidden nodes = 3, $\sigma^2 = 0.90$. For Gibbs sampling: learning rate = 1, hidden nodes = 3 and $\sigma^2 = 0.45$.

trons confined to move in a harmonic oscillator trap. Their idea was to use a Boltzmann machine to represent the wave function. In this article, the Boltzmann machine was used in combination with the 3 MCMC methods; Metropolis Brute force sampling, Metropolis Importance sampling and Gibbs sampling. The goal was to find the ground state energy of the system of electrons.

The analysis started with testing the algorithm on a non-interacting system with 1 particle in 2 dimensions. The convergence properties of the sampling methods was studied as well. The Gibbs samples performed best, and had the best accuracy as well as the smallest variance. The two other methods also gave good results. The absolute error of the method using Gibbs sampling was 0.000034 a.u. with a variance of 1.43e-09. When extending the non-interacting system to 2 particles in 3 dimensions, the Gibbs sampler no longer gave the best results. For this system, the methods using importance sampling had the best accuracy. It gave an estimated ground state energy of 3.000751 a.u.. The exact value was 3 a.u.. In general, the smoothness of the convergence for the Gibbs sampler seems to be the best, but for the larger system it had problems with converging to the exact value. In general, it seems like the higher the number of hidden nodes in the Boltzmann machine, the better the network is at representing the wave function. In addition, a learning rate of 1.0 seems to be the best, since it provides high convergence rate while maintaining accuracy.

When the methods worked properly on the non-interacting system, the system was extended to include the interactions. All of the methods had problems getting the same accuracy as for the non-interacting system. However, with the optimal parameters for the Gibbs sampler, it obtained an estimate of the ground state energy of 3.078 a.u.. The exact value was 3 a.u., so the estimate was not to bad. The same trends about the number of hidden nodes and learning rate was found for the interacting system as well.

To conclude, it seems like the Gibbs sampler was the method that performed the best overall. In addition, the number of hidden nodes in the Boltzmann machine seems to have quite a large effect on the accuracy of the estimates. Also, the learning rate of 1.0 gave the best results for all of the test conducted in this report.

An idea for further work can be to test the methods on more complicated systems. This can for example be interacting system with more than two particles. In addition, it would be interesting to run optimization for longer on the interacting case, to see if any of the combinations with smaller learning rates gave better results if given enough time to converge properly. Unfortunately, I did not have time to do this for this article. It would also be interesting to try other generative networks for representing the wave function. More modern generative methods could probably get even better results. However, the Boltzmann machine gave surprisingly good results, at least for the non-interacting system.

6 APPENDIX

In this part derivations of the main results from the theory are presented.

6.1 Finding the local energy

The derivation of the local energy is inspired by the derivation in []. The local energy is given by

$$E_L = \frac{1}{\psi}\hat{\mathbf{H}}\psi \tag{37}$$

where $\hat{\mathbf{H}}$ is the Hamiltonian operator given by

$$\hat{\mathbf{H}} = \sum_i^P(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2) + \sum_{i<j}\frac{1}{r_{ij}} \tag{38}$$

and ψ is the wave function. The wave function is given by

$$\psi(\boldsymbol{X}) = \frac{1}{Z}e^{-\sum_i^M (x_i-a_i)^2/2\sigma^2}\prod_j^N\left(1+e^{b_j+\sum_i^M X_i w_{ij}}\right) \tag{39}$$

Inserting this gives that

$$\begin{aligned}
E_L &= \frac{1}{\psi}\boldsymbol{H}\psi = \frac{1}{\psi}\left(\sum_i^P(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2) + \sum_{i<j}\frac{1}{r_{ij}}\right)\psi \\
&= \frac{1}{\psi}\left(\sum_i^P(-\frac{1}{2}\nabla_i^2\psi + \frac{1}{2}\omega^2 r_i^2\psi) + \psi\sum_{i<j}\frac{1}{r_{ij}}\right) = -\frac{1}{2\psi}\sum_i^P\nabla_i^2\psi + \frac{1}{2}\omega^2\sum_j^P r_j^2 + \sum_{p<q}\frac{1}{r_{pq}} \\
&= -\frac{1}{2\psi}\sum_i^P\sum_d^D\frac{\partial^2\psi}{\partial x_{id}^2} + \frac{1}{2}\omega^2\sum_j^P\sum_d^D x_{jd}^2 + \sum_{p<q}\frac{1}{r_{pq}} \\
&= \frac{1}{2}\sum_p^P\sum_d^D\left(-\left(\frac{\partial}{\partial x_{pd}}\ln\psi\right)^2 - \frac{\partial^2}{\partial x_{pd}^2}\ln\psi + \omega^2 x_{pd}^2\right) + \sum_{p<q}\frac{1}{r_{pq}}
\end{aligned} \tag{40}$$

where P is the number of particles, D is the number of dimensions, and x_{ij} is the j'th coordinate of the i'th particle. The last step comes from the fact that

$$-\left(\frac{\partial}{\partial x_{pd}}\ln\psi\right)^2 - \frac{\partial^2}{\partial x_{pd}^2}\ln\psi = -\frac{1}{\psi^2}\left(\frac{\partial\psi}{\partial x_{pd}}\right)^2 - \left(-\frac{1}{\psi^2}\left(\frac{\partial\psi}{\partial x_{pd}}\right)^2 + \frac{1}{\psi}\frac{\partial^2\psi}{\partial x_{pd}^2}\right) = \frac{1}{\psi}\frac{\partial^2\psi}{\partial x_{pd}^2} \tag{41}$$

Since the visible vector is one dimensional and represents the position of the particles, each coordinate takes up one space of the vector. The vector is therefore of length $P \cdot D$. If the visible layer is represented by the vector $\boldsymbol{v}$, then the equation can be simplified to

$$\boxed{E_L = \frac{1}{2}\sum_m^M\left(-\left(\frac{\partial}{\partial v_m}\ln\psi\right)^2 - \frac{\partial^2}{\partial v_m^2}\ln\psi + \omega^2 v_m^2\right) + \sum_{p<q}\frac{1}{r_{qp}} \qquad (42)}$$

where m is the elements of the visible layer.

6.2 Finding the gradient with respect to biases and weights

We can use that

$$\frac{1}{\psi}\frac{\partial}{\partial x}\psi = \frac{\partial}{\partial x}\ln\psi \tag{43}$$

and then take the derivative of the logarithm of the wave function. The logarithm of the wave function is given by

$$\ln(\psi(\boldsymbol{X}) = -\ln(Z) - \sum_i^M\frac{(x_i-a_i)^2}{2\sigma^2} + \sum_j^N\ln(1+e^{b_n+\sum_i^M\frac{X_i w_{ij}}{\sigma^2}}). \tag{44}$$

Taking the derivative with respect to the nodes in the visible layer gives

$$\frac{\partial}{\partial a_m}\ln\psi = -\frac{\partial}{\partial a_m}\sum_i^M \frac{(x_i - a_i)^2}{2\sigma^2} = (x_m - a_m)/\sigma^2. \tag{45}$$

Taking the derivative with respect to the nodes in the hidden layer gives

$$\frac{\partial}{\partial b_n}\ln\psi = \frac{\partial}{\partial b_n}\sum_j^N \ln(1 + e^{b_j + \sum_i^M \frac{X_i w_{ij}}{\sigma^2}}) = \frac{1}{(1 + e^{b_j + \sum_i^M \frac{X_i w_{ij}}{\sigma^2}})}\frac{\partial}{\partial b_n}(1 + e^{b_n + \sum_i^M \frac{X_i w_{in}}{\sigma^2}}) \tag{46}$$

$$\frac{\partial}{\partial b_n}\ln\psi = \frac{1}{(\exp\left[-b_n - \frac{1}{\sigma^2}\sum_i^M X_i w_{in}\right] + 1)} \tag{47}$$

Taking the derivative of the logarithm of the wave function with respect to the weights gives

$$\frac{\partial}{\partial w_{mn}}\ln\psi = \frac{\partial}{\partial w_{mn}}\sum_j^N \ln(1 + e^{b_j + \sum_i^M \frac{X_i w_{ij}}{\sigma^2}}) = \frac{1}{(1 + e^{b_j + \sum_i^M \frac{X_i w_{ij}}{\sigma^2}})}\frac{\partial}{\partial w_{mn}}(1 + e^{b_n + \sum_i^M \frac{X_i w_{in}}{\sigma^2}}) \tag{48}$$

which gives that

$$\frac{\partial}{\partial w_{mn}}\ln\psi = \frac{X_m}{\sigma^2\Big(\exp\Big[-b_n - \frac{1}{\sigma^2}\sum_i^M X_i w_{in}\Big] + 1)\Big)} \tag{49}$$

by using the chain rule.

References

[1] G. Carleo and M. Troyer, Science **355**, Issue 6325, pp. 602-606 (2017)

[2] M. Taut, Pyhs. Rev. A **48**, Number 5 (1993)

[3] M. Hjorth-Jensen, *From Variational Monte Carlo to Boltzmann Machines and Machine Learning.* `http://compphysics.github.io/ComputationalPhysics2/doc/pub/NeuralNet/html/NeuralNet.html` (2019)

[4] *Variational Monte Carlo methods* M. Hjorth-Jensen `http://compphysics.github.io/ComputationalPhysics2/doc/pub/vmc/html/vmc.html`

[5] M. K. Hovden, *Neural Networks for classification and regression* `https://github.com/MartinKHovden/FYS-STK4155/blob/master/Project2/FYSSTK4155_PROJECT2_REPORT.pdf` (2019)

[6] M. K. Hovden *VMC for calculating the ground state energy of a trapped hard sphere Bose gas* `https://github.com/MartinKHovden/ComputationalPhysics2/blob/master/Project1_/CompPhys2_PROJECT1_Martin_Krokan_Hovden.pdf` (2020)

[7] M. Hjorth-Jensen, *Computational Physics, lecture notes 2015*, Oslo, 2015.

[8] M. Jonsson, *Standard Error Estimation by an Automated Blocking Method*, Phys. Rev. E **98**, 043304 (2018).

[9] G. H. Givens and J. A. Hoeting, Computational Statistics, Wiley, New Jersey 2013.

[10] M. Hjorth-Jensen *The restricted Boltzmann machine applied to the quantum many body problem,* `https://github.com/CompPhysics/ComputationalPhysics2/blob/gh-pages/doc/Projects/2020/Project2/Project2ML/pdf/Project2ML.pdf`(2020)