Quantum Boltzmann Machines and Auto-encoders

Henrik Modahl Breitenstein

April 5, 2024

Abstract

In this paper we

Contents

Chapter 1

Introduction

Machine learning is becoming more and more prevalent in everyday life. As hardware has become stronger and learning models has been refined, machine learning has become a powerful tool to solve a wide variety of problems. Physics is no exception to this, as the last few years has shown a number of fields in physics furthered by machine learning.
[fields in physics using machine learning].
Quantum physics has often avenues for direct application of machine learning, as one wants to fit a wavefunction within the bounds of the hamiltonian. As such there has been several ways of approach. One of them boils down to having the output of the neural net being the wavefunction amplitude, as done by [feed forward neural net article] on the [model] model. Another approach is to have the machine be the wavefunction and letting the output be measured states, for example done by [rbm article] on the [model] model. The Ising and Heisenberg models has seen much attention with respect to machine learning methods, and we will therefor use the less popular Lipkin model [source] as well.
Another computational field that has seen a great increase in interest is the field of quantum computation. Pioneered by [person] and [person], showing that quantum computation has some serious advantages to its classical counterparts in certain areas. A widely used algorithm from quantum computation is the variational quantum eigensolver. And as quantum physics problems can often be simplified as an eigenvalue-problem, we will use this algorithm as a way to compare the machine learning method with an alternative.

Part I

Theory

Chapter 2

Machine Learning

Machine learning is a machine adapting the way it processes some information by looking at its own output from said processing. A neural network is such a machine.

2.1 Basic Principles

2.1.1 Nodes and Layers

The fundamental building block of neural networks are nodes, which often is analogously seen as an neuron from in a brain, hence the name neural network. In a node we have the bias and the activation function and a weight for each connected node.

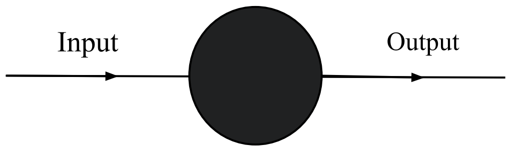


Figure 2.1: A singular node. Has an input which is affected by the weight, bias and activation function within the node.

In the figure above the node takes in a input from the left, x. The weight, w, and bias, b, create the intermediate output of the node accordingly:

$$n = wx + b \; . \tag{2.1}$$

Then n is passed through an activation function, σ and we get the output of the node

$$z = \sigma\left(n\right) \; . \tag{2.2}$$

Combining several nodes which takes the same input we get a layer. Layers are then connected where the previous layer's output is sent as input to the next via these connections. This is the simplest network, called a feedforward network, as we pass the input through layer by layer without any inter-layer connections.

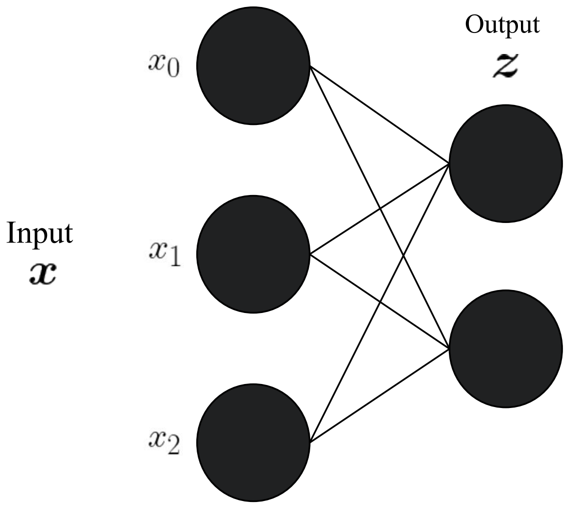


Figure 2.2: A simple feedforward neural network where we have an input layer, individual connections to the input, and a output layer. The output layer's nodes are all connected to each of the input layer's nodes.

Here the first layer is called an input layer as it has a one to one correspondence with a input data point. The input and output layer is fully connected however, where the output of each of the input layer's nodes are sent to each of the output layer's, where each connection has its own weight. Using vectors to represent the layers we have

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \tag{2.3}$$

as input, which is then inserted in the input layer's nodes as shown in 2.2. The weights now has an extra dimension as each nodes is connected to each node in the next layer, so we can represent it with a matrix:

$$W = \begin{bmatrix} w_{00} & w_{10} & w_{20} \\ w_{01} & w_{11} & w_{21} \end{bmatrix} \tag{2.4}$$

where the weight w_{kj} is the weight of the connection between node k of the input layer and node j of the output layer. Feeding the input forward we first get the intermediate output

$$\boldsymbol{n} = W\boldsymbol{x} + \boldsymbol{b} \tag{2.5}$$

where we use simple matrix multiplication. The output layer often has a unique activation function to make the output more applicable to a certain context. For example, in classification problems the softmax function

$$\sigma_s(\boldsymbol{z}')_i = \frac{e^{z'_i}}{\sum_{j=1}^{N} e^{z'_j}} \ , \tag{2.6}$$

which normalizes the output to a probability distribution, is often used. Using the softmax activation function, we have the final output

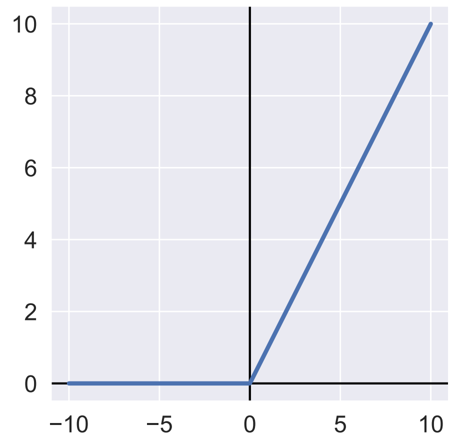
$$\boldsymbol{z} = \sigma_s\left(\boldsymbol{n}\right) \ . \tag{2.7}$$

The network model is easily expandable to more layers as one just sends the output of a layer into the next by following equation 2.5.

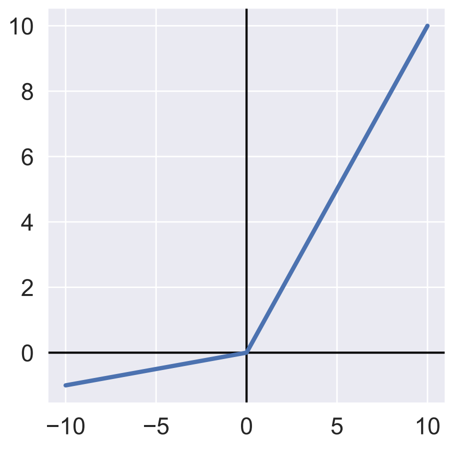
2.1.2 Activation function

The term activation function is inspired from the way biological neurons 'activate' after a certain signal threshold is met. The activation function chosen for a given network can have large impact on the finished trained model. Some common activation functions are

ReLU
$max\left(0, x\right)$



The Rectified Linear Unit activation function is one of the most basic activation functions. Since it does not have an upper bound, the activation function can have problems with values becoming too large. ReLU is however one of the most popular activation function because it has an derivative easy to calculate and avoids the vanishing gradient problem during learning.

Leaky ReLU $max\left(ax, x\right)\ a \in \langle 0, 1\rangle$	

Leaky ReLU adds the possibility for negative values to impact the output of the node.

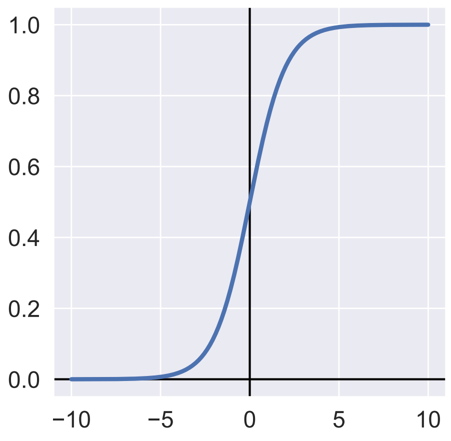
Sigmoid $\sigma\left(x\right) = \frac{1}{1+e^{-x}}$	

Table 2.1

The Sigmoid activation function has both a upper and lower bound, preventing the values from escalating out of control, but has an more computationally expensive derivative and can struggle with the vanishing gradient problem during learning.

2.1.3 Training a neural network

Training a neural network is done by changing the weights and biases such that the output comes closer to a desired goal. This goal is to minimize a cost function C, which can be any function. As an example, an common cost function for regression is the squared error

$$C = \sum_k \left(t_k - z_k\right)^2 \tag{2.8}$$

where t is a target value and z is the model output. We want to change the weights and biases in such a way that the cost function decreases. A natural way would be to use the gradient of the cost function in terms of the weights and biases, and decrease the variables by a proportional amount, such that we approach the minima of the cost function. Finding the gradient of the cost function in terms of the weights and biases is done by backpropagation, a method introduced by S. Linnainmaa in 1960 [1], completed by F. Rosenblatt in 1976 [2] and popularized by D. E. Rumelhart in 1982 [3]. The method is derived by application of the chain rule starting from the output and going layer by layer to get to the input.
We expand our simple input-output model 2.2 and define a subscript for the different layers and nodes as shown here:

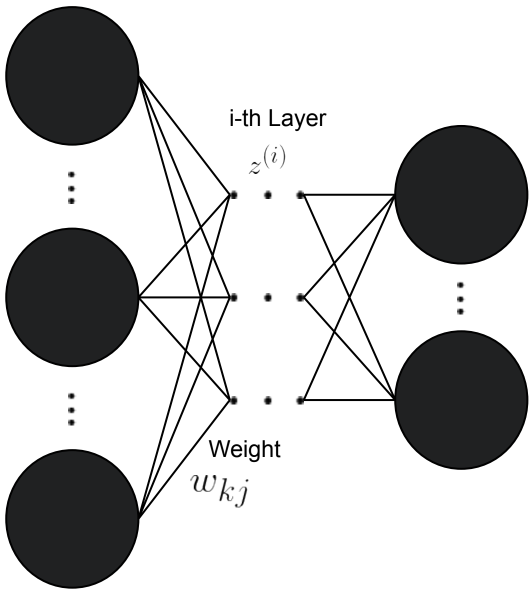


Figure 2.3: A more general network where we w_{kj} is the weight matrix

We have that:

$\boldsymbol{z}^{(i)}$ is the i-th layer vector output. (i) is the layer subscript, going from the input layer $z^{(0)}$ to the output layer $z^{(L)}$.

$\boldsymbol{n}$ is the intermediate output of a layer before it is sent through an activation function.

σ is an activation function.

$W^{(i)}$ is a weight matrix with weights w_{kj}, which is the weight from k-th node of the i-th layer to node j of layer $i+1$.

Starting from the output we want to find

$$\Delta W^{(L-1)} = -\eta \frac{\partial C}{\partial W^{(L-1)}} , \tag{2.9}$$

where η is a chosen learning rate. The change in a single weight

$$\Delta w_{kj}^{(L-1)} = -\eta \frac{\partial C}{\partial w_{kj}^{(L-1)}} \ , \tag{2.10}$$

where we can add the dependency on the activation function and intermediate output

$$\Delta w_{kj}^{(L-1)} = -\eta \frac{\partial C}{\partial z_k^{(L)}} \frac{\partial z_k^{(L)}}{\partial n_k^{(L)}} \frac{\partial n_k^{(L)}}{\partial w_{kj}^{(L-1)}} \tag{2.11}$$

Where we have that

$$\frac{\partial C}{\partial z_k^{(L)}} = -2\left(t_k - z_k^{(L)}\right) \tag{2.12}$$

and using the sigmoid activation function, 2.1, we have

$$\frac{\partial z_k^{(L)}}{\partial n_k^{(L)}} = \frac{\partial\left(\frac{1}{1=e^{-n_k}}\right)}{\partial n_k} = \frac{e^{-n_k}}{\left(1+e^{-n_k}\right)^2} = n_k(1-n_k) \tag{2.13}$$

and then lastly the intermediate output

$$\frac{\partial n_k^{(L)}}{\partial w_{kj}^{(L-1)}} = \frac{\partial\left(w_{kj}^{(L-1)} z_j^{(L-1)} + b_j^{(L-1)}\right)}{\partial w_{kj}^{(L-1)}} = z_j^{(L-1)} \tag{2.14}$$

So we can write the change of weight as

$$\Delta w_{kj}^{(L-1)} = \eta\left(t_k - z_k^{(L)}\right) z_k^{(L)} \left(1 - z_k^{(L)}\right) z_j^{(L-1)} \ . \tag{2.15}$$

We simplify by defining

$$\delta_k^{(L)} = \left(t_k - z_k^{(L)}\right) z_k^{(L)} \left(1 - z_k^{(L)}\right) \tag{2.16}$$

and get

$$\Delta w_{kj}^{(L-1)} = \eta \delta_k^{(L)} z_j^{(L-1)} \ , \tag{2.17}$$

where we have shortened the -2 factor into η as well. For the next layer, $(L-2)$, we have that a individual node is connected to each of the previous layer's nodes. We then have the change

$$\Delta w_{jm}^{(L-2)} = \eta\left[\sum_k \frac{\partial C}{\partial z_k^{(L)}} \frac{\partial z_k^{(L)}}{\partial n_k^{(L)}} \frac{\partial n_k^{(L)}}{\partial z_j^{(L-1)}}\right] \frac{\partial z_j^{(L-1)}}{\partial n_j^{(L-1)}} \frac{\partial n_j^{(L-1)}}{\partial w_{jm}^{(L-2)}} \ , \tag{2.18}$$

where we have calculated the expression within the sum already

$$\frac{\partial C}{\partial z_k^{(L)}} \frac{\partial z_k^{(L)}}{\partial n_k^{(L)}} \frac{\partial n_k^{(L)}}{\partial z_j^{(L-1)}} = \left(t_k - z_k^{(L)}\right) z_k^{(L)} \left(1 - z_k^{(L)}\right) w_{kj}^{(L-1)} = \delta_k^{(L)} w_{kj}^{(L-1)} \ . \tag{2.19}$$

The expression outside the sum is calculated as in 2.13 and 2.14

$$\frac{\partial z_j^{(L-1)}}{\partial n_j^{(L-1)}} \frac{\partial n_j^{(L-1)}}{\partial w_{jm}^{(L-2)}} = z_j^{(L-1)} \left(1 - z_j^{(L-1)}\right) z_m^{(L-2)} \ . \tag{2.20}$$

We can then write the change in weight as

$$\Delta w_{jm}^{(L-2)} = \eta \left[\sum_k \delta_k^{(L)} w_{kj}^{(L-1)} \right] z_j^{(L-1)} \left(1 - z_j^{(L-1)}\right) z_m^{(L-2)} \ , \tag{2.21}$$

which we can further simplify by defining

$$\delta_j^{(L-1)} = \left[\sum_k \delta_k^{(L)} w_{kj}^{(L-1)} \right] z_j^{(L-1)} \left(1 - z_j^{(L-1)}\right) \tag{2.22}$$

and we get

$$\Delta w_{jm}^{(L-2)} = \eta \delta_j^{(L-1)} z_m^{(L-2)} \ . \tag{2.23}$$

This process can then be repeated easily with the fact that

$$\delta_k^{i-1} = \left[\sum_k \delta_k^{(i)} w_{kj}^{(i-1)} \right] z_j^{(i-1)} \left(1 - z_j^{(i-1)}\right) \tag{2.24}$$

for any layers further in.

2.2 Boltzmann Machine

In this thesis we are going to use a slightly different type of neural network, called a Boltzmann machine. First major contribution to Boltzmann machines comes from G. E Hinton and T.J. Sejnowski in 1983 [4]. The neural net is a generative model which learns by matching the probability distribution of its inputs.

2.2.1 Structure and training

A Boltzmann machine has interconnected nodes within a layer.

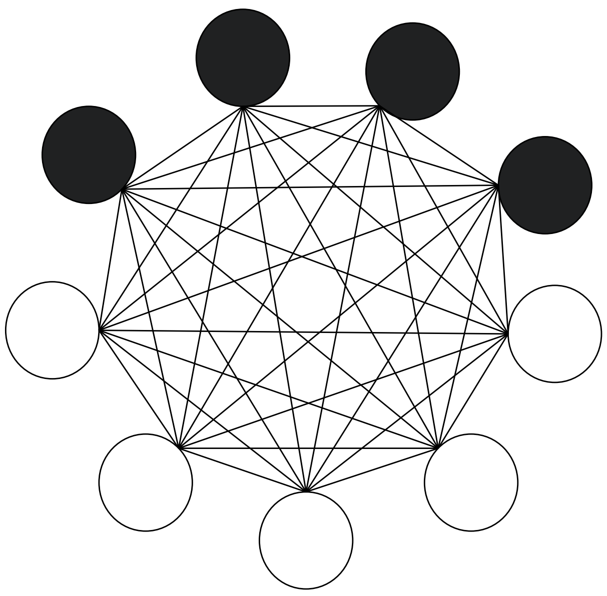


Figure 2.4: A unrestricted Boltzmann machine where every node is connected. Here the black nodes are the visible layer while the white nodes constitute the hidden layer.

The nodes are binary, being able to take the value 0 or 1, have a weights w_{ij} for connection strength between node v_i and h_j and biases a_i for the visible layer and b_j for the hidden layer. For a system of N_h hidden neurons and N_v visible neurons we have the probability distribution of nodes taking the value 1 defined as

$$P(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{Z} e^{-E(\boldsymbol{v}, \boldsymbol{h})} \,, \tag{2.25}$$

where the energy of the model is given by

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_{i}^{N_v} a_i v_i - \sum_{j}^{N_h} b_j h_j - \sum_{i}^{N_v} \sum_{j}^{N_h} v_i w_{ij} h_j \tag{2.26}$$

and the normalization factor

$$Z = \sum_{\boldsymbol{v}, \boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})} \,, \tag{2.27}$$

where we sum over all possible states of the model, which increases exponentially as $2^{(N_v + N_h)}$. The marginal distribution over the visual layer can be written as

$$P(\boldsymbol{v}) = \sum_{\boldsymbol{h}} \frac{1}{Z} e^{-E(\boldsymbol{v}, \boldsymbol{h})} \tag{2.28}$$

To train a Boltzmann machine we need to have a cost function that compares the predicted distribution of the model and the actual distribution of the data set. As such we will use the Kullback-Leibler divergence as a cost function:

$$KL(\boldsymbol{W}, \boldsymbol{a}, \boldsymbol{b}) = \sum_{(\boldsymbol{v},\boldsymbol{h})} R(\boldsymbol{v}) \log \frac{R(\boldsymbol{v})}{P(\boldsymbol{v})} \; , \tag{2.29}$$

where $R(\boldsymbol{v})$ is the distribution we want to approximate and $P(\boldsymbol{v})$ is the distribution of the neural network model. Following the derivations of A.L. Yuille [5] we have that

$$\frac{\partial KL(\boldsymbol{W}, \boldsymbol{a}, \boldsymbol{b})}{\partial w_{ij}} = - \sum_{(\boldsymbol{v},\boldsymbol{h})} \frac{R(\boldsymbol{v})}{P(\boldsymbol{v})} \frac{\partial P(\boldsymbol{v})}{\partial w_{ij}} \; , \tag{2.30}$$

where we further have

$$\frac{\partial P(\boldsymbol{v})}{\partial w_{ij}} = \frac{1}{Z} \frac{\partial}{\partial w_{ij}} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} - \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} \frac{\partial \log Z}{\partial w_{ij}} \tag{2.31}$$

which we can express as

$$\frac{\partial P(\boldsymbol{v})}{\partial w_{ij}} = - \sum_{\boldsymbol{h}} v_i h_j P(\boldsymbol{v}, \boldsymbol{h}) + \sum_{\boldsymbol{h}} \left[P(\boldsymbol{v}, \boldsymbol{h}) \sum_{\boldsymbol{v},\boldsymbol{h}} v_i h_j P(\boldsymbol{v}, \boldsymbol{h}) \right] \; . \tag{2.32}$$

Then

$$\frac{\partial P(\boldsymbol{v})}{\partial w_{ij}} = - \sum_{\boldsymbol{h}} v_i h_j P(\boldsymbol{v}, \boldsymbol{h}) + P(\boldsymbol{v}, \boldsymbol{h}) \sum_{\boldsymbol{v},\boldsymbol{h}} v_i h_j P(\boldsymbol{v}, \boldsymbol{h}) \; . \tag{2.33}$$

Using the result of equation 2.33 in equation 2.30 we get that

$$\frac{\partial KL(\boldsymbol{W}, \boldsymbol{a}, \boldsymbol{b})}{\partial w_{ij}} = \sum_{\boldsymbol{v},\boldsymbol{h}} v_i h_j \frac{P(\boldsymbol{v}, \boldsymbol{h})}{P(\boldsymbol{v})} R(\boldsymbol{v}) - \left[\sum_{\boldsymbol{v},\boldsymbol{h}} R(\boldsymbol{v}) \right] \sum_{\boldsymbol{v},\boldsymbol{h}} v_i h_j P(\boldsymbol{v}, \boldsymbol{h}) \; , \tag{2.34}$$

which we can simplify

$$\frac{\partial KL(\boldsymbol{W}, \boldsymbol{a}, \boldsymbol{b})}{\partial w_{ij}} = \sum_{\boldsymbol{v},\boldsymbol{h}} v_i h_j P(\boldsymbol{h}|\boldsymbol{v}) R(\boldsymbol{v}) - \sum_{\boldsymbol{v},\boldsymbol{h}} v_i h_j P(\boldsymbol{v}, \boldsymbol{h}) \; , \tag{2.35}$$

where we require that

$$\frac{\partial \log Z}{\partial w_{ij}} = \sum_{\boldsymbol{v},\boldsymbol{h}} v_i h_j P(\boldsymbol{v}, \boldsymbol{h}) \; . \tag{2.36}$$

We then define the expectation, or correlation, values

$$\langle v_i h_j \rangle_{\text{data}} = P(\boldsymbol{h}|\boldsymbol{v}) R(\boldsymbol{v}) \tag{2.37}$$

and

$$\langle v_i h_j \rangle_{\text{model}} = P(\boldsymbol{v}, \boldsymbol{h}) \; . \tag{2.38}$$

This gives us the update rule

$$\Delta w_{ij} = -\eta \left(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}} \right) \; . \tag{2.39}$$

2.2.2 Restricted Boltzmann machine

Estimating $\langle v_i h_j \rangle_{\text{data}}$ and $\langle v_i h_j \rangle_{model}$ is done by Gibbs sampling, which is explained in a later chapter, but can be inefficient and take a long time to converge for complex models. Removing the weights between nodes within the same layer we can alleviate much of the computational cost of training. This type of Boltzmann machine is called a restricted Boltzmann machine, or RBM:

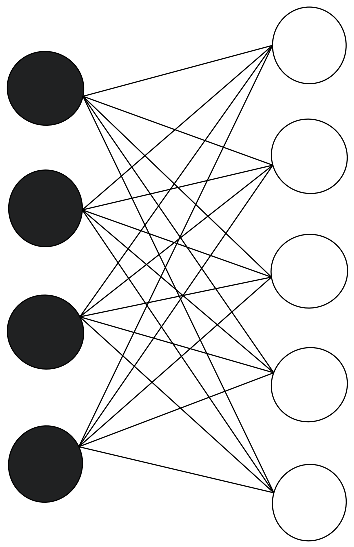


Figure 2.5: A restricted Boltzmann machine where there are no connections between nodes within the same layer. The white nodes are hidden while the black ones are the visible nodes.

Making the nodes independent of nodes in the same layer means we can write the conditional distributions as

$$P(\boldsymbol{v}|\boldsymbol{h}) = \prod_{i \in \boldsymbol{v}} P(v_i|\boldsymbol{h}) \ , \tag{2.40}$$

and

$$P(\boldsymbol{h}|\boldsymbol{v}) = \prod_{j \in \boldsymbol{h}} P(h_j|\boldsymbol{v}) \ . \tag{2.41}$$

In a RBM we first have a forward pass where we insert the input data into the visual layer, then we sample the hidden layer by the distribution:

$$p(h_j^{(0)} = 1|\boldsymbol{z}_v^{(0)}) = \sigma\left(\boldsymbol{z}_v^{(0)} \otimes \boldsymbol{W} + \boldsymbol{b}\right) \ , \tag{2.42}$$

where our activation function σ is the Sigmoid function 2.1. The index (0) indicate that it is the first pass-through the neural network. As the nodes are binary we then take a sample from $p(h_j^{(0)} = 1|\boldsymbol{z}_v^{(0)})$ as a Bernoulli distribution, which means each $z_{h,j}^{(0)}$ takes the value 1 with probability $h_j^{(0)}$. After the forward pass we have a backward pass where we sample from the hidden layer

$$p(v_j^{(1)} = 1|\boldsymbol{z}_h^{(0)}) = \sigma\left(\boldsymbol{z}_h^{(0)} \otimes \boldsymbol{W} + \boldsymbol{a}\right) , \tag{2.43}$$

where we then convert it to binary values as well. For a continuous valued output it is optional to let the last visual output to remain as a probability distribution. Estimating $\langle v_i h_j \rangle_{\text{data}}$ is done by sampling from $P(\boldsymbol{h}|\boldsymbol{v})$

$$\langle \boldsymbol{vh} \rangle_{\text{data}} = \boldsymbol{z}_v^{(0)} \otimes p(\boldsymbol{h}^{(0)}|\boldsymbol{z}_v^{(0)}) \tag{2.44}$$

,
while estimating $\langle v_i h_j \rangle_{\text{model}}$ requires that we let the model sufficiently affect the output. To do this we do Gibbs sampling through k iterations of the forward and backward passes. Then we have

$$\langle \boldsymbol{vh} \rangle_{\text{model}} = \boldsymbol{z}_v^{(k)} \otimes p(\boldsymbol{h}^{(k)}|\boldsymbol{z}_v^{(k)}) \tag{2.45}$$

.
And from 2.39 the change in weight becomes

$$\Delta \boldsymbol{W} = \eta \left[\boldsymbol{z}_v^{(0)} \otimes p(\boldsymbol{h}^{(0)}|\boldsymbol{z}_v^{(0)}) - \boldsymbol{z}_v^{(k)} \otimes p(\boldsymbol{h}^{(k)}|\boldsymbol{z}_v^{(k)})\right] . \tag{2.46}$$

2.2.3 Neural net quantum state

To apply the restricted Boltzmann machine on a quantum mechanical system, a way to represent the system's state is needed. As the visual layer the output of the machine, and can here be viewed as the collapsed state of the system after measurement, we use the marginal distribution of the visual layer, the probability distribution of the visual layer states given the state of the hidden layer, to represent the wavefunction of the system.

$$\Psi_{rbm}(\boldsymbol{v}) = \frac{1}{Z}\sum_{\boldsymbol{h}} \exp\{-E(\boldsymbol{v}, \boldsymbol{h})\} \tag{2.47}$$

$$= \frac{1}{Z} e^{-\sum_i^{N_v} \frac{(x_i - a_i)^2}{2}} \prod_j^{N_h} (1 + e^{b_j + \sum_i^{N_v} x_i w_{ij}}) , \tag{2.48}$$

This is most often referred to as a neural net quantum state, or abbreviated NQS.

2.2.4 Minimizing local energy

We want a restricted boltzmann machine to match the distribution of the ground state of a quantum mechanical systems. This poses a problem with the use of the change in weight, 2.46, as we do not have any data to train the model on. Instead we will use the variational principle with the fact that

$$E\left[\Psi_r bm\right] \geq E_0 , \tag{2.49}$$

where $\Psi_r bm$ is the machine state and E_0 is the ground state of the quantum system we want the machine state to match. Because of 2.49 we can variationly minimize the expected energy and be certain that it approaches the ground state. This poses another problem, though, because the machine state is not directly accessible, but one can take samples from the machine's distribution. So instead of using the machine state directly, one approximates it by a sufficient amount of samples, essentially taking repeated measurements of the system and constructing the wavefunction from the result. For N total number of samples, and M_k the number of samples that are the basis state $|b_k\rangle \in \boldsymbol{B}$, then we have the approximate machine state:

$$\Psi_{rbm} \approx \sqrt{\frac{M_0}{N}}\,|b_0\rangle + \sqrt{\frac{M_1}{N}}\,|b_1\rangle + \cdots + \sqrt{\frac{M_N}{N}}\,|b_N\rangle \ , \tag{2.50}$$

where we assume the wavefunction is real and positive definite. We use the samples to then approximate the energy by finding the expectation value of their local energy. The local energy is defined as:

$$E_L(s) = \frac{\langle s|\, H\, |\Psi_{rbm}\rangle}{\langle s|\Psi_{rbm}\rangle} \ , \tag{2.51}$$

where $\langle s|$ is a sample. The system energy can then be approximated

$$\langle E\rangle \approx \langle E_L\rangle = \frac{1}{N}\sum_{k=0}^{N} E_L(s_k) \ . \tag{2.52}$$

So minimizing the energy of the machine state can be done through minimizing the local energy. To derive the cost function, our gradient for the gradient decent calculations, we will take inspiration from this derivation [6] of unknown author. Starting of with the expected value of the hamiltonian

$$\langle H\rangle = \frac{\int \mathrm{d}\Psi(X)^* H\Psi(X)}{\int \mathrm{d}X\,\Psi^*(X)\Psi(X)} \ , \tag{2.53}$$

we then have the derivative by using the chain rule

$$\frac{\partial}{\partial\alpha}\langle H\rangle = \frac{\int \mathrm{d}X\,\Psi_\alpha^* H\Psi + \Psi^* H\Psi_\alpha}{\int \mathrm{d}X\,\Psi^*\Psi} - \frac{\left(\int \mathrm{d}X\,\Psi^* H\Psi\right)\left(\int \mathrm{d}X\,\Psi_\alpha\Psi + \Psi^*\Psi_\alpha\right)}{\left(\int \mathrm{d}X\,\Psi^*\Psi\right)^2} \ , \tag{2.54}$$

where Ψ_α is used to denote $\frac{\partial\Psi}{\partial\alpha}$. Now we expand the integrals of the numerator of the second term by $\Psi^*\Psi$ and we get

$$\frac{\partial}{\partial\alpha}\langle H\rangle = \frac{\int \mathrm{d}X\ \Psi_\alpha^* H\Psi + \Psi^* H\Psi_\alpha}{\int \mathrm{d}X\ \Psi^*\Psi} - \langle E_L\rangle\left\langle \frac{\partial}{\partial\alpha}\ln|\Psi|^2\right\rangle \ . \tag{2.55}$$

We then use the fact that the hamiltonian is Hermitian:

$$\int \mathrm{d}X\ \Psi^* H\Psi_\alpha = \int \mathrm{d}X\ \Psi_\alpha (H\Psi)^* \ , \tag{2.56}$$

and we get

$$\frac{\partial}{\partial\alpha}\langle H\rangle = \frac{\int \mathrm{d}X\ \Psi_\alpha^* H\Psi + \Psi_\alpha (H\Psi)^*}{\int \mathrm{d}X\ \Psi^*\Psi} - \langle E_L\rangle\left\langle \frac{\partial}{\partial\alpha}\ln|\Psi|^2\right\rangle . \tag{2.57}$$

then expanding the first terms numerator integrals by $\Psi^*\Psi$ we end up with

$$\frac{\partial}{\partial \alpha}\langle H \rangle = \left\langle \frac{\Psi^*_\alpha}{\Psi^*}\frac{H\Psi}{\Psi} + \frac{\Psi_\alpha}{\Psi}\left(\frac{H\Psi}{\Psi}\right)^*\right\rangle - \langle E_L\rangle \left\langle \frac{\partial}{\partial\alpha}\ln|\Psi|^2\right\rangle \tag{2.58}$$

$$= \left\langle \frac{\Psi^*_\alpha}{\Psi^*}E_L + \frac{\Psi_\alpha}{\Psi}E_L^*\right\rangle - \langle E_L\rangle \left\langle \frac{\partial}{\partial\alpha}\ln|\Psi|^2\right\rangle \ . \tag{2.59}$$

If we now assume that Ψ is real, and then E_L would also be real, we can shorten the gradient

$$\frac{\partial}{\partial\alpha}\langle H\rangle = 2\left(\langle E_L\frac{1}{\Psi}\frac{\partial\Psi}{\partial\alpha}\rangle - \langle E_L\rangle\langle\frac{1}{\Psi}\frac{\partial\Psi}{\partial\alpha}\rangle\right) \ , \tag{2.60}$$

where α then is our collection of weights and biases

2.2.5 Change in weights and biases

For practical use of the cost function 2.60 defined above, we need to derive the change in weights and biases. With the fact that $\frac{1}{\Psi}\frac{\partial \ln\Psi}{\partial\alpha} = \frac{\partial\ln\Psi}{\partial\alpha}$ together with

$$\ln\Psi(\boldsymbol{v}) = -\ln Z - \sum_i^{N_v}\frac{(v_i-a_i)}{2} + \sum_k^{N_h}\ln\left(1+\exp\left\{b_k+\sum_j^{N_v}v_jw_{jk}\right\}\right) \ , \tag{2.61}$$

we can then find that

$$\frac{\partial}{\partial a_i}\ln\Psi = v_i - a_i \tag{2.62}$$

$$\frac{\partial}{\partial b_j}\ln\Psi = \left(\exp\left\{-b_j - \sum_k^{N_v}v_kw_{kj}\right\}+1\right)^{-1} \tag{2.63}$$

$$\frac{\partial}{\partial w_{ij}}\ln\Psi = v_i\left(\exp\left\{-b_j - \sum_k^{N_v}v_kw_{kj}\right\}+1\right)^{-1} \ . \tag{2.64}$$

2.3 Monte Carlo Methods

Monte Carlo methods is a way to gain insight into a probability distribution by using random samples from said distribution. As an example problem one is to calculate the integral

$$I(h,P) = \int h(x)P(x)\,dx,$$

where $h(x)$ is an arbitrary function and $P(x)$ is a probability distribution. Then by taking random samples from $P(x)\to x_s$, one could estimate the integral by

$$I(h,P) \approx \frac{1}{N}\sum_{s=1}^{N}h(x_s).$$

Which is simple enough when one can sample from the target distribution, here $P(x)$, something that is not always the case.

2.3.1 Importance sampling

When the distribution $P(x)$ is unknown we can instead draw our sample from a proposed distribution $Q(x)$ and then use importance weights to correct it. Then continuing with the example above we have

$$I(h, P) = \int \frac{h(x)P(x)}{Q(x)} Q(x)\, dx.$$

And our approximate becomes

$$I(h, P) \approx \frac{1}{N} \sum_{s=1}^{N} \frac{h(x_s)P(x_s)}{Q(x_s)}.$$

This requires a that $Q(x)$ is somewhat close to that of $P(x)$, which is not necessarily easy to construct. It is therefore better to use a adaptive proposal distribution instead.

2.3.2 Metropolis-Hastings algorithm

The Metropolis-Hastings algorithm is a Markov chain Monte Carlo method and introduces feature adaptive proposal distributions. For a desired distribution $P(x)$, a Markov chain describes the probability of a series of events where we have the transition probability $P(x'|x)$ of transitioning from state x to x'. Adapting the our proposed distribution $Q(x)$ by using Markov chains we move towards a stationary distribution where we then are at a equilibrium:

$$P(x'|x)P(x) = P(x|x')P(x')\ , \tag{2.65}$$

where the transition from x to x' is equally likely both ways. We can rewrite this as

$$\frac{P(x'|x)}{P(x|x')} = \frac{P(x')}{P(x)}\ . \tag{2.66}$$

The Metropolis-Hastings method is to set up a proposal $Q(x'|x)$ and then use a acceptance distribution $A(x', x)$ to either accept or reject samples from the proposed distribution. We can then rewrite the transition probability as

$$P(x'|x) = Q(x'|x)A(x', x)\ . \tag{2.67}$$

And then equilibrium equation 2.66 can then be written as

$$\frac{A(x', x)}{A(x|x')} = \frac{P(x')Q(x|x')}{P(x)Q(x'|x)}\ , \tag{2.68}$$

where we define the acceptance ratio

$$A(x', x) = \min\left(1, \frac{P(x')Q(x|x')}{P(x)Q(x'|x)}\right)\ . \tag{2.69}$$

For each iteration i we have a the state x' taken at random from $Q(x'|x_i)$ which then is either accepted by $x_{i+1} = x'$ or rejected by $x_{i+1} = x_i$. The distribution of $\{x_i\}$ will then approach our desired distribution $P(x)$ after a sufficient number of iterations.

2.3.3 Gibbs Sampling

Gibbs sampling is a special use case where we accept every suggested state. It is then necessary to make the proposed distribution as close to the actual distribution as possible. How we accomplish this is covered in the implementation section, see **??**.

2.4 Controls and Errors

Seeing how accurate a model is can be use full both during training, to see how well it optimizes, and for a trained model. The simplest way to check the accuracy is to look at the difference from the true value of whatever the model was supposed to find. If we define the model's predicted answer as z_m and the true answer as z_t we have that the answer is:

$$Error = |z_t - z_m| \ , \tag{2.70}$$

where we take the absolute value as the sign of the error is not too interesting. For a set of model predictions $\boldsymbol{z_m}$ and the equivalent set of the true values $\boldsymbol{z_t}$, we might want to reduce the error set,

$$\mathbf{Error} = \boldsymbol{z}_t - \boldsymbol{z}_m \ ,$$

into one number to make it easier to compare models. A intuitive solution is to take the mean error, but often it is desirable to emphasize larger differences and for this there are several ways of approach. One approach is to take the mean of the squared error:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{z}_{t,i} - \boldsymbol{z}_{m,i})^2 \ . \tag{2.71}$$

But these error can only be calculated if we have the target solution $\boldsymbol{z_t}$. To compare models without the true solution we can look at the accuracy of our Monte Carlo samples. The ground state energy of a hamiltonian is supposed to have zero total variance in the local energy of the samples. For a discrete set of values $\boldsymbol{X} = \{x_1, x_2, \dots, x_n\}$ with probabilities $\boldsymbol{P} = \{p_1, p_2, \dots, p_n\}$, the variance is defined as:

$$Var[\boldsymbol{X}] = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 \ , \tag{2.72}$$

where μ is the mean:

$$\mu[\boldsymbol{X}] = \frac{1}{n}\sum_{i=1}^{n} x_i \ . \tag{2.73}$$

The local energy of a sample $|s\rangle \in \boldsymbol{S}$ is:

$$E_L(s) = \frac{\langle s| H |\psi_{rbm}\rangle}{\langle s|\psi_{rbm}\rangle} \ , \tag{2.74}$$

where $|\psi_{rbm}\rangle$ is the machine state. We can then insert the set of all the local energies,

$$\boldsymbol{E}_L = E_L(s_1), E_L(s_2), \dots, E_L(s_n) \ , \tag{2.75}$$

into the definition of variance, 2.72.

$$Var[\boldsymbol{E}_L] = \frac{1}{n}\sum_{i=1}^{n}\left(E_L(s_i) - \mu\right)^2 \ . \tag{2.76}$$

And we get

$$Var[\boldsymbol{E}_L] = \frac{1}{n}\sum_{i=1}^{n}\left(E_L(s_i) - \langle E_L\rangle\right)^2 \ , \tag{2.77}$$

which can be rewritten as

$$Var[\boldsymbol{E}_L] = \frac{1}{n}\sum_{i=1}^{n}\left(E_L(s_i)^2 - 2\langle E_L\rangle E_L(s_i) + \langle E_L\rangle^2\right) \ . \tag{2.78}$$

And we get

$$Var[\boldsymbol{E}_L] = \frac{1}{n}\sum_{i=1}^{n}[E_L(s_i)^2] - 2\langle E_L\rangle\frac{1}{n}\sum_{i=1}^{n}[E_L(s_i)] + \langle E_L\rangle^2 \tag{2.79}$$

$$Var[\boldsymbol{E}_L] = \langle E_L^2\rangle - \langle E_L\rangle^2 \tag{2.80}$$

If we take a look at the expectation values $\langle E_L\rangle$ and $\langle E_L^2\rangle$. We estimate the true expectation value of the energy as:

$$\langle E\rangle \approx \langle E_L\rangle = \frac{1}{n}\sum_{k=1}^{n}E_L(s_k) = \frac{1}{n}\sum_{k=1}^{n}\frac{\langle s_k|\, H\, |\psi_{rbm}\rangle}{\langle s_k|\psi_{rbm}\rangle} \ , \tag{2.81}$$

which for a correct trial wavefunction ψ_{rbm} we would have

$$\langle E_L\rangle = \frac{1}{n}\sum_{k=1}^{n}E\frac{\langle s_k|\psi_{rbm}\rangle}{\langle s_k|\psi_{rbm}\rangle} = \langle E\rangle \ . \tag{2.82}$$

And for $\langle E_L^2\rangle$ we get

$$\langle E_L^2\rangle = \frac{1}{n}\sum_{k=1}^{n}\frac{\langle s_k|\, H^2\, |\psi_{rbm}\rangle}{\langle s_k|\psi_{rbm}\rangle} = \frac{1}{n}\sum_{k=1}^{n}E^2\frac{\langle s_k|\psi_{rbm}\rangle}{\langle s_k|\psi_{rbm}\rangle} = \langle E\rangle^2 \ . \tag{2.83}$$

Inserting this into 2.80 we get

$$Var[\boldsymbol{E}_L] = \langle E\rangle^2 - \langle E\rangle^2 = 0 \ . \tag{2.84}$$

So for a machine state that is close to the true wavefunction we would get a low variance, and if it is perfectly aligned with the true wavefunction we would only need one sample to determine the energy.

Chapter 3

Quantum Mechanics

Quantum mechanics is the physics and mathematics describing how things behave at the smallest of scales. Certainty becomes no more and everything devolves into probabilistic happenstance. The systems we plan to solve are of quantum mechanical nature, so we will start with a small introduction to the field.

3.1 Basic Principles

3.1.1 Wavefunction and superposition

A quantum state is described with the help of a wavefunction, which for one dimension we write as

$$\psi(x) : \mathbb{R} \to \mathbb{C} \ .$$

In and of itself the wavefunction does not have a good physical interpretation, but the squared absolute value becomes a probability distribution, such that

$$P(x) = |\psi(x)|^2 \ , \tag{3.1}$$

meaning that $|\psi(x)|^2$ is the probability of finding the particle at position x. The wavefunction therefor needs to be normalized, such that

$$\int_{-\inf}^{\inf} \mathrm{d}x \psi^* \psi = 1 \ . \tag{3.2}$$

It then also means the particle is in a undetermined position before measurement. The quantum state is a combination of all the possible positions it can be in, weighted by $\psi(x)$, in intervals for a continuum of eigenstates $|\phi\rangle$

$$|\psi\rangle = \int \mathrm{d}x \psi(x) \, |\phi\rangle \ . \tag{3.3}$$

And for a discrete set of eigenstates

$$|\psi\rangle = \sum_i \psi(x_i) \, |\phi\rangle \ . \tag{3.4}$$

Such a state $|\psi\rangle$ is called a superposition.

3.1.2 Operators and the Schrödinger equation

Affecting the wavefunction is done mathematically through operators. An operator maps a state to another, transforming the wavefunction. For a generalized operator $\hat{O}$ and the state $|\psi\rangle$ we have

$$\hat{O}\,|\psi\rangle = |\psi'\rangle \ , \tag{3.5}$$

where $|\psi'\rangle$ is a new state. When an operator acts on a eigenstate:

$$\hat{O}\,|\phi\rangle = \varepsilon\,|\phi\rangle \ , \tag{3.6}$$

the eigenvalue ε is a quantity of what $\hat{O}$ represents. For this quantity to be something physically measurable, the operator needs to be hermitian

$$\hat{O} = \hat{O}^\dagger \ , \tag{3.7}$$

such that $\varepsilon \in \mathbb{R}$. An important observable operator is the hamiltonian, representing the energy of the system. The hamiltonian maps a state to its energy distribution, indicating its time evolution which is dictated by the Schrödinger equation:

$$i\hbar\frac{\partial}{\partial t}|\Psi(t)\rangle = \hat{H}|\Psi(t)\rangle \ , \tag{3.8}$$

where t is time. Eigenstates of the hamiltonian are the stable energy states of the system

$$H\,|\psi_n\rangle = E_n\,|\psi_n\rangle \ , \tag{3.9}$$

where E_n is the energy of the state.

3.1.3 Global and local energy

The global energy is the energy of the whole system, the possible are values of energy that can be measured. This is the energy eigenvalues of the hamiltonian:

$$E_n = \frac{\langle\psi_n|\,H\,|\psi_n\rangle}{\langle\psi_n|\psi_n\rangle} \ . \tag{3.10}$$

The local energy is, however, the energy associated with any particular basis state:

$$E_{\text{loc}}(\phi) = \frac{\langle\phi|\,H\,|\psi\rangle}{\langle\phi|\psi\rangle} \ , \tag{3.11}$$

where then ϕ is the basis state, a specific point in the space of the system, and ψ is the state of the system.

3.1.4 Pauli Exlusion Principle

The Pauli Exclusion Principle says that, within a quantum system, identical fermions are limited to one per quantum state. This is expressed mathematically with a anti-symmetric wavefunction:

$$\Psi(\dots, x_i, \dots, x_j, \dots) = -\Psi(\dots, x_j, \dots, x_i, \dots) \ , \tag{3.12}$$

where x_i and x_j is two particles being switched.

3.2 Measurement in Quantum Mechanics

Measurement in quantum mechanics is different from classical measurement in the way that the act of measuring a state affects the state itself. When a quantum state is measured the output will be a real, classical, value. As such, if we have a wavefunction $|\Psi\rangle$ of a particle in the superposition

$$|\Psi\rangle = \alpha_{\downarrow}\psi_{\downarrow} + \alpha_{\uparrow}\psi_{\uparrow} \ , \tag{3.13}$$

where $\psi_{\downarrow}$ is the spin down state and $\psi_{\uparrow}$ is the spin up state of the particle. A measurement, here indicated with an M, can yield either

$$M(|\Psi\rangle) = \psi_{\downarrow} \text{ or } M(|\Psi\rangle) = \psi_{\uparrow} \ .$$

But as the measurement is done the wavefunction 'collapses' into the measured state

$$M(|\Psi\rangle) = \psi_{\downarrow|\uparrow} \to |\Psi\rangle = \psi_{\downarrow|\uparrow}$$

Most often we are interested in the original superposition wavefunction and a measurement result does not tell us anything meaningful about it. To get a look at the wavefunction we need to take multiple measurements, where the system is put back into its original state, and estimate $\alpha_{\downarrow}$ and $\alpha_{\uparrow}$ with the averages

$$\alpha_{\downarrow} \approx \sqrt{\frac{N_{\downarrow}}{N}} \tag{3.14}$$

$$\alpha_{\uparrow} \approx \sqrt{\frac{N_{\uparrow}}{N}} \ , \tag{3.15}$$

where N is the total number of measurements and $N_{\downarrow|\uparrow}$ is the number of times the measurement yields the respective state.

3.3 Entanglement

Chapter 4

Many-Body Methods

4.1 Basic Principles

4.1.1 Slater Determinants

Given a system with 2 fermions. The first particle particle has quantum numbers i and spin value α, while the second particle has quantum numbers k and spin value β. Since both particles are indistinguishable form each other, we have no way of knowing which particle is where. As a result the wavefunction needs to reflect this by being indifferent to particle permutation. A naive product wavefunction of the two fermions would look like

$$|\psi\left(x_1, x_2\right)\rangle = \varphi_{i\alpha}(x_1)\varphi_{k\beta}(x_2) \ . \tag{4.1}$$

If we try to switch the positions of x_1 and x_2 we get that

$$|\psi\left(x_2, x_1\right)\rangle = \varphi_{i\alpha}(x_2)\varphi_{k\beta}(x_1) \ , \tag{4.2}$$

which fails the anti-symmetric requirement for fermionic wavefunctions, which makes them distinguishable from each other. Instead we would want a wave function accounts for both scenarios in the first place

$$|\psi\left(\mathbf{x}_1, \mathbf{x}_2\right)\rangle = \frac{1}{\sqrt{2}}\left(\varphi_{i\alpha}(x_1)\varphi_{k\beta}(x_2) - \varphi_{i\alpha}(x_2)\varphi_{k\beta}(x_1)\right) \ , \tag{4.3}$$

where the wavefunction is a normalized combination of the two product wavefunctions. Here we have the antisymmetry

$$|\psi\left(\mathbf{x}_1, \mathbf{x}_2\right)\rangle = -\left|\psi\left(x_2, x_1\right)\right\rangle \ . \tag{4.4}$$

Which can be further generalized for a system with N fermions.

$$\psi(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \varphi_1(\mathbf{x}_1) & \varphi_2(\mathbf{x}_1) & \cdots & \varphi_N(\mathbf{x}_1) \\ \varphi_1(\mathbf{x}_2) & \varphi_2(\mathbf{x}_2) & \cdots & \varphi_N(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_1(\mathbf{x}_N) & \varphi_2(\mathbf{x}_N) & \cdots & \varphi_N(\mathbf{x}_N) \end{vmatrix} \ . \tag{4.5}$$

where it gets its name from.

4.2 Second quantization

4.2.1 Creation and annihilation operators

Starting with the vacuum state $|0\rangle$. The creation operator creates a single-particle state α_i:

$$\hat{a}^{\dagger}_{\alpha_1} |0\rangle = |\alpha_1\rangle \ , \tag{4.6}$$

which can be extended for more particles

$$\hat{a}^{\dagger}_{\alpha_1}\hat{a}^{\dagger}_{\alpha_2}\dots\hat{a}^{\dagger}_{\alpha_n} |0\rangle = |\alpha_1\alpha_2\dots\alpha_n\rangle \ . \tag{4.7}$$

The annihilation operator destroys the particle

$$\hat{a}_{\alpha_1} |\alpha_1\rangle = |0\rangle \ . \tag{4.8}$$

They are hermitian conjugate, which means

$$\hat{a}_{\alpha_i} = \left(\hat{a}^{\dagger}_{\alpha_i}\right)^{\dagger} \ . \tag{4.9}$$

Furthermore, the Pauli principle says we cannot have two particles in the same state, which gives us

$$\hat{a}^{\dagger}_{\alpha_i}\hat{a}^{\dagger}_{\alpha_i} = 0 \ . \tag{4.10}$$

Similarly we cannot annihilate a particle that does not exist in the first place

$$\alpha \neq \{\alpha_i\}$$

$$\hat{a}_{\alpha} |\alpha_1\alpha_2\dots\alpha_n\rangle = 0 \ . \tag{4.11}$$

For fermions the state wavefunction is anti symmetric, therefor switching position of two particles yields a factor of -1, and we get that

$$\hat{a}^{\dagger}_{\alpha_i}\hat{a}^{\dagger}_{\alpha_k} = -\hat{a}^{\dagger}_{\alpha_k}\hat{a}^{\dagger}_{\alpha_i} \ .$$

From this we have the commutation relations

$$\left\{\hat{a}^{\dagger}_{\alpha}\hat{a}^{\dagger}_{\beta}\right\} = 0 \tag{4.12}$$

$$\left\{\hat{a}_{\alpha}\hat{a}_{\beta}\right\} = 0 \tag{4.13}$$

$$\left\{\hat{a}^{\dagger}_{\alpha}\hat{a}_{\beta}\right\} = \delta_{\alpha\beta} \tag{4.14}$$

4.2.2 Operators in second quantization

`https://github.com/ManyBodyPhysics/FYS4480/blob/master/doc/pub/week36/ipynb/week36.ipynb`

For a one-body operator in coordinate space we have

$$\hat{H}_0 = \sum_i \hat{h}_0(x_i) \ , \tag{4.15}$$

and using the anti-symmetric Slater determinant from 4.5, which we can write as:

$$\Phi(x_1, x_2, \dots, x_n, \alpha_1, \alpha_2, \dots, \alpha_n) = \frac{1}{\sqrt{n!}} \sum_p (-1)^p \hat{P} \psi_{\alpha_1}(x_1)\psi_{\alpha_2}(x_2) \dots \psi_{\alpha_n}(x_n) \ , \tag{4.16}$$

we can define

$$\hat{h}_0(x_i)\psi_{\alpha_i}(x_i) = \sum_{\alpha'_k} \psi_{\alpha'_k}(x_i) \langle \alpha'_k | \hat{h}_0 | \alpha_k \rangle \ . \tag{4.17}$$

So for each one-particle function in the Slater determinant we gain a contribution to $\hat{H}_0 \left|\Phi\right\rangle$. Our Slater determinant can be written in second quantization as simply

$$|\Phi\rangle = |\alpha_1, \alpha_2, \dots, \alpha_n\rangle \ . \tag{4.18}$$

With this we have

$$\begin{aligned}
\hat{H}_0 |\alpha_1, \alpha_2, \dots, \alpha_n\rangle = &\sum_{\alpha'_1} \langle \alpha'_1 | \hat{h}_0 | \alpha_1 \rangle |\alpha'_1 \alpha_2 \dots \alpha_n\rangle \\
&+ \sum_{\alpha'_2} \langle \alpha'_2 | \hat{h}_0 | \alpha_2 \rangle |\alpha_1 \alpha'_2 \dots \alpha_n\rangle \\
&+ \dots & (4.19) \\
&+ \sum_{\alpha'_n} \langle \alpha'_n | \hat{h}_0 | \alpha_n \rangle |\alpha_1 \alpha_2 \dots \alpha'_n\rangle \ , & (4.20)
\end{aligned}$$

where we go over each possible permutation of each particle. Furthermore, if we use the fact that we can write

$$|\alpha_1 \alpha_2 \dots \alpha'_k \dots \alpha_n\rangle = a^\dagger_{\alpha'_k} a_{\alpha_k} |\alpha_1 \alpha_2 \dots \alpha_k \dots \alpha_n\rangle \ , \tag{4.21}$$

we can then shorten the expression to

$$\hat{H}_0 = \sum_{\alpha\beta} \langle \alpha | \hat{h}_0 | \beta \rangle a^\dagger_\alpha a_\beta \ . \tag{4.22}$$

As a generalized one-body operator, that preserves the number of particles, in second quantization. A two-body operator in coordinate space can be written as

$$\hat{H}_I = \sum_{i<j} V(x_i, x_j) \ , \tag{4.23}$$

where V is some interaction force between two particles. Using our Slater determinant again we have that

$$V(x_i, x_j)\psi_{\alpha_k}(x_i)\psi_{\alpha_l}(x_j) = \sum_{\alpha'_k \alpha'_l} \psi'_{\alpha_k}(x_i)\psi'_{\alpha_l}(x_j) \langle \alpha'_k \alpha'_l | \hat{v} | \alpha_k \alpha_l \rangle \tag{4.24}$$

Once again summing over all possible permutations of each combination of two-particle pairs, we get

$$\begin{aligned} H_I|\alpha_1\alpha_2\dots\alpha_n\rangle = &\sum_{\alpha_1',\alpha_2'}\langle\alpha_1'\alpha_2'|\hat{v}|\alpha_1\alpha_2\rangle|\alpha_1'\alpha_2'\dots\alpha_n\rangle \\ &+\dots \\ &+\sum_{\alpha_1',\alpha_n'}\langle\alpha_1'\alpha_n'|\hat{v}|\alpha_1\alpha_n\rangle|\alpha_1'\alpha_2\dots\alpha_n'\rangle \\ &+\dots \\ &+\sum_{\alpha_2',\alpha_n'}\langle\alpha_2'\alpha_n'|\hat{v}|\alpha_2\alpha_n\rangle|\alpha_1\alpha_2'\dots\alpha_n'\rangle \\ &+\dots \\ &+\sum_{\alpha_{n-1}',\alpha_n'}\langle\alpha_{n-1}'\alpha_n'|\hat{v}|\alpha_{n-1}\alpha_n\rangle|\alpha_1\alpha_{n-1}'\dots\alpha_n'\rangle\ . \end{aligned} \tag{4.25}$$

Then, using the fact that

$$a^\dagger_{\alpha_k'}a^\dagger_{\alpha_l'}a_{\alpha_l}a_{\alpha_k}|\alpha_1\alpha_2\dots\alpha_k\dots\alpha_l\dots\alpha_n\rangle == |\alpha_1\alpha_2\dots\alpha_k'\dots\alpha_l'\dots\alpha_n\rangle\ , \tag{4.26}$$

we end up with

$$\begin{aligned} H_I|\alpha_1\alpha_2\dots\alpha_n\rangle &= \sum_{\alpha_1',\alpha_2'}\langle\alpha_1'\alpha_2'|\hat{v}|\alpha_1\alpha_2\rangle a^\dagger_{\alpha_1'}a^\dagger_{\alpha_2'}a_{\alpha_2}a_{\alpha_1}|\alpha_1\alpha_2\dots\alpha_n\rangle \\ &= \sum_{\alpha,\beta,\gamma,\delta}{}' \langle\alpha\beta|\hat{v}|\gamma\delta\rangle a^\dagger_\alpha a^\dagger_\beta a_\delta a_\gamma|\alpha_1\alpha_2\dots\alpha_n\rangle\ , \end{aligned} \tag{4.27}$$

where $\alpha,\ \beta$ are single-particle states while $\gamma,\ \delta$ are pairs of single-particle states. We can remove this distinction with the fact that

$$\langle\alpha\beta|\hat{v}|\gamma\delta\rangle = \langle\beta\alpha|\hat{v}|\delta\gamma\rangle\ . \tag{4.28}$$

And we end up with

$$\hat{H}_I = \frac{1}{2}\sum_{\alpha\beta\gamma\delta}\langle\alpha\beta|\hat{v}|\gamma\delta\rangle a^\dagger_\alpha a^\dagger_\beta a_\delta a_\gamma\ , \tag{4.29}$$

where all indices are summed over single-particle states only.

4.2.3 Hamiltonian in second quantization

The way we calculate the local energy in practicality is connected to the hamiltonian's effect on a state. As such we will go over a concrete example to make our implementation exelanation, see **??** simpler to understand. For an example hamiltonian

$$H = \sum_{p,\sigma}\varepsilon_{p,\sigma}\hat{a}^\dagger_{p,\sigma}\hat{a}_{p,\sigma}\ , \tag{4.30}$$

where p is the number of fermions, here 2, and σ is the up or down spin. We can then apply the hamiltonian to each of the basis states.

$$H|00\rangle = \left(\varepsilon_{0,+}\hat{a}^{\dagger}_{0,+}\hat{a}_{0,+} + \varepsilon_{0,-}\hat{a}^{\dagger}_{0,-}\hat{a}_{0,-} + \varepsilon_{1,+}\hat{a}^{\dagger}_{1,+}\hat{a}_{1,+}\varepsilon_{1,-}\hat{a}^{\dagger}_{1,-}\hat{a}_{1,-}\right)|00\rangle \tag{4.31}$$

$$H|01\rangle = \left(\varepsilon_{0,+}\hat{a}^{\dagger}_{0,+}\hat{a}_{0,+} + \varepsilon_{0,-}\hat{a}^{\dagger}_{0,-}\hat{a}_{0,-} + \varepsilon_{1,+}\hat{a}^{\dagger}_{1,+}\hat{a}_{1,+}\varepsilon_{1,-}\hat{a}^{\dagger}_{1,-}\hat{a}_{1,-}\right)|01\rangle \tag{4.32}$$

$$H|10\rangle = \left(\varepsilon_{0,+}\hat{a}^{\dagger}_{0,+}\hat{a}_{0,+} + \varepsilon_{0,-}\hat{a}^{\dagger}_{0,-}\hat{a}_{0,-} + \varepsilon_{1,+}\hat{a}^{\dagger}_{1,+}\hat{a}_{1,+}\varepsilon_{1,-}\hat{a}^{\dagger}_{1,-}\hat{a}_{1,-}\right)|10\rangle \tag{4.33}$$

$$H|11\rangle = \left(\varepsilon_{0,+}\hat{a}^{\dagger}_{0,+}\hat{a}_{0,+} + \varepsilon_{0,-}\hat{a}^{\dagger}_{0,-}\hat{a}_{0,-} + \varepsilon_{1,+}\hat{a}^{\dagger}_{1,+}\hat{a}_{1,+}\varepsilon_{1,-}\hat{a}^{\dagger}_{1,-}\hat{a}_{1,-}\right)|11\rangle \ . \tag{4.34}$$

The local energy of a state $|s\rangle$ is:

$$E_{local} = \frac{\langle s| H |\psi\rangle}{\langle s|\psi\rangle} \ , \tag{4.35}$$

4.3 Methods and Algorithms

4.3.1 Full Configuration Interaction Theory

We want to show that diagonalizing the Hamiltonian matrix yields the energies of the different levels. Starting of with the time-independent Schrodinger equation

$$H|\psi\rangle = E|\psi\rangle \ . \tag{4.36}$$

Assuming we can express $|\psi\rangle$ in terms of an orthonormal basis $\{|n\rangle\}$:

$$|\psi\rangle = \sum_n c_n |n\rangle \ , \tag{4.37}$$

we can then insert this into the time-independent Schrodinger equation and get

$$\sum_n c_n H|n\rangle = E\sum_n c_n |n\rangle \ . \tag{4.38}$$

Since the basis $\{|n\rangle\}$ is orthonormal we that the identity matrix can be expressed as

$$I = \sum_k |k\rangle\langle k| \ ,$$

where $|k\rangle \in \{|n\rangle\}$. Inserting this into equation 4.38 we get that

$$\sum_k\sum_n \langle k| H |n\rangle c_n |k\rangle = E\sum_n c_n |n\rangle \ . \tag{4.39}$$

Multiplying from the left side by $|m\rangle \in \{|n\rangle\}$ we get

$$\sum_n \langle k| H |n\rangle c_n \langle m|k\rangle = E \sum_n c_n \langle m|n\rangle \ , \tag{4.40}$$

where

$$\langle m|k\rangle = \delta_{mk} \langle m|n\rangle \qquad\qquad = \delta_{mn} \ ,$$

such that

$$\sum_n \langle m| H |n\rangle c_n = E c_m \ . \tag{4.41}$$

$\langle m| H |n\rangle$ is the element m, n of the Hamiltonian matrix, so 4.41 can be expressed as

$$\sum_n H_{mn} c_n = E c_m \ . \tag{4.42}$$

And in matrix form this becomes

$$HC = EC \ , \tag{4.43}$$

where

$$C = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} \ .$$

Diagonalizing H will then give us the eigenvalues E as the energy of the different levels of the system.

4.4 Neural network quantum states

To use a neural network to solve a quantum mechanical system we need a way to represent the quantum state with the neural network. The solution is using neural network quantum states, abbreviated NQS, which was introduced by G. Carlo and M. Troyer in 2017 [7]. A quantum state $|\Psi\rangle$ can be expressed by a neural network as

$$\langle s_1 \dots s_N | \Psi; \boldsymbol{W}, \boldsymbol{B}\rangle = F(s_1 \dots s_N; \boldsymbol{W}, \boldsymbol{B}) \ , \tag{4.44}$$

where we have N input variables, with accordance to the number of degrees of freedom the state $|\Psi\rangle$ has, as well as weights $\boldsymbol{W}$ and biases $\boldsymbol{B}$ of the neural network F.

Chapter 5

Models

5.1 The Lipkin-Meshkow-Glick model

`https://arxiv.org/pdf/1805.12442.pdf`
One of problem that many-body physics encounter in complex, real world, system is the fact that the many-body Shrödinger equation isn't exactly solvable. This may require either to approximat the Shrodinger equation or limit the number of particles in a system. Therefor it is more common to test many-body methods on simplified models where the exact solution is available without approximation. One of these models is the Lipkin-Meshkow-Glick model, abbreviated LMG in this thesis, first built in the 1960's [8].

5.1.1 The model system

The idea behind the LMG model is to have two energy levels separated by an energy-value ε, one just below the Fermi level and one just above. In our case we fill up the base energy level with any number of particles, which then can be excited up to the second energy level. 5.1 shows a visualized example with two particles:

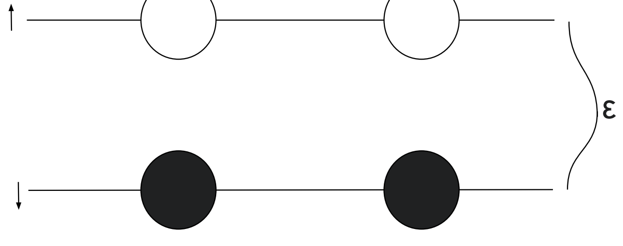


Figure 5.1: The LMG model with two particles and two holes where the particles can move two. The two levels are separated by a constant ε.

For a N-fermion system the levels are N-fold degenerate, represented by the different positions the particles can be in in 5.1, with two characteristic quantum numbers associated with each particle. The σ quantum number is assumed to be -1 in the lower level and $+1$ in the higher level, which can be seen as particle spin. We will use p to denote the degenerate state in which a particle resides in. So σ denotes which level the particle is on and p denotes where in that level it resides. The Hamiltonian proposed by Lipkin, Glick and Meshkow is as follows:

$$H = \sum_{p\sigma} \left(\frac{1}{2}\sigma\varepsilon\right) \hat{a}^\dagger_{p\sigma}\hat{a}_{p\sigma} - \frac{V}{2}\sum_{pp'\sigma} \hat{a}^\dagger_{p\sigma}\hat{a}^\dagger_{p'\sigma}\hat{a}_{p'-\sigma}\hat{a}_{p-\sigma} - \frac{W}{2}\sum_{pp'\sigma} \hat{a}^\dagger_{p\sigma}\hat{a}^\dagger_{p'-\sigma}\hat{a}_{p'\sigma}\hat{a}_{p-\sigma}\ , \tag{5.1}$$

where the operators $\hat{a}^\dagger_{p\sigma}$ creates and $\hat{a}_{p\sigma}$ destroys a particle in the energy level associated with σ and in the p position within that level. As an example, if we start out with two particles in the lower level:

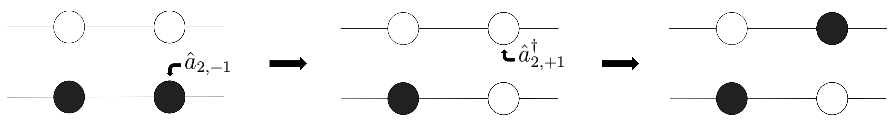


Figure 5.2: Two particles in the $\sigma = -1$ level filling out the $p = 1$ and $p = 2$ state of the LMG model. The particle at $p = 2$ and $\sigma = -1$ is destroyed and then a particle is created at $p = 2$, $\sigma = +1$.

Here we excite one of the particles in the lower layer up to the $\sigma = +1$ layer by destroying and then creating a particle. The Hamiltonian has three parts. Firstly we have the single-particle energy of each particle in the system. Then the a term that moves pairs of particle from one level to another, with an interaction strength proportional to V. Lastly we have a term that splits pairs of particles, with an interaction strength proportional to W, as in the example above.
In the $N = 2$ fermions example above we have in total four possible states, since two fermions cannot have the same quantum numbers. Essentially we want to look at each of these states and determine their contribution to the ground state energy by how likely the system is to be measured in that state. In our case we will simplify the Hamiltonian at 5.1 by defining $W = 0$ such that we are left with:

$$H = \sum_{p\sigma} \left(\frac{1}{2}\sigma\varepsilon\right) \hat{a}^\dagger_{p\sigma}\hat{a}_{p\sigma} - \frac{V}{2}\sum_{pp'\sigma} \hat{a}^\dagger_{p\sigma}\hat{a}^\dagger_{p'\sigma}\hat{a}_{p'-\sigma}\hat{a}_{p-\sigma} \tag{5.2}$$

5.1.2 Rewriting the Hamiltonian

An advantage of the LMG model is the fact that the two-body interaction does not change the value of p. Together with the two-valued σ, each particle can only exist in two possible states. This suggests to use the quasi-spin operators to rewrite the Hamiltonian. These quasi-spin operators are defined as follows:

$$
\begin{aligned}
\hat{J}_+ &= \sum_p^N \hat{a}^\dagger_{p,+}\hat{a}_{p,-} \\
\hat{J}_- &= \sum_p^N \hat{a}^\dagger_{p,-}\hat{a}_{p,+} \\
\hat{J}_z &= \frac{1}{2}\sum_p^N \left(\hat{a}^\dagger_{p,+}\hat{a}_{p,+} - \hat{a}^\dagger_{p,-}\hat{a}_{p,-}\right)
\end{aligned}
\tag{5.3}
$$

Where the $+, -$ indicates the quantum number $\sigma = \{+1, -1\}$ as associated with spin up and spin down respectively. The $\hat{J}_+$ accounts for the energy of a particle being excited up a level, the $\hat{J}_-$ accounting for a particle falling down a level. While the $\hat{J}_z$ takes into account the difference in single-particle energy of the two energy levels. All for each degenerate states p. Using these quasi-spin operators we can rewrite the Hamiltonian as:

$$
H = \varepsilon\hat{J}_z - \frac{V}{2}\left(\hat{J}_+\hat{J}_+ + \hat{J}_-\hat{J}_-\right) - \frac{W}{2}\left(\hat{J}_+\hat{J}_- + \hat{J}_-\hat{J}_+\right) \tag{5.4}
$$

5.1.3 Analytical Solution

For an exact solution of the LMG model one can use the full configuration interaction theory described in section 4.3.1. However, for a given total spin J the spin of a state can overlap with systems with fewer particles. For example, a system with $J = 2$ and another with $J = 1$ can both have the same spin projection $J_z = -1$ as seen below.

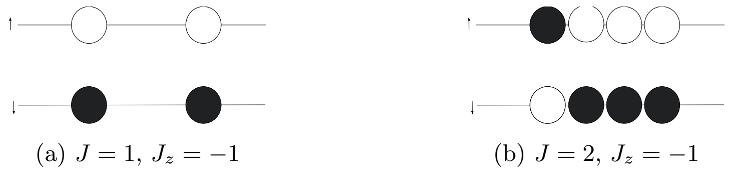


Figure 5.3: Two systems with different total spin but both are in a state where $J_z = -1$.

Therefor a more complete Hamiltonian would differentiate between these states. For a four-particle system we would then have

$$
H_4 = \begin{bmatrix} H_{J=2} & & 0 \\ & H_{J=1} & \\ 0 & & H_{J=0} \end{bmatrix} . \tag{5.5}
$$

Since the Hamiltonian commutes with J^2,

$$
\left[H, J^2\right] = 0 , \tag{5.6}
$$

J is a good quantum number and all other elements in the H_4 Hamiltonian becomes zero. As a diagonal block matrix we can then instead diagonalize the J-specific Hamiltonians separately. For $J = 2$ we have:

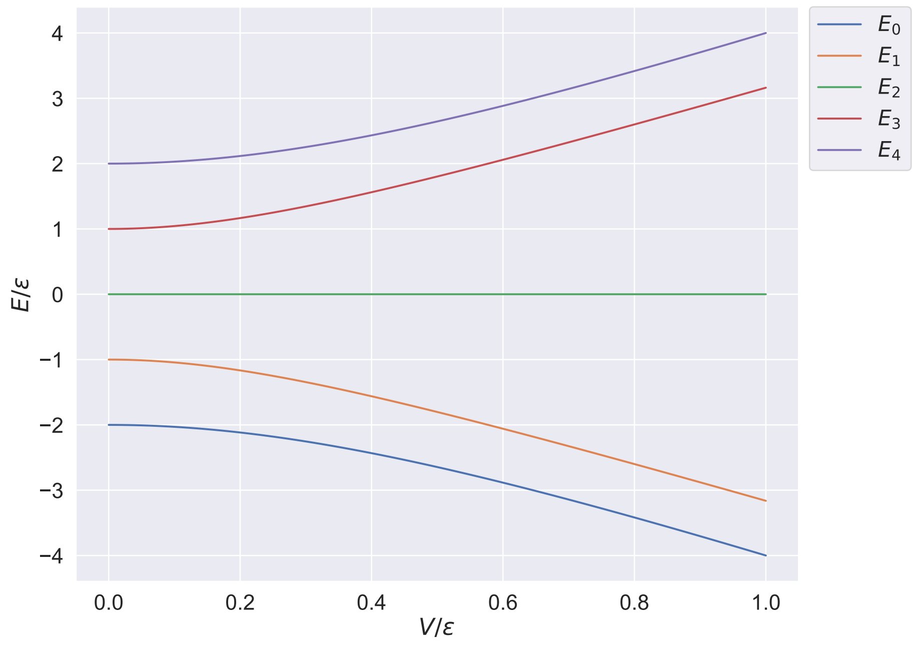


Figure 5.4: The analytical solution for the LMG model with total spin $J = 2$, $\varepsilon = 1$ and $W = 0$.

And for $J = 1$:

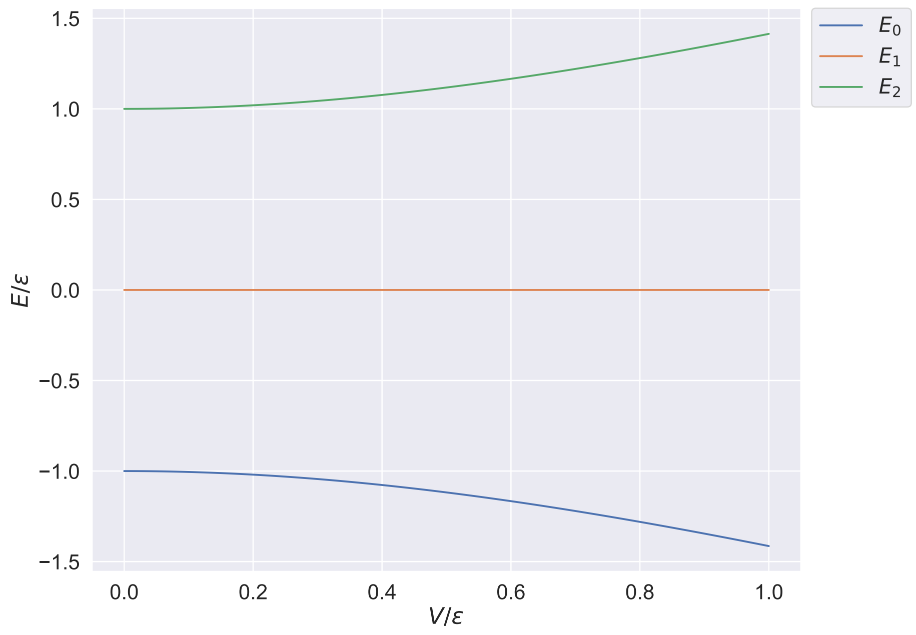


Figure 5.5: The analytical solution for the LMG model with total spin $J = 1$, $\varepsilon = 1$ and $W = 0$.

5.2 The Ising model

The Ising model consists of a discrete set of particles where nearest neighbors interact. We will look at the one dimensional Ising model, with the particles arranged in a line as follows:
image of particles in a line
Where the state of the whole system is denoted $\boldsymbol{\sigma}$ and the individual states are denoted σ. Each particle can be in either spin up or spin down state which gives their respective interactive factor

$$\sigma \in \{+1, -1\} . \tag{5.7}$$

We define the interaction strength between particles as a constant J. We sum each neighbor pair's contribution and get the total energy for that specific configuration, which then is the local energy of that configuration:

$$H(\boldsymbol{\sigma}) = J \sum_{\langle i,j \rangle} \sigma_i \sigma_j . \tag{5.8}$$

5.3 The Heisenberg model

The Heisenberg model is similar to the Ising model, only with a slight variation. Instead of considering nearest neighbors in both directions, the dipoles interact only in one direction. The local energy of a configuration is defined

$$H(\boldsymbol{\sigma}) = J \sum_i \sigma_i \sigma_{i+1} + h \sum_i \sigma_i , \tag{5.9}$$

where we have the coupling constant J and single particle energy h.

5.4 The Pairing model

Part II

Implementation

Chapter 6

Libraries and hardware

6.0.1 CUDA and torch

When we have large amounts of data and do the same unconditional operations on them, we can reduce the time it takes for the RBM to find the ground state by parallelizing parts of the calculations. The GPU, graphical processing unit, is designed for such tasks. To use the GPU we need a interface that can communicate with it. There are several options for GPU interfaces, but for NVIDIA[9] GPU's there is the CUDA[10] interface. For simplicity we will use a python library: PyTorch[11]. PyTorch is a machine learning library which has CUDA integration possibilities. To enable the use of the GPU we first initialize the GPU as a device:

```python
import torch

device = torch.device('cuda')
```

Then when we create a tensor we do so with the GPU as its device:

```python
tensor = torch.tensor(elements, dtype=torch.float64, device=device)
```

where `elements` is the whole object we want on the GPU. Operations on `tensor` will then be run on the processors in the GPU if possible.

Chapter 7

Restricted Boltzmann Machine

7.0.1 The base structure

We have implemented the restricted Boltzmann machine in separated parts. The parts are connected to each other through the main file, `RBMmodules\\main.py`. The main `run` function takes in the options for that specific run, the model (the quantum system) in question and the machine setup as arguments. These arguments are sent to the different parts of the RBM: the initializer, the local energy function and finally the solver, which outputs the predicted ground state energy. In this section we will go over the machine model initialization briefly, while going more in depth into the local energy function and the solver, as these are less straight forward.

7.0.2 Model initialization

The model initialization is just the how the RBM's structure is desired, but there are some caveats. The RBM really only consists of a few arrays: the visua layer bias, the hidden layer bias and the weights. Defining the size of these is for the most part all that is needed to 'create' an RBM. But, for the use of CUDA through PyTorch we have to define where these arrays are stored, and we would want flexibility as well, so that the code is useable with and without a CUDA graphics card. We do this as such:

```
def set_up_model(visual_n, hidden_n, precision, device):
    visual_bias = torch.zeros(visual_n, dtype=precision, device=device)
    hidden_bias = torch.zeros(hidden_n, dtype=precision, device=device)
    W = W_scale*torch.rand(visual_n, hidden_n, dtype=precision, device=device)

    model = create_model_dataclass(precision, device)

    init_model = model(
        visual_bias = visual_bias,
        hidden_bias = hidden_bias,
        weights = W,
        device = device,
        precision = precision)

    return init_model
```

The supporting functions have been omitted to save space. The `create_model_dataclass(...)` is only for the possibility of changing precision with ease, even though everything in the thesis is done in 64-bit floating point precision. At the top of the function is the true creation of the RBM and the rest is for packaging it into one unit together with their data types.

7.0.3 The solver

Solving a system here means training the RBM some length of time or until a certain threshold is met, and get the machines last guess at the system's ground state energy. To train the machine we take first need to implement the sampling of each layer.

```
def sample_visual(hidden, visual_bias, W):
    given = S(hidden@W.T + visual_bias)
    binary = torch.bernoulli(given)
    return given, binary
```

This is done in accordance to the what has been derived mathematically, where the @ symbol indicates matrix multiplication. The `S(...)` function is the sigmoid function. The given probability distribution of the layer is then sent through a bernoulli function, which returns a binary array based on the given array's distribution. Both versions are returned, so it is only minor changes that is required to go from continuous to binary visual layer. The equivalent `sampel_hidden` has only the two bias arrays switched in the `given` definition. The next step is to find the cost function derivative based on the biases and weights. A estimation of the machine state is needed, and this is here done with two different methods. First of we have the Metropolis-Hastings algorithm which accept states conditionally, then secondly we have Gibbs sampling, which accepts all states. The use of a condition makes Metropolis-Hastings unable to be vectorized, and therefor severely slower than Gibbs sampling. Both methods are combined into the same function, here we have left some variable definitions out for ease of reading or lack of relevance.

```
def MonteCarlo(cycles, local_energy_func, gibbs_k, model):
    hidden = torch.bernoulli(torch.rand(
        cycles,
        hidden_n,
        dtype=model.precision,
        device=model.device
    ))
    _, samples = p_visual(hidden)
    hidden, dist_s = p_sampler(samples)
    dPsidvb = (dist_s - vb)
    dPsidhb = 1/(torch.exp(-hb-dist_s@W)+ 1)
    dPsidW = dist_s[:, :, None]*dPsidhb[:, None, :]
    E_local = local_energy_func(dist_s)
    E_mean = torch.mean(E_local)
    E_diff = E_local - E_mean
    DeltaVB = torch.mean(E_diff[:, None]*dPsidvb, axis=0)
    DeltaHB = torch.mean(E_diff[:, None]*dPsidhb, axis=0)
    DeltaW = torch.mean(E_diff[:, None, None]*dPsidW, axis=0)
    dE = torch.mean(E_local - E_mean)
    return DeltaVB, DeltaHB, DeltaW
```

The functions prefixed with a "p" are partial versions, to not needing to pass already defined argument. In the `p_sampler(...)` function we split between either of the two sampling methods. Looking at Metropolis-Hastings first:

```
def metropolis_hastings(input, visual_bias, hidden_bias, W):

    size = input.size(dim=0)
    given_h, hidden = sample_hidden(input, hidden_bias, W)
    rand_nums = torch.rand(size)
    previous = input[0]
    E_prev = net_Energy(previous, visual_bias, hidden, hidden_bias, W)
    for i in range(input.size(dim=0)):
        E_current = net_Energy(input[i], visual_bias, hidden, hidden_bias, W)
        acc_ratio = E_current/E_prev
        if acc_ratio <= rand_nums[i]:
            input[i] = previous
        previous = input[i]
        E_prev = E_current

    return input
```

We have a simple loop that generates a random number and checks it against the acceptance rate by the machine energy ratio. The Gibbs sampling is done by back and forth sampling of the layers.

```
def gibbs_update(input, visual_bias, hidden_bias, W, k):
    given_h, hidden = sample_hidden(input, hidden_bias, W)
    given_v, visual_k = sample_visual(hidden, visual_bias, W)
    for _ in range(k):
        given_h, hidden_k = sample_hidden(given_v, hidden_bias, W)
        given_v, visual_k = sample_visual(given_h, visual_bias, W)

    return hidden_k, visual_k
```

Where the number of cycles `k`, often referred to as `gibbs_k` in the rest of the codebase, is predefined by the user. The two first lines does one Gibbs cycle, which means the total number of cycles is $k + 1$.

Then we have taken to use of all the derived equations for the derivative of the wave function and cost function based on the model variables. Something to note here is the use of automatic casting functionality PyTorch has, done by expanding the correct dimension in the arrays one does arithmetic with. Because of its avid use and the confusing syntax, we will explain a few of the above lines thoroughly.

```
dPsidW = dist_s[:, :, None]*dPsidhb[:, None, :]
```

Both `dist_s` and `dPsidhb` are two-dimensional arrays. Inside the square brackets the `:` sign means that all elements of that dimension is to be included. The `None` creates an empty dimension to the array in the specified depth. For example `dist_s` has `None` in the lowest depth, at the scalar level. This replaces the scalar element with a 1×1 array with the scalar element in it. Like so:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \rightarrow \begin{bmatrix} [a_{11}] & [a_{12}] \\ [a_{21}] & [a_{22}] \end{bmatrix}$$

for a imagined 2×2 `dist_s` with dummy elements. With `dPsidhb[:, None, :]` we have the rows being wrapped up in the same way:

$$\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \rightarrow \begin{bmatrix} [b_{11} & b_{12}] \\ [b_{21} & b_{22}] \end{bmatrix}$$

though as a extra layer above the vector of the row of elements stored in memory. The broadcasting happens when a missing or empty dimension comes up against another missing or empty dimension. PyTorch then makes duplicates of one dimension to fill the missing pieces, and this can be used to get the end result we want. A better example for understanding how broadcasting works is this line here:

```
DeltaVB = torch.mean(E_diff[:, None]*dPsidvb, axis=0)
```

The array `E_diff` is one-dimensional and is stored in memory as a $1 \times n$ matrix, called row major storage for reference, and by adding a wrapper around every element, they themself are stored as a row of only one element. This is then the same as transposing the vector:

$$E_{diff} = [a_1, a_2, \ldots, a_n] \rightarrow \begin{bmatrix} [a_1] \\ [a_2] \\ \vdots \\ [a_n] \end{bmatrix}$$

The transposed array, though now with an extra superficial dimension, is then multiplied element-wise with a one-dimensional array. But since the elements of `E_diff` is now arrays in their own right, `dPsidvb` is copied for each row of `E_diff`. Result in

$$\begin{bmatrix} [a_1] \\ [a_2] \\ \vdots \\ [a_n] \end{bmatrix} * [b_1, b_2, \ldots, b_n] \rightarrow \begin{bmatrix} a_1 \cdot [b_1, b_2, \ldots, b_n] \\ a_2 \cdot [b_1, b_2, \ldots, b_n] \\ \vdots \\ a_n \cdot [b_1, b_2, \ldots, b_n] \end{bmatrix}$$

Where * is element-wise multiplication and $\cdot$ is the scalar product. Going back to the first example we then have

$$\begin{bmatrix}[a_{11}] & [a_{12}] \\ [a_{21}] & [a_{22}]\end{bmatrix} * \begin{bmatrix}[b_{11} & b_{12}] \\ [b_{21} & b_{22}]\end{bmatrix} \rightarrow \begin{bmatrix}[a_{11}] \cdot [b_{11} & b_{12}] & [a_{12}] \cdot [b_{11} & b_{12}] \\ [a_{21}] \cdot [b_{21} & b_{22}] & [a_{22}] \cdot [b_{21} & b_{22}]\end{bmatrix} \rightarrow \begin{bmatrix}a_{11}b_{11} + a_{11}b_{12} & a_{12}b_{11} + a_{12}b_{12} \\ a_{21}b_{21} + a_{21}b_{22} & a_{22}b_{21} + a_{22}b_{22}\end{bmatrix}$$

Where it is important to notice that we have refrained from using the = sign as this is all meant as a way to visualize the process as apposed to being mathematically correct.
We then come to the update part of the solver, which has been severly shortened here as there is much of it that is used for logging different run statistics.

```
def find_min_energy(...):

  ...

  for n in range(epochs):
    DeltaVB, DeltaHB, DeltaW = MonteCarlo(...)

    adapt_lr = adapt_func(...)

    model.visual_bias -= adapt_lr*DeltaVB
    model.hidden_bias -= adapt_lr*DeltaHB
    model.weights -= adapt_lr*DeltaW


  ...

return stats
```

For each epoch it calls the `MonteCarlo(...)` function for the derivatives of the cost function based on the different variables we want to optimize. Then there is an adaptive learning rate function, which is defined by the user at the main `run(...)` call, and then we update the machine variables.

7.1 Calculation of the local energy

At the sampling part we skipped over an important line.

```
  E_local = local_energy_func(dist_s)
```

Where we have to calculate the local energy of the quantum system in question. We know that the local energy of a particular sample state $|s\rangle$ can be calculated as

$$E_{loc} = \frac{\langle s| H |\psi_{rbm}\rangle}{\langle s|\psi_{rbm}\rangle} , \tag{7.1}$$

but here we will explain in more detail how this is done for the each of the different systems we are looking at. First of all it is important to remember the structure of the input to our function calculating the local energy. We want to vectorize the calculations as much as possible so the input will be all our samples, taken with the Gibbs or Metropolis-Hastings algorithm, together in one array:

$$\mathbf{S} = [s_0, s_1, \ldots, s_n] \ . \tag{7.2}$$

Our operations will then be done on all the samples at once, decreasing computation time by vectorization as well as opening up for use of the GPU. To use the definition of the local energy, 7.1, we need to extract the machine state from our set of samples. As described in 4.4 we check the distribution by extracting all unique states in our set of samples and setting the amplitudes as the square root of their relative occurrence in the set, presuming that all are positive definite.

```
def unique_states_amplitude(samples):
    size = samples.shape[0]
    unique, weight = torch.unique(samples, dim=0, return_counts=True)
    weight = torch.sqrt(weight/size)

    return unique, weight
```

We then have a approximation of our machine state $|\psi\rangle$. The next step of using each state in the sample set is varies with the hamiltonian of the system.

7.1.1 The Lipkin model

The Lipkin model hamiltonian is defined as

$$H = \sum_{p\sigma} \left(\frac{1}{2}\sigma\varepsilon\right) \hat{a}^\dagger_{p\sigma}\hat{a}_{p\sigma} + \frac{V}{2}\sum_{pp'\sigma} \hat{a}^\dagger_{p\sigma}\hat{a}^\dagger_{p'\sigma}\hat{a}_{p'-\sigma}\hat{a}_{p-\sigma} + \frac{W}{2}\sum_{pp'\sigma} \hat{a}^\dagger_{p\sigma}\hat{a}^\dagger_{p'-\sigma}\hat{a}_{p'\sigma}\hat{a}_{p-\sigma} \ , \tag{7.3}$$

For a more structured explanation we separate the hamiltonian in three parts:

$$H_\varepsilon = \sum_{p\sigma} \left(\frac{1}{2}\sigma\varepsilon\right) \hat{a}^\dagger_{p\sigma}\hat{a}_{p\sigma} \tag{7.4}$$

$$H_V = \frac{V}{2}\sum_{pp'\sigma} \hat{a}^\dagger_{p\sigma}\hat{a}^\dagger_{p'\sigma}\hat{a}_{p'-\sigma}\hat{a}_{p-\sigma} \tag{7.5}$$

$$H_W = \frac{W}{2}\sum_{pp'\sigma} \hat{a}^\dagger_{p\sigma}\hat{a}^\dagger_{p'-\sigma}\hat{a}_{p'\sigma}\hat{a}_{p-\sigma} \ . \tag{7.6}$$

A state $|b\rangle$ contributes for the different parts as follows:

- H_ε - the same state $|b\rangle$.
- H_V - states that are a excited or deexcited pair away from the state $|b\rangle$.
- H_W - states one excited or deexcited particle away from $|b\rangle$.

For the H_ε contribution we start of by calculating the number of particles in the first and second layer.

```
N_0 = torch.sum(unique == 0, dim=-1)
N_1 = torch.sum(unique == 1, dim=-1)
```

The particles in the first layer contribute with negative spin and the particles in the second layer contribute positivly, so we get that.

```
H_0 = 0.5*eps*(N_1-N_0)
```

We then have an array of the machine states amplitudes multiplied by ε. Applying `mask` to this array then selects the correct value for each sample without having to calculate duplicates more than once. We then have the H_ε contribution for each sample directly:

```
H_eps = H_0[mask]
```

We then calculate the difference between the basis states of the machine state, which is the states in `unique`, and the number of particles that change layer.

```
diff_unique = abs(torch.sum(unique[:, None] - unique), dim=-1)
diff_N1 = abs(N_1[:, None] - N_1)
```

The vector arrays are being broadcast to a $N \times N$ matrix. If we denote `unique` $= \boldsymbol{u}$ we have that `unique[:, None] - unique` becomes

$$\boldsymbol{u}^T - \boldsymbol{u} = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_n \end{bmatrix} - \begin{bmatrix} u_0 & u_1 & \dots & u_n \end{bmatrix} \rightarrow \begin{bmatrix} u_0 - u_0 & u_0 - u_1 & \dots & u_0 - u_n \\ u_1 - u_0 & u_1 - u_1 & \dots & u_1 - u_n \\ & & \vdots & \\ u_n - u_0 & u_n - u_1 & \dots & u_n - u_n \end{bmatrix} \tag{7.7}$$

Taking the absolute value of the sum of these $u_i - u_j$ elements we get the total number of particles that either excite or de-excite from one state to the next. For the H_V contributions we first need to find each possible one-pair-different basis states. The `diff_unique` array does not confirm that a two-particles difference comes the same layer, however, so we need to check with the change in particles in one of the layers as well. We calculate the H_V contribution for the basis states as follows:

```
one_pair = np.bitwise_and(diff_unique==2, diff_N1==2)
H_1 = V*torch.sum(weight[:, None]*one_pair, dim=0)/weight
```

where the `np.bitwise_and` outputs a 1 only if both inputs are 1.
We check each of the unique states, $|b_i\rangle$, with all the other unique states, $|b_j\rangle$, and see if there are a one-pair-difference between them, setting the element (i, j) to 1 if that is the case. As a result we end up with a $N \times N$ matrix, $\mathbf{V}$, where N is the number of unique states. Each pair represents the element

$$\boldsymbol{V}_{i,j} = \frac{\langle b_i| H_V |b_j\rangle}{\langle b_i|\psi_{rbm}\rangle} = \frac{V\alpha_j}{\alpha_i} , \tag{7.8}$$

where α_i and α_j is respectively the amplitude of the basis state $|b_i\rangle$ and $|b_j\rangle$ in the machine state ψ_{rbm}. This means we need to scale our matrix $\mathbf{V}$ with the interaction strength V. Then scale each row i with $\frac{1}{\alpha_i}$ and each column j with the corresponding states amplitude α_j. Each row then represents the elements of

$$E_{loc,V}(|b_i\rangle) = \frac{\langle b_i| H_V |\psi_{rbm}\rangle}{\langle b_i|\psi_{rbm}\rangle} ,$$

and since the amplitudes are already assumed to be positive definite, we can then calculate the total contribution of the basis state $|b_i\rangle$:

$$E_{loc,V}(|b_i\rangle) = \sum_j \mathbf{V}_{i,j} , \tag{7.9}$$

which we can then apply `mask` to and get the H_V contribution for each sample:

```
    H_1 = V*torch.sum(
        weight[:, None]*(
            abs(torch.sum((unique[:, None] - unique), dim=-1)) == 2), dim=0)/weight
    H_V = H_1[mask]
```

For H_W we similarly find the unique states that are one particle different from each other. Scaling it with the interaction strength W, the rows and columns with the amplitude of the respective basis state and sum along the second axis:

```
    H_2 = W*torch.sum(
        weight[:, None]*(
            abs(torch.sum((unique[:, None] - unique), dim=-1)) == 1) , dim=0)/weight
    H_W = H_2[mask]
```

The total hamiltonian we have as

$$H = H_\varepsilon + H_V + H_W ,$$

which we can calculate directly:

```
    E = (H_eps + H_V + H_W)
    return E
```

7.1.2 The Ising model

For the Ising model we can calculate the local energy directly with:

$$H(\boldsymbol{\sigma}) = J \sum_{<i,j>} \sigma_i \sigma_j , \tag{7.10}$$

where J is the interaction strength between nearest neighbors. To calculate this efficiently we take the whole set and shift it:

```
  def ising_local(samples):
    spin = 2*samples - 1
    shifted = torch.roll(spin, dim=1)
```

where we then assume the boundary conditions:

$$\sigma_0 = \sigma_N \ , \tag{7.11}$$

as explained in ??. We can then easily calculate the energy of the configuration with a sum:

```
H_J = J*torch.sum(shifted*spin, dim=1)
H_L = L*torch.sum(spin, dim=1)
return H_J + H_L
```

where H_J is the iarenteraction part and H_L is the external field part.

7.1.3 The Pairing model

The Pairing model hamiltonian is defined as:

$$H = \dots \tag{7.12}$$

We then start with

7.1.4 The Heisenberg model

The Heisenberg model hamiltonian we have as:

$$H = \dots \tag{7.13}$$

7.2 Implementation tests

As we have implemented the restricted Boltzmann machine and the relevant hamiltonians, it would be good to know for certain that it produces meaningful result for each model. To accomplish this we will test the RBM on some small systems of the different models. This section is only meant to verify our implementation, so here we can sacrifice accuracy for less computation time. For all calculations in this section we iterate through 500 epochs with 10000 samples and a learning rate $\eta = 0.5$.

7.2.1 The Lipkin model

For the Lipkin model we first want to see the machine find the correct energy with no interaction between particles, single particle energy only. We make sure the machine handles increasing complexity by looking at systems with increasing size: 2, 4, 8 and 16 particles. With only single particle energy the ground state is the state where all particles occupying the lower level, which is for a N-particle system:

$$E_0 = \frac{1}{2}\varepsilon N \ . \tag{7.14}$$

To keep the output consistent, and make it easier to analyze, we will set

$$\varepsilon = 1$$

all throughout this section. Combining the convergence graphs into one we get:

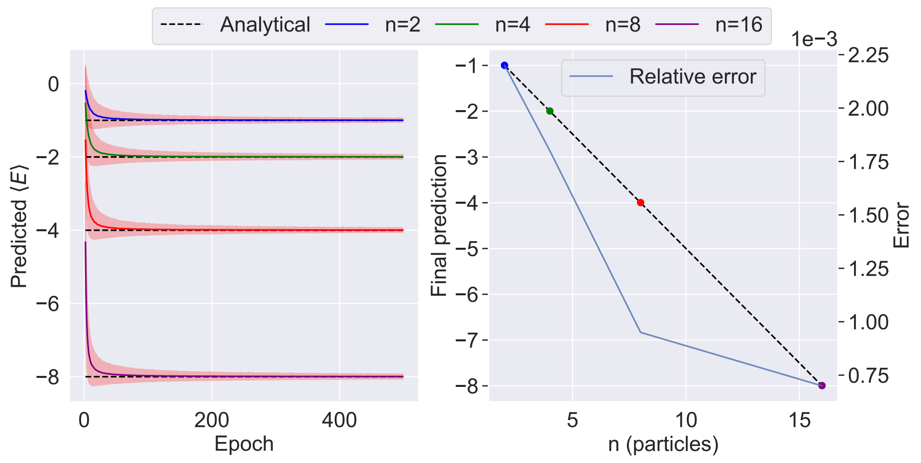


Figure 7.1: The predicted ground state energy for the Lipkin model without interaction and $\varepsilon = 1$. On the left is convergence plotted for each of the different number of particles and the colored area indicates the standard deviation of the local energy of the different samples. The first few data points of each convergence line is removed for readability. On the right the final output of the machine is shown with a dot for n particles 2, 4, 8 and 16 together with the relative error. The dashed lines are the analytical ground state energies.

We see that the RBM manages to find the ground state energy for each system size. We can also see the standard deviation approaching zero. This tells us the RBM wavefunction is close to the true wavefunction because, as explained in **??**:

$$\begin{gathered}\psi_{rbm} = \psi_{true} \\ \Downarrow \\ Var[E_L] = 0\ .\end{gathered}$$

The right side plot reiterates that the predicted energies follows the analytical solution, but we also see that the relative error is decreasing, although not with a factor equal to the decrease in ground state energy. Which means the error is increasing somewhat with the size of the system. Looking at the relative standard deviation:

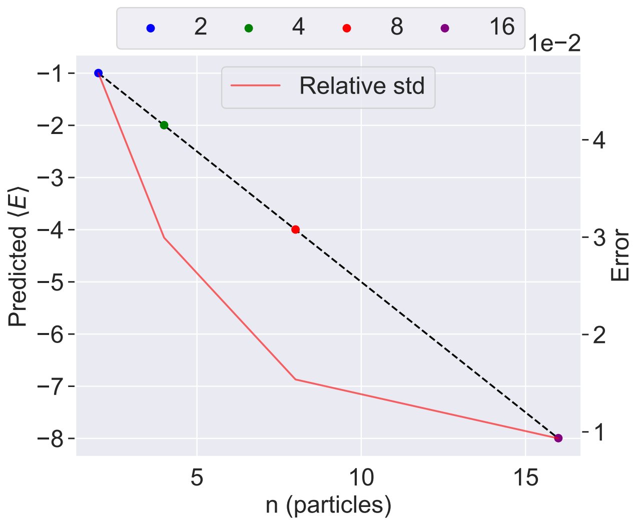


Figure 7.2: The found ground state energy by the restricted Boltzmann machine for the Lipkin model without interaction together with the relative standard deviation error.

we can see that it also decreases with the number of particles. However, the relative standard deviation decreases more, relatively from $n = 2$ to $n = 16$, than the relative error does. This would indicate that the overall increase in error mostly comes from the increase in local energy of the basis states that are wrongly part of the machine state rather than a increase in the machine state's spread.

7.2.2 The Ising model

7.2.3 The Pairing model

7.2.4 The Heisenberg model

Part III

Results and Discussion

Chapter 8

Simple Model

Chapter 9

Main Model

Part IV

Conclusion

Chapter 10

Conclusion

Bibliography

[1] F. Rosenblatt, *Principles of neurodynamics; perceptrons and the theory of brain mechanisms.* Washington: Spartan Books, 1962, bibliography : p. 609-616. [Online]. Available: http://hdl.handle.net/2027/mdp.39015039846566

[2] S. Linnainmaa, "Taylor expansion of the accumulated rounding error," *BIT Numerical Mathematics*, vol. 16, no. 2, pp. 146–160, Jun 1976. [Online]. Available: https://doi.org/10.1007/BF01931367

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct 1986. [Online]. Available: https://doi.org/10.1038/323533a0

[4] G. E. Hinton and T. J. Sejnowski, "Analyzing cooperative computation," *Fifth annual connference of the Cogniitive Science Society*, 1983.

[5] A. Yuille, "Boltzmann machine," *JHU*, 2016.

[6] U. author. (2019) Derivation of gradient of the expectation of local energy. [Online]. Available: https://physics.stackexchange.com/questions/473533/derivation-of-gradient-of-the-expectation-of-local-energy

[7] Vol. 355, feb 2017. [Online]. Available: https://doi.org/10.1126%2Fscience.aag2302

[8] H. Lipkin, N. Meshkov, and A. Glick, "Validity of many-body approximation methods for a solvable model: (i). exact solutions and perturbation theory," *Nuclear Physics*, vol. 62, no. 2, pp. 188–198, 1965. [Online]. Available: https://www.sciencedirect.com/science/article/pii/002955826590862X

[9] NVIDIA, "Nvidia," 2024. [Online]. Available: https://www.nvidia.com/en-us/about-nvidia/#About%20Us

[10] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda compilation tools, release 12.3, v12.3.103," 2023. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[11] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.