Solving quantum mechanical problems with Machine Learning

Morten Hjorth-Jensen[1,2]

[1]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University, USA
[2]Department of Physics and Center for Computing in Science Education, University of Oslo, Norway

Jan 16, 2019

What is this talk about?

The main aim is to give you a short and pedestrian introduction to how we can use Machine Learning methods to solve quantum mechanical many-body problems. And why this could be of interest.

The hope is that after this talk you have gotten the basic ideas to get you started. Peeping into https://github.com/mhjensenseminars/MachineLearningTalk, you'll find a Jupyter notebook, slides, codes etc that will allow you to reproduce the simulations discussed here, and perhaps run your own very first calculations.

Try it out and please don't hesitate to swing by if something is unclear.

Why?

How can we avoid the dimensionality curse? Many possibilities

1. smarter basis functions
2. resummation of specific correlations
3. stochastic sampling of high-lying states (stochastic FCI, CC and SRG/IMSRG)
4. many more

Machine Learning and Quantum Computing hold also great promise in tackling the ever increasing dimensionalities. Here we will focus on Machine Learning.

Overview

- Short intro to Machine Learning

© 1999-2019, Morten Hjorth-Jensen. Released under CC Attribution-NonCommercial 4.0 license

- Variational Monte Carlo (Markov Chain Monte Carlo, MC^2) and two-electron quantum dots, solving quantum mechanical problems in a stochastic way. It will serve as our motivation for switching to Machine Learning.
- From Variational Monte Carlo to Boltzmann Machines and Machine Learning

What are the Machine Learning calculations here based on?

This work is inspired by the idea of representing the wave function with a restricted Boltzmann machine (RBM), presented recently by G. Carleo and M. Troyer, Science **355**, Issue 6325, pp. 602-606 (2017). They named such a wave function/network a *neural network quantum state* (NQS). In their article they apply it to the quantum mechanical spin lattice systems of the Ising model and Heisenberg model, with encouraging results.

Thanks to Jane Kim (MSU), Vilde Flugsrud (UiO), Alfred Alocias Mariadason (UiO) for many discussions and interpretations.

A new world

Machine learning (ML) is an extremely rich field, in spite of its young age. The increases we have seen during the last three decades in computational capabilities have been followed by developments of methods and techniques for analyzing and handling large date sets, relying heavily on statistics, computer science and mathematics. The field is rather new and developing rapidly.

Popular software packages written in Python for ML are

- Scikit-learn,
- Tensorflow,
- PyTorch and
- Keras.

These are all freely available at their respective GitHub sites. They encompass communities of developers in the thousands or more. And the number of code developers and contributors keeps increasing.

Lots of room for creativity

Not all the algorithms and methods can be given a rigorous mathematical justification, opening up thereby for experimenting and trial and error and thereby exciting new developments.

A solid command of linear algebra, multivariate theory, probability theory, statistical data analysis, understanding errors and Monte Carlo methods is important in order to understand many of the various algorithms and methods.

Job market, a personal statement: A familiarity with ML is almost becoming a prerequisite for many of the most exciting employment opportunities. And add quantum computing and there you are!

Knowledge of Statistical analysis and optimization of data

Some key elements that enter much of the discussion on ML:

1. Basic concepts, expectation values, variance, covariance, correlation functions and errors;
2. Simpler models, binomial distribution, the Poisson distribution, simple and multivariate normal distributions;
3. Central elements of Bayesian statistics and modeling;
4. Central elements from linear algebra
5. Gradient methods for data optimization
6. Monte Carlo methods, Markov chains, Metropolis-Hastings algorithm;
7. Estimation of errors using cross-validation, blocking, bootstrapping and jackknife methods;
8. Practical optimization using Singular-value decomposition and least squares for parameterizing data.
9. Principal Component Analysis.

Some members of the ML family

1. Linear regression and its variants, Logistic regression
2. Decision tree algorithms, from simpler to more complex ones like random forests
3. Bayesian statistics
4. Support vector machines and finally various variants of
5. Artifical neural networks and deep learning
6. Convolutional NN, autoenconders
7. and many more

What are the basic ingredients?

Almost every problem in ML and data science starts with the same ingredients:

- The dataset $\mathbf{x}$ (could be some observable quantity of the system we are studying)
- A model which is a function of a set of parameters α that relates to the dataset, say a likelihood function $p(\mathbf{x}|\alpha)$ or just a simple model $f(\alpha)$
- A so-called **cost** function $\mathcal{C}(\mathbf{x}, f(\alpha))$ which allows us to decide how well our model represents the dataset.

We seek to minimize the function $\mathcal{C}(\mathbf{x}, f(\alpha))$ by finding the parameter values which minimize $\mathcal{C}$. This leads to various minimization algorithms.

What is Machine Learning?

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

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

You will notice however that many of the basic ideas discussed do come from Physics!

Types of Machine Learning

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

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

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

- Regression: Finding a functional relationship between an input data set and a reference data set. The goal is to construct a function that maps input data to continuous output values.
- Clustering: Data are divided into groups with certain common traits, without knowing the different groups beforehand. It is thus a form of unsupervised learning.

References

- An excellent reference, Mehta *et al.*, arXiv:1803.08823 and Physics Reports in press (2018)
- A cute paper by Utama and Piekarewicz, Validating neural-network refinements of nuclear mass models, Phys. Rev. C 97, 014306
- Every issue of Physical Review Letters has now one or more articles on ML
- Books and lectures notes and see also the course FYS-STK3155/4155
- See also Metha and Schwab, arXiv.1410.3831, how to link Variational renormalization group theory with deep learning (recommended read)

Another interesting article

Here we will use so-called **reduced Boltzmann Machines** to simulate quantum many-body problems. For Monte Carlo aficionados, there is a very close similarity with what are called **shadow wave functions**, see the work of Pederiva and Kalos and collaborators, Phys Rev. E 90, 053304 (2014).

Courses on Data science and Machine Learning at UiO

The link here gives an excellent overview of courses on Machine learning at UiO.

1. STK2100 Machine learning and statistical methods for prediction and classification.
2. IN3050 Introduction to Artificial Intelligence and Machine Learning. Introductory course in machine learning and AI with an algorithmic approach.
3. STK-INF3000/4000 Selected Topics in Data Science. The course provides insight into selected contemporary relevant topics within Data Science.
4. IN4080 Natural Language Processing. Probabilistic and machine learning techniques applied to natural language processing.

5. STK-IN4300 – Statistical learning methods in Data Science. An advanced introduction to statistical and machine learning. For students with a good mathematics and statistics background.

6. INF4490 Biologically Inspired Computing. An introduction to self-adapting methods also called artificial intelligence or machine learning.

7. IN-STK5000 Adaptive Methods for Data-Based Decision Making. Methods for adaptive collection and processing of data based on machine learning techniques.

8. IN5400/INF5860 – Machine Learning for Image Analysis. An introduction to deep learning with particular emphasis on applications within Image analysis, but useful for other application areas too.

9. TEK5040 – Deep learning. The course addresses advanced algorithms and architectures for deep learning with neural networks. The course provides an introduction to how deep-learning techniques can be used in the construction of key parts of advanced autonomous systems that exist in physical environments and cyber environments.

Additional courses of interest

1. STK4051 Computational Statistics

2. STK4021 Applied Bayesian Analysis and Numerical Methods

Decision trees and Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

steps=250

distance=0
x=0
distance_list=[]
steps_list=[]
while x<steps:
    distance+=np.random.randint(-1,2)
    distance_list.append(distance)
    x+=1
    steps_list.append(x)
plt.plot(steps_list,distance_list, color='green', label="Random Walk Data")

steps_list=np.asarray(steps_list)
distance_list=np.asarray(distance_list)

X=steps_list[:,np.newaxis]
```

```
#Polynomial fits

#Degree 2
poly_features=PolynomialFeatures(degree=2, include_bias=False)
X_poly=poly_features.fit_transform(X)

lin_reg=LinearRegression()
poly_fit=lin_reg.fit(X_poly,distance_list)
b=lin_reg.coef_
c=lin_reg.intercept_
print ("2nd degree coefficients:")
print ("zero power: ",c)
print ("first power: ", b[0])
print ("second power: ",b[1])

z = np.arange(0, steps, .01)
z_mod=b[1]*z**2+b[0]*z+c

fit_mod=b[1]*X**2+b[0]*X+c
plt.plot(z, z_mod, color='r', label="2nd Degree Fit")
plt.title("Polynomial Regression")

plt.xlabel("Steps")
plt.ylabel("Distance")

#Degree 10
poly_features10=PolynomialFeatures(degree=10, include_bias=False)
X_poly10=poly_features10.fit_transform(X)

poly_fit10=lin_reg.fit(X_poly10,distance_list)

y_plot=poly_fit10.predict(X_poly10)
plt.plot(X, y_plot, color='black', label="10th Degree Fit")

plt.legend()
plt.show()


#Decision Tree Regression
from sklearn.tree import DecisionTreeRegressor
regr_1=DecisionTreeRegressor(max_depth=2)
regr_2=DecisionTreeRegressor(max_depth=5)
regr_3=DecisionTreeRegressor(max_depth=7)
regr_1.fit(X, distance_list)
regr_2.fit(X, distance_list)
regr_3.fit(X, distance_list)

X_test = np.arange(0.0, steps, 0.01)[:, np.newaxis]
y_1 = regr_1.predict(X_test)
y_2 = regr_2.predict(X_test)
y_3=regr_3.predict(X_test)

# Plot the results
plt.figure()
plt.scatter(X, distance_list, s=2.5, c="black", label="data")
plt.plot(X_test, y_1, color="red",
         label="max_depth=2", linewidth=2)
plt.plot(X_test, y_2, color="green", label="max_depth=5", linewidth=2)
plt.plot(X_test, y_3, color="m", label="max_depth=7", linewidth=2)

plt.xlabel("Data")
```

```
plt.ylabel("Darget")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```

Artificial neurons

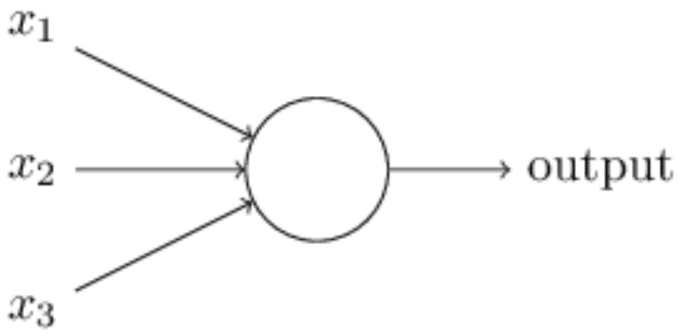
The field of artificial neural networks has a long history of development, and is closely connected with the advancement of computer science and computers in general. A model of artificial neurons was first developed by McCulloch and Pitts in 1943 to study signal processing in the brain and has later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

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

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

Here, the output y of the neuron is the value of its activation function, which have as input a weighted sum of signals $x_i, \ldots, x_n$ received by n other neurons.

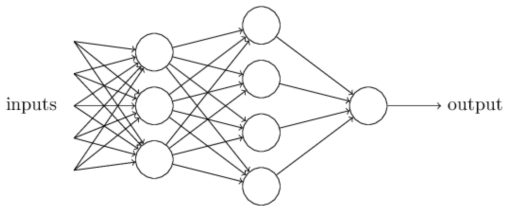
A simple perceptron model



Neural network types

An artificial neural network (NN), is a computational model that consists of layers of connected neurons, or *nodes*. It is supposed to mimic a biological nervous system by letting each neuron interact with other neurons by sending

signals in the form of mathematical functions between layers. A wide variety of different NNs have been developed, but most of them consist of an input layer, an output layer and eventual layers in-between, called *hidden layers*. All layers can contain an arbitrary number of nodes, and each connection between two nodes is associated with a weight variable.



The system: two electrons in a harmonic oscillator trap in two dimensions

The Hamiltonian of the quantum dot is given by

$$\hat{H} = \hat{H}_0 + \hat{V},$$

where $\hat{H}_0$ is the many-body HO Hamiltonian, and $\hat{V}$ is the inter-electron Coulomb interactions. In dimensionless units,

$$\hat{V} = \sum_{i<j}^{N} \frac{1}{r_{ij}},$$

with $r_{ij} = \sqrt{\mathbf{r}_i^2 - \mathbf{r}_j^2}$.

This leads to the separable Hamiltonian, with the relative motion part given by ($r_{ij} = r$)

$$\hat{H}_r = -\nabla_r^2 + \frac{1}{4}\omega^2 r^2 + \frac{1}{r},$$

plus a standard Harmonic Oscillator problem for the center-of-mass motion. This system has analytical solutions in two and three dimensions (M. Taut 1993 and 1994).

Quantum Monte Carlo Motivation

Given a hamiltonian H and a trial wave function Ψ_T, the variational principle states that the expectation value of $\langle H \rangle$, defined through

$$\langle E \rangle = \frac{\int d\boldsymbol{R} \Psi_T^*(\boldsymbol{R}) H(\boldsymbol{R}) \Psi_T(\boldsymbol{R})}{\int d\boldsymbol{R} \Psi_T^*(\boldsymbol{R}) \Psi_T(\boldsymbol{R})},$$

is an upper bound to the ground state energy E_0 of the hamiltonian H, that is

$$E_0 \leq \langle H \rangle.$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. Traditional integration methods such as the Gauss-Legendre will not be adequate for say the computation of the energy of a many-body system.

Quantum Monte Carlo Motivation

Basic steps. Choose a trial wave function $\psi_T(\boldsymbol{R})$.

$$P(\boldsymbol{R},\boldsymbol{\alpha}) = \frac{|\psi_T(\boldsymbol{R},\boldsymbol{\alpha})|^2}{\int |\psi_T(\boldsymbol{R},\boldsymbol{\alpha})|^2 d\boldsymbol{R}}.$$

This is our model, or likelihood/probability distribution function (PDF). It depends on some variational parameters $\boldsymbol{\alpha}$. The approximation to the expectation value of the Hamiltonian is now

$$\langle E[\boldsymbol{\alpha}] \rangle = \frac{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R},\boldsymbol{\alpha})H(\boldsymbol{R})\Psi_T(\boldsymbol{R},\boldsymbol{\alpha})}{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R},\boldsymbol{\alpha})\Psi_T(\boldsymbol{R},\boldsymbol{\alpha})}.$$

Quantum Monte Carlo Motivation

Define a new quantity.

$$E_L(\boldsymbol{R},\boldsymbol{\alpha}) = \frac{1}{\psi_T(\boldsymbol{R},\boldsymbol{\alpha})} H\psi_T(\boldsymbol{R},\boldsymbol{\alpha}),$$

called the local energy, which, together with our trial PDF yields

$$E[\boldsymbol{\alpha}] = \int P(\boldsymbol{R})E_L(\boldsymbol{R},\boldsymbol{\alpha})d\boldsymbol{R} \approx \frac{1}{N}\sum_{i=1}^{N} E_L(\boldsymbol{R_i},\boldsymbol{\alpha})$$

with N being the number of Monte Carlo samples.

Quantum Monte Carlo

The Algorithm for performing a variational Monte Carlo calculations runs thus as this

- Initialisation: Fix the number of Monte Carlo steps. Choose an initial $\boldsymbol{R}$ and variational parameters α and calculate $|\psi_T(\boldsymbol{R},\boldsymbol{\alpha})|^2$.
- Initialise the energy and the variance and start the Monte Carlo calculation by looping over trials.

- Calculate a trial position $\boldsymbol{R}_p = \boldsymbol{R} + r * step$ where r is a random variable $r \in [0,1]$.
- Metropolis algorithm to accept or reject this move $w = P(\boldsymbol{R}_p, \boldsymbol{\alpha})/P(\boldsymbol{R}, \boldsymbol{\alpha})$.
- If the step is accepted, then we set $\boldsymbol{R} = \boldsymbol{R}_p$.
- Update averages

- Finish and compute final averages.

Observe that the jumping in space is governed by the variable *step*. This is often called brute-force sampling. Need importance sampling to get more relevant sampling.

The trial wave function

We want to perform a Variational Monte Carlo calculation of the ground state of two electrons in a quantum dot well with different oscillator energies, assuming total spin $S = 0$. Our trial wave function has the following form

$$\psi_T(\boldsymbol{r}_1, \boldsymbol{r}_2) = C \exp\left(-\alpha_1 \omega (r_1^2 + r_2^2)/2\right) \exp\left(\frac{r_{12}}{(1+\alpha_2 r_{12})}\right), \tag{1}$$

where the αs represent our variational parameters, two in this case.

Why does the trial function look like this? How did we get there? **This will be our main motivation** for switching to Machine Learning.

The correlation part of the wave function

To find an ansatz for the correlated part of the wave function, it is useful to rewrite the two-particle local energy in terms of the relative and center-of-mass motion. Let us denote the distance between the two electrons as r_{12}. We omit the center-of-mass motion since we are only interested in the case when $r_{12} \to 0$. The contribution from the center-of-mass (CoM) variable $\boldsymbol{R}_{\rm CoM}$ gives only a finite contribution. We focus only on the terms that are relevant for r_{12} and for three dimensions. The relevant local energy becomes then

$$\lim_{r_{12}\to 0} E_L(R) = \frac{1}{\mathcal{R}_T(r_{12})}\left(2\frac{d^2}{dr_{ij}^2} + \frac{4}{r_{ij}}\frac{d}{dr_{ij}} + \frac{2}{r_{ij}} - \frac{l(l+1)}{r_{ij}^2} + 2E\right)\mathcal{R}_T(r_{12}) = 0.$$

Set $l = 0$ and we have the so-called **cusp** condition

$$\frac{d\mathcal{R}_T(r_{12})}{dr_{12}} = -\frac{1}{2(l+1)}\mathcal{R}_T(r_{12}) \qquad r_{12} \to 0$$

Resulting ansatz

The above results in

$$\mathcal{R}_T \propto \exp{(r_{ij}/2)},$$

for anti-parallel spins and

$$\mathcal{R}_T \propto \exp{(r_{ij}/4)},$$

for anti-parallel spins. This is the so-called cusp condition for the relative motion, resulting in a minimal requirement for the correlation part of the wave fuction. For general systems containing more than say two electrons, we have this condition for each electron pair ij.

The VMC code

```
# Importing various packages
from math import exp, sqrt
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import sys

#Trial wave function for quantum dots in two dims
def WaveFunction(r,alpha,beta):
    r1 = r[0,0]**2 + r[0,1]**2
    r2 = r[1,0]**2 + r[1,1]**2
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = r12/(1+beta*r12)
    return exp(-0.5*alpha*(r1+r2)+deno)

#Local energy  for quantum dots in two dims, using analytical local energy
def LocalEnergy(r,alpha,beta):

    r1 = (r[0,0]**2 + r[0,1]**2)
    r2 = (r[1,0]**2 + r[1,1]**2)
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = 1.0/(1+beta*r12)
    deno2 = deno*deno
    return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha + 1.0/r12+deno2*(alpha*r12-deno2+2*beta*deno-1

# The Monte Carlo sampling with the Metropolis algo
def MonteCarloSampling():

    NumberMCcycles= 100000
    StepSize = 1.0
    # positions
    PositionOld = np.zeros((NumberParticles,Dimension), np.double)
    PositionNew = np.zeros((NumberParticles,Dimension), np.double)
    # seed for rng generator
    seed()
    # start variational parameter
    alpha = 0.9
    for ia in range(MaxVariations):
        alpha += .025
        AlphaValues[ia] = alpha
```

```
            beta = 0.2
            for jb in range(MaxVariations):
                beta += .01
                BetaValues[jb] = beta
                energy = energy2 = 0.0
                DeltaE = 0.0
                #Initial position
                for i in range(NumberParticles):
                    for j in range(Dimension):
                        PositionOld[i,j] = StepSize * (random() - .5)
                wfold = WaveFunction(PositionOld,alpha,beta)

                #Loop over MC MCcycles
                for MCcycle in range(NumberMCcycles):
                    #Trial position
                    for i in range(NumberParticles):
                        for j in range(Dimension):
                            PositionNew[i,j] = PositionOld[i,j] + StepSize * (random() - .5)
                    wfnew = WaveFunction(PositionNew,alpha,beta)

                    #Metropolis test to see whether we accept the move
                    if random() < wfnew**2 / wfold**2:
                       PositionOld = PositionNew.copy()
                       wfold = wfnew
                       DeltaE = LocalEnergy(PositionOld,alpha,beta)
                    energy += DeltaE
                    energy2 += DeltaE**2

                #We calculate mean, variance and error ...
                energy /= NumberMCcycles
                energy2 /= NumberMCcycles
                variance = energy2 - energy**2
                error = sqrt(variance/NumberMCcycles)
                Energies[ia,jb] = energy
        return Energies, AlphaValues, BetaValues


    #Here starts the main program with variable declarations
    NumberParticles = 2
    Dimension = 2
    MaxVariations = 10
    Energies = np.zeros((MaxVariations,MaxVariations))
    AlphaValues = np.zeros(MaxVariations)
    BetaValues = np.zeros(MaxVariations)
    (Energies, AlphaValues, BetaValues) = MonteCarloSampling()

    # Prepare for plots
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    # Plot the surface.
    X, Y = np.meshgrid(AlphaValues, BetaValues)
    surf = ax.plot_surface(X, Y, Energies,cmap=cm.coolwarm,linewidth=0, antialiased=False)
    # Customize the z axis.
    zmin = np.matrix(Energies).min()
    zmax = np.matrix(Energies).max()
    ax.set_zlim(zmin, zmax)
    ax.set_xlabel(r'$\alpha$')
    ax.set_ylabel(r'$\beta$')
    ax.set_zlabel(r'$\langle E \rangle$')
    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
```

```
    # Add a color bar which maps values to colors.
    fig.colorbar(surf, shrink=0.5, aspect=5)
    plt.show()
```

Technical aspect, improvements and how to define the cost function

The above procedure is not the smartest one. Looping over all variational parameters becomes expensive. Also, we don't use importance sampling and optimizations of the standard deviation (blocking, bootstrap, jackknife). Such codes are included in the above Github address.

We can also be smarter and use minimization methods to find the **optimal** variational parameters with fewer Monte Carlo cycles and then fire up our heavy artillery.

One way to achieve this is to minimize the energy as function of the variational parameters.

Energy derivatives

To find the derivatives of the local energy expectation value as function of the variational parameters, we can use the chain rule and the hermiticity of the Hamiltonian.

Let us define

$$\bar{E}_{\alpha_i} = \frac{d\langle E_L\rangle}{d\alpha_i}.$$

as the derivative of the energy with respect to the variational parameter α_i We define also the derivative of the trial function (skipping the subindex T) as

$$\bar{\Psi}_i = \frac{d\Psi}{d\alpha_i}.$$

Derivatives of the local energy

The elements of the gradient of the local energy are then (using the chain rule and the hermiticity of the Hamiltonian)

$$\bar{E}_i = 2\left(\langle\frac{\bar{\Psi}_i}{\Psi}E_L\rangle - \langle\frac{\bar{\Psi}_i}{\Psi}\rangle\langle E_L\rangle\right).$$

From a computational point of view it means that you need to compute the expectation values of

$$\langle\frac{\bar{\Psi}_i}{\Psi}E_L\rangle,$$

and

$$\langle\frac{\bar{\Psi}_i}{\Psi}\rangle\langle E_L\rangle$$

These integrals are evaluted using MC intergration (with all its possible error sources). We can then use methods like stochastic gradient or other minimization methods to find the optimal variational parameters (I don't discuss this topic here, but these methods are very important in ML).

How do we define our cost function?

We have a model, our likelihood function.
How should we define the cost function?

Meet the variance and its derivatives

Why the variance? Suppose the trial function (our model) is the exact wave function. The action of the hamiltionan on the wave function

$$H\Psi = \text{constant} \times \Psi,$$

The integral which defines various expectation values involving moments of the hamiltonian becomes then

$$\langle E^n \rangle = \langle H^n \rangle = \frac{\int d\boldsymbol{R}\Psi^*(\boldsymbol{R})H^n(\boldsymbol{R})\Psi(\boldsymbol{R})}{\int d\boldsymbol{R}\Psi^*(\boldsymbol{R})\Psi(\boldsymbol{R})} = \text{constant} \times \frac{\int d\boldsymbol{R}\Psi^*(\boldsymbol{R})\Psi(\boldsymbol{R})}{\int d\boldsymbol{R}\Psi^*(\boldsymbol{R})\Psi(\boldsymbol{R})} = \text{constant}.$$

This gives an important information: If I want the variance, the exact wave function leads to zero variance! The variance is defined as

$$\sigma_E = \langle E^2 \rangle - \langle E \rangle^2.$$

Variation is then performed by minimizing both the energy and the variance.

The variance defines the cost function

We can then take the derivatives of

$$\sigma_E = \langle E^2 \rangle - \langle E \rangle^2,$$

with respect to the variational parameters. The derivatives of the variance can then be used to defined the so-called Hessian matrix, which in turn allows us to use minimization methods like Newton's method or standard gradient methods.

This leads to however a more complicated expression, with obvious errors when evaluating integrals by Monte Carlo integration. Less used, see however Filippi and Umrigar. The expression becomes complicated

$$\bar{E}_{ij} = 2\left[\langle(\frac{\bar{\Psi}_{ij}}{\Psi} + \frac{\bar{\Psi}_j}{\Psi}\frac{\bar{\Psi}_i}{\Psi})(E_L - \langle E \rangle)\rangle - \langle\frac{\bar{\Psi}_i}{\Psi}\rangle\bar{E}_j - \langle\frac{\bar{\Psi}_j}{\Psi}\rangle\bar{E}_i\right] + \langle\frac{\bar{\Psi}_i}{\Psi}E_{Lj}\rangle + \langle\frac{\bar{\Psi}_j}{\Psi}E_{Li}\rangle - \langle\frac{\bar{\Psi}_i}{\Psi}\rangle\langle E_{Lj}\rangle\langle\frac{\bar{\Psi}_j}{\Psi}\rangle\langle E_{Li}\rangle.$$

Evaluating the cost function means having to evaluate the above second derivative of the energy.

Why Boltzmann machines?

What is known as restricted Boltzmann Machines (RMB) have received a lot of attention lately. One of the major reasons is that they can be stacked layer-wise to build deep neural networks that capture complicated statistics.

The original RBMs had just one visible layer and a hidden layer, but recently so-called Gaussian-binary RBMs have gained quite some popularity in imaging since they are capable of modeling continuous data that are common to natural images.

Furthermore, they have been used to solve complicated quantum mechanical many-particle problems or classical statistical physics problems like the Ising and Potts classes of models.

Boltzmann Machines

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

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

Some similarities and differences from DNNs

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

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

Boltzmann machines (BM)

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

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

It turns into a Hopfield network if we choose deterministic rather than stochastic units. In contrast to a Hopfield network, a BM is a so-called generative model. It allows us to generate new samples from the learned distribution.

A standard BM setup

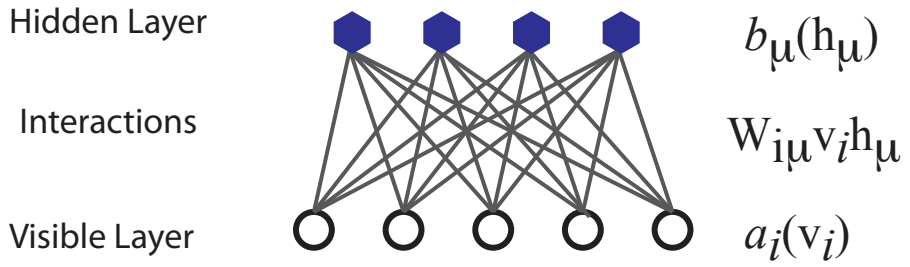
A standard BM network is divided into a set of observable and visible units $\hat{x}$ and a set of unknown hidden units/nodes $\hat{h}$.

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

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

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

The structure of the RBM network



The network

The network layers:

1. A function $\mathbf{x}$ that represents the visible layer, a vector of M elements (nodes). This layer represents both what the RBM might be given as training input, and what we want it to be able to reconstruct. This might for example be the pixels of an image, the spin values of the Ising model, or coefficients representing speech.

2. The function $\mathbf{h}$ represents the hidden, or latent, layer. A vector of N elements (nodes). Also called "feature detectors".

Goals

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

Examples of this trick being employed in physics:

1. The Hubbard-Stratonovich transformation

2. The introduction of ghost fields in gauge theory

3. Shadow wave functions in Quantum Monte Carlo simulations

The network parameters, to be optimized/learned:

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

2. $\mathbf{b}$ represents the hidden bias, a vector of same lenght as $\mathbf{h}$.

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

Joint distribution

The restricted Boltzmann machine is described by a Boltzmann distribution

$$P_{rbm}(\mathbf{x}, \mathbf{h}) = \frac{1}{Z} e^{-\frac{1}{T_0} E(\mathbf{x}, \mathbf{h})}, \tag{2}$$

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

$$Z = \int \int e^{-\frac{1}{T_0} E(\mathbf{x}, \mathbf{h})} d\mathbf{x} d\mathbf{h}. \tag{3}$$

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

Network Elements, the energy function

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

An expression for the energy function is

$$E(\hat{x}, \hat{h}) = -\sum_{ia}^{NA} b_i^a \alpha_i^a(x_i) - \sum_{jd}^{MD} c_j^d \beta_j^d(h_j) - \sum_{ijad}^{NAMD} b_i^a \alpha_i^a(x_i) c_j^d \beta_j^d(h_j) w_{ij}^{ad}.$$

Here $\beta_j^d(h_j)$ and $\alpha_i^a(x_j)$ are so-called transfer functions that map a given input value to a desired feature value. The labels a and d denote that there can be multiple transfer functions per variable. The first sum depends only on the visible units. The second on the hidden ones. **Note** that there is no connection between nodes in a layer.

The quantities b and c can be interpreted as the visible and hidden biases, respectively.

The connection between the nodes in the two layers is given by the weights w_{ij}.

Defining different types of RBMs

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

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

$$E(\mathbf{x}, \mathbf{h}) = -\sum_{i}^{M} x_i a_i - \sum_{j}^{N} b_j h_j - \sum_{i,j}^{M,N} x_i w_{ij} h_j, \tag{4}$$

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

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

$$E(\mathbf{x}, \mathbf{h}) = \sum_{i}^{M} \frac{(x_i - a_i)^2}{2\sigma_i^2} - \sum_{j}^{N} b_j h_j - \sum_{i,j}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2}. \tag{5}$$

More about RBMs

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

Other types of units include:

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

Sampling: Metropolis sampling

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

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

$$A(\boldsymbol{x}^k \rightarrow \boldsymbol{x}^{k+1}) = \min(1, \frac{P(\boldsymbol{x}^{k+1})}{P(\boldsymbol{x}^k)}), \tag{6}$$

where we need the marginalized probability

$$P(\boldsymbol{x}) = \sum_{\mathbf{h}} P_{rbm}(\mathbf{x}, \mathbf{h}) \tag{7}$$

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

Sampling: Gibbs sampling

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

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

$$P(h_j = 1|\boldsymbol{x}) = \frac{1}{1 + e^{-b_j - \sum_i x_i w_{ij}}} \tag{9}$$

$$P(x_i = 1|\boldsymbol{h}) = \frac{1}{1 + e^{-a_j - \sum_j h_j w_{ij}}}, \tag{10}$$

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

Gaussian RBM

For the Gaussian-Binary RBM the conditional probabilities are

$$P(x_i|\mathbf{h}) = \mathcal{N}(x_i; a_i + \sum_j h_j w_{ij}, \sigma^2) \tag{11}$$

$$P(h_j = 1|\mathbf{x}) = \frac{1}{1 + e^{-b_j - \frac{1}{\sigma^2} \sum_i x_i w_{ij}}}, \tag{12}$$

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

Cost function

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

$$\mathcal{L}(\{\theta_i\}) = \langle \log P_\theta(\boldsymbol{x}) \rangle_{data} \tag{13}$$

$$= -\langle E(\boldsymbol{x}; \{\theta_i\}) \rangle_{data} - \log Z(\{\theta_i\}), \tag{14}$$

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

Optimization / Training

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

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla \mathcal{C}(\boldsymbol{\theta}_k) \tag{15}$$

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

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

$$\frac{\partial \mathcal{C}(\{\theta_i\})}{\partial \theta_i} = \langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \rangle_{data} + \frac{\partial \log Z(\{\theta_i\})}{\partial \theta_i} \tag{16}$$

$$= \langle O_i(\boldsymbol{x}) \rangle_{data} - \langle O_i(\boldsymbol{x}) \rangle_{model}, \tag{17}$$

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

$$O_i(\boldsymbol{x}) = \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i}, \tag{18}$$

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

$$\langle O_i(\boldsymbol{x}) \rangle_{model} = \mathrm{Tr} P_\theta(\boldsymbol{x}) O_i(\boldsymbol{x}) = -\frac{\partial \log Z(\{\theta_i\})}{\partial \theta_i}. \tag{19}$$

More on RBMs

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

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

$$\frac{\partial \mathcal{C}(w_{ij}, a_i, b_j)}{\partial w_{ij}} = \langle x_i h_j \rangle_{data} - \langle x_i h_j \rangle_{model} \tag{20}$$

$$\frac{\partial \mathcal{C}(w_{ij}, a_i, b_j)}{\partial a_{ij}} = \langle x_i \rangle_{data} - \langle x_i \rangle_{model} \tag{21}$$

$$\frac{\partial \mathcal{C}(w_{ij}, a_i, b_j)}{\partial b_{ij}} = \langle h_i \rangle_{data} - \langle h_i \rangle_{model}. \tag{22}$$

$$\tag{23}$$

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

Which sampling to use

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

RBMs for the quantum many body problem

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

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

Choose the right RBM

Carleo and Troyer applied the RBM to the quantum mechanical spin lattice systems of the Ising model and Heisenberg model, with encouraging results. Our goal is to test the method on systems of moving particles. For the spin lattice systems it was natural to use a binary-binary RBM, with the nodes taking values

of 1 and -1. For moving particles, on the other hand, we want the visible nodes to be continuous, representing position coordinates. Thus, we start by choosing a Gaussian-binary RBM, where the visible nodes are continuous and hidden nodes take on values of 0 or 1. If eventually we would like the hidden nodes to be continuous as well the rectified linear units seem like the most relevant choice.

Representing the wave function

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

$$F_{rbm}(\mathbf{x}, \mathbf{h}) = \frac{1}{Z} e^{-\frac{1}{T_0} E(\mathbf{x}, \mathbf{h})}. \tag{24}$$

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

$$F_{rbm}(\mathbf{x}) = \sum_{\mathbf{h}} F_{rbm}(\mathbf{x}, \mathbf{h}) \tag{25}$$

$$= \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{x}, \mathbf{h})}. \tag{26}$$

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

$$\Psi(\mathbf{X}) = F_{rbm}(\mathbf{x}) \tag{27}$$

$$= \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\mathbf{x}, \mathbf{h})} \tag{28}$$

$$= \frac{1}{Z} \sum_{\{h_j\}} e^{-\sum_i^M \frac{(x_i - a_i)^2}{2\sigma^2} + \sum_j^N b_j h_j + \sum_{i,j}^{M,N} \frac{x_i w_{ij} h_j}{\sigma^2}} \tag{29}$$

$$= \frac{1}{Z} e^{-\sum_i^M \frac{(x_i - a_i)^2}{2\sigma^2}} \prod_j^N (1 + e^{b_j + \sum_i^M \frac{x_i w_{ij}}{\sigma^2}}). \tag{30}$$

$$\tag{31}$$

Choose the cost function

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

$$G_i = \frac{\partial \langle E_L \rangle}{\partial \theta_i} = 2(\langle E_L \frac{1}{\Psi} \frac{\partial \Psi}{\partial \theta_i} \rangle - \langle E_L \rangle \langle \frac{1}{\Psi} \frac{\partial \Psi}{\partial \theta_i} \rangle), \tag{32}$$

where the local energy is given by

$$E_L = \frac{1}{\Psi}\hat{\mathbf{H}}\Psi. \tag{33}$$

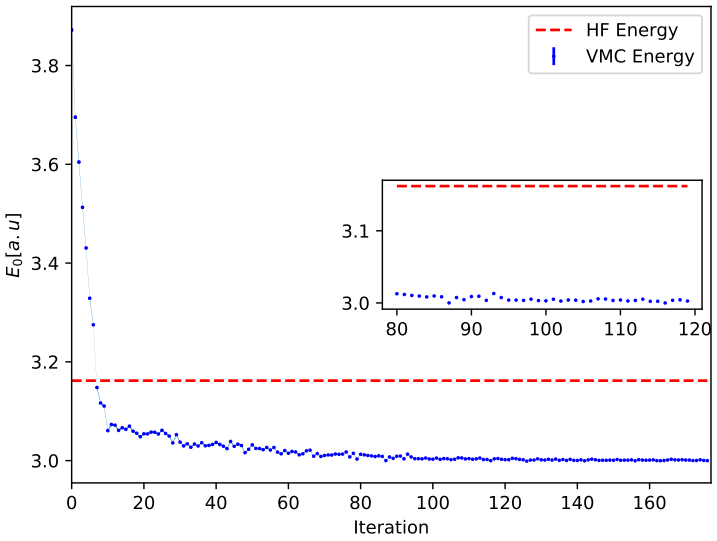
Running the codes

You can find the codes for the simple two-electron case at the Github repository https://github.com/mhjensenseminars/MachineLearningTalk/tree/master/doc/Programs/MLcpp/src. Python codes to come, only c++ as of now.

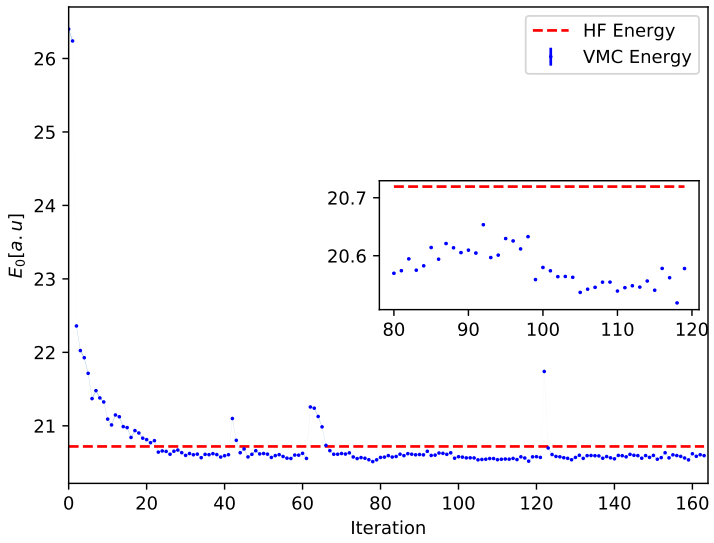
The trial wave function is based on the product of a Slater determinant with Gaussian orbitals, a simple Jastrow factor $\exp(r_{ij})$ and the reduced Boltzmann machines.

The Broyden-Fletcher-Goldfarb-Shanno algorithm was used for the minimization. We used 14 hidden nodes in the calculations below.

Energy as function of iterations, $N = 2$ electrons



Energy as function of iterations, $N = 6$ electrons



Conclusions and where do we stand

- A simple extension of the work of G. Carleo and M. Troyer, Science **355**, Issue 6325, pp. 602-606 (2017) gives excellent results for two-electron systems as well as good agreement with standard VMC calculations for $N = 6$ and $N = 12$ electrons.
- Minimization problem can be tricky.
- Anti-symmetry dealt with multiplying the trail wave function with an optimized Slater determinant.
- To come: Analysis of wave function from ML and compare with diffusion and Variational Monte Carlo calculations as well as the analytical results of Taut for the two-electron case.
- Extend to more fermions. How do we deal with the antisymmetry of the multi-fermion wave function?

1. Here we used standard Hartree-Fock theory to define an optimal Slater determinant. Takes care of the antisymmetry. What about constructing an anti-symmetrized network function?
2. Use thereafter ML to determine the correlated part of the wafe function (including a standard Jastrow factor).
3. Test this for multi-fermion systems and compare with other many-body methods.

- Can we use ML to find out which correlations are relevant and thereby diminish the dimensionality problem in say CC or SRG theories?

Additional material

Kullback-Leibler relative entropy

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

$$\mathrm{KL}(p||q) = \int_{-\infty}^{\infty} p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} d\boldsymbol{x}. \tag{34}$$

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

$$\begin{aligned}
\mathrm{KL}(f(\boldsymbol{x})||p(\boldsymbol{x}|\boldsymbol{\theta})) &= \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log \frac{f(\boldsymbol{x})}{p(\boldsymbol{x}|\boldsymbol{\theta})} d\boldsymbol{x} && (35)\\
&= \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log f(\boldsymbol{x}) d\boldsymbol{x} - \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log p(\boldsymbol{x}|\boldsymbol{\theta}) d\boldsymbol{x} && (36)\\
&= \langle \log f(\boldsymbol{x}) \rangle_{f(\boldsymbol{x})} - \langle \log p(\boldsymbol{x}|\boldsymbol{\theta}) \rangle_{f(\boldsymbol{x})} && (37)\\
&= \langle \log f(\boldsymbol{x}) \rangle_{data} + \langle E(\boldsymbol{x}) \rangle_{data} + \log Z && (38)\\
&= \langle \log f(\boldsymbol{x}) \rangle_{data} + \mathcal{C}_{LL}. && (39)
\end{aligned}$$

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

Optimizing the cost function

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

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

$$\langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \rangle_{model} = \int p(\boldsymbol{x}|\boldsymbol{\theta}) \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} d\boldsymbol{x} = -\frac{\partial \log Z(\theta_i)}{\partial \theta_i}. \tag{40}$$

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

Setting up for gradient descent calculations

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

$$\frac{\partial \mathcal{C}_{LL}}{\partial \theta_i} = \langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \rangle_{data} + \frac{\partial \log Z(\theta_i)}{\partial \theta_i} \tag{41}$$

$$= \langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \rangle_{data} - \langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \rangle_{model} \tag{42}$$

$$\tag{43}$$

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

More interpretations

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

As in supervised machine learning problems, the goal is also here to perform well on **unseen** data, that is to have good generalization from the training data.

The distribution $f(x)$ we approximate is not the **true** distribution we wish to estimate, it is limited to the training data. Hence, in unsupervised training as well it is important to prevent overfitting to the training data. Thus it is common to add regularizers to the cost function in the same manner as we discussed for say linear regression.