Week 18 May 1-5: Neural networks and project 2

Morten Hjorth-Jensen Email morten.hjorth-jensen@fys.uio.no[1,2]

[1]Department of Physics and Center fo Computing in Science Education, University of Oslo, Oslo, Norway
[2]Department of Physics and Astronomy and Facility for Rare Ion Beams, Michigan State University, East Lansing, Michigan, USA

May 4, 2023

Overview of week 18, May 1-5

- Neural networks and project 2
  1. Discussions of neural network and solution of project 2
  2. Discussion of codes. See also code for Boltzmann machine included in the notes here.

Neural networks and Boltzmann machines (BMs)

We have introduced Boltzmann machines as generative models where we train a neural network with a probability distribution, which in our case is the Boltzmann distribution. This distribution is used as an ansatz for the trial wave function to be used in a Variational Monte Carlo calculation

An important benefit of using BMs is that we can reuse or VMC codes with the Metropolis family of sampling methods and our basic Markov-chain Monte Carlo machinery.

Cost function

But we do not optmize the maximum likelihod (log). The function we optimize is the expectation value of the energy which depends on the parameters that define the Boltzmann distribution. This is where we deviate from what is common in machine learning. Rather than defining a cost function based on some dataset, our cost function is the energy of the quantum mechanical system.

From the variational principle we know that minizing this energy should lead to the ground state wavefunction.

That is

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

Irrespective of whether we use Boltzmann machines or neural networks, this is the function we end up optmizing.

Calculating gradients

And as was done in our VMC code, the gradient we need to evaluate is

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

where θ_i are the biases and weights of a neural network.

In order to exploit that the trial wave function is often on an exponential form, it is convenient to use that

$$\frac{1}{\Psi}\frac{\partial \Psi}{\partial \theta_i} = \frac{\partial \ln \Psi}{\partial \theta_i}.$$

Python version for the two non-interacting particles

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



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

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

    for ih in range(NumberHidden):
        Psi2 *= (1.0 + np.exp(Q[ih]))
    Psi1 = np.exp(-Psi1/(2*sig2))

    return Psi1*Psi2

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

    Q = Qfac(r,b,w)

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

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

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

                locenergy += 1/sqrt(distance)

    return locenergy

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

    sigma=1.0
    sig2 = sigma**2

    Q = Qfac(r,b,w)

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

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

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

    return  WfDer

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

    sigma=1.0
    sig2 = sigma**2
    qforce = np.zeros((NumberParticles,Dimension), np.double)
    sum1 = np.zeros((NumberParticles,Dimension), np.double)

    Q = Qfac(r,b,w)

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

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

    return qforce

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

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

    Q = b + temp

    return Q

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

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

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

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


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

    #Loop over MC MCcycles
    for MCcycle in range(NumberMCcycles):
        #Trial position moving one particle at the time
        for i in range(NumberParticles):
            for j in range(Dimension):
                PositionNew[i,j] = PositionOld[i,j]+normalvariate(0.0,1.0)*sqrt(TimeStep)+\
                                       QuantumForceOld[i,j]*TimeStep*D
            wfnew = WaveFunction(PositionNew,a,b,w)
            QuantumForceNew = QuantumForce(PositionNew,a,b,w)

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

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

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

        energy += DeltaE

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

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


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

interaction=False
```

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

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


    iter += 1

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

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

Neural networks

To implement a standard neural network (feed forward NN), the function we will optimize is the same as for Boltzmann machines, except that now the ansatz for the trial wave function is the neural network itself, with its parameters and architecture given by

1. Number of hidden layers and nodes in each layer;
2. Actitvation functions for the various nodes;
3. Hyperparamters of the type from an l_2-norm or l_1-norm or a mix of various norms;

4. Gradient algorithms for optimization with various ways to optimize the **learning rate**;
5. Back propagation algorithm and automatic differentiation for computing the updates of the various parameters θ_i;
6. The cost/loss function to be optimized;

In this lecture we will review briefly the structure of neural networks. These notes are taken from the course on Machine Learning FYS-STK4155.

In particular we will review the material from weeks 40 and 41.

Developing a code for Neural Networks

We have seen that Boltzmann machines are straightforward to implement, however due to the ansatz made in the construction of the so-called energy function, they are at the end less flexible if we need to change the way we describe the visible and hidden layers. Computing the final marginal probability which defines the trial wave function grows extremely complicated with other ways of defining the variables of the hidden and visible layers.

Neural networks as alternatives

A neural network on the other hand offers much more flexibility in the training. The price we have to pay is however an additional computational cost due to the many more parameters to train (hidden layers and nodes) and the implementation of the back propagation algorithm and automatic differentiation.

Basic elements in codes

We consider an unpolarized gas of fermions in $d = 3$ dimensions, whose dynamics is modeled by the nonrelativistic Hamiltonian

$$H = -\frac{\hbar^2}{2m}\sum_i \nabla_i^2 + \sum_{ij} v_{ij}\,, \tag{3}$$

where the attractive two-body interaction could be a Coulomb interaction or any other type of interaction. A popular example is

$$v_{ij} = -2v_0\frac{\hbar^2}{m}\frac{\mu^2}{\cosh^2(\mu r_{ij})}, \tag{4}$$

which acts only between opposite-spin pairs. We will use $\boldsymbol{r}_i \in \mathbb{R}^d$ and $s_i \in \{-1, 1\}$ to denote the spatial coordinates and spin projection on the z-axis of the i-th particle. The parameters v_0 and μ tune the scattering length a and effective range r_e of the potential. The interaction above, called Pöschl-Teller, has been employed in several previous QMC calculation. It provides an analytic solution of the two-body problem, and the unitary limit corresponding to the zero-energy ground state between two particles is with $v_0 = 1$ and $r_e = 2/\mu$.

Neural-network quantum states

The codes we link to are tailored to problems with fermions only.

We can solve the Schrödinger equation associated with the above Hamiltonian using various different families of neural-network quantum states that respect periodic boundary conditions by construction. All ans"atzes have the general form

$$\Psi(X) = e^{J(X)}\Phi(X), \tag{5}$$

where the Jastrow correlator $J(X)$ is symmetric under particle exchange and $\Phi(X)$ is antisymmetric. In the above equation, we used $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ with $\boldsymbol{x}_i = (\boldsymbol{r}_i, s_i)$ to compactly represent the set of all single-particle positions and spins.

The antisymmetric part of the Slater-Jastrow (SJ) family of states can be written as

$$\Phi_{SJ}(X) = \det \begin{bmatrix} \phi_1(\boldsymbol{x}_1) & \phi_1(\boldsymbol{x}_2) & \cdots & \phi_1(\boldsymbol{x}_N) \\ \phi_2(\boldsymbol{x}_1) & \phi_2(\boldsymbol{x}_2) & \cdots & \phi_2(\boldsymbol{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_N(\boldsymbol{x}_1) & \phi_N(\boldsymbol{x}_2) & \cdots & \phi_N(\boldsymbol{x}_N) \end{bmatrix}. \tag{6}$$

In the fixed-node approximation, we take the single-particle states to be products of spin eigenstates and plane wave (PW) orbitals

$$\phi_i^{PW}(\boldsymbol{x}_j) = \mathrm{e}^{i\boldsymbol{k}_i\cdot\boldsymbol{r}_j}\delta_{s_i,s_j}\,, \tag{7}$$

with discrete momenta $\boldsymbol{k} = 2\pi\boldsymbol{n}/L$, $\boldsymbol{n} \in \mathbb{Z}^d$, and spin states $s_i \in \{-1, 1\}$.

The nodal structure of the above Slater determinant can be improved by means of backflow (BF) transformations. For brevity, we will use the notation $\tilde{\boldsymbol{x}}_i \equiv (\boldsymbol{x}_i, \{\boldsymbol{x}_j\}_{j\neq i})$ to indicate dependency on a specific particle i, and permutation invariance over all other particles $j \neq i$.

Inputs to neural networks

It is possible to implement the aforementioned neural quantum state (NQS) using X as direct inputs to the appropriate NN, but it is advantageous to devise new inputs that already capture a large portion of the correlations. One approach is to employ a permutation-equivariant message-passing neural network (MPNN) to iteratively build correlations into new one-body and two-body features from the original "visible" features. The visible features are chosen to be

$$\boldsymbol{v}_i = (s_i)\,, \tag{8}$$

$$\boldsymbol{v}_{ij} = (\boldsymbol{r}_{ij}, \|\boldsymbol{r}_{ij}\|, s_i s_j)\,, \tag{9}$$

with the separation vectors $\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j$ and distances $\|\boldsymbol{r}_{ij}\| = r_{ij}$ replaced by their L-periodic surrogates

$$\boldsymbol{r}_{ij} \mapsto (\cos(2\pi\boldsymbol{r}_{ij}/L), \sin(2\pi\boldsymbol{r}_{ij}/L)) \ , \tag{10}$$

$$\|\boldsymbol{r}_{ij}\| \mapsto \|\sin(\pi\boldsymbol{r}_{ij}/L)\|. \tag{11}$$

More information

Note that we have excluded explicit dependence on the particle positions $\boldsymbol{r}_i$ in the visible one-body features, thereby enforcing translational invariance in the new features. Linear transformations are applied to and concatenated with each feature to obtain the initial hidden features

$$\boldsymbol{h}_i^{(0)} = (\boldsymbol{v}_i, A\boldsymbol{v}_i), \tag{12}$$

$$\boldsymbol{h}_{ij}^{(0)} = (\boldsymbol{v}_{ij}, B\boldsymbol{v}_{ij}). \tag{13}$$

Codes on neural networks applied to many-body problems

In addition to the above mentioned codes we have several popular available code sets

1. Neural network quantum states and NetKet

2. Ferminet