Time-independent Schrödinger Equation and Convolutional Neural Network

Shaohao Chen[1]

[1]Research Computing Services, Boston University, Boston, MA 02215

April 30, 2018

Abstract

In this work, I show an example of quantum machine learning in the sense that classical learning algorithms are applied to quantum systems. I apply a deep neural network model, more specifically a convolutional neural network (CNN) model, to deal with the time-independent Schrödinger Equation for a two-dimensional helium atom. I build an effective CNN model to map the spatial structure of the input potential (image) to the ground-state energy. The mapping function is learned from data. The convergence of training error and test error have been obtained. The predicted energies for test cases are very close to the true energies. The computer code is well scalable on high-performance computing resources.

1 Introduction

In the past decades, there have been fruitful research results in the field of machine learning. Lots of effective statistical methods for regression and classification have been developed[1]. Neural network is one of them. In recent years, deep learning algorithms based on deep (multi-layer) neural network (DNN) become polular, due to the following two facts: (1) the rapid increase of speed of computer processors (including CPU, GPU and TPU). (2) the availability of big data boosted from internet technique.

Due to rapid development of machine learning and deep learning algorithms, many artificial intelligence (AI) techniques have been invented. The application of AI techniques has made significant influence on many fields, such as image classification, handwriting recognition, computer vision, speech recognition, bioinformatics, medical diagnosis, playing Go game, predicting sports game, and financial analysis. Applications of machine learning and deep learning methods are very broad nowadays. This work is related to their applications in the following fields: quantum physics, atomic and molecular physics, and partial differential equation (PDE) methods.

Quantum machine learning, an emerging interdisciplinary research area at the intersection of quantum physics and machine learning has drawn broad interests recently [2]. There are three different approaches to combine the disciplines of quantum physics and machine learning: (1) use quantum learning algorithms to deal with classical systems; (2) use classical learning algorithms to deal with quantum systems; (3) use quantum learning algorithms to deal with quantum systems. In this work, I will show an example of quantum machine learning in the sense that classical learning algorithms are applied to quantum systems.

Machine learning methods have also been applied in atomic and molecular physics [3, 4, 5]. The central idea is to build a map between physical features of the systems and the concerned physical quantities, then fit the necessary parameters by statistical methods. Instead of mapping physical features, deep neural network methods are more focus on spatial structure of the systems. deep neural network methods have been applied in atomic and molecular physics too [6, 7, 8].

It has been shown that machine learning algorithms can be applied to solve partial differential equations (PDE) [9, 10]. A deep neural network scheme has been proposed to solve the time-independent Schrödinger Equation [8], which is at the heart of quantum mechanics. The central idea is to map spatial features of the input potential function to energies. The true mapping function is complicated. It is known that a sufficiently large artificial neural network can approximate any continuous mapping [11, 12], but the cost of optimizing such a network can be very expensive. Fortunately, deep neural network models are parallelizable and scalable and and thus can benefit from state-of-the-art high-performance computing techniques.

In this project, I apply a deep neural network model, more specifically a convolutional neural network (CNN) model, to deal with the time-independent Schrödinger Equation for a two-dimensional (2D) helium atom. The central idea was proposed in reference [8]. CNN, a kind of DNN, is focus on local spatial structure of input data. I will build a CNN model to map the spatial structure of the input potential function to the the ground-state energy. I will use this model to explore a prototype system in atomic physics, i.e. the helium atom with one nucleus and two electrons, which is an ideal three-body system.

2 Theory and Computation

2.1 Schrödinger Equation and Deep Learning

The time-independent Schrödinger Equation can be written as,

$$H\Psi = \left(-\frac{1}{2}\nabla^2 + V\right)\Psi = \varepsilon\Psi \tag{1}$$

where H is the Hamiltonian, $-\frac{1}{2}\nabla^2$ is the kinetic-energy operator (∇^2 is the Laplacian), V is the potential function, ε is the eigen energy and Ψ is the wavefunction. Expanded on a basis of Hilbert space, H is a matrix and Ψ is a vector. In traditional methods, the Hamiltonian matrix H is diagonalized to obtain the eigen energies (eigen values) and wavefunctions (eigen vectors). Here I try to obtain the energies using a deep learning method instead. Following is a brief description of the theory.

Potential functions vary for different quantum systems. Once a potential function is given, the energies of the system are fixed. Therefore, the energy ε is a function of potential V,

$$\varepsilon = f(V), \tag{2}$$

Therefore, to solve the time-independent Schrödinger Equation can be considered in the following scenarial: given an input — potential function V, we need to find the response — energies ε. Usually the ground state and a few low-lying excited stats (corresponding to several smallest eigen values) are interesting. The true function f that maps V to ε is complicated. Fortunately it has been shown that complicated functions can be approached by deep (multilayer) neural network models, so I intend to apply a DNN model to find out the approach function $\hat{f}$, and thus predict the energy $\hat{\varepsilon}$ for the given input V,

$$\hat{\varepsilon} = \hat{f}(V). \tag{3}$$

Due to its convolutional features, convolutional neural network is known to be efficient to recognize 2D images. In reference [8], four kinds of 2D potential functions have been investigated by a CNN model. In this project, I focus on another kind of 2D potential, that is, the 2D soft Coulomb potential for helium atom,

$$V(z_1, z_2) = -\frac{2}{\sqrt{z_1^2 + a^2}} - \frac{2}{\sqrt{z_2^2 + a^2}} + \frac{1}{\sqrt{(z_1 - z_2)^2 + b^2}}. \tag{4}$$

The first two terms are the Coulomb interactions between the nucleus and the two electrons, where the number 2 is the nucleus charge of helium atom and the negative sign means that the charge of the nucleus and the charges of the electrons have opposite signs. The third term is the electron-electron interaction, where the positive sign means that the charges of the two electrons have the same sign. z_1 and z_2 are orthogonal Cartesian coordinates of the two electrons of helium atom. a and b are two nonzero soft-coulomb parameters. a^2 is introduced to eliminate the electron-nuleus Coulomb singularity at $z_1 = 0$ or $z_2 = 0$, and b^2 is introduced to eliminate the electron-electron Coulomb singularity at $z_1 = z_2$. Atomic unit (denoted as $a.u.$) is adopted throughout this paper. This soft Coulomb potential is widely used in atomic physics and optical physics.

Expanded on discrete grids indexed by all possible values of z_1 and z_2, the potential function is represented by a vector. The kinetic energy operator (the Laplacian) is calculated by three-points finite difference formula, and thus is represented by a symmetric tridiagonal matrix. Adding the elements of the potential vector to the diagonal elements of the kenetic-energy matrix, we obtain the Hamiltonian matrix, which is also a symmetric tridiagonal matrix. Thus the eigen energies are real scalars. Our goal is to predict several smallest eigen energies (real scalars) based on the input of a 2D potential function (represented by a vector). This process is very similar to the process of 2D image recognition, where several numbers (e.g. representing cat, dog, airplane, car, etc.) are predicted based on the input of a 2D image (where pixel intensities are represented by a vector). It has been shown in many cases that CNN models due to the convolutional features are efficient in recognizing 2D images, so it is reasonable to apply a CNN model to solve our problem.

In the following two subsections, I will describe details of the computations.

2.2 Dataset generation

I vary the soft-coulomb parameters a and b to obtain a number (N) of 2D-helium soft-coulomb potential $\{V_i\}_{i=1}^N$ using Eq (4). The dimension p of V_i is equal to the product of the grid numbers of z_1 and z_2, i.e. $p = n_{z_1} n_{z_2}$. For each potential V_i, I solve the eigen problem (1) using traditional linear algebra methods. I use the *linalg.eigvals* function in Numpy library to diagonalize the input Hamiltonian matrix and obtain the output eigen energies. The Numpy library is linked to Intel MKL and is parallelized by multithreading technique. The computer code is attached in appendix A.

The eigen energies $\{\varepsilon_{ik}\}_{k=0}^K$ will be treated as the true energies in the following. Here K and k respectively denote the total number and the index of required eigen states, e.g. $k = 0$ for the ground state, $k = 1$ for the first excited state, and so on. Finally I obtain N data points $\{V_i, \varepsilon_{ik}\}_{i=1}^N$ for each eigen energy.

Here are more details for choosing parameters. To make the problem simple and quickly converged, I only predict the ground state ($k = 0$ and $K = 1$) in the following, so the final output is

one scalar. To make the computational efforts less and thus obtain the results within a reasonable time, the grids is limited in a small range: $-2a.u. < z_1 < 2a.u.$ and $-2a.u. < z_2 < 2a.u.$. The grid stride is $dz_1 = dz_2 = 0.1$. This makes the dimensions of V_i: $p = 40 \times 40 = 1600$. I choose $a^2 = 0.5$ and $b^2 = 0.0399$ to obtain $\varepsilon = 2.78a.u.$, which is close to real ground-state energy observed by experiments. Then choose 200 uniformly distributed a^2 in the the range of $[0.48, 0.52]$ and 200 uniformly distributed b^2 in the range of $[0.379, 0.419]$. This makes total $200 \times 200 = 40000$ sample points. I will use 90% as training points and 10% as test points, so there are $N = 36000$ training points and $N_0 = 4000$ test points. Here the number of training points is much larger than the dimension of the input vector (or number of input features), i.e. $N >> p$, and thus the data set is sufficiently large.

2.3 Convolutoinal Neural Network

From the previous subsection we know that the true energy is known, and we want to predict the energy from data using a learning algorithm, so this is a supervised learning problem. As discussed above, we will use a deep convolutional neural network to map the input potential (vector) to the energies (scalars).

2.3.1 Neural Network Layers

To build the deep neural network, the following three types of layers will be employed.

I. Fully connected layer: In a fully connected layer, each point is connected to all points of the previous layer.

First apply a linear model to the input, then use an activation function to bring in nonlinearity. The activation function is chosen as the so-called rectified linear unit (ReLU). The definition of ReLU (denoted as $Relu$) is the maximum of the input value and 0,

$$Relu(t) = max(0, t). \tag{5}$$

The length-q output vector y can be written as a function of the length-p input vector x,

$$y = Relu(b + Wx) \tag{6}$$

where W is a $q \times p$ matrix (called weights) and b is a length-q vector (called bias). W and b are regression parameters, which will be obtained by regression methods described later. A fully connected layer requires $q(p + 1)$ parameters. Usually p and q are large numbers, and thus the number of parameters are very large.

II. Convolutional Layer: In a convolutional layer, each point is connected only to the points in a local region of the previous layer. Here is a description of a convolutional Layer.

First the length-p input vector x is converted to a 2D image X sized $\sqrt{p} \times \sqrt{p}$. A weight matrix $W_m(i, j)$ is multiplied to a square local region $\widetilde{X}(i, j)$ of the image X. Here i and j are the row and column indexes of the image X. Both W_m and $\widetilde{X}$ are matrices sized $s \times s$ and are centered at the (i, j) point of the image. Here s is an odd number and $s << \sqrt{p}$. Then all the s^2 elements of the product $W_m\widetilde{X}$ is summed up, and a bias b_m is added too. The square matrix W_m is called a filter. The filter moves at a stride r to cover all points of the input image. The process described above is known as convolution integral in calculus. By applying filters to local regions of a 2D image, the convolutional layer captures local spacial structure of the image. Then apply an activation function ReLU to the result to bring in nonlinearity. The index m denotes a channel. To sufficiently capture

local features, usually multiple channels ($m = 1, 2, ..., M$) are applied, and thus multiple output images are generated, e.g. the m-th channel generates an output image Y_m.

Summarizing the above description, the element of the i'-th row and the j'-th column of the m-th output image can be written as,

$$Y_m(i', j') = Relu\left[b_m + \sum_S W_m(i, j)\widetilde{X}(i, j)\right] \tag{7}$$

where $W_m(i, j)$ is a filter of the m-th channel, $\widetilde{X}(i, j)$ is a local region of the input image X, S means the area of the local region, and b_m is the bias. Both $W_m(i, j)$ and $\widetilde{X}(i, j)$ are matrices sized $s \times s$ and centered at the (i, j) point of the input image. The input index (i, j) can be mapped to output index (i', j'), and the input size $\sqrt{p} \times \sqrt{p}$ can be mapped to the output size $\sqrt{p'} \times \sqrt{p'}$ too. Especially, if the stride of the filter is equal to 1, we have $i' = i$, $j' = j$ and $p' = p$.

In order to apply a filter to the edge points of the image, $(s-1)/2$ rows and columns with zero elements are added beyond the boundary of the original image. This process is called padding. If a convolutional Layer is followed by a fully-connected layer, all M sized $\sqrt{p'} \times \sqrt{p'}$ output image is first converted to a vector sized $p'M$, then it is used as an input of the following layer. This process is called flattening.

A convolutional layer requires $(s^2 + 1)mp/r^2$ parameters. Usually the number of filters m and the filter size s are much smaller than the size of the output vector q in a fully-connected layer, so the number of parameters in a convolutional layer is smaller than that in a fully-connected layer, and one can increase the stride r to further reduce the number of parameters.

III. Maximum Pooling Layer: In a maximum pooling layer, only the maximum value of a local region in the previous layer is kept.

Usually a convolutional layer is followed by a pooling layer. Maximum pooling makes sense because main features of a local region are represented by its maximum value. A pooling region has a size $s_p \times s_p$ and a stride r_p. This layer is introduced to reduce the size of the output image and the number of regression parameters, and thus reduce computational efforts. Usually the number of parameters are reduced by a factor of r_p^2.

2.3.2 Multi-layer Neural Network

Using the three types of layers described above, I build the entire multi-layer neural network as following. The architecture diagram is shown in Fig. 1.

Layer 1: The input 2D image sized 40×40.

Layer 2: A convolutional layer with 32 channels. The filter size is 3×3 and the stride is 1. The output is 32 images each sized 40×40.

Layer 3: A pooling layer. The pool size is 2×2 and the stride is 2. The output is 32 vectors each sized 20×20.

Layer 4: A convolutional layer with 16 channels. The filter size is 3×3 and the stride is 1. The output is 16 images each sized 20×20.

Layer 5: A pooling layer. The pool size is 2×2 and the stride is 2. The output is 16 images each sized 10×10.

Layer 6: A convolutional layer with 16 channels. The filter size is 3×3 and the stride is 1. The output is 16 images each sized 10×10.

Layer 7: A pooling layer. The pool size is 2×2 and the stride is 2. The output is 16 images each sized 5×5.

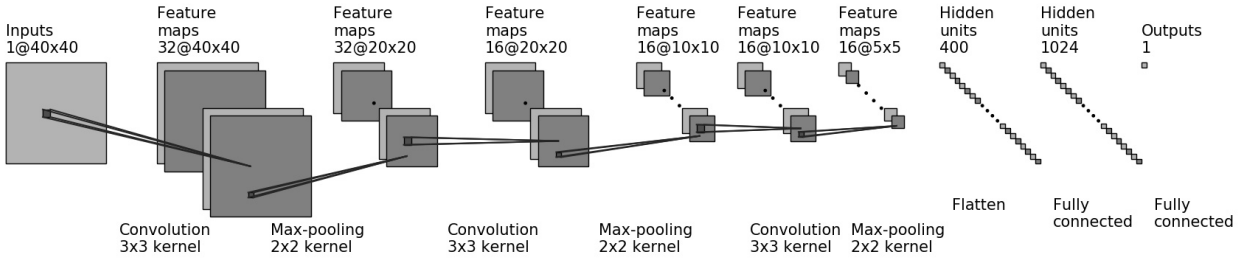

Figure 1: Architecture diagram of convolutional neural network.

Layer 8: A fully connected layer. The input 2D image from the previous layer is converted to a vector sized $5 \times 5 \times 16 = 400$, and then it is fully connected to the output vector sized 1024.

Layer 9: A fully connected layer. The output is a scalar sized 1.

The initial input is a 2D potential V, and the final output is the predicted energy $\hat{\varepsilon}$. The initial weights are normal distribution with mean 0 and standard derivation 0.1. i.e. $W \sim N(0, 0.1)$. The initial bias is a constant $b = 0.1$.

2.3.3 Training

Given a data set $\{V_i, \varepsilon_{ik}\}_{i=1}^{N}$, the parameters W and b used in the neural network are obtained by a regression procedure that minimizes a loss function. This process is called training.

The loss function is defined as the mean square error between the true energy and the predicted energy,

$$L = \frac{1}{N} \sum_{i=1}^{N} (\varepsilon_i - \hat{\varepsilon_i})^2 \tag{8}$$

This is also the training error. I use the Adadelta gradient descent algorithm [13] to minimize the loss function. Backpropagation method is employed. The learning rate is 0.001. The regression parameters W and b are adjusted towards the direction of decent gradient of the loss function in each training step until convergence is obtained.

The training data set is divided into 36 batches. The batch size is 1000. Each batch of data is randomly picked up in the original training set, then it is used to run one training step. I found that after 500 epochs (500 times through all the training examples), the loss function no longer decreased significantly. Then the regression parameters are fixed, and the test error is estimated using the test data set $\{V_i, \varepsilon_{ik}\}_{i=1}^{N_0}$, where $N_0 = 4000$. Note that the test data set is not exposed to the training process.

2.3.4 Implementation of Computation

I use Tensorflow [14] in Python interface to build the computer program. The computer code is attached in appendix B. It is convenient to build multiplayer CNN using Tensorflow functions. For example, building a convolutional layer is as simple as writing one line of code by using functions $tf.nn.conv2d$ and $tf.nn.relu$. The computation behind the scene of the code has been described in section 2.3.1. The code is parallelized and accelerated on Graphics processing unit (GPU). The program ran on BU Shared Computing Cluster (SCC).

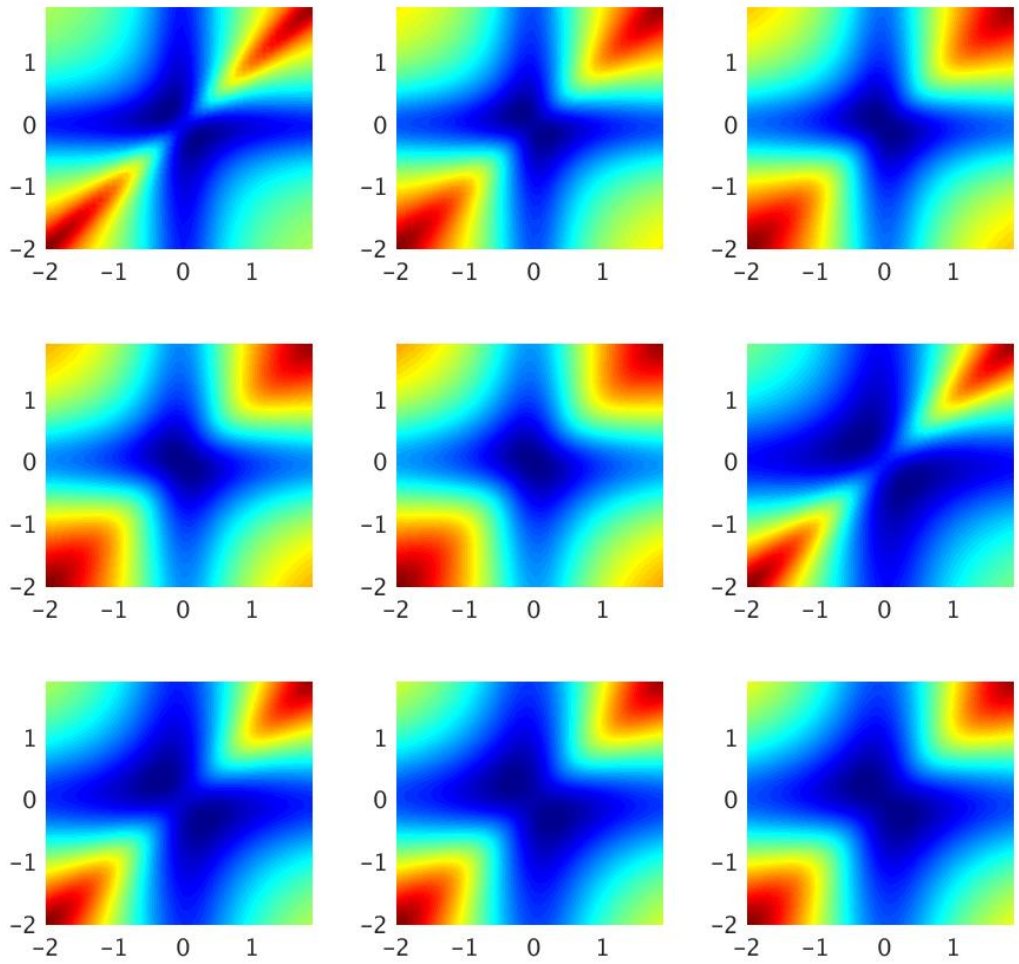

Figure 2: 2D potentials as a function of coordinates of the two electrons z_1 and z_2. Blue means small value and red means large value. Subplots are for 9 different Coulomb parameters a and b. These images show similar but different spacial structures. There is a symmetry to the two coordinates z_1 and z_2. The CNN model captures the spacial structures of the images. Once the model has been "trained" on a large number of labeled images, it "learns" the map between the image (potential) and the label (energy). Then the model can predict the label (energy) for a new image (potential) with a similar spacial structure.

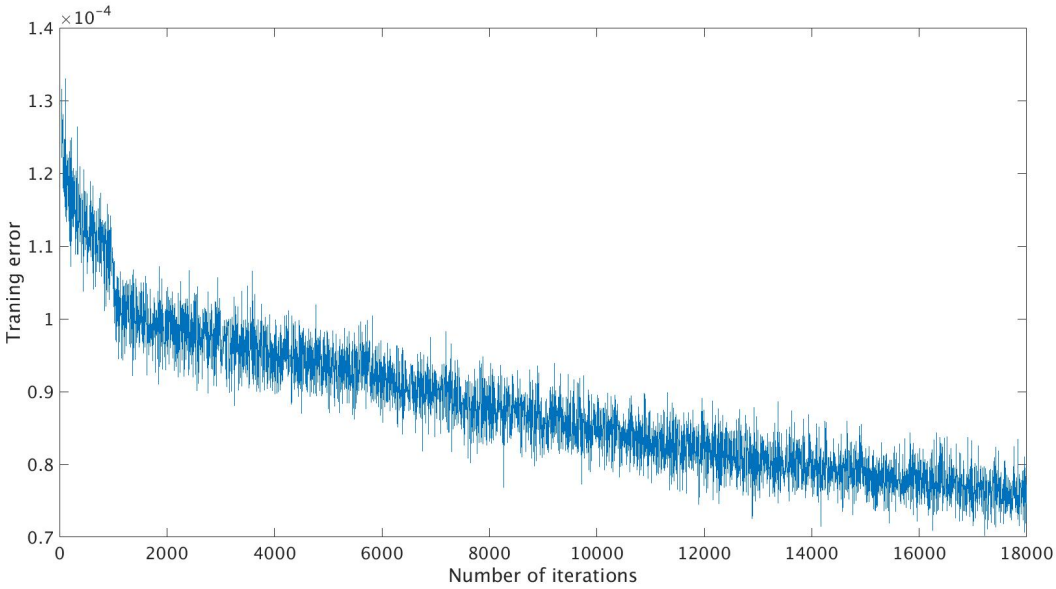

Figure 3: Training error as a function of iteration number. 500 epochs (500 times through all the training examples) is reached.

3 Results and Discussions

Fig. 2 shows 2D-helium potentials as a function of the two electron coordinates z_1 and z_2 for 9 different sets of soft-coulomb parameters a and b. The primary spatial structure is the symmetry to the two coordinates z_1 and z_2. There are other main spacial structures shown in two regions. One region is around the origin ($z_1 = 0$ and $z_2 = 0$). Here the two negative terms of nucleus-electron interactions are large, so the value of the potential is small. The positive term of electron-electron interaction makes the value in this region larger when it becomes dominant. The other region is around the down-left corner and the up-right corner ($z_1 = z_2$, $z_1 >> 0$ and $z_2 >> 0$). Here the electron-electron interaction is large, while nucleus-electron interaction is relatively small, so the value of the potential is large. The detail spacial structure varies as the soft-coulomb parameters varies. The CNN model captures the spacial structure of a potential. Once the model has been "trained" on a large number of labeled images, it "learns" the map between the image (potential) and the label (energy). Then the model can predict the label (energy) for a new image (potential) with a similar spacial structure.

Fig. 3 shows the convergence of training error. The training error no longer decreased significantly after 500 epochs. This indicates convergence of the computation. The final training error is 7.48×10^{-5}. The test error on a new data set with 4000 data points is 7.61×10^{-5}. The test error is very close to the training error, so the model is not overfitting.

Fig. 4 shows predicted energies versus true energies. Four different test data sets are used. Each set has 1000 data points. In all of the four cases, all the 1000 predicted energies are close to the true energies. As a result, the points are located in the vicinity of the 45 degree diagonal line.

Tab. 1 shows some selected predicted energies, the corresponding true energies, and the percentage difference. The energies are in atomic unit. The percentage differences are less than 0.4%.

The small difference between the predicted energies and the true energies indicates that the CNN

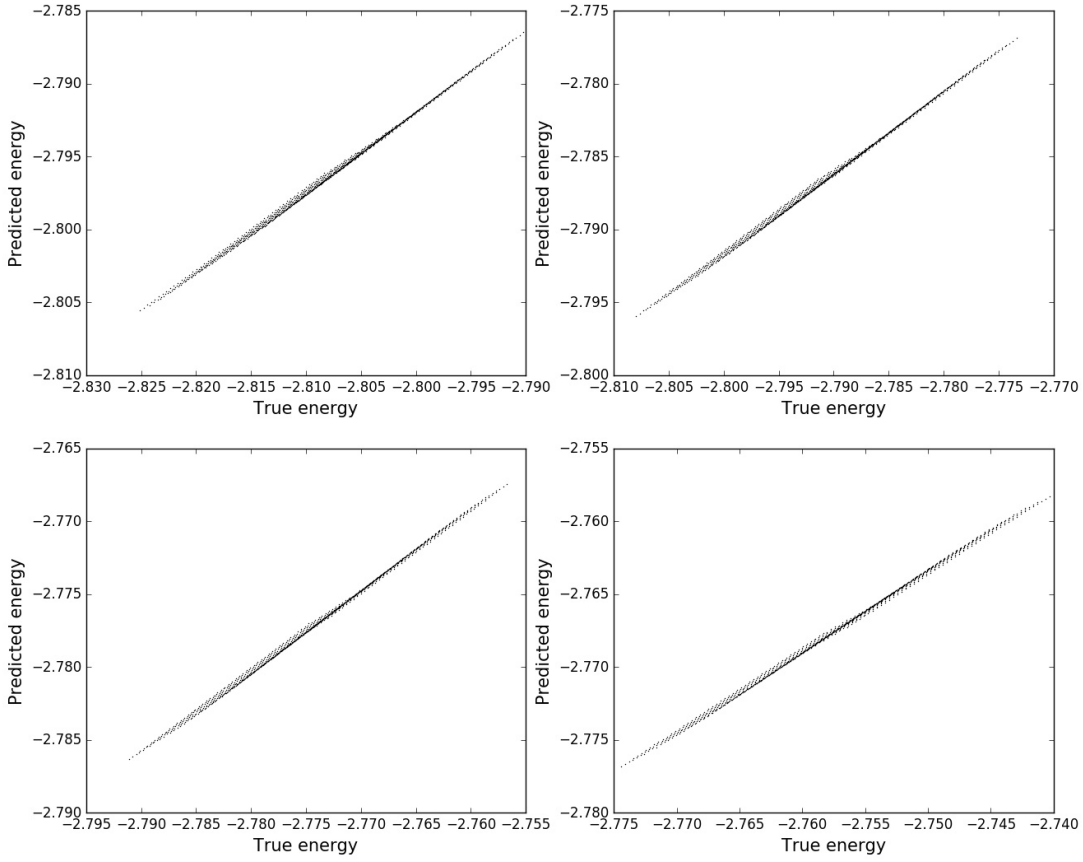

Figure 4: Predicted energies vs. true energies. Subplots are for 4 different test data sets each sized 1000. Points are located in the vicinity of the 45 degree diagonal line, indicating that the predicted energies are close to the true energies.

True energies	Predicted energies	Percentage difference
-2.807	-2.796	0.39%
-2.800	-2.791	0.30%
-2.791	-2.787	0.18%
-2.775	-2.777	-0.08%
-2.758	-2.768	-0.34%

Table 1: Selected predicted energies, the corresponding true energies, and the percentage difference.

module works well for the problem. An important question is to ask why deep neural network works so well. A general interpretation is that any continuous function can be approached by a multi-layer neural network with a sufficiently large number of parameters [11, 12], but the computational cost is expensive. In deep neural network, the mapping scheme is complicated and a large number of parameters are involved, so it is hard to understand the reason why deep neural network really works. There have been some approaches to understand it [15], but further research is needed to answer this question.

CNN is effective in capturing spatial structure of the input image, so CNN requires less parameters than fully-connected neural network does. In particular, the 2D helium potential is symmetric to z_1 and z_2. This symmetry in space makes the mapping function less complicated, and thus this symmetric image requires less layers and parameters in the model, compared to a random (asymmetric) one. As a consequence, a good prediction is obtained by employing only 7 hidden layers, and the number of employed parameters is much less than the number of all possible mapping functions [15].

The computer program has run on a 16-core Intel Sandybridge CPU and an Nvidia P100 GPU separately. The computation on GPU is around 30 times faster than that on the CPU. The number of CUDA cores of the GPU is more than 3500, so it is powerful for computations that are ideally parallelizable. Since the training process is highly parapplelizable, it is well scalable on GPU. The well scalability allows us to investigate on larger problems with more parameters.

4 Conclusion and Prospect

In this work, I built a CNN model and used it deal with the time-independent Schrödinger equation for a 2D helium atom. The CNN model effectively captures the spacial structure of given potentials, and precisely predicts ground-state energies. The convergence of training error have been obtained. The test error is very close to the training error, so there is no overfitting. The predicted energies for test cases are very close to the true energies, indicating that the CNN model works well for the problem. The computer code is well scalable on high-performance computing resources.

Finally I prospect possible future work. (1) The current grid size is not big enough to capture interesting physics of helium atom. It is necessary to increase the grid size in future work. This will increase the size of the input vector, and consequently, the size of training data set, the number of CNN layers, and the number of regression parameters need to be increased considerably, So the computational effort will be increased considerably too. (2) Use other types of learning algorithms such as kernel ridge regression and random forest to investigate the problem. It will be interesting to see how these two models are compared to CNN. (3) Use the CNN model to predict other interesting physical quantities such as the first excited state energy and the kinetic energy. This

is straightforward to do under the current framework. To predict the wavefunction (eigen vector) is also interesting, since most physical quantities can be computed based on wavefuction. But this will substantially increase the size of the output vector and thus makes the learning process much more difficult. (4) Investigate on other quantum systems, such as single-active-electron atoms, 6D Helium atom, and 3D hydrogen molecule. These systems are interesting in atomic physics. (5) Apply the CNN model to solve time-dependent Schrödinger equation. Not like the time-independent Schrödinger equation, the time-dependent Schrödinger equation is expressed in a domain of complex numbers. A map between real-number domain and complex-number domain is required, making the mapping function more difficult to be learned.

Appendices

A The computer code for the eigen problem

The computer code of solving the eigen problem 1 was built with the Numpy library in python interface. The code is attached as following.

```
import numpy as np

# A function to compute potential and eigen energies
def he2D_potientials_energies(n, dz, na, nb, da, db, n_epsilon):
    n_sq = n ** 2 # total 2D size
    z1 = np.zeros(n)
    z2 = np.zeros(n)
    dz2 = dz ** 2
    two_inverse_dz2 = 2 / dz2
    mhalf_inverse_dz2 = -0.5 / dz2
    z_start = -n/2*dz
    a2 = np.zeros(na)
    b2 = np.zeros(nb)
    a2_start = 0.5 - na/2*da
    b2_start = 0.339 - na/2*db
    n_pot = na*nb # number of potentials
    pot = np.zeros((n_sq, n_pot))
    ham = np.zeros((n_sq, n_sq))
    epsilon = np.zeros((n_epsilon, n_pot))
    au_to_ev = 27.211

    for j in range(n):
        z1[j] = z_start + j*dz
        z2[j] = z_start + j*dz
    for j in range(na):
        a2[j] = a2_start + j*da
    for j in range(nb):
        b2[j] = b2_start + j*db

    for i in range(na): # loops of parameters a2, b2
        for j in range(nb):
            i_pot = i*nb + j
```

```
            i_sq = 0
            for ii in range(n): # loops of coordinates z1, z2
                for jj in range(n):
                    i_sq = ii*n + jj
                    pot[i_sq, i_pot] = 1.0/np.sqrt((z1[ii]-z2[jj])*(z1[ii]-z2[jj]) + b2[j]) - 2.0/np.sqrt(z1[ii]*z1[ii] +
                        a2[i]) - 2.0/np.sqrt(z2[jj]*z2[jj] + a2[i]) # soft Coulomb potential for 2D Helium atom
                    # Compute Hamiltonian: H = K + V, where K is computed by three-points finite difference
                    ham[i_sq, i_sq] = two_inverse_dz2 + pot[i_sq, i_pot] # diagonal elements of H
                    if ii > 0:
                        ham[i_sq, (ii - 1) * n + jj] = mhalf_inverse_dz2 # ii-1 elements of H
                    if ii < n-1:
                        ham[i_sq, (ii + 1) * n + jj] = mhalf_inverse_dz2 # ii+1 elements of H
                    if jj > 0:
                        ham[i_sq, ii * n + jj - 1] = mhalf_inverse_dz2 # jj-1 elements of H
                    if jj < n-1:
                        ham[i_sq, ii * n + jj + 1] = mhalf_inverse_dz2 # jj+1 elements of H
            eigen_energies = np.sort(np.linalg.eigvals(ham)) # compute ordered eigen energies for each potential
            epsilon[..., i_pot] = eigen_energies[0:n_epsilon] # select the first sevearl eigen energies
    return pot, epsilon

# ====== start main program ====
# Input parameters
n = 40
dz = 0.1
na = 200
nb = 200
da = 0.0001
db = 0.0001
n_epsilon = 5
au_to_ev = 27.211

# Call the function compute potentials and energies
he2D_pot, he2D_epsilon = he2D_potientials_energies(n, dz, na, nb, da, db, n_epsilon)

# save potentials and energies
np.savetxt("he2d_potentials.csv", he2D_pot, delimiter=",")
np.savetxt("he2d_energies.csv", he2D_epsilon, delimiter=",")
```

B The computer code for the CNN model

The computer code of the CNN model was built with Google Tenowflow in python interface. The code is attached as following.

```
import numpy as np
import tensorflow as tf
np.set_printoptions(threshold=np.nan)

# user defined functions
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1) # W is initialized as ~ N(0, 0.1^2)
  return tf.Variable(initial)
```

```
def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape) # b is initialized as constants: 0.1
  return tf.Variable(initial)

def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# ====== start main program ======
# read data
pot_file='../pot_eng7/he2d_potentials.csv'
epsilon_file='../pot_eng7/he2d_energies.csv'

n = 40
n_grids = n ** 2
n_out = 1
na = 200
nb = 200
n_samples = na*nb
rtest = 10
n_test = n_samples / rtest
n_train = n_samples - n_test
ind_training=(1,)
ind_test=(0,)
for i in range(2,n_samples):
    if i%rtest == 0:
        ind_test = ind_test + (i,) # index of test set
    else:
        ind_training = ind_training + (i,) # index of training set

n_col = n_out + n_grids
train_data = np.zeros((n_train, n_col))
test_data = np.zeros((n_test, n_col))

eps = np.genfromtxt(epsilon_file, delimiter=',', usecols=(ind_training), max_rows=1 ) # read the first row
train_data[:,0] = np.array([eps]).T[:,0]
train_data[:,1:n_col] = np.matrix.transpose( np.genfromtxt(pot_file, delimiter=',', usecols=(ind_training) ) )

eps0 = np.genfromtxt(epsilon_file, delimiter=',', usecols=(ind_test), max_rows=1 ) # read the first row
test_data[:,0] = np.array([eps0]).T[:,0]
test_data[:,1:n_col] = np.matrix.transpose( np.genfromtxt(pot_file, delimiter=',', usecols=(ind_test) ) )
np.savetxt("true_eps.csv", test_data[:,0:1]) # save true epsilon in one column

# Declare placeholder to hold input data
x = tf.placeholder(tf.float32, shape=[None, n_grids])
y_ = tf.placeholder(tf.float32, shape=[None, n_out])

# ======= build cnn ===========
fsize = 3
```

```
nchanel1 = 32
nchanel2 = 16
nchanel3 = 16
print nchanel1, nchanel2, nchanel3

# 1st layer: input data
x_image = tf.reshape(x, [−1, n, n, 1])

# 2nd and 3rd layer: conv + pooling
W_conv1 = weight_variable([fsize, fsize, 1, nchanel1])
b_conv1 = bias_variable([nchanel1])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

# 4th and 5th layer: conv + pooling
W_conv2 = weight_variable([fsize, fsize, nchanel1, nchanel2])
b_conv2 = bias_variable([nchanel2])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

# 6th and 7th layer: conv + pooling
W_conv3 = weight_variable([fsize, fsize, nchanel2, nchanel3])
b_conv3 = bias_variable([nchanel3])
h_conv3 = tf.nn.relu(conv2d(h_pool2, W_conv3) + b_conv3)
h_pool3 = max_pool_2x2(h_conv3)

# 8th and 9th layer: flatten + fully connected.
n_pool = 3
pool_size = 2
fac_reduce = pool_size ** n_pool
n_full = n / fac_reduce
W_fc1 = weight_variable([n_full * n_full * nchanel3, 1024])
b_fc1 = bias_variable([1024])
h_flat = tf.reshape(h_pool3, [−1, n_full*n_full*nchanel3])
h_fc1 = tf.nn.relu(tf.matmul(h_flat, W_fc1) + b_fc1)

keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob) # Optional: Drop to avoid overfitting

# last layer: fully connected −−−> output
W_fc2 = weight_variable([1024, n_out])
b_fc2 = bias_variable([n_out])
y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

# Loss function = Training error
mean_squared_error = tf.reduce_mean( tf.square(tf.subtract( y_, y_conv)) )

# Use Adadelta optimizer to minimize loss function
lrate = 0.001 # learning rate
train_step = tf.train.AdadeltaOptimizer(lrate).minimize( mean_squared_error )

# Declare an interactive session and initialize W and b
```

```
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())

# Divde training data into multiple batches
bsize =1000
n_batch = n_train / bsize
interv = 4
n_epoch = 500 #100
n_iter = n_batch * n_epoch
n_eval = n_iter / interv #+ n_epoch # number of output error
print bsize, n_batch, n_epoch, n_iter
pot_batch = np.zeros((bsize, n_grids))
eps_batch = np.zeros((bsize, 1))
err_train = np.zeros((n_eval,3))

# Really run training and evaluation
k = 0
iter = 0
for j in range(n_epoch): # loop of epochs
  # random shuffle training data
  np.random.shuffle(train_data)

  for i in range(n_batch): # loop of batches in one epoch
    # Get data for the i-th batch
    istart = i*bsize
    iend = istart + bsize
    pot_batch = train_data[istart:iend, 1:n_col]
    eps_batch = train_data[istart:iend, 0:1]

    if (i+1)\%interv == 0: # save valication error
      err_train[k,0] = iter
      err_train[k,1] = mean_squared_error.eval(feed_dict={x: pot_batch, y_: eps_batch, keep_prob: 1.0})

    # Run one tranining step
    train_step.run(feed_dict={x: pot_batch, y_: eps_batch, keep_prob: 1.0})

    if (i+1)\%interv == 0: # save training error
      err_train[k,2] = mean_squared_error.eval(feed_dict={x: pot_batch, y_: eps_batch, keep_prob: 1.0})
      print("training_error:_\%g"\%err_train[k,2])
      k = k + 1
    iter = iter + 1 # count iterations

np.savetxt("training_error.csv", err_train, delimiter=",")

# Evaluate with test data
n_batch0 = n_test / bsize
pot_batch0 = np.zeros((bsize, n_grids))
eps_batch0 = np.zeros((bsize, 1))
err_test = np.zeros(n_batch0)
for i in range(n_batch0):
   istart0 = i*bsize
   iend0 = istart0 + bsize
```

```
    pot_batch0 = train_data[istart0:iend0, 1:n_col]
    eps_batch0 = train_data[istart0:iend0, 0:1]
    err_test[i] = mean_squared_error.eval(feed_dict={x: pot_batch0, y_: eps_batch0, keep_prob: 1.0})
    print("test_error_of_a_batch:_%g"% err_test[i] )

# Save predicted energies
y_pred = y_conv.eval(feed_dict={x: test_data[:,1:n_col], y_: test_data[:,0:1], keep_prob: 1.0})
np.savetxt("predicted_eps.csv", y_pred, delimiter=",")
```

References

[1] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elememt of Machine Learning.* Springer.

[2] Jacob Biamonte, et al. Nature 549, 195 (2017).

[3] J. C. Snyder, M. Rupp, K. Hansen, K.-R. Mller, and K. Burke, Phys. Rev. Lett. 108, 253002 (2012).

[4] F. Brockherde, L. Vogt, L. Li, M. E. Tuckerman, K. Burke, and K.-R. Mller, arXiv:1609.02815.

[5] K. Yao and J. Parkhill, J. Chem. Theory Comput. 12, 1139 (2016).

[6] P. Mehta and D. J. Schwab, arXiv:1410.3831.

[7] K. T. Schtt, F. Arbabzadah, S. Chmiela, K. R. Mller, and A. Tkatchenko, Nat. Commun. 8, 13890 (2017).

[8] Kyle Mills, Michael Spanner, and Isaac Tamblyn, Physical Review A 96, 042113 (2017).

[9] B. P. van Milligen, V. Tribaldos, and J. A. Jimnez, Phys. Rev. Lett. 75, 3594 (1995).

[10] G. Carleo and M. Troyer, Science 355, 602 (2017).

[11] K. I. Funahashi, Neural Networks 2, 183 (1989).

[12] J. L. Castro, C. J. Mantas, and J. M. Bentez, Neural Networks 13, 561 (2000).

[13] M. D. Zeiler, arXiv:1212.5701.

[14] M. Abadi et al., arXiv:1603.04467 (2015).

[15] H.W. Lin, M. Tegmark, and D. Rolnick, J. Stat. Phys. 168, 1223 (2017).