ISTANBUL TECHNICAL UNIVERSITY

FACULTY OF SCIENCE AND LETTERS

Advanced Physics Project Report

Machine Learning and Nonlinear Schödinger Equation

Hüseyin Talha Şenyaşa

Department : Physics Engineering
Student ID : 090120132
Advisor : Assoc. Prof. A. Levent Subaşı

FALL 2017

Summary

We train an artificial neural network to estimate the ground state energy of a one-dimensional Bose-Einstein condensate in harmonic trapping potential. Such a system can be described by the solution of a non-linear Schrödinger equation also called a Gross-Pitaevskii equation. We also use the method for the inverse problem of predicting the non-linearity parameter using the ground state density profile for a given harmonic trapping potential.

Contents

1 Introduction and Motivation

Machine learning is a growing research area, not only its theoretical base but also its share in applications in different areas. One of the reason why machine learning is used increasingly today is that artificial neural networks used in machine learning can approximate any continuous function within desired accuracy. This means that if we take $\epsilon > 0$ as desired accuracy, $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$ output of the network and $\boldsymbol{g}(\boldsymbol{x})$ real value of the function, then it is guaranteed that there exists a network that satisfies the relation $|\boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{x})| < \epsilon$. From this point of view, a process or calculation that can be thought of as a function can be represented by a corresponding neural network that mimics this function in desired accuracy [1]. With this in mind, neural networks have the potential to learn general functions and be exploited for their advantages. Approximate value of the quantity can be determined for different scenarios.

Many different kind of applications of machine learning have already been implemented in physics[1]. For example, In [3, 4] machine learning is applied to quantum many body problems. A machine learning method called Unsupervised Learning to detect patterns in big datasets is used for discovering phase transitions [5]. There are also developed techniques in machine learning inspired from physics such as quantum tensor networks [6]. Relation between physics and machine learning also caused foundation of a new branch called Quantum Machine Learning which aims to implement a quantum software to make machine learning faster than its classical version [7].

In [8], machine learning approach is applied to a 2D Schrödinger Equation with random potentials. The authors built a convolutional deep neural network, and trained it to predict ground state energy of the system under four different two dimensional confining potentials including some random potentials. Their study showed that machine learning is a promising alternative in electronic structure calculations of quantum systems. In our study inspired from the article mentioned above, we apply machine learning method to the nonlinear Schrödinger equation to predict ground state energy of a Bose-Einstein condensate at absolute zero temperature in one dimensional harmonic trapping potentials.

2 Gross Pitaevskii Equation

A Bose-Einstein Condensate (BEC) is described by Gross Pitaevskii Equation (GPE) also known as non-linear Schrodinger Equation (NLSE). General form of

[1]For a detailed list, see [2]

GPE in three dimension is given as,

$$i\hbar\frac{\partial\Psi}{\partial t} = \frac{-\hbar^2}{2m}\nabla^2\Psi + V(\boldsymbol{r},t)\Psi + g|\Psi|^2\ \Psi \tag{1}$$

where $\hbar$ is Planck constant, $\boldsymbol{r}$ is position vector, t is time, $\Psi(\boldsymbol{r},t)$ is the wave function, m is mass, ∇^2 is the Laplacian operator, V is the potential, g is a constant interaction parameter and it is defined as

$$g = \frac{4\pi\hbar^2 a_s}{m} \tag{2}$$

where, a_s is the s wave scattering length. Non-linearity of GPE comes from the interaction term. If there is no interaction GPE reduces to the Schrödinger Equation (SE) and becomes a linear equation. By definition, g can be positive or negative. If $g > 0$, it represents repulsive interaction, and if $g < 0$, it represents attractive interactions[9].

In Eq. (1), the first term on the right side is the kinetic term, the second term represents the potential energy, and the last term is the interaction term of the Hamiltonian. The expectation value of the total energy can be written as;

$$\langle E\rangle = \int \Psi^*\hat{H}\Psi d^3\boldsymbol{r} \tag{3}$$

$$E = \int\left(\frac{\hbar^2}{2m}|\nabla\Psi|^2 + V|\Psi|^2 + g|\Psi|^4\right)d^3\boldsymbol{r} \tag{4}$$

If potential is not function of time, then, total energy of the system is conversed.

We assume a time-independent harmonic trapping potential. In such a case, the potential is only function of position $V(\boldsymbol{r})$ and it can be written as

$$V(x,y,z) = \frac{1}{2}m(\omega_x^2x^2 + \omega_y^2y^2 + \omega_z^2z^2) \tag{5}$$

where ω is the angular frequency. Then method of separation of variables can be applied and it can be obtained that there is a solution in stationary form. In this case the factorized wave function is,

$$\Psi(\boldsymbol{r},t) = \psi(\boldsymbol{r})e^{-i\mu t/\hbar} \tag{6}$$

Here, μ is the chemical potential. Time independent GPE becomes,

$$\mu\psi = \frac{-\hbar^2}{2m}\nabla^2\psi + V(\boldsymbol{r})\psi + g|\psi|^2\psi \tag{7}$$

BECs can also be studied in one dimension by making angular frequencies asymmetric such that $\omega_x, \omega_y \gg \omega_z$ and keeping their energy order much greater

than condensate's energy $\hbar(\omega_x\omega_y)^{1/2} \gg \mu$. This enables us to confine the dynamics of the system in one dimension and it can be described with a corresponding one dimensional GPE equation [9]. To obtain one dimensional GPE one can rewrite wave function,

$$\psi(x, y, z, t) = \psi_z(z, t)\psi_x(x)\psi_y(y) \tag{8}$$

ψ_x and ψ_y are the ground state solution of the transverse potential;

$$\begin{aligned} \psi_x(x) &= \frac{1}{(\pi l_x^2)^{1/4}} e^{-x^2/2l_x^2} \\ \psi_y(y) &= \frac{1}{(\pi l_y^2)^{1/4}} e^{-y^2/2l_y^2} \end{aligned} \tag{9}$$

where $l_x = \sqrt{\hbar/m\omega_x}$ and $l_y = \sqrt{\hbar/m\omega_x}$ are the harmonic oscillator lengths that related to the density distribution of the condensate. Normalization of the wave function is chosen such that $\int \psi_x(x)dx = \int \psi_y(y)dy = 1$, and $\int \psi_z(z,t) = N$. If we plug in the factorized wave function to Eq. (7), then, the equation becomes;

$$\mu'\psi_z = \frac{-\hbar^2}{2m}\frac{d^2\psi_z}{dz^2} + \frac{1}{2}m\omega_z^2 z^2\psi_z + g'|\psi_z|^2\psi_z \tag{10}$$

where μ' and g' one dimensional effective chemical potential and interaction strength respectively

$$\mu' = \mu - \frac{\hbar}{2}(\omega_x - \omega_y), \quad g' = \frac{g}{2\pi l_x l_y} \tag{11}$$

From now on, we refer to Eq. (10) as GPE.

It is convenient to work with dimensionless quantities, to make equation dimensionless, one can define $z = a_0\widetilde{z}$. In this case Eq. (10) becomes,

$$\mu\psi = \frac{-\hbar^2}{2ma_0^2}\frac{d^2\psi}{d\widetilde{z}^2} + \frac{1}{2}m\omega^2 a_0^2\widetilde{z}^2\psi + g|\psi|^2\psi \tag{12}$$

(Here we dropped primes). After that, setting $\hbar\omega = \hbar^2/ma_0^2$, we find $a_0^2 = \hbar\omega/m$ .Plugging a_0^2 in to the Eq. (12), we get

$$\mu\psi = -\frac{1}{2}m\omega^2\frac{d^2\psi}{d\widetilde{z}^2} + \frac{\hbar^2}{2m}\widetilde{z}^2\psi + g|\psi|^2\psi \tag{13}$$

If we divide Eq. (13) by $\hbar\omega$ and define $\widetilde{\mu} = \mu/\hbar\omega$, we obtain

$$\widetilde{\mu}\psi = -\frac{1}{2\sqrt{a_0}}\frac{d^2\psi}{d\widetilde{z}^2} + \frac{1}{2\sqrt{a_0}}\widetilde{z}^2\psi + \frac{g|\psi|^2\psi}{\hbar\omega} \tag{14}$$

From the normalization condition,

$$\int |\psi|^2 dz = \int |\psi|^2 a_0 d\widetilde{z} = N \tag{15}$$

we can define $\widetilde{\psi} = (\sqrt{a_0}/\sqrt{N})\psi$. The third term on the right in Eq. (14) becomes,

$$\frac{g|\psi|^2}{\hbar\omega} = \frac{ga_0}{N\hbar\omega}|\widetilde{\psi}|^2 \tag{16}$$

Finally, defining

$$\frac{ga_0}{N\hbar\omega_z} = \widetilde{g} \tag{17}$$

the dimensionless GPE can be written as,

$$\widetilde{\mu}\widetilde{\psi} = -\frac{1}{2}\frac{d^2\widetilde{\psi}}{d\widetilde{z}^2} + \frac{1}{2}\widetilde{z}^2\widetilde{\psi} + \widetilde{g}|\widetilde{\psi}|^2\widetilde{\psi} \tag{18}$$

There is no known analytic solution of GPE for harmonic trapping potential except the case in which interaction parameter is zero. In this case, GPE reduces to the SE and solution is well known with the ground state energy;

$$\widetilde{\mu} = \frac{1}{2} \tag{19}$$

we used this case to check our numerical solutions.

2.1 Numerical Solution and XMDS Framework

In the previous section we stated that there is no general analytic solution of GPE. It is known for only few cases such as for a uniform condensate $V = 0$ constant. Most of the time GPE is solved with numerical techniques or approximations. In this study, we used a framework called XMDS [10], implemented specifically to solve differential equation systems with well optimized numerical methods. In this framework the partial differential equations system can be described by a markup language called XML. When equation system is declared properly, XMDS produces a source code written in C++ that solves the equation with pre-specified numerical method. In our program, XMDS takes angular frequency, shift of equilibrium point, value of interaction parameter externally. It generates trapping potential and other quantities internally with supplied expressions.

Cross-check of the framework has done in two ways. First, since there exists an analytical solution for $g = 0$, we compared the results of the XMDS with analytical ones, and we also used another program here [11], and compared few results for situations where analytical solution does not exist.

3 Problem Statement and Dataset Generation

GPE can be solved with stated methods above. These methods must be applied every time when there is a change in parameters of the equation such as applying different trapping potential or changing interaction parameter. For example, when interaction parameter is changed, if there is no analytic solution, numerical method must be reapplied. With machine learning these steps can be encoded in network and information about the system can be obtained instantly with only one time cost which is training process of the network. In[8], authors apply deep learning method to the Schrödinger Equation to obtain ground state energy of the system under different potentials.

We try to built an artificial neural network to predict the ground state energy of a BEC for a given harmonic trapping potential and interaction parameter. We also try the inverse problem which is the prediction of interaction parameter for a given potential and density.

An artificial neural network is made of layers and these layers contain simple units called neurons. These neurons were insprired by the neurons in the human brain. A neuron takes one or more input and generates an output. To do this, it uses its internal variables which are called weights and biases. In general, the number of weights in a neuron is equal to the size of the input and there is one bias value. There is no restriction on the range of weights and biases except computational, they can even take on complex values [12]. Input output relation of a neuron can be described in the following way; let $\boldsymbol{x} = [x_1, x_2, x_3, ..., x_n]$ be the input of the neuron and f be the function that describes behavior of the neuron.

$$f(\boldsymbol{x}, \boldsymbol{\omega}, b) = \sum_{i=1}^{n} \omega_i x_i + b \tag{20}$$

Here, ω_i are the weights and b is the bias. A layer of a network involves neurons that implement with this function. In our example, x_n inputs represent the potential expression respect to the position or represents density for the inverse problem. The range of the function f is infinite and it can be used directly. However, there are various ways to interpret result of the function f. The most primitive version of interpretation is sending the result of f as an argument to unit step function to generate 0 (False) or 1 (True). In this case,

$$a(f) = u_1(f(\boldsymbol{x}, \boldsymbol{\omega}, b)) \tag{21}$$

where a is called as the activation function of the neuron and it is a unit step function in this case. However, the usage of the step function introduces discontinuity and does not allow to use differential methods to calculate behavior of the

neurons when a small change in weights or biases is applied. Therefore, a more common activation function is sigmoid function denoted by σ which is continuous version of the step function and it is defined as,

$$\sigma(f) = \frac{1}{1 + e^{-f}} \tag{22}$$

If a neuron defined in this way such that its output given by Eq. (22), then it is called a sigmoid neuron. In the former case which is the step function, the neuron is called as perceptron.

If we denote j^{th} neuron in the l^{th} layer with f_j^l, and the weight from k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the layer l^{th} with ω_{jk}^l, finally the bias of the j^{th} neuron in the l^{th} layer with b_j^l, then, the output of the j^{th} neuron in the l^{th} layer will be;

$$f_j^l(\sigma(f_k^{l-1}), \boldsymbol{\omega}, b) = \sum_{k=1}^{n} \omega_{jk}^l \sigma(f_k^{l-1}) + b_j^l \tag{23}$$

and the activation of the output can be written as;

$$\sigma(f_j^l(\sigma(f_k^{l-1}), \boldsymbol{\omega}, b) \tag{24}$$

Here we use sigmoid function as activation function but it can be any suitable activation function such as hyperbolic tangent $\tanh(f)$ or Rectified Linear Unit (ReLU).

Connection complexity of the neurons may vary. Output of each neuron in one layer can be input of every neuron in next layer. Such networks are called Fully Connected Networks (FCN) or multilayer perceptrons (MLP)[1]. If the result of one layer is directly sent to the next layer without any circulation or feedback, then the network is called as Feed Forward Network [1].

We are going to change these weights and biases in such a way that the difference between output generated by network and real value of the corresponding quantity will be minimized. To do that, we are going to use a proxy relation between output and real value which is called as Cost Function [1]. There are different kind of cost functions such as quadratic cost function, cross entropy cost function etc. We are going to use quadratic cost function also called as mean squared error to show the general mechanism and to introduce few notions that directly effects the behavior of the network.

The quadratic cost function is defined as;

$$C(\omega, b) = \frac{1}{2n} \sum_x ||\boldsymbol{y} - a(f_L(\boldsymbol{x}))||^2 \tag{25}$$

where n, ω, b are number of examples in the training set, weights and biases, respectively. The subscript L indicates the last layer which is the output, so $f_L(\boldsymbol{x})$ output produced by network and $\boldsymbol{y}$ is real value of the quantity. To minimize C, we can take derivative of the function and can find extremum values, but this method is extremely costly because the total number of neurons in the network is enormous [1]. This problem can be overcame by an iterative algorithm called Gradient Descent. Since C is also a scalar field the algorithm tries to determine a direction which points to the decrement and moves in that direction with small steps. In each iteration, algorithm repeats itself to reach minimum value.

A small displacement in arbitrary direction which corresponds to a small change in weight or bias or both can be written as;

$$\Delta C = \frac{\partial C}{\partial \omega}\Delta\omega + \frac{\partial C}{\partial b}\Delta b \tag{26}$$

This expression gives information about what happens when small displacement is made. In order to the determine direction of decrement we can rewrite this expression in terms of the gradient since it points to direction of maximum rate of increase.

We can define gradient operator as;

$$\boldsymbol{\nabla} C = \left(\frac{\partial C}{\partial \omega}, \frac{\partial C}{\partial b}\right)^T \tag{27}$$

In this case, Eq. (26) can be written as;

$$\Delta C = \boldsymbol{\nabla} C \cdot \Delta \boldsymbol{l} \tag{28}$$

where $\boldsymbol{l} = (\omega, b)^T$. We want to minimize the cost function, thus; in each iteration value of the C must be smaller than the previous one which means that ΔC must be smaller than zero. Therefore, the direction of displacement must be in the opposite direction of the gradient. Rather than calculating the opposite direction for each iteration we can define $\Delta \boldsymbol{l}$ in such a way so that it always points to opposite direction of the gradient.

$$\Delta \boldsymbol{l} = -\eta \boldsymbol{\nabla} C \tag{29}$$

In this case, Eq. (28) becomes;

$$\Delta C = -\eta ||\boldsymbol{\nabla} C||^2 \tag{30}$$

where η is positive definite number and it is called the learning rate. Since it is guaranteed that $||\boldsymbol{\nabla} C||^2 \geq 0$, therefore; $\Delta C \leq 0$. In this situation, learning rate

is the step size in each iteration. From Eq. (30) it is intuitive that η can not be a large number because in such a case, the algorithm can miss the minimum point. This situation also brings up the subject that gradient descent does not give guarantee to find the minimum point.

Gradient Descent algorithm has an statistical version called Stochastic Gradient Descent to increase the speed of training process. It is assumed that gradient of m randomly selected examples in the dataset is nearly equal to gradient of the whole dataset and it can be expressed as;

$$\frac{1}{m}\sum_{k=1}^{m}\nabla C_{X_j} \approx \frac{1}{n}\sum_{x=1}^{n}\nabla C_x = \nabla C \tag{31}$$

where ∇C_x is the gradient of a single training input and m is also called as mini-batch size.

It is worth noting that approaching a minimum value is proportional to the number of iterations which equals to number of examples in the dataset, therefore; it is a corollary that there can be two situations[2]. First, the algorithm cannot reach the minimum point due to lack of training examples, and secondly, the algorithm reaches a global or local minimum point and starts to oscillate around it[13]. There are mechanisms to prevent such cases and other optimization problems like direction sensitivity and they are called as adaptive learning rate [8]. In our network we used an algorithm called Adam [14] provided by the framework we used.

The final step in the gradient descent is updating the weights and biases. We use Eq. (30) to perform this task. If we combine Eq. (30) with Eq. (31) and write it in open form, the expression to update the weight and bias can be written as;

$$\begin{aligned}\omega' &= \omega - \frac{\eta}{m}\sum_{j=1}^{m}\frac{\partial C_{X_j}}{\partial \omega}\\ b' &= b - \frac{\eta}{m}\sum_{j=1}^{m}\frac{\partial C_{X_j}}{\partial b}\end{aligned} \tag{32}$$

Numerical calculation of the partial derivatives in the equation is one of the costliest computational operation in the training process. To reduce computational cost, there is a well known algorithm called backpropagation [15]. We use Pytorch Framework in our study and it provides similar mechanism called automatic differentiation.

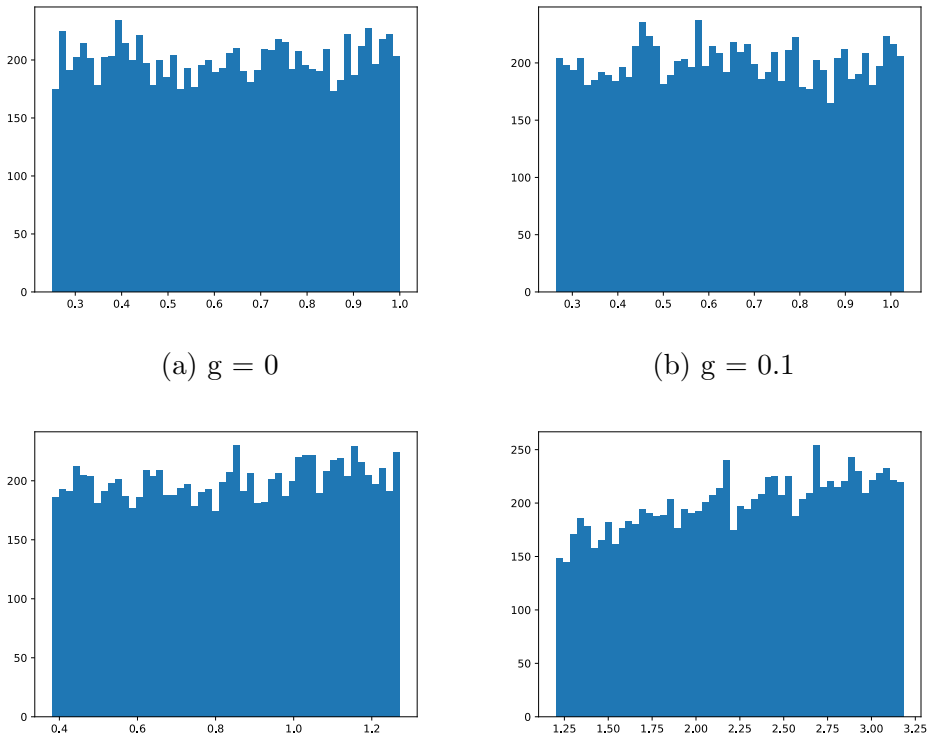
In this section, we reviewd the general concept in artificial neural networks

[2]The third case is that algorithm may diverge but we ignore this situation here.

and introduced notions we are going to use in our example.

3.1 Dataset and Dataset Generation

We solved GPE for four different interaction parameters, therefore; there are four different corresponding datasets and each of them contains 10000 elements. We used 8500 of them as training examples and 1500 of them for testing if otherwise is specified. An element of a dataset involves an array containing potential values as a function of position and an another array containing interaction, kinetic, potential and total energy values respectively.



(a) g = 0 (b) g = 0.1

(c) g = 1 (d) g = 10

Figure 1: Total energy distributions of the generated solution for different interaction parameter values.

The array containing potential energy is generated according to harmonic trap expression in XMDS, thus; we only supplied angular frequency and shift of equilibrium to the XMDS randomly within predetermined limits. The dimensionless angular frequency can take values between 0.5 and 2. Shift of the equilibrium point is determined due to boundary conditions since density must be zero at the boundaries. In the numerical solution of the GPE, domain of the z dimension is between -10 to 10. Taking the maximum magnitude of the shift as ∓ 5 is enough to ensure that density function goes to zero at infinity. In both angular frequency

and shift, we used uniformly generated random numbers. Total energy distribution is given in Fig 1. Another type distribution is not required in our problem since the potential type is fixed but there are examples such as here [8], which different distribution is used.

4 Machine Learning for GPE

We used Pytorch Framework [16] to build our neural networks. It allows the client codes to work on both CPU and GPU via its internal python object called Tensor. If any CUDA supported GPU is available then Pytorch can use GPU without any change in the code. Code have three main parts, first one is dataloaders; it reads train and test data for specified interaction parameter from corresponding file and generates tensor dataset object. In this part, manipulation on dataset can be applied such as normalization, shuffling etc. Second part is implementation of the network. Architecture of the network is represented with a python class inherited from Pytorch's base class for networks called Module. This class also includes a forward method which is responsible for how data will be sent to the next layer. The last part is training and testing. In the training part, the network is iterated with training dataset and loss (result of cost function) is calculated at the end of each iteration. After that, weights and bias are updated accordingly.

Our aim is to show that the machine learning methods can be used to bypass GPE equation and physical features of the BEC can be determined directly. To do that, we implemented two different types of neural network; fully connected network (FCN) and convolutional neural network (CNN). CNN is more complex than FCN and it is developed to handle data which have grid-like topology [15]. Since our dataset involves translation, we expect that CNN can handle such a situation better than FCN.

4.1 Architecture

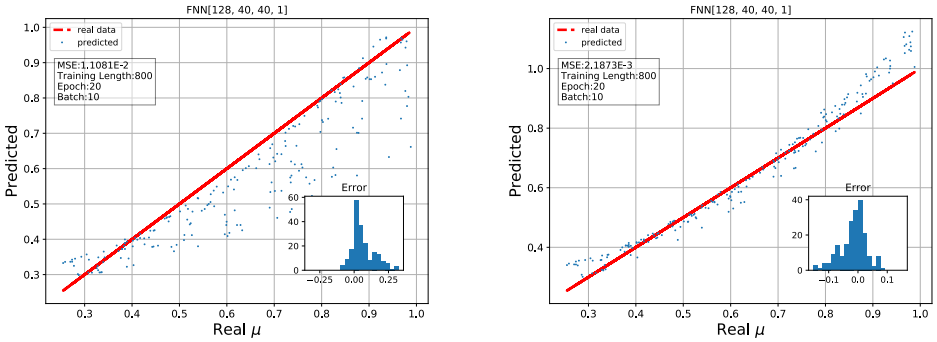
FCN involves 128 input neurons as input layer, next layer is the first hidden layer with 30 neurons, the second is another hidden layer same as the first, the next one is last hidden layer with 10 neurons and the last layer is the output layer. Totally there are 5 layers in our FCN and it will be denoted as FCN$[128, 30, 30, 10, 4]$. The most common activation function in feedforward networks rectified linear unit (ReLU) [8], is used for each forward feed except output. No operation is applied to the output neuron. Learning rate of the FCN is fixed and it is 0.001. Cost function is mean squared error (MSE) and optimization is

done with Adam.

CNN has two convolution layers, two maxpool layers and three fully connected layer and the last layer of the fully connected part is the output layer. Maxpooling is applied to output of the first and second convolution layers. ReLu is also applied each forward except output neuron same as in the FCN. Fully connected part of the CNN is FCN$[310, 100, 20, 1]$. Learning rate, cost function and optimizer of the CNN is same as FCN which are 0.001, MSE and Adam respectively.

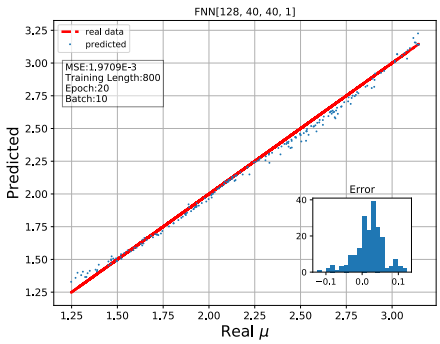
4.2 Hyperparameters

We started with small dataset which has 800 elements for training 200 for test to see response of the architecture and to obtain a range for learning rate. The size of this dataset is small but it can give an idea about the range of learning rate since stochastic gradient descent is a statistical approach. We first set learning rate, η to 3 and start to decrease it, until 0.01 prediction of the network was extremely poor. For lower values, predictions were more encouraging. In this section, we show the effect of learning rate for two different architectures and interaction parameters.

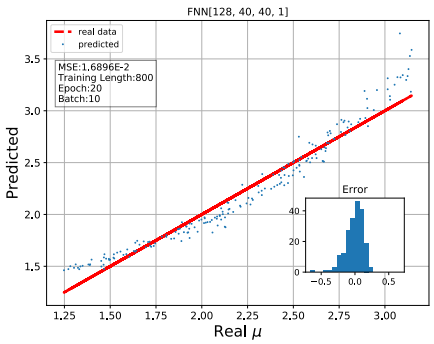


(a) FCN[128, 40, 40, 1], $\eta = 0.003$ (b) FCN[128, 40, 40, 1], $\eta = 0.001$

Figure 2: True Energy of Ground State vs Predicted Ground State Energy. Their hyperparameters are identical except the learning rate. Total number of epoch is 20 and batch size is 10. In this example interaction parameter g, is zero. Even the precision of the learning rate 0.001 seems higher, it is trivial to expect that change in interaction parameter will effect the result.



(a) FCN[128, 40, 40, 1], lr = 0.003

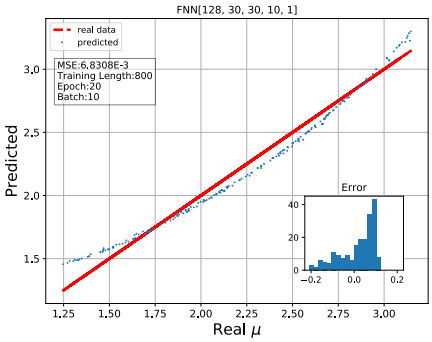


(b) FCN[128, 40, 40, 1], lr = 0.001

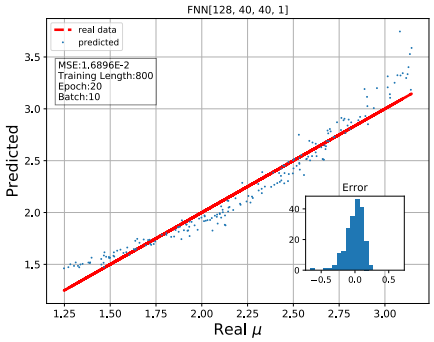
Figure 3: Here the same network with different interaction parameter which is 10.

It is clear that precision of the network with learning rate 0.003 is higher than 0.001. Of course, dramatic increase in the training dataset length may affect them both to converge same precision but our intention here to show that small change in learning rate causes different results.

After determination of learning rate we added extra one hidden layer to the network and tried some different number of neurons combination in the layers.

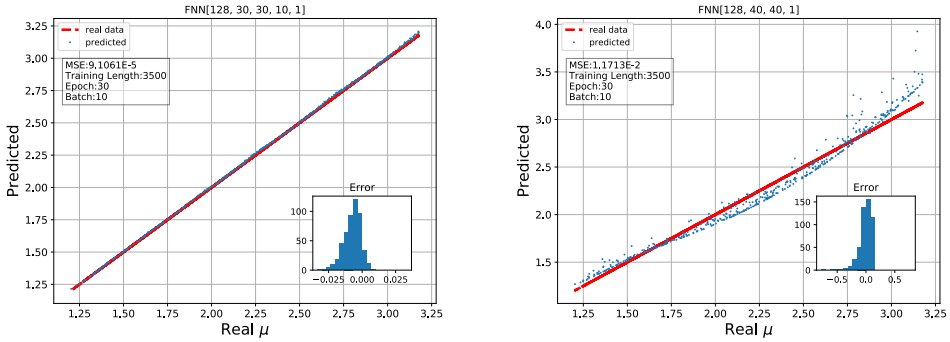


(a) FCN[128, 30, 30, 10, 1]



(b) FCN[128, 40, 40, 1]

Figure 4: Interaction paramater g, is 10. Batch size is 10 and total number of epoch is 20 for this example. Precision of the network of 5 layer (a) is higher than 4 layer (b). Bias in (a) can be eliminated by increasing training dataset length.



(a) FCN[128, 30, 30, 10, 1] (b) FCN[128, 40, 40, 1]

Figure 5: Total training dataset length is increased to 3500 and test dataset length is 500. Batch size, Learning rate are same as example given in Fig. 4 but the epoch is 30. The bias is eliminated. Precision and accuracy of the network of 5 layers has improved.

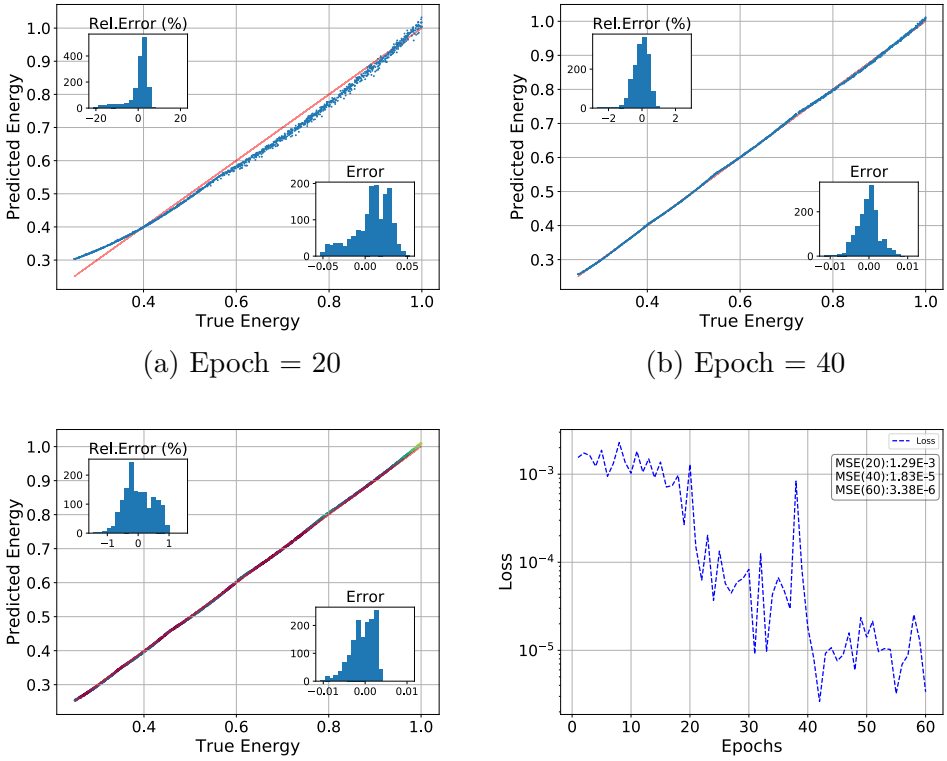
In conclusion, we chose learning rate η as 0.001 and started the training process. The other two hyperparameters epoch and mini batch size are determined in the training process. After number of different tries, we set the total number of epochs to 60, and mini batch size to 30.

4.3 Training Results

We give the result of total energy predictions per 20 epochs for both FCN and CNN also to show how predictions change. Results of FCN$[128, 30, 30, 10, 4]$ used in prediction of interaction, potential, kinetic and total energy separately are given directly. For the inverse problem, three different scenarios are presented. All trainings are done with identical networks and only the training and test datasets are changed.

4.3.1 Non-interacting System

In non-interacting systems, GPE reduces to SE which does not involve non-linear term.



(a) Epoch = 20

(b) Epoch = 40

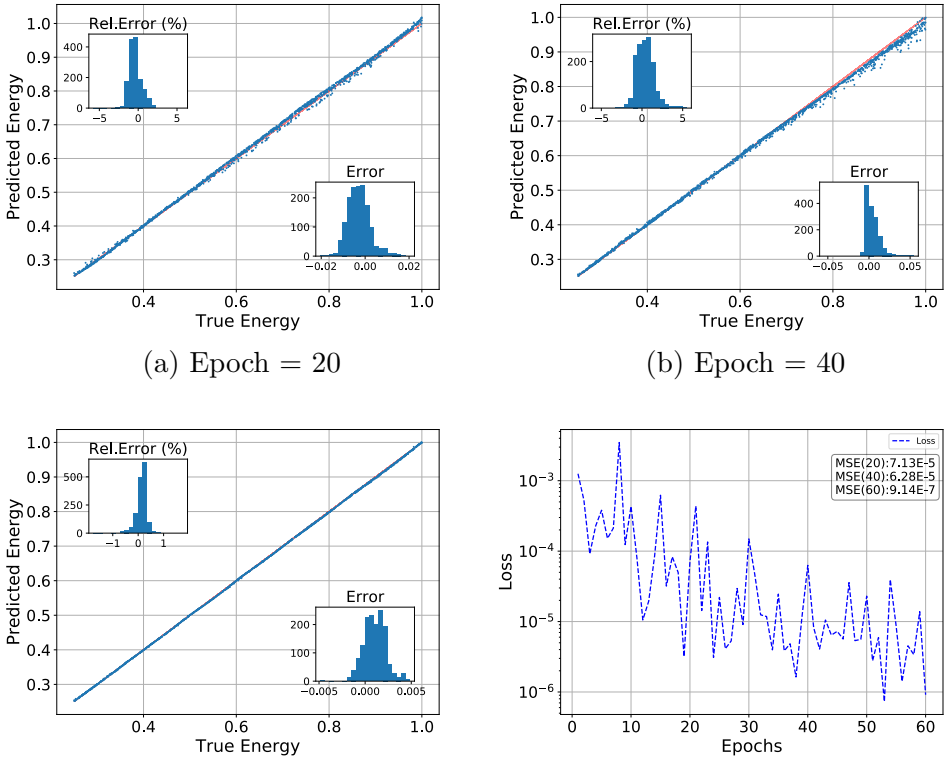
(c) Epoch = 60

(d) Loss

Figure 6: Result of FCN[128, 30, 30, 10, 1] for non-interacting system.

In figures, x axis represents true dimensionless total energy values, and y is predicted energy by trained network. Inset histogram at the left upper corner is relative error given as percentage and the inset histogram at the right corner is difference error.

Since the system has no interaction parameter, problem is relatively easy when it is compared to systems that involve interaction. In Fig. 6(a) it is obvious that the network requires more training examples and it reaches a satisfactory level in Fig. 6(b). Supplying another 20 more epochs reduces both relative error and MSE as shown in Fig. 6(d)



(a) Epoch = 20

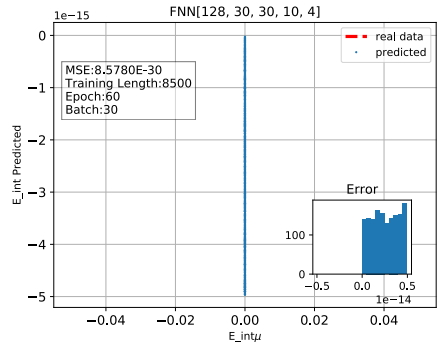
(b) Epoch = 40

(c) Epoch = 60

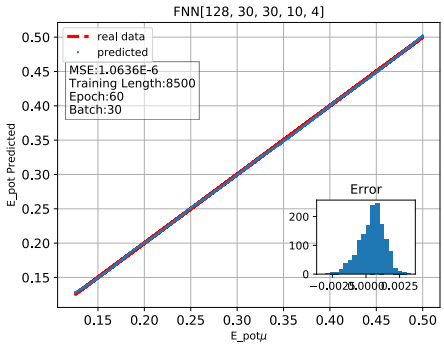
(d) Loss

Figure 7: CNN results for $g = 0$

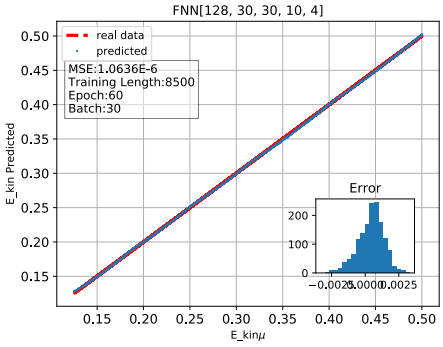
The bias in Fig. 6(a) does not appear in Fig. 7(a). One reason why there is less or no bias in CNN predictions could be because CNN's have the ability to handle translations in dataset. In our dataset, translation corresponds to shift of equilibrium point.



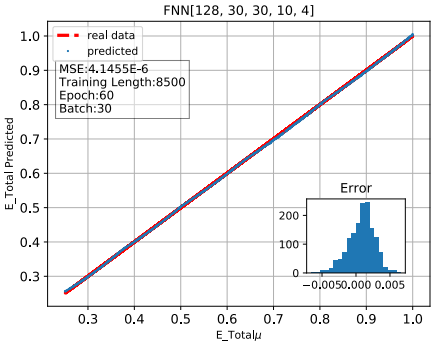
(a) FCN[128, 30, 30, 10, 4]



(b) FCN[128, 30, 30, 10, 4]



(c) FCN[128, 30, 30, 10, 4]



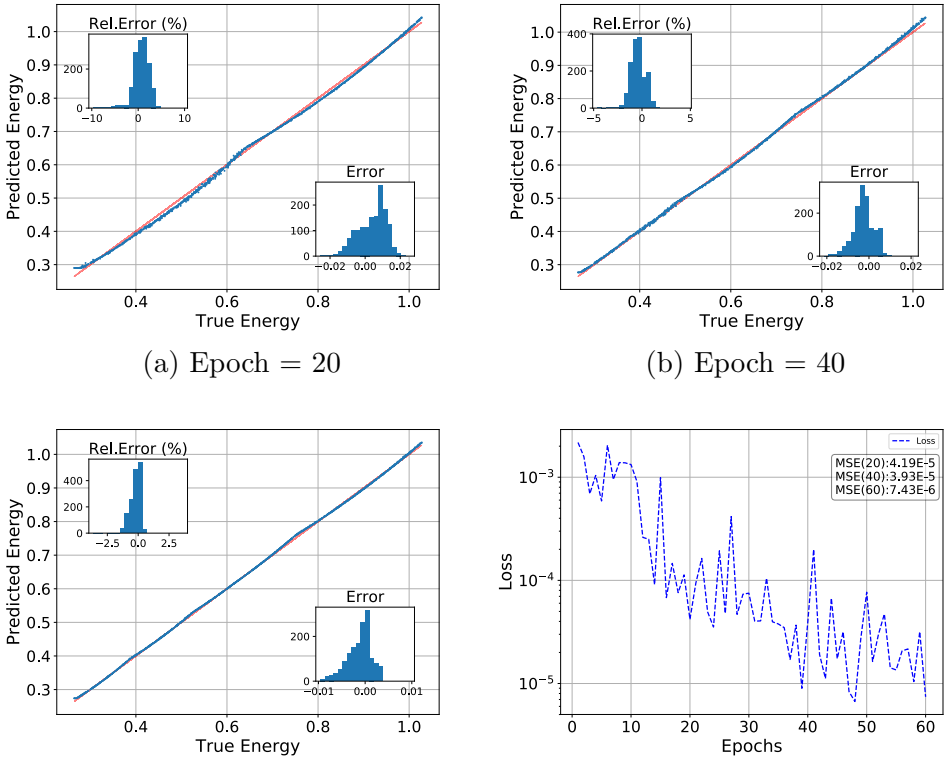
(d) FCN[128, 30, 30, 10, 4]

Figure 8: FCN[128, 30, 30, 10, 4] results for $g = 0$. Here (a) is interaction energy, (b) potential energy, (c) kinetic energy and (d) total energy predictions.

Only number of neurons in the output layer is increased to 4, and still network predicts total energy of the system within same sensitivity as former FCN network.

4.3.2 Interacting Systems

In interacting systems, values of interaction parameter g are 0.1, 1, 10, 20 and the results are given in this order. In this stage, the values of the interaction parameter g can be thought directly as a coefficient of a nonlinear partial differential equation rather than its physical meaning. In $g = 1$ scenario, contribution of interaction parameter to Eq. (18) will be more vivid. (We do not take into account the attractive interaction case which is represented by negative g. In such a situation, BEC collapses [9].)



(a) Epoch = 20

(b) Epoch = 40

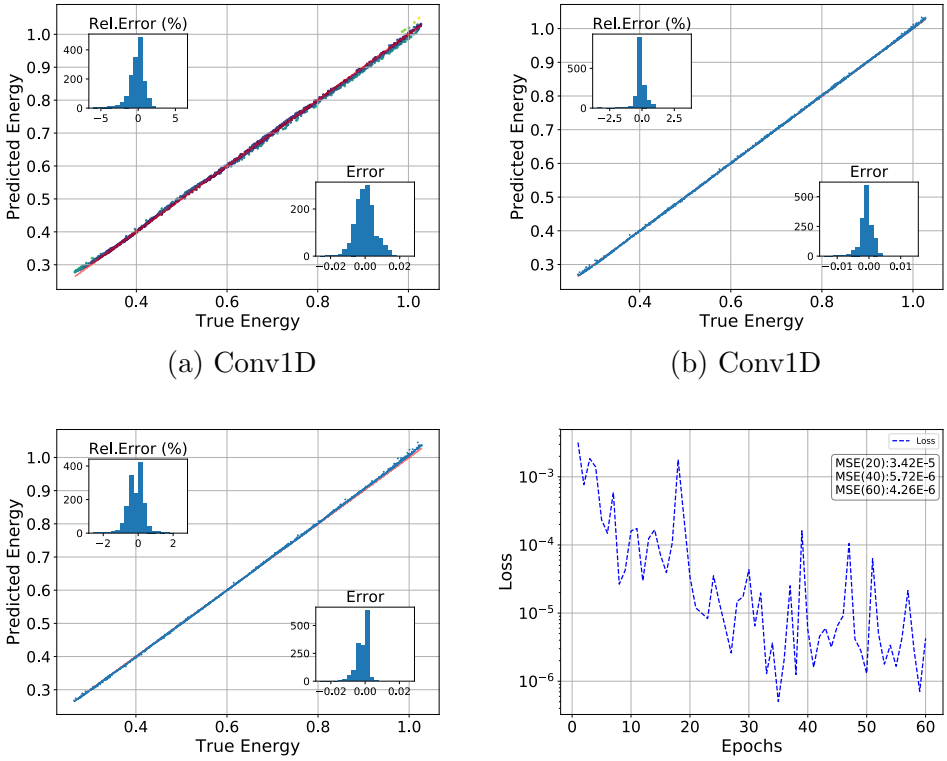
(c) Epoch = 60

(d) Loss

Figure 9: FCN[128, 30, 30, 10, 1] results for $g = 0.1$.

In first 20 epoch Fig. 9(a), there are relatively higher deviations in lower energies. In Fig. 9(b), deviations become smaller but still visible. In Fig. 9(c), prediction line is smoother but still deviations remains in lower energy levels.

Although the accuracy of the network is high, loss figure Fig. 9(d) still has fluctuation.

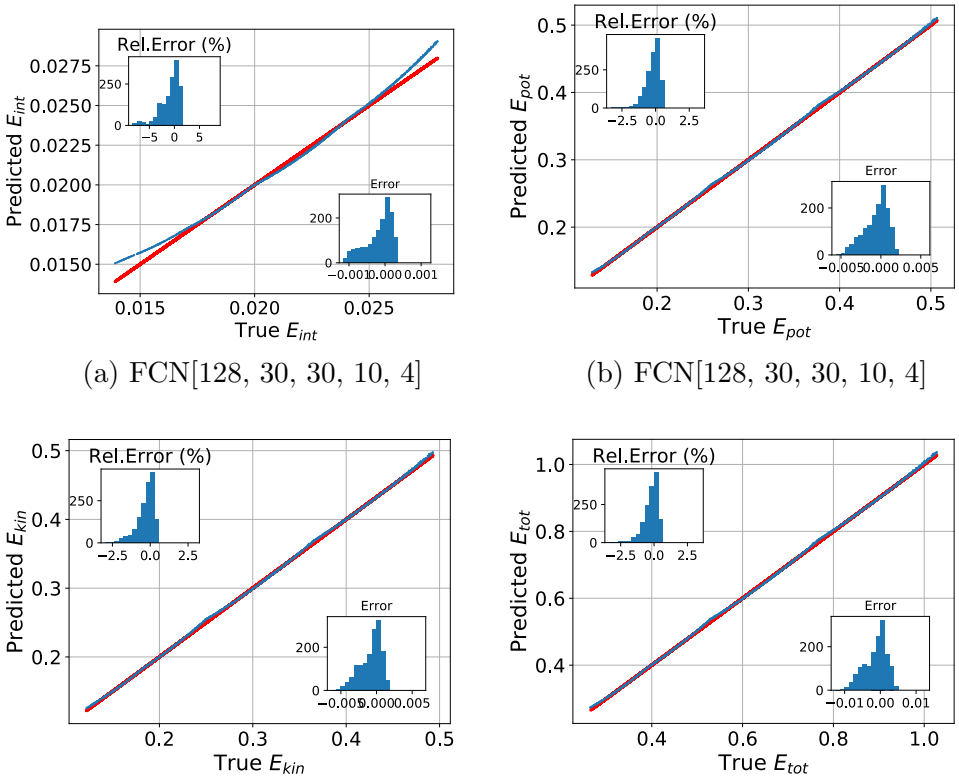


(a) Conv1D (b) Conv1D

(c) Conv1D (d) Conv1D

Figure 10: CNN results for $g = 0.1$

In Fig. 10(a) deviation has nearly same characteristic with FCN in lower energies. After 60 epochs Fig. 10(c), there is a visible shift in higher energies but this does not occur in Fig. 10(b). Local peeks can be seen on Fig. 10(d) which increases error. Different techniques may be applied to recover the state at Fig. 10(b). For instance, before final test, network can check itself and compare different losses at different epochs and can select the minimum case.



(a) FCN[128, 30, 30, 10, 4]

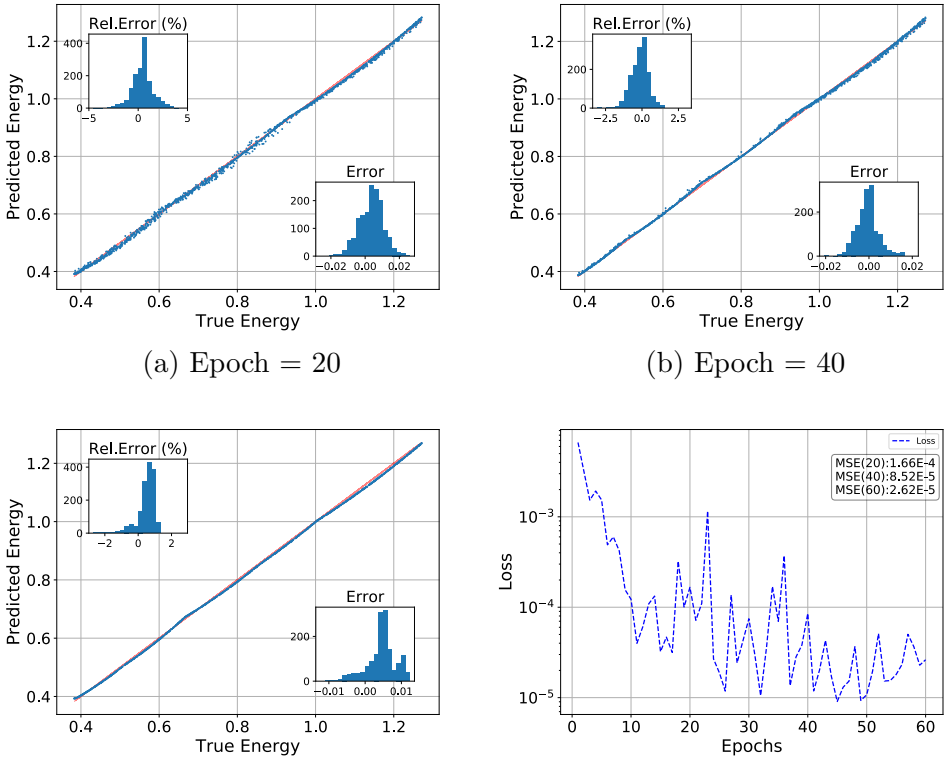
(b) FCN[128, 30, 30, 10, 4]

(c) FCN[128, 30, 30, 10, 4]

(d) FCN[128, 30, 30, 10, 4]

Figure 11: Separate Energy Predictions for $g = 0.1$

Output layer of the network involves four layer and since our network is fully connected, some of the weights at previous layers are shared indirectly. The order of interaction energy is much smaller than the others. This is the reason of bias in Fig 11(a). Because, the order of incoming inputs are same as other energy levels and network has not enough example to reduce weights and bias values in order to increase prediction accuracy. Applying a proper normalization to the variables may reduce the error. In this time, incoming inputs will be in the same order.



(a) Epoch = 20

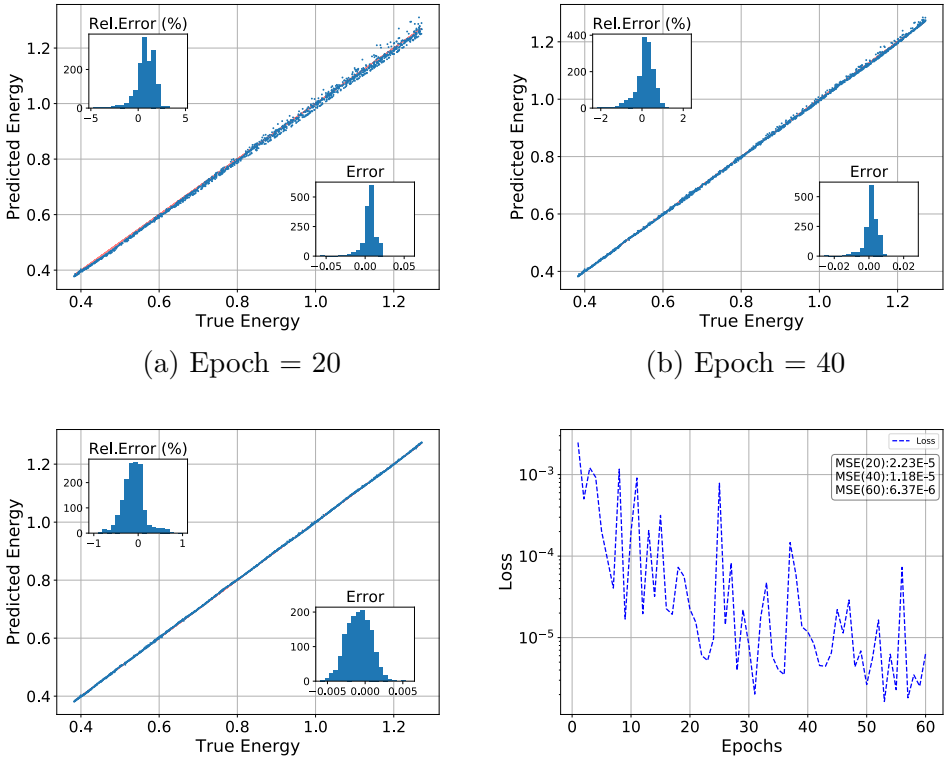
(b) Epoch = 40

(c) Epoch = 60

(d) Loss

Figure 12: FCN[128, 30, 30, 10, 1] results for $g = 1$

Relative error gets smaller after each 20 epoch and network goes into saturation after 40 epoch.



(a) Epoch = 20

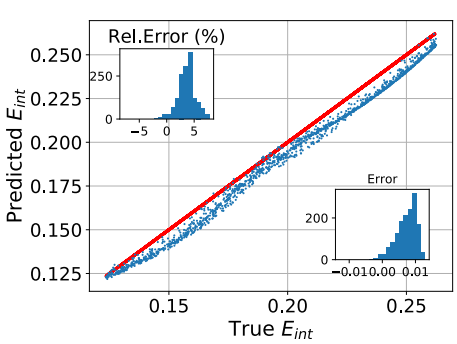
(b) Epoch = 40

(c) Epoch = 60

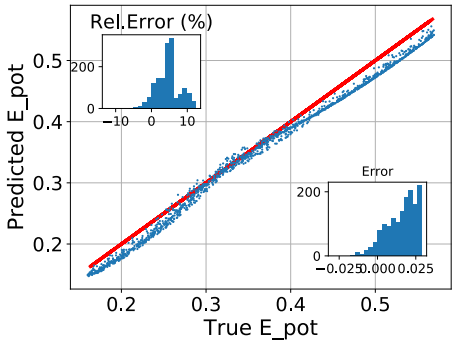
(d) Loss

Figure 13: CNN results for $g = 1$

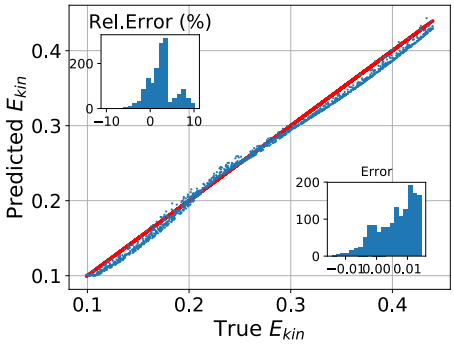
Even the loss of the CNN decreases slower than FCN Fig. 12(d) and involves local peeks, its relative error at epoch 40 and 60 is smaller.



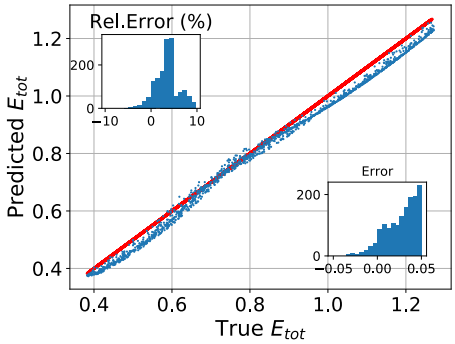
(a) FCN[128, 30, 30, 10, 4]



(b) FCN[128, 30, 30, 10, 4]



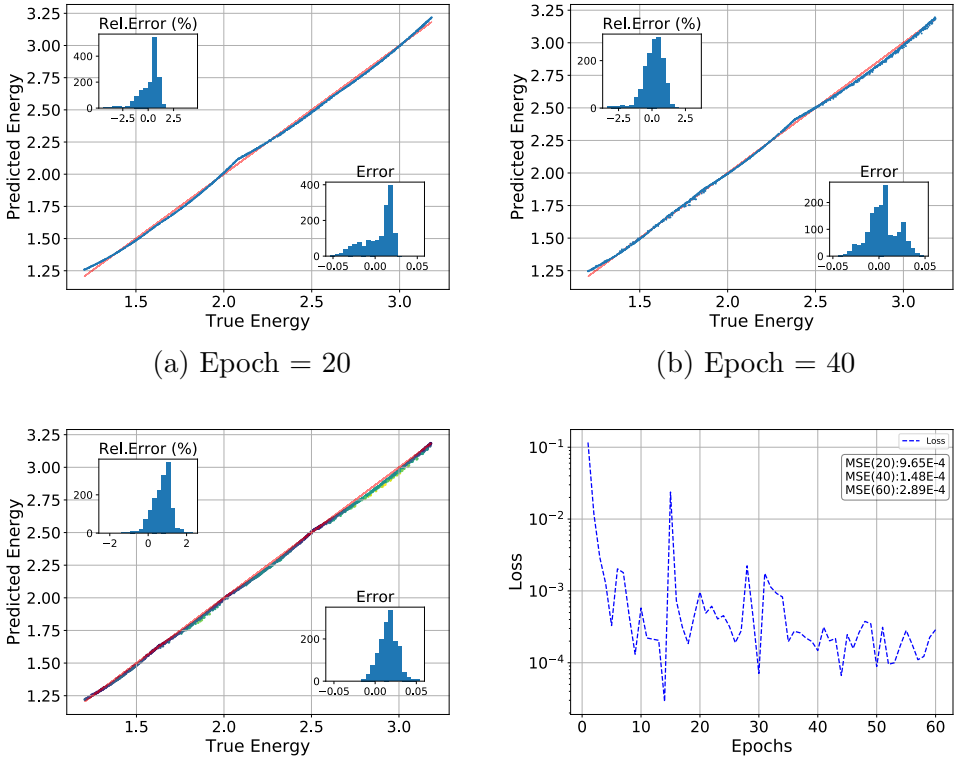
(c) FCN[128, 30, 30, 10, 4]



(d) FCN[128, 30, 30, 10, 4]

Figure 14: FCN[128, 30, 30, 10, 4], Separate energy predictions for $g = 1$.

This is the worst case in our training results. In first sight, it can be thought that a proper normalization may reduce the error and increase the accuracy but the difference between energy values are not as higher as in the case $g = 0.1$, so the problem is not due to normalization. Here, contribution of interaction parameter to Eq. (18) is in same order with other terms. So, it can be said that the degree of freedom of the network is not enough to handle such a situation.



(a) Epoch = 20

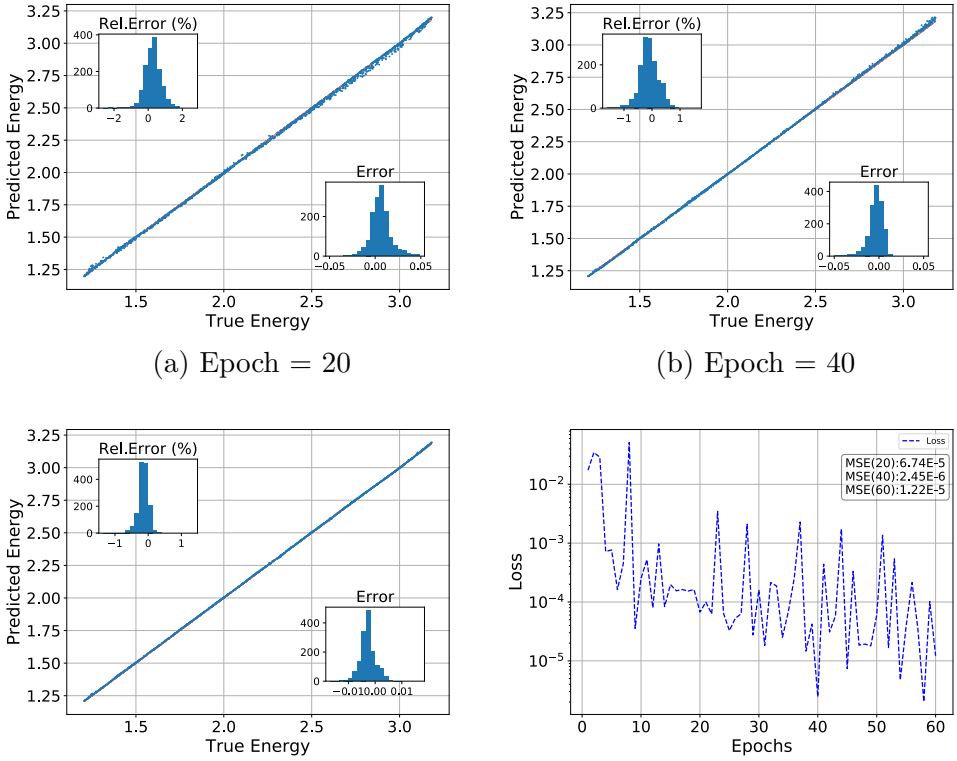
(b) Epoch = 40

(c) Epoch = 60

(d) Loss

Figure 15: FCN[128, 30, 30, 10, 1] predictions for $g = 10$

In Fig. 15(a) same deviation occurs in lower and higher energies as in previous results. Loss of the network decreases rapidly but order of it greater than the case of which $g = 1$



(a) Epoch = 20

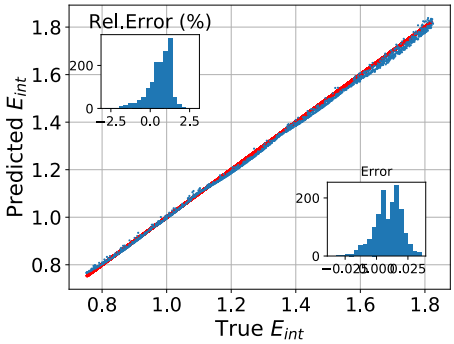
(b) Epoch = 40

(c) Epoch = 60

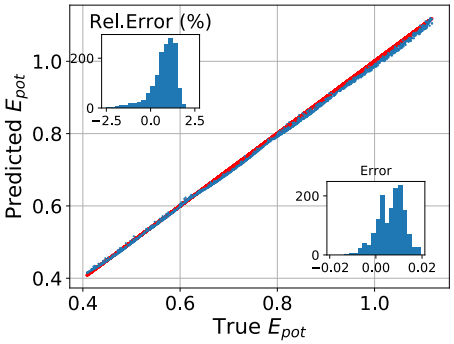
(d) Loss

Figure 16: CNN results for $g = 10$

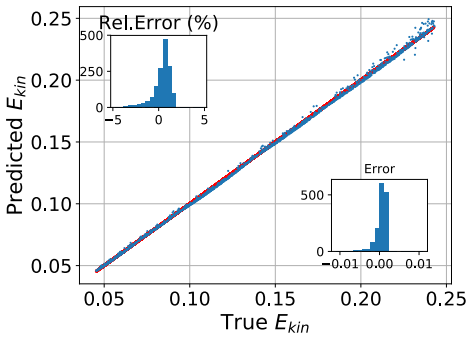
Accuracy and precision of the CNN are greater than FCN's, both relative error and MSE are smaller than FCN predictions. Also unlike FCN, there is no bias in first 20 epochs. Another important subject is that last 20 epochs increases MSE but reduces relative error slightly.



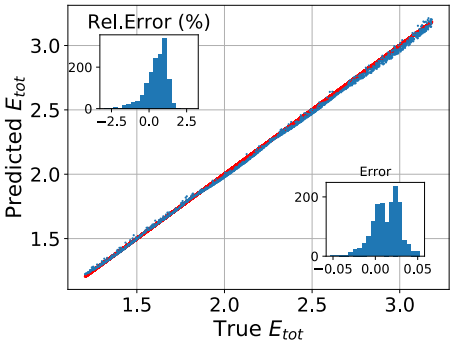
(a) FCN[128, 30, 30, 10, 4]



(b) FCN[128, 30, 30, 10, 4]



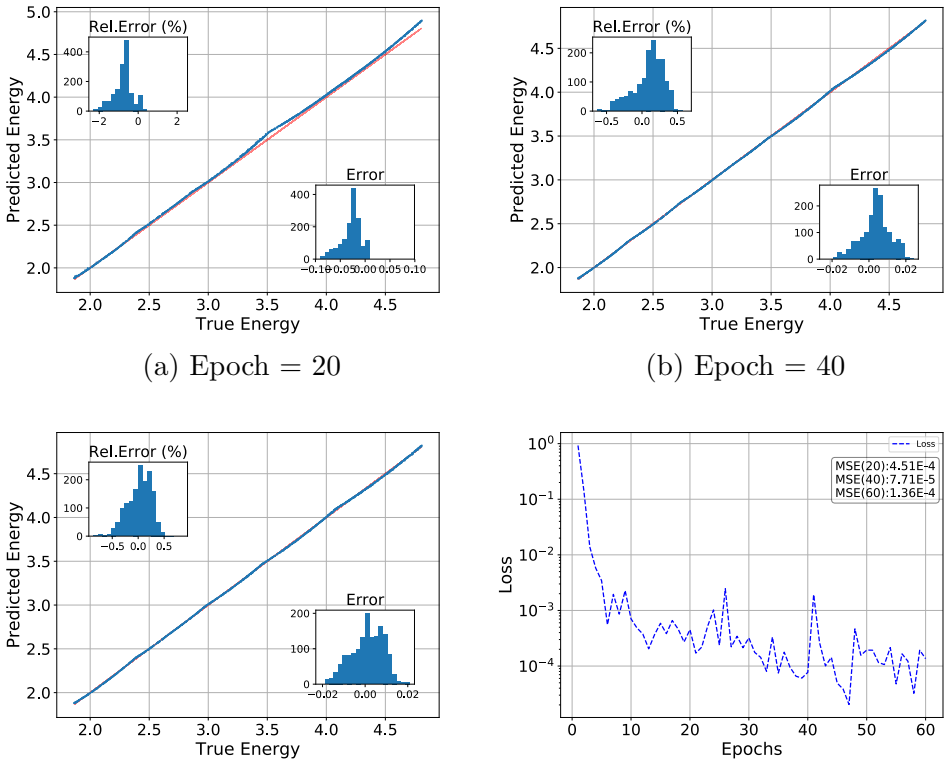
(c) FCN[128, 30, 30, 10, 4]



(d) FCN[128, 30, 30, 10, 4]

Figure 17: Separate prediction g=10

Result of separate energy prediction for $g = 10$ is not poor as $g = 1$. This also shows that dominance of a term in the equation directly affects the prediction.



(a) Epoch = 20

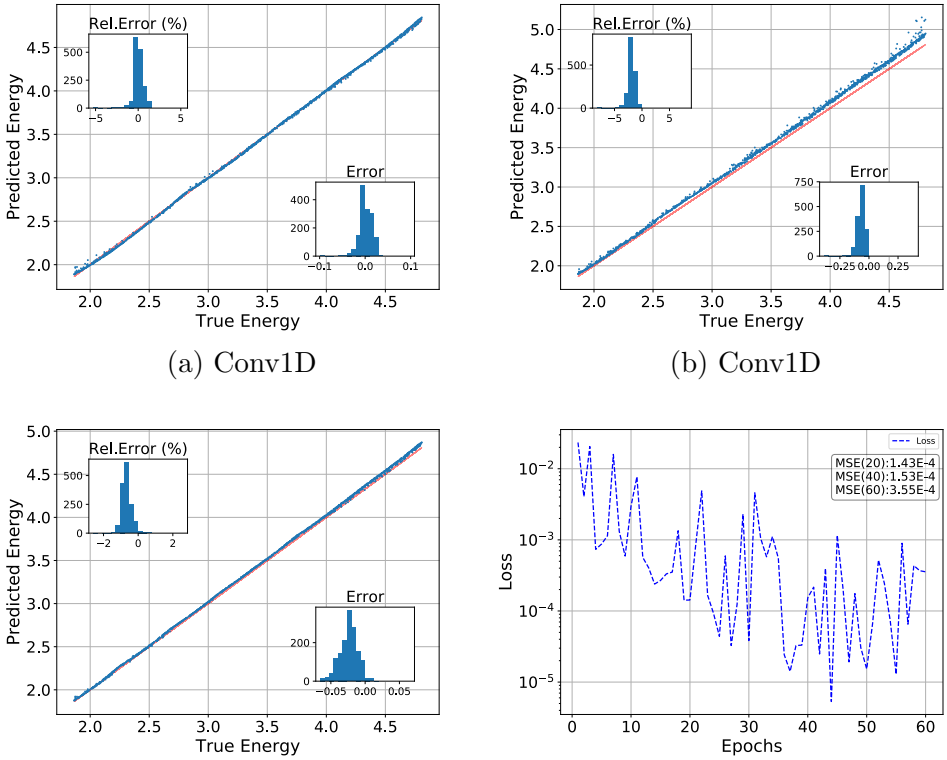
(b) Epoch = 40

(c) Epoch = 60

(d) Loss

Figure 18: FCN[128, 30, 30, 10, 1] predictions for g = 20

In Fig. 18(d), the network goes into saturation before 10 epochs. This shows that how the arrangement of the dataset and initialization of the network affects the predictions. A shuffling on both dataset and network initialization may produce different loss graph.

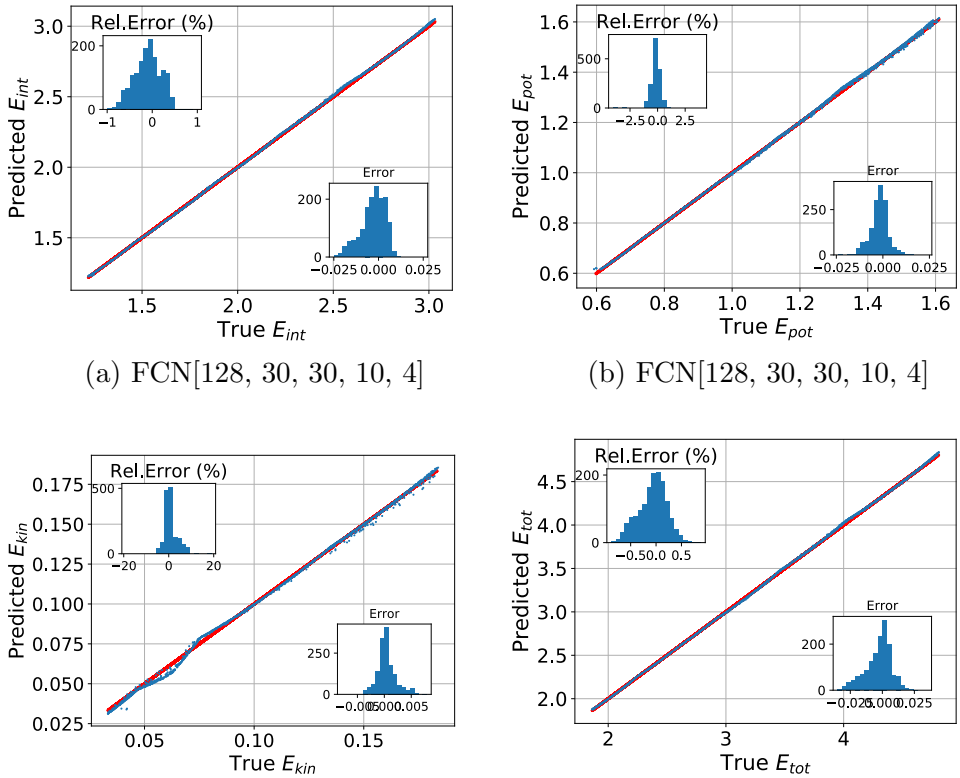


(a) Conv1D (b) Conv1D

(c) Conv1D (d) Conv1D

Figure 19: CNN predictions for $g = 20$

Unbiased predictions in first 20 epochs becomes biased after 40 epochs. It is because of energy distribution in the dataset. The number of examples at higher energies is greater than number of lowers.



(a) FCN[128, 30, 30, 10, 4]

(b) FCN[128, 30, 30, 10, 4]

(c) FCN[128, 30, 30, 10, 4]

(d) FCN[128, 30, 30, 10, 4]

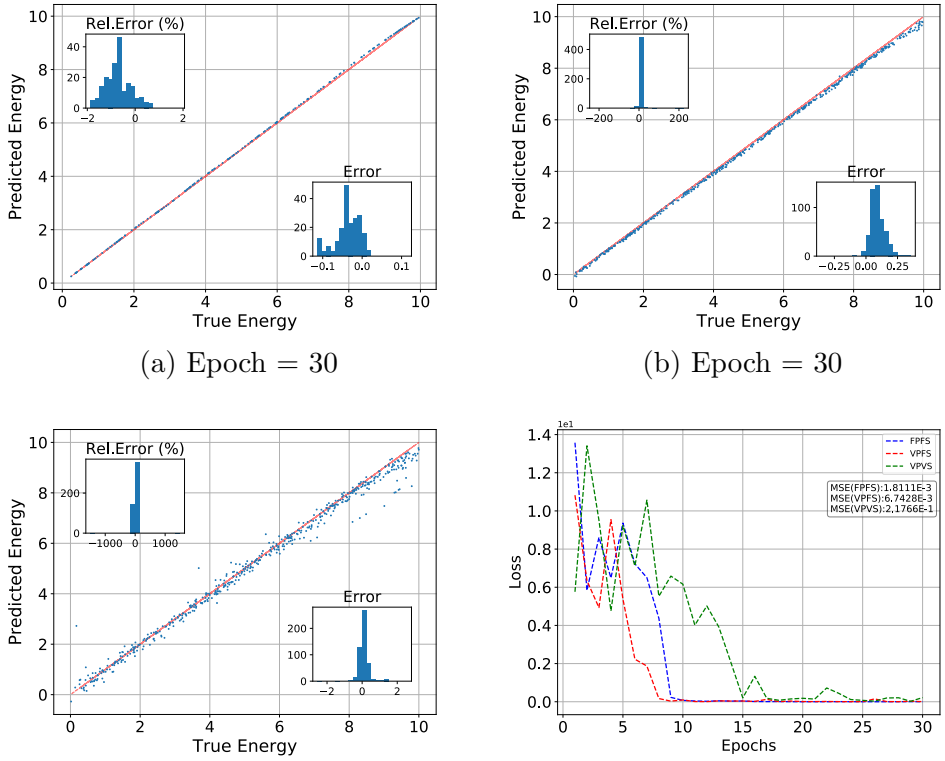
Figure 20: Separate predictions for g=20

The problem in Fig. 20(c) is same as in the case $g = 0.1$ which is shown in Fig. 11(a). The order of kinetic energy is lower than other energies.

4.3.3 Inverse Problem Prediction of g

In this section, we try to predict the interaction parameter for the density. We accept that g is positive and the value of it can vary between 0 to 10. We used modified version of CNN to predict interaction parameter. Modification only involves a change in input layer of the network. We increased the number of channels to 2 and one of the channels represents potential the other is density. Dataloaders also updated to handle new input representation.

There are three different cases. The first one is that both potential and shift are fixed, only the interaction parameter varies. In the second one we used random angular frequencies. In final case, we both randomized angular frequency and shift. **fix axis labels**



(a) Epoch = 30 (b) Epoch = 30

(c) Epoch = 30 (d) Loss

Figure 21: In (a) is angular frequency is fixed and there is no shift (FPFS). In (b) angular frequency is randomized and no shift (VPFS). In (c) both angular frequency and shift are randomized (VPVS).

When there is no randomization in parameters the network handles the situation easily but when angular frequency and shift starts to vary, the network produces poorer results. It is an expected situation and it does not show that the network is not capable of to predict interaction parameter. As shown in Fig. 21(c), precision of the network is not good but the accuracy is. Expansion in the training dataset and increase in total number of epochs will fix the problem.

5 Conclusion

In this study, we implemented an artificial neural network to show a proof of concept that machine learning methods can also be applied to non-linear Schrödinger Equation. Our results show that when the harmonic trapping potential is given, ground state energy of the Bose-Einstein can be obtained without solving the corresponding Gross-Pitaevskii Equation. We also managed to predict interaction, kinetic, potential and total energy of a system separately. Error in our results differs case to case. The worst result occurred in separate energy predictions when the interaction parameter is 1 and the relative error came out about %10. Because of the contribution of the interaction parameter to the Gross-Pitaevskii equation, all terms become same order and the degree of freedom of the network is not enough to make predictions. However, in most cases except separate energy predictions relative error is lower than %2. In inverse problem, we built another neural network to predict interaction parameter when potential and density are given. The prediction accuracy decreases when angular frequency of the trapping potential and shift of equilibrium point is not fixed but mean of the relative error for it is 1.15.

Our work only involves harmonic trapping potential and limited interaction parameters. Each trained network only accept inputs for a fixed interaction parameter. Our future work will focus on more generic situations such that the network will be able to predict ground state energy of a condensate when both potential and interaction parameter is random. It is also possible to work with two dimensional trapping potential. Our aim is first successful implementation of random interaction parameter and one dimensional random potentials. After that, we will try two dimensional case.

References

[1] M. A. Nielsen, "Neural networks and deep learning," 2015.

[2] PhysicsML, "Papers."

[3] G. Carleo and M. Troyer, "Solving the quantum many-body problem with artificial neural networks," *Science*, vol. 355, no. 6325, pp. 602–606, 2017.

[4] Z. Cai, "Approximating quantum many-body wave-functions using artificial neural networks," *arXiv preprint arXiv:1704.05148*, 2017.

[5] L. Wang, "Discovering phase transitions with unsupervised learning," *Physical Review B*, vol. 94, no. 19, p. 195105, 2016.

[6] E. M. Stoudenmire and D. J. Schwab, "Supervised learning with quantum-inspired tensor networks," *arXiv preprint arXiv:1605.05775*, 2016.

[7] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, "Quantum machine learning, 2016," *arXiv preprint arXiv:1611.09347*.

[8] K. Mills, M. Spanner, and I. Tamblyn, "Deep learning and the schrödinger equation," *Physical Review A*, vol. 96, no. 4, p. 042113, 2017.

[9] C. F. Barenghi and N. G. Parker, *A primer on quantum fluids*. No. arXiv: 1605.09580, Springer, 2016.

[10] G. R. Dennis, J. J. Hope, and M. T. Johnsson, "Xmds2: Fast, scalable simulation of coupled stochastic partial differential equations," *Computer Physics Communications*, vol. 184, no. 1, pp. 201–208, 2013.

[11] P. Muruganandam and S. K. Adhikari, "Fortran programs for the time-dependent gross–pitaevskii equation in a fully anisotropic trap," *Computer Physics Communications*, vol. 180, no. 10, pp. 1888–1912, 2009.

[12] H.-G. Zimmermann, A. Minin, and V. Kusherbaeva, "Comparison of the complex valued and real valued neural networks trained with gradient descent and random search algorithms.," in *ESANN*, 2011.

[13] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[14] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[16] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

A APPENDIX A

A.1 Comment on Python Codes

There are two neural network codes and three utility codes. Code of FCN is feedforwardnetwork.py, CNN is cnnnetwork1d.py. analyzer.py is kind of a network tracer. All information about network internals and iteration information is saved and traced by class inside this module. readdata.py reads the data generated by XMDS and produces proper representations of the data. sampletrainloader.py takes the data from readdata.py and produces Pytorch's tensors. run_training.py automatizes the training process.

Codes are not optimized and involves badly chosen inconsistent naming conventions. But volume of the code is small and it is self explanatory.

A.2 Neural Network and Utility Codes

Fully Connected Network (FCN)
(feedforwardnetwork.py)

```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data_utils
from torchvision import datasets, transforms
from torch.autograd import Variable

import matplotlib.pyplot as plt
import numpy as np
import readmnistdata as rm
import sampletrainloader as tl
import analyzer as an
import os.path

# Training settings
parser = argparse.ArgumentParser(description='Fully Connected
↪  FeedForwardNetwork for nonlinearSE')

parser.add_argument('--batch-size',        type=int,             default=30,
↪  metavar='N',    help = 'input batch size for training (default: 64)')
parser.add_argument('--test-batch-size',   type=int,             default=2000,
↪  metavar='N',    help = 'input batch size for testing (default: 1000)')
```

```
parser.add_argument('--epochs',            type=int,             default=60,
↪  metavar='N',    help = 'number of epochs to train (default: 10)')
parser.add_argument('--lr',                type=float,
↪  default=0.001,              metavar='LR',   help = 'learning rate
↪  (default: 0.01)')
parser.add_argument('--momentum',          type=float,           default=0.5,
↪  metavar='M',    help = 'SGD momentum (default: 0.5)')
parser.add_argument('--no-cuda',           action='store_true',
↪  default=False,                              help = 'disables CUDA
↪  training')
parser.add_argument('--seed',              type=int,             default=1,
↪  metavar='S',    help = 'random seed (default: 1)')
parser.add_argument('--log-interval',      type=int,             default=10,
↪  metavar='N',    help = 'how many batches to wait before logging training
↪  status')
parser.add_argument('--network-arch',      type=int,             default=[128,
↪  40, 40, 20, 1],   nargs='+',  help = 'Network arch : (default:
↪  256-40-20-1)')
parser.add_argument('--training-len',      type=int,             default=8500,
↪  help = 'Training len (default: 3500)')
parser.add_argument('--test-len',          type=int,             default=1500,
↪  help = 'Test len (default: 500)')
parser.add_argument('--runtime-count',     type=int,             default=0,
↪  help = 'this parameter counts that how many times the program is runned')
parser.add_argument('--show-progress',     action='store_true',
↪  default=False,                              help = 'display progress
↪  (default:False)')
parser.add_argument('--data-file',         type=str,
↪  default="potential-g-10-.dat",              help = 'data file to read
↪  (default = "potential.dat")')
parser.add_argument('--label-file',        type=str,
↪  default="energy-g-10-.dat",                 help = 'label file to read
↪  (default = "energy.dat")')
parser.add_argument('--inter-param',       type=float,           default=0.0,
↪  help = 'interaction parameter program uses this parameter to choose which
↪  file to open (default: 0)')

args = parser.parse_args()
args.cuda = not args.no_cuda and torch.cuda.is_available()

torch.manual_seed(args.seed)
if args.cuda:
    torch.cuda.manual_seed(args.seed)

kwargs = {'num_workers': 1, 'pin_memory': True} if args.cuda else {}

#input_size, hidden_size, hidden2_size, num_classes = args.network_arch
input_size, hidden_size, hidden2_size, hidden3_size, num_classes =
↪  args.network_arch
num_epochs = args.epochs
batch_size = args.batch_size
learning_rate = args.lr
training_len = args.training_len
test_len = args.test_len
data_file = args.data_file
label_file = args.label_file

if (args.inter_param).is_integer():
```

```
    args.inter_param = int(args.inter_param)
print("FFN running, Interaction param: {}".format(args.inter_param))

data_file = "potential-g-{}-.dat".format(args.inter_param)
label_file = "energy-g-{}-.dat".format(args.inter_param)

t = tl.nonlinear1D(data_file, label_file, training_len, test_len)
train_dataset, test_dataset = t.init_tensor_dataset()

train_loader = data_utils.DataLoader(train_dataset, batch_size =
↪  args.batch_size, shuffle=True, **kwargs)
test_loader = data_utils.DataLoader(test_dataset, batch_size =
↪  args.test_batch_size, shuffle=False, **kwargs)

class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, hidden2_size)
        self.fc3 = nn.Linear(hidden2_size, hidden3_size)
        self.fc4 = nn.Linear(hidden3_size, num_classes)


    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        out = self.relu(out)
        out = self.fc4(out)
        return out



net = Net(input_size, hidden_size, num_classes)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
res = an.analyzer(args)

def train(epoch):
    for batch_idx, (data, labels) in enumerate(train_loader):
        data = Variable(data)
        labels = Variable(labels).float()
        optimizer.zero_grad()
        outputs = net(data)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

#         if (i) % res.batch_size == 0 and args.show_progress == True:
#             print('Epoch [%d/%d], Step [%d/%d], Loss: %.4f' %(res.cur_epoch,
↪  res.epochs, i, res.training_len // res.batch_size, loss.data[0]))
#             res.step(loss.data[0])

        if batch_idx % args.log_interval == 0 and args.show_progress == True:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
```

```
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.data[0]))

    res.step(loss.data[0])


info_file_name = "../figs/FFNTEST/" + os.path.splitext(data_file)[0]


def test():
    for data, labels in test_loader:
        data = Variable(data)
        outputs = net(data)
        predicted = outputs.data.numpy()
        real = test_dataset.target_tensor.numpy()
        real = real.reshape([test_len, 1])
        res.calc_error(real, predicted)
        #res.display_plot()

#         global info_file_name
#         file_name = info_file_name + "epoch-{}-.inf".format(res.cur_epoch)
#         an.save_info(res, file_name)

    return predicted

while res.cur_epoch != res.epochs + 1:
    train(res.cur_epoch)
#     if res.cur_epoch % (res.epochs / 3) == 0:
#         test()
    res.cur_epoch +=1

res.cur_epoch = res.epochs
test()
```

Convolutional Neural Network (CNN)

(cnnnetwork1d.py)

```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data_utils
from torchvision import datasets, transforms
from torch.autograd import Variable

import matplotlib.pyplot as plt
import numpy as np
import readmnistdata as rm
import sampletrainloader as tl
import analyzer as an
import os.path

# Training settings
parser = argparse.ArgumentParser(description='CNN for nonlinearSE')

parser.add_argument('--batch-size',        type=int,             default=30,
↪  metavar='N',    help = 'input batch size for training (default: 64)')
```

```
parser.add_argument('--test-batch-size',   type=int,             default=1500,
↪  metavar='N',    help = 'input batch size for testing (default: 1000)')
parser.add_argument('--epochs',            type=int,             default=60,
↪  metavar='N',    help = 'number of epochs to train (default: 10)')
parser.add_argument('--lr',                type=float,           default=1e-3,
↪  metavar='LR',   help = 'learning rate (default: 0.01)')
parser.add_argument('--momentum',          type=float,           default=0.2,
↪  metavar='M',    help = 'SGD momentum (default: 0.5)')
parser.add_argument('--no-cuda',           action='store_true',
↪  default=False,                               help = 'disables CUDA
↪  training')
parser.add_argument('--seed',              type=int,             default=1,
↪  metavar='S',    help = 'random seed (default: 1)')
parser.add_argument('--log-interval',      type=int,             default=10,
↪  metavar='N',    help = 'how many batches to wait before logging training
↪  status')
parser.add_argument('--network-arch',      type=str,
↪  default="conv1d",           nargs='+',      help = 'Network arch :
↪  (default: 256-40-20-1)')
parser.add_argument('--training-len',      type=int,             default=8500,
↪  help = 'Training len (default: 3500)')
parser.add_argument('--test-len',          type=int,             default=1500,
↪  help = 'Test len (default: 500)')
parser.add_argument('--runtime-count',     type=int,             default=0,
↪  help = 'this parameter counts that how many times the program is runned')
parser.add_argument('--show-progress',     action='store_true',
↪  default=False,                               help = 'display progress
↪  (default:False)')
parser.add_argument('--data-file',         type=str,
↪  default="potential-g-20-.dat",               help = 'data file to read
↪  (default = "potential.dat")')
parser.add_argument('--label-file',        type=str,
↪  default="energy-g-20-.dat",                  help = 'label file to read
↪  (default = "energy.dat")')
parser.add_argument('--inter-param',       type=float,           default=0.0,
↪  help = 'interaction parameter program uses this parameter to choose which
↪  file to open (default: 0)')



args = parser.parse_args()
args.cuda = not args.no_cuda and torch.cuda.is_available()

torch.manual_seed(1)

kwargs = {'num_workers': 1, 'pin_memory': True} if args.cuda else {}

#input_size, hidden_size, hidden2_size, num_classes = args.network_arch
num_epochs = args.epochs
batch_size = args.batch_size
learning_rate = args.lr
training_len = args.training_len
test_len = args.test_len
#data_file = args.data_file
#label_file = args.label_file

if (args.inter_param).is_integer():
```

```
    args.inter_param = int(args.inter_param)

print("CNN running, Interaction param: {}".format(args.inter_param))


data_file = "potential-g-{}-.dat".format(args.inter_param)
label_file = "energy-g-{}-.dat".format(args.inter_param)


t = tl.nonlinear1D(data_file, label_file, training_len, test_len, cnn = True)
train_dataset, test_dataset = t.init_tensor_dataset()

train_loader = data_utils.DataLoader(train_dataset, batch_size =
↪  args.batch_size, shuffle=True, **kwargs)
test_loader = data_utils.DataLoader(test_dataset, batch_size =
↪  args.test_batch_size, shuffle=False, **kwargs)


class CnnNet(nn.Module):
    def __init__(self):
        super(CnnNet, self).__init__()
        self.conv1 = nn.Conv1d(1, 10, 2)
        self.conv2 = nn.Conv1d(5, 20, 2)
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(310, 100)
        self.fc2 = nn.Linear(100, 20)
        self.fc3 = nn.Linear(20, 1)


    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x

model = CnnNet()
criterion = F.mse_loss
optimizer = optim.Adam(model.parameters(), lr = args.lr)
res = an.analyzer(args)

def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = Variable(data), Variable(target).float()
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0 and (args.show_progress ==
        ↪  True):
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
```

```
                100. * batch_idx / len(train_loader), loss.data[0]))
    res.step(loss.data[0])


info_file_name = "../figs/CNN/" + os.path.splitext(data_file)[0]

def test():
    model.eval()
    test_loss = 0
    correct = 0
    for data, target in test_loader:
        data, target = Variable(data, volatile = True),
        ↪  Variable(target).float()
        outputs = model(data)
        predicted = outputs.data.numpy()
        real = test_dataset.target_tensor.numpy()
        real = real.reshape([test_len, 1])
        res.calc_error(real, predicted)
        #res.display_plot()

        global info_file_name
        file_name = info_file_name +
        ↪  "conv1d-epoch-{}-.inf".format(res.cur_epoch)
        an.save_info(res, file_name)

    return predicted


while res.cur_epoch != res.epochs + 1:
    train(res.cur_epoch)
    if res.cur_epoch % (res.epochs / 3) == 0:
        test()
    res.cur_epoch +=1

res.cur_epoch = res.epochs
```

Utility modules:

(analyzer.py)

```
import numpy as np
import matplotlib
matplotlib.use("pgf")
import matplotlib.pyplot as plt
import argparse
import pickle
import torch
from scipy.optimize import curve_fit
from decimal import Decimal

class analyzer(object):
    testdataset = None
    predicted = None
    error = None
    def __init__(self, args):
        self.learning_rate = args.lr
        self.arch = args.network_arch
        self.training_len = args.training_len
        self.test_len = args.test_len
```

```
        self.seed = args.seed
        self.batch_size = args.batch_size
        self.epochs = args.epochs
        self.cur_epoch = 1
        self.runtime_count = args.runtime_count
        self.loss = np.empty(0)

    def calc_error(self, testdataset, predicted):
        self.testdataset = testdataset
        self.predicted = predicted
        #self.error = np.array([(self.predicted[i] - self.testdataset[i])**2
        ↪  for i in range(self.test_len)])
        self.error = sum((self.predicted - self.testdataset)**2)
        self.error = (self.error/self.test_len)
        self.relative_error = sum(np.abs(self.predicted - self.testdataset) /
        ↪  sum(self.testdataset))

    def display_plot(self, file_name = None):
        err = self.testdataset - self.predicted
        relative_err = ((self.testdataset - self.predicted) /
        ↪  (self.testdataset)) * 100
        fig, ax1 = plt.subplots()
        left, bottom, width, height = [0.65, 0.20, .2, .2]
        inset = fig.add_axes([left, bottom, width, height])
        left2, bottom2, width2, height2 = [0.19, 0.60, .2, .2]
        inset2 = fig.add_axes([left2, bottom2, width2, height2])
        ax1.plot(self.testdataset, self.testdataset, "--r", label=None,
        ↪  linewidth = 1, alpha=0.5)
        #ax1.scatter(self.testdataset, self.predicted, c = np.abs(err), s = 4)
        ax1.plot(self.testdataset, self.predicted, ".", label = None,
        ↪  markersize = 2)
        #ax1.set_title("FNN{}".format(self.arch))
        ax1.set_xlabel("True Energy", fontsize = 20)
        ax1.set_ylabel("Predicted Energy", fontsize = 20)
        ax1.tick_params(labelsize = 18)
        ax1.legend()
        ax1.grid()
        #props = dict(boxstyle='square', facecolor='white', alpha=0.5)
        #textstr = "MSE:{:.4E}\nTraining
        ↪  Length:{}\nEpoch:{}\nBatch:{}".format(Decimal(float(self.error)),
        ↪  self.training_len, self.cur_epoch, self.batch_size)
        #ax1.text(0.03, 0.85, textstr, transform=ax1.transAxes, fontsize=11,
        ↪  verticalalignment='top', bbox=props)
        #ax1.text(, ,
        ↪  "{}\nlr={}\nepoch={}\ntrain_len={}\ntest_len={}\nerror={}".format(self.arch,
        ↪  self.learning_rate, self.cur_epoch, self.training_len,
        ↪  self.test_len, self.error))

        inset.hist(err, range=[-np.amax(np.abs(err)), np.amax(np.abs(err))],
        ↪  bins=20)
        inset2.hist(relative_err, range=[-np.amax(np.abs(relative_err)),
        ↪  np.amax(np.abs(relative_err))], bins=20)
        inset.set_title("Error", fontsize = 18)
        inset.tick_params(labelsize=12)
        inset2.set_title("Rel.Error (%)", fontsize = 18)
        inset2.tick_params(labelsize=12)

        if file_name == None:
```

```
            figure = plt.gcf()
            figure.set_size_inches(8,6)
            plt.show()
        else:
            figure = plt.gcf()
            figure.set_size_inches(8,6)
            plt.savefig(file_name + ".svg", format = "svg", dpi=1200)
            plt.clf()




    def display_plot2(self, file_name = None):
        features = ['E_int', 'E_kin', 'E_pot', 'E_Total']
        features_l = ['$E_{int}$', '$E_{kin}$', '$E_{pot}$', '$E_{tot}$']
        err = []
        relative_err = []

        for i in range(len(features)):
            err.append(self.testdataset[:,i] - self.predicted[:,i])
            relative_err.append(((self.testdataset[:,i] - self.predicted[:,i])
            ↪  / (self.testdataset[:,i])) * 100)
            fig, ax1 = plt.subplots(num=features[i])
            left, bottom, width, height = [0.65, 0.20, .2, .2]
            inset = fig.add_axes([left, bottom, width, height])
            left2, bottom2, width2, height2 = [0.19, 0.60, .2, .2]
            inset2 = fig.add_axes([left2, bottom2, width2, height2])
            ax1.plot(self.testdataset[:,i], self.testdataset[:,i], "--r",
            ↪  linewidth = 3)
            ax1.plot(self.testdataset[:,i], self.predicted[:,i], ".",
            ↪  markersize = 2)
            #ax1.set_title("FNN{}".format(self.arch))
            ax1.set_xlabel("True " + features_l[i], fontsize = 20)
            ax1.set_ylabel("Predicted " + features_l[i], fontsize = 20)
            ax1.tick_params(labelsize = 18)
            ax1.legend()
            ax1.grid()
            #props = dict(boxstyle='square', facecolor='white', alpha=0.5)
            #textstr = "MSE:{:.4E}\nTraining
            ↪  Length:{}\nEpoch:{}\nBatch:{}".format(Decimal(float(self.error[i])),
            ↪  self.training_len, self.cur_epoch, self.batch_size)
            #ax1.text(0.03, 0.85, textstr, transform=ax1.transAxes,
            ↪  fontsize=11, verticalalignment='top', bbox=props)


            inset.hist(err[i], range=[-np.amax(np.abs(err[i])),
            ↪  np.amax(np.abs(err[i]))], bins=20)
            inset2.hist(relative_err[i],
            ↪  range=[-np.amax(np.abs(relative_err[i])),
            ↪  np.amax(np.abs(relative_err[i]))], bins=20)
            inset.set_title("Error")
            inset.tick_params(labelsize=12)
            inset2.set_title("Rel.Error (%)", fontsize = 18)
            inset2.tick_params(labelsize=12)


            if file_name != None:
```

```
                fig.savefig(file_name + "{}-".format(features[i]) + ".svg",
                ↪  format = "svg")
                fig.clf()

        if file_name == None:
            plt.show()
            plt.clf()




    def plot_error(self):
        error = self.testdataset - self.predicted
        plt.hist(error)
        plt.show()

    def step(self, loss_val):
        #self.cur_epoch += 1
        self.loss = np.append(self.loss, loss_val)

#    def gauss(x, a, x0, sigma):
#        return a * np.exp(-(x-x0)**2 / (2*sigma**2))
#
#    def gaussian_dist():
#        err = self.testdataset - self.predicted)
#        x_r = np.amax(np.abs(err))
#        x = np.linspace(-x_r, x_r, self.test_len)
#        mean = np.mean(err)
#        sigma = np.std(err)
#        popt,pcov = curve_fit(gauss, x, err, p0 = [1,mean,sigma])
#        plt.hist(err)
#        plt.plot(x, gauss(x, *popt))
#        plt.show()


def save_info(obj, file_name):
    f = open(file_name, "wb")
    pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)


def load_info(file_name):
    f = open(file_name, "rb")
    return pickle.load(f)


def save_state(model, optimizer, epoch, file_name):
    state = {'epoch': epoch,
             'model_state_dict': model.state_dict(),
             'optim_state_dict': optimizer.state_dict() }
    torch.save(state, file_name)


def load_state(file_name):
    state = torch.load(file_name)
    epoch = state['epoch']
    mst = (state['model_state_dict'])
    ost = (state['optim_state_dict'])
```

```
    return epoch, mst, ost


def combine_plots(datum, file_name = None):
    fig, axarr = plt.subplots(1, len(datum), sharey=True)
    #left, bottom, width, height = [0.65, 0.20, .2, .2]

    err = []
    for i, data in enumerate(datum):
        err.append(data.testdataset - data.predicted)
        #inset = fig.add_axes([left, bottom, width, height])
        axarr[i].plot(data.testdataset, data.testdataset, "--r", label="real
        ↪  data", linewidth = 3)
        axarr[i].plot(data.testdataset, data.predicted, ".", label =
        ↪  "predicted", markersize = 2)
        axarr[i].set_title("FNN{}".format(data.arch))
        axarr[i].set_xlabel("Real $\\mu$", fontsize = 18)
        axarr[i].set_ylabel("Predicted", fontsize = 18)
        axarr[i].tick_params(labelsize = 12)
        axarr[i].legend()
        axarr[i].grid()
        props = dict(boxstyle='square', facecolor='white', alpha=0.5)
        textstr = "MSE:{:.4E}\nTraining
        ↪  Length:{}\nEpoch:{}\nBatch:{}".format(Decimal(float(data.error)),
        ↪  data.training_len, data.cur_epoch, data.batch_size)
        axarr[i].text(0.03, 0.85, textstr, transform=axarr[i].transAxes,
        ↪  fontsize=11, verticalalignment='top', bbox=props)

    plt.show()
```

(readdata.py)

```
import os
import numpy as np


def read_data(data_file, label_file, train_len=800, test_len=200):


    data_file = os.path.join("../data/nonlinearSE/", data_file)
    label_file = os.path.join("../data/nonlinearSE/", label_file)

    #data = np.load(data_file)
    data = np.loadtxt(data_file)
    label = np.loadtxt(label_file)

    total_len = train_len + test_len
    train_data = data[0:train_len]
    train_label = label[0:train_len]

    test_data = data[train_len:total_len]
    test_label = label[train_len:total_len]

    return train_data, train_label, test_data, test_label


def read_data2(data_file, label_file, train_len=800, test_len=200):
```

```
    data_file = os.path.join("../data/nonlinearSE/", data_file)
    label_file = os.path.join("../data/nonlinearSE/", label_file)

    data = np.load(data_file)
    label = np.loadtxt(label_file)

    total_len = train_len + test_len
    train_data = data[0:train_len]
    train_label = label[0:train_len]

    test_data = data[train_len:total_len]
    test_label = label[train_len:total_len]

    return train_data, train_label, test_data, test_label
```

(sampletrainloader.py)

```
from __future__ import print_function
import torch
import torch
import torch.utils.data as data_utils
import numpy as np
import readmnistdata as rm
import readdata as rd

class nonlinearSE(object):
    def __init__(self, root = "../data"):
        self.train_len = 50000
        self.test_len = 10000

        data = list(rm.read())
        self.train_data = np.zeros([self.train_len, 28, 28])
        self.train_label = np.zeros([self.train_len], dtype = 'int')

        self.test_data = np.zeros([self.test_len, 28, 28])
        self.test_label = np.zeros([self.test_len], dtype = 'int')

        for i in range(self.train_len):
            self.train_data[i], self.train_label[i] = data[i]

        for i in range(self.test_len):
            self.test_data[i], self.test_label[i] = data[i + self.train_len]

        self.train_data_tensor = torch.from_numpy(self.train_data).float()
        self.train_label_tensor = torch.from_numpy(self.train_label).long()
        self.train_data_tensor = self.train_data_tensor.unsqueeze(1)

        self.test_data_tensor = torch.from_numpy(self.test_data).float()
        self.test_label_tensor = torch.from_numpy(self.test_label).long()
        self.test_data_tensor = self.test_data_tensor.unsqueeze(1)


    def get_data(self, train = True):
        if train:
            return (self.train_data_tensor, self.train_label_tensor)
        else:
            return (self.test_data_tensor, self.test_label_tensor)
```

```
class nonlinear1D__(object):
    def __init__(self, root = "../data", label_vectorize = False):
        self.train_len = 50000
        self.test_len = 10000

        data = list(rm.read(matrix = False))

        self.train_data = np.zeros([self.train_len, 784])
        self.test_data = np.zeros([self.test_len, 784])

        if label_vectorize:
            self.train_label = np.zeros([self.train_len, 10], dtype = 'int')
            self.test_label = np.zeros([self.test_len, 10], dtype = 'int')
        else:
            self.train_label = np.zeros([self.train_len], dtype = 'int')
            self.test_label = np.zeros([self.test_len], dtype = 'int')


        for i in range(self.train_len):
            x, y = data[i]
            self.train_data[i], self.train_label[i] = x, get_label(y,
            ↪  label_vectorize)

        for i in range(self.test_len):
            x, y = data[i + self.train_len]
            self.test_data[i], self.test_label[i] = x, get_label(y,
            ↪  label_vectorize)

        self.train_data_tensor = torch.from_numpy(self.train_data).float()
        self.train_label_tensor = torch.from_numpy(self.train_label).long()
#         self.train_data_tensor = self.train_data_tensor.unsqueeze(1)

        self.test_data_tensor = torch.from_numpy(self.test_data).float()
        self.test_label_tensor = torch.from_numpy(self.test_label).long()
#         self.test_data_tensor = self.test_data_tensor.unsqueeze(1)


    def get_data(self, train = True):
        if train:
            return (self.train_data_tensor, self.train_label_tensor)
        else:
            return (self.test_data_tensor, self.test_label_tensor)

class nonlinear1D(object):
    def __init__(self, data_file, label_file, train_len, test_len,
    ↪  label_vectorize = False, unsqueeze = False):
        self.train_len = train_len
        self.test_len = test_len

        self.train_data, self.train_label, self.test_data, self.test_label =
        ↪  rd.read_data(data_file, label_file, self.train_len, self.test_len)
        self.train_data_tensor = torch.from_numpy(self.train_data).float()
        self.train_label_tensor = torch.from_numpy(self.train_label).float()
        if unsqueeze == True: self.train_data_tensor =
        ↪  self.train_data_tensor.unsqueeze(1)

        self.test_data_tensor = torch.from_numpy(self.test_data).float()
        self.test_label_tensor = torch.from_numpy(self.test_label).float()
```

```
        if unsqueeze == True: self.test_data_tensor =
        ↪  self.test_data_tensor.unsqueeze(1)

    def get_data(self, train = True):
        if train:
            return (self.train_data_tensor, self.train_label_tensor)
        else:
            return (self.test_data_tensor, self.test_label_tensor)

    def init_tensor_dataset(self):
        train_dataset = data_utils.TensorDataset(*self.get_data())
        test_dataset = data_utils.TensorDataset(*self.get_data(train = False))
        return train_dataset, test_dataset

def get_label(j, label_vectorize):
    if not label_vectorize:
        return j
    e = np.zeros((10))
    e[j] = 1.0
    return e
```

(run_trainings.py)

```
import shlex, subprocess, time
import random as rnd
import time as time


inter_params = [0, 0.1, 1, 10, 20]
epoch = 60

net_cnn = "cnnnetwork1d.py"
net_ffn = "feedforwardnetwork2.py"


start = time.time()
for i in range(len(inter_params)):

    arguments = "--inter-param={} --epochs={}".format(inter_params[i], epoch)

    cmdline = "python {} {}".format(net_ffn, arguments)
    args = shlex.split(cmdline)
    p = subprocess.Popen(args)
    p.wait()

#    cmdline = "python {} {}".format(net_cnn, arguments)
#    args = shlex.split(cmdline)
#    p = subprocess.Popen(args)
#    p.wait()

print("Total Time = {}".format(time.time() - start))
```