BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC PHENIKAA

BÁO CÁO ĐỒ ÁN LIÊN NGÀNH

HỌC PHẦN ĐỒ ÁN LIÊN NGÀNH

ĐỀ TÀI

Ứng dụng mô hình học sâu mô phỏng động lực học phân tử

Giảng viên hướng dẫn:	Ts. Phạm Tiến Lâm		
Sinh viên thực hiện:	Trần Minh Hiếu	21011601	K15CNTT3
	Từ Lê Tú Uyên	21012401	K15KHMT
Khoa:	Công Nghệ Thông Tin		

HÀ NỘI, THÁNG 7 2024

Contents

List of Figures

Abstract

Molecular dynamics simulations are essential for studying atomic-scale physical and chemical processes, demanding precise calculations of energies and forces. Although electronic structure methods such as density functional theory provide reliable results, they are computationally intensive. This study proposes a comprehensive approach using a deep neural network (NN) to enhance the computation of force and energy in molecular dynamics simulations. The NN extracts embedded features from pairwise interactions among atoms and their neighbors, aggregating these to predict atomic forces and potential energies. By fine-tuning these pairwise interaction features, we optimize model performance while considering many-body effects and other atomic interaction physics. Integrating the Coulomb matrix of local structures alongside pairwise data improves force and energy predictions, particularly for silicon systems. The models exhibit high accuracy and transferability to larger systems, underscoring their robustness in practical applications.

1 Introduction

The computation of energy, particularly the forces within a chemical system, is pivotal in the computer simulation of matter and material design. These computations facilitate sampling in the phase space through molecular dynamics (MD) or Monte Carlo simulation. Typically, energies and forces are derived from electronic structure calculations based on density functional theory (DFT). While DFT has become a standard method in the simulation of matter, it has limitations in versatility, accuracy, and computational efficiency. To reduce computational costs, the energies and forces of large systems are often approximated using empirical models, which construct the potential energy surface (PES) from low-dimensional terms representing covalent bonds, angles, and dihedral angles. Forces are then calculated from the gradient of the PES, based on atomic coordinates. Although these methods are efficient for simulating large biosystems, they struggle to accurately describe chemical reactions and processes involving the formation or dissociation of covalent bonds.

Recently, machine learning approaches have been developed to learn the PES from materials structures and corresponding DFT energies. These methods use algorithms to determine the functional relationship between structure and energy, often decomposing the total energy of a material into contributions from constituent atoms, based on interactions within a cutoff radius. For instance, local chemical environments of atoms can be represented using atom-distribution-based symmetry functions or atomic density distributions, with the similarity between structures estimated by overlapping their atomic densities.

Calculating atomic forces, derived from the gradient of total energy, presents challenges when dealing with complex learned relationships between structure and energy, especially in deep learning models with sparse representations. Therefore, complementing energy models with force representations can enhance MD calculations. Despite the extensive research on learning atomic PES from data, fewer studies focus on learning atomic forces. This study proposes a deep neural network model that considers two-body terms as pairwise interactions within a chemical environment. By extracting hidden features from these interactions, the model estimates forces and energies, showing high accuracy and transferability to larger systems. The model's performance was validated with silicon-lithium systems, demonstrating its effectiveness in reproducing DFT-derived forces and energies.

The remainder of this paper is organized as follows: the second section covers the data used in our study, while Section 3 details the methodology. Section 4 introduces the convolutional neural network architecture. Section 5 presents the experimental results and discussion, followed by a summary in the conclusion section.

2 Data

The dataset used in this study consists of the atomic configurations and corresponding energies and forces for a silicon crystal structure with 64 atoms. This dataset is stored in the 'Si_64.xyz' file and follows a specific format that includes detailed information about the unit cell parameters, atomic coordinates, and forces acting on each atom.

2.1 Dataset Format and Structure

The 'Si_64.xyz' file is structured as follows:

- **Number of Atoms:** The first line of the file indicates the number of atoms in the system, which is 64.
- **Unit Cell Parameters:** The second line provides the unit cell parameters and the total energy of the system. The format of this line is:

```
a_x   a_y   a_z   b_x   b_y   b_z   c_x   c_y   c_z   total_energy
```

Here, **a**, **b**, and **c** are the vectors defining the unit cell, and *total_ energy* represents the total energy of the system.

- **Atomic Data:** Each subsequent line represents an atom in the system and includes the atom type, atomic coordinates, and forces. The format for these lines is:

  ```
  atom_type  x  y  z  fx  fy  fz
  ```

  where:

  - *atom_ type* is the chemical symbol of the atom (e.g., Si for silicon).
  - x, y, z are the Cartesian coordinates of the atom.
  - f_x, f_y, f_z are the components of the force acting on the atom.

2.2 Dataset Characteristics

- **Unit Cell:** The unit cell is cubic with a side length of 10.85677 Å.
- **Atomic Positions:** The positions are given in Cartesian coordinates relative to the unit cell.
- **Forces:** The forces are provided for each atom in all three Cartesian directions.

This dataset represents a 64-atom silicon crystal structure, providing comprehensive information on the atomic positions within the unit cell, the forces acting on each atom, and the overall unit cell parameters. This detailed dataset allows for the analysis of structural and energetic properties of the silicon crystal using advanced machine learning models.

2.3 Data Processing

In this section, we detail the processes and methods employed to handle and transform the data extracted from XYZ files. The XYZ file format is commonly used to store molecular structures and their properties, making it an essential part of our data pipeline. We utilized several custom Python functions to facilitate the extraction and organization of this data.

2.3.1 Reading XYZ Files

The primary step in our data processing pipeline involves reading the XYZ files. We employed four distinct functions, each tailored to extract specific information from the files:

- **read_xyz_md(name)**:
    - **Purpose**: Extracts cell vectors, atomic coordinates, atom types, and energies.
    - **Process**:
        - Reads the number of atoms from the first line.
        - Parses the cell vectors and energies from the second line.
        - Extracts atomic coordinates and atom types from subsequent lines.
        - Aggregates data into lists for cell vectors, coordinates, atoms, and energies.
- **read_xyz_force(name)**:
    - **Purpose**: Extracts cell vectors, atomic coordinates, atom types, forces, and energies.
    - **Process**:
        - Similar to `read_xyz_md`, with additional parsing for atomic forces.
        - Aggregates data into lists for cell vectors, coordinates, atoms, forces, and energies.

2.3.2 Data Aggregation and Validation

The extracted data from the XYZ files are aggregated into lists, ensuring that each frame of molecular data is correctly aligned. The functions ensure the integrity and consistency of the data by:

- Parsing the number of atoms and ensuring it matches the expected count.
- Validating the structure of each frame to ensure all necessary data points are present.
- Handling potential inconsistencies by initializing new frames appropriately.

The data processing pipeline effectively extracts and organizes molecular data from XYZ files. The tailored functions ensure that all relevant properties, such as cell vectors, atomic coordinates, atom types, forces, spins, and descriptors, are accurately parsed and aggregated. This processed data serves as a foundation for further analysis and modeling in our study.

3 Methodology

The utilization of machine learning in materials science presents unique challenges, primarily because the input to a predictive model for a property of a chemical system can vary in size and shape depending on the system being studied. Traditional machine learning algorithms typically require a fixed-dimensional vector or matrix as input. Consequently, chemical systems often need to be converted into fixed-dimensional feature vectors, which are considered fixed during the training process.

Material properties can often be decomposed into contributions from individual constituent structural elements, such as chemical bonds, bond angles, and dihedral angles. For instance, the total energy of a chemical system can be represented by considering bonds, bond angles, Lennard–Jones potentials, and other forms of classical potentials. Small structural fragments can also be used to represent the energy of a chemical system in cluster expansion formulations.

Based on this principle, our methodology decomposes a chemical structure into its structural elements. We then designed a machine learning model to extract hidden information from these structural elements to estimate material properties. The workflow of our method is illustrated in Figure 1. A chemical structure is decomposed into collections of pairwise interactions within a certain chemical environment. We employ a deep neural network (NN) to extract the embedding representation of these interactions for predicting atomic forces and total energy. The network is designed to apply to all pairs of atoms, allowing it to be used for systems of any size.

3.1 Model for Energy by Pairwise Interactions

The total energy of a solid system is a function of its atomic structure (i.e., unit cell vectors and atomic coordinates) and is invariant under translations and rotations. To model the total energy E, we decompose it into contributions from constituent atoms, referred to as atomic energy (partial energy):

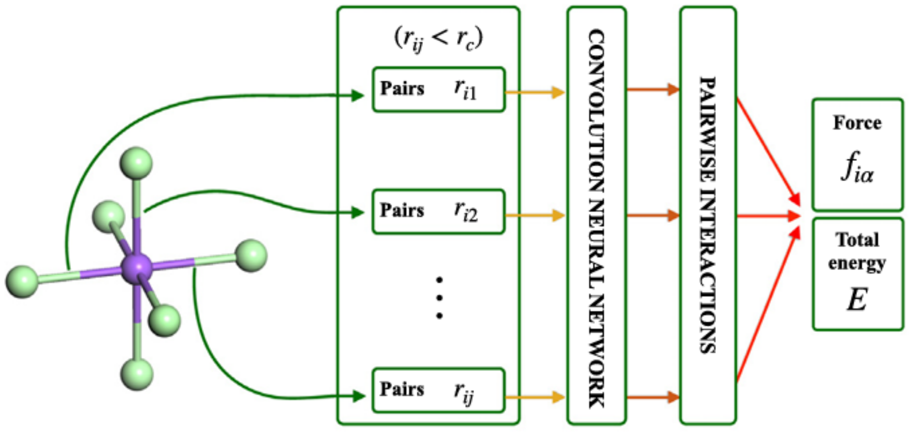


Figure 1: Workflow of our proposed framework

$$E = \sum_{i=1}^{N} E_i \tag{1}$$

where E_i is the fictitious energy of the i-th atom in the system, and N is the number of atoms. The energy of an atom is determined by its interactions with neighboring atoms within a cutoff radius. The energy can be expressed as:

$$E = \sum_{i=1}^{N} H(\mathbf{x}_i) \tag{2}$$

where $\mathbf{x}_i$ is a feature vector derived from the set of pairwise terms representing interactions between the central atom and its neighbors. Using one hidden layer, the atomic energy can be computed as:

$$E_i = H(\mathbf{x}_i) = \mathbf{W}_2 \cdot g(\mathbf{W}_1 \cdot \mathbf{x}_i) \tag{3}$$

where $\mathbf{W}_1$ and $\mathbf{W}_2$ are the weights of the hidden and output layers, respectively, and g is a nonlinear activation function.

The feature vector $\mathbf{x}_i$ is derived from pairwise terms $\mathbf{b}_{ij}$ representing interactions between atoms i and j. These are embedded using a deep NN with three hidden layers, expressed as:

$$\mathbf{a}_{ij} = \mathbf{w}_3 \cdot g(\mathbf{w}_2 \cdot g(\mathbf{w}_1 \cdot \mathbf{b}_{ij})) \tag{4}$$

where $\mathbf{w}_1$, $\mathbf{w}_2$, and $\mathbf{w}_3$ are the weights of the hidden layers.

To handle varying numbers of neighboring atoms, we use padding to ensure all vectors have the same dimensions, resulting in a 3D input tensor representation for a structure. We employ the convolutional networks to implement our model as shown in Figure. 2. The Convolutional networks (Conv2D and Conv1D layers) are then used to extract hidden features and compute atomic energies.

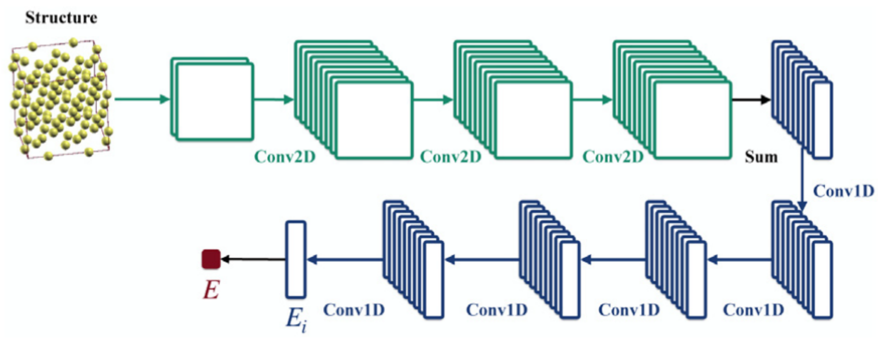


Figure 2: Deep neural network model for predicting the total energy.

3.2 Atomic Force from Pairwise Interactions

Atomic forces arise from interactions with neighboring atoms within a cutoff radius. The force on an atom is a superposition of pairwise forces:

$$f_i^\alpha = \sum_j f_{ij}^\alpha = \sum_j F_{ij} \frac{r_{ij}^\alpha}{|r_{ij}|} f_c(r_{ij}) \tag{5}$$

where f_i^α is the α component of the force on atom i, F_{ij} is the pairwise-force function, and $f_c(r_{ij})$ is a smooth cutoff function. The cutoff function f_c is defined as:

$$f_c = \frac{1}{2}\left[\cos\left(\frac{\pi r_{ij}}{r_c}\right) + 1\right] \tag{6}$$

where r_c is the cutoff radius. The pairwise-force function F_{ij} quantifies the strength of the force between atoms i and j.

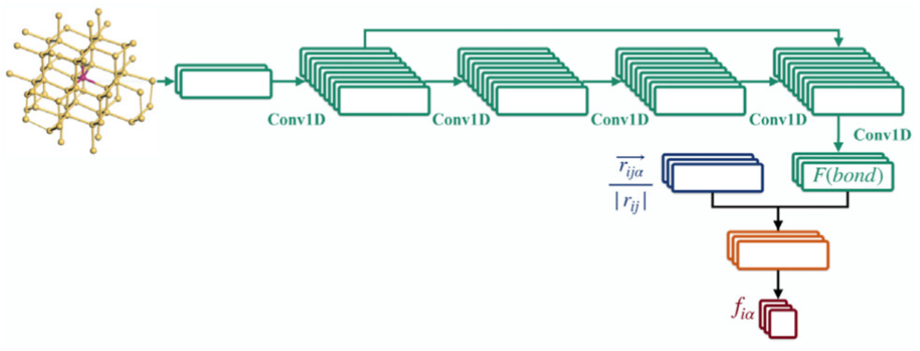


Figure 3: Deep neural network for predicting atomic forces.

We utilize a convolutional network technique to implement the force model, as shown in Figure 3. For each atom, we derive the feature vectors for all pairwise interactions, b_{ij}, within a cutoff radius and the unit vectors $\frac{r_{ij}}{|r_{ij}|}$ for all pairs of atoms. These vectors are stacked in descending order of the distance to the center atom, forming the feature matrix, $\mathbf{B}$, and the unit vector matrix, $\mathbf{N}$. To ensure consistent matrix dimensions for all atoms, we employ a padding technique.

We then apply Conv1D layers to the feature matrix $\mathbf{B}$. The final layer has a single neuron, producing a list of pairwise forces. By multiplying these forces by the corresponding unit vectors, we obtain the forces exerted by neighboring atoms on the central atom. Finally, we use Eq. (5) to calculate the net force on the atom of interest.

4 Convolutional Neural Networks

4.1 Overview of Convolutional Neural Networks

In machine learning, classifiers assign class labels to data points. For example, an image classifier assigns class labels (e.g., dog, cat) to objects present in an image. Convolutional Neural Networks [**?**], or CNNs, are a type of classifier that excels at this task!

A CNN [**?**]is a neural network: an algorithm used to recognize patterns in data. Neural networks generally consist of a set of neurons organized into layers, each with its own learnable weights and biases. Let's break down CNN into its basic building blocks:

1. Tensor: A tensor can be understood as a multidimensional data structure, similar to a matrix but with n dimensions. In CNNs, tensors typically have three

dimensions (height, width, and channel) until reaching the output layer. Tensors are crucial in multi-dimensional data processing, such as images in CNNs.

2. Neuron: A neuron can be seen as a function that receives multiple inputs and produces a single output. In CNNs, neurons are often represented by color maps ranging from red to blue, called activation maps. Neurons perform computations on input data to create useful features for image classification.

3. Layer: Layers in CNNs are groups of neurons that perform the same operation. Neurons in the same layer usually share the same hyperparameters, and layers are fundamental building blocks in the architecture of CNNs.

4. Kernel weights and biases: These are the parameters for each neuron in CNNs. Weights are used to weight the inputs, while biases are used to adjust the neuron's output. These values are fine-tuned during the network's training process, allowing the network to adapt to the provided data.

5. Class score: A CNN outputs a differentiable score function, represented as class scores in its output layer. This score function is continuous and can be integrated and differentiated, which is important during the neural network's training process. In the output layer of CNNs, class scores are calculated for each class or label that the network is trained to classify. These scores usually represent the network's confidence in classifying an image into different classes. By comparing scores across classes, we can determine the class that the network predicts as the most accurate for the input image.

CNNs have proven to be powerful tools for computer vision applications and have achieved impressive performance in solving image processing problems. Next, we will delve into the functioning of the main layers in a CNN.

4.2 Layers in CNNs

4.2.1 Input Layer

The input layer in a CNN is the first part of the network, and its task is to receive and process input data, typically images or similarly structured data. Below are some important points about the input layer in CNNs:

1. Input Data: The input layer contains the input data, usually an image or a multidimensional tensor representing an image. This data typically has a fixed size

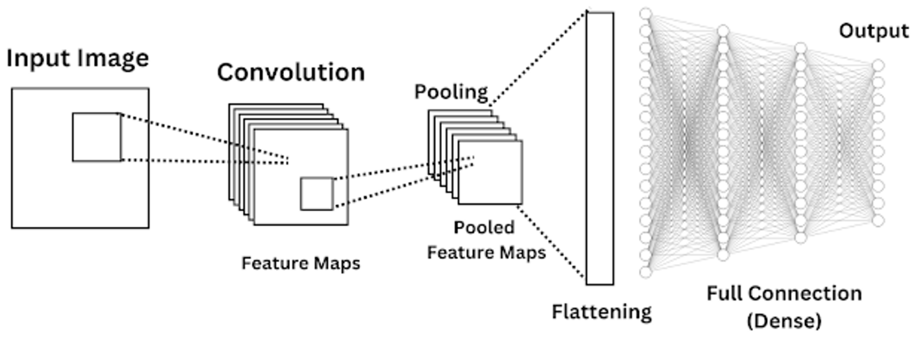


Figure 4: Basic structure of CNN

and is processed by converting it into features suitable for extracting information from the image.

2. Input Size: A CNN needs to know the size of the input data to set the output size for subsequent convolutional layers. Generally, the input size is specified before building the network.

3. Input Channels: Color images typically have three channels (RGB), while grayscale images have only one channel. The input layer must have a number of channels compatible with the input.

4. Input Preprocessing: In some cases, preprocessing can be performed on the input data to prepare it for the CNN. This may include normalizing pixel values or resizing the image to fit the network size.

5. Output Information: The input layer does not perform any calculations on the data; it simply receives the input data and passes it down to the first convolutional layers in the CNN.

The input layer is a crucial part of a CNN as it ensures that the input data is correctly prepared and passed into the network to start the learning and feature extraction process from the images.

4.2.2 Convolutional Layer

CNNs utilize a special type of layer called the convolutional layer, which makes them highly suitable for learning from image data and image-like data. Regarding image data, CNNs can be used for various computer vision tasks, such as image processing, classification, segmentation, and object detection.

The convolutional layer in CNNs allows the network to learn how to detect specific features in images, such as edges, corners, or other important characteristics. This makes CNNs very efficient in image processing because they can leverage weight sharing and use convolutional layers to extract local information from images.

4.2.2.1 Convolution Operation

- 1D Convolution (conv1D): Consider a one-dimensional signal $a(t)$ and a filter $w(t)$. The convolution of the signal and the filter is a new one-dimensional signal $b(t)$ defined by the formula:

$$b(t) = \int_u a(u)w(t-u)du \tag{7}$$

Assuming that $a(t)$ and $w(t)$ are defined only on integer values of t, the one-dimensional convolution can be defined as:

$$b(t) = \sum_{u=-\infty}^{\infty} x(u)w(t-u) \tag{8}$$

An example of discrete one-dimensional convolution (without padding) is illustrated in Figure 5.

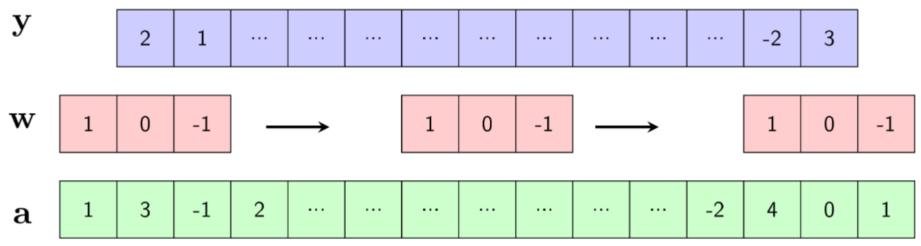


Figure 5: One-dimensional convolution

- 2D Convolution (conv2D): Similarly, consider a two-dimensional input signal $I(i,j)$ and a two-dimensional kernel $K(i,j)$. Assuming I, K are defined only on integer values of i, j, i.e., two-dimensional matrices, the two-dimensional convolution with variables is defined as:

$$S(i,j) = (I * K)(i,j) = \sum_u \sum_v I(u,v)K(i-u, j-v) \tag{9}$$

Due to the commutative property of convolution, the two-dimensional convolution of I and K can be written as:

$$S(i, j) = (K * I)(i, j) = \sum_u \sum_v I(i - u, j - v)K(u, v) \tag{10}$$

When the filter or kernel is symmetric, the convolution operation coincides with the cross-correlation. In many machine learning documents, convolution is implemented using cross-correlation:

$$S(i, j) = (I * K)(i, j) = \sum_u \sum_v I(i + u, j + v)K(u, v) \tag{11}$$

When performing convolution, we need to consider several hyperparameters for the filter: size, stride, and padding, which define how the filter moves over the input and how it interacts with the data to extract features.

1. Size: This is the size of the filter, usually defined by the width and height of the filter. For example, a 3x3 filter is a matrix with 3 rows and 3 columns. If the kernel is a square matrix, we consider a kernel of size $f \times f$.

2. Stride: Stride is the distance between consecutive positions where the filter is applied on the input. Stride determines how the filter moves over the input data. For example, with a stride of 1, the filter moves one pixel at a time, while with a stride of 2, the filter jumps over one pixel each time. Denoted as s.

3. Padding, denoted as p: Padding involves adding zeroes or other values around the input before applying the filter. Padding can help maintain the output size after applying the filter and can also help extract features from the edges of the image more effectively. There are two common types of padding: "same" (maintaining size: $p = (f - 1)/2$) and "valid" (no padding: $p = 0$).

After applying the two-dimensional convolution to the input signal I of size $n \times n$ with a kernel of size $f \times f$, stride s, and padding p, we obtain a new image of size $n_1 \times n_1$ with $n_1 = \lfloor \frac{n+2p-f}{s} + 1 \rfloor$. An example of applying two-dimensional convolution for some values of f, s, p is presented in Figure 6 and Figure 7.

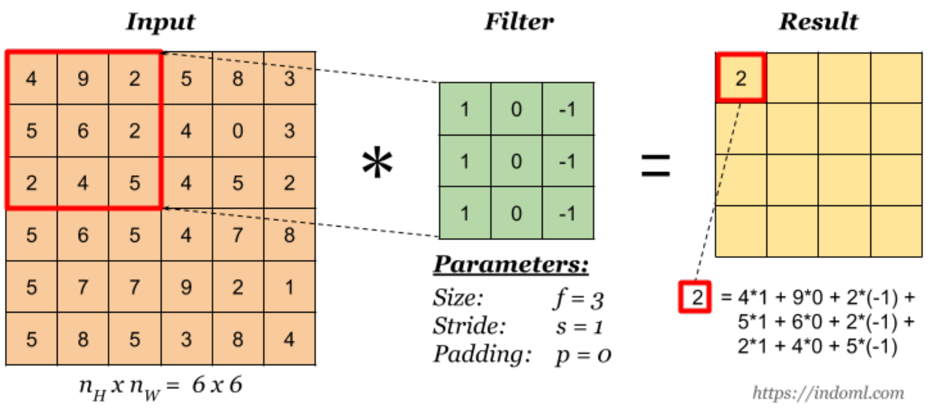


Figure 6: Two-dimensional convolution for a 6×6 image $(f = 3, s = 1, p = 0)$

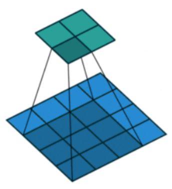
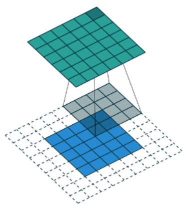
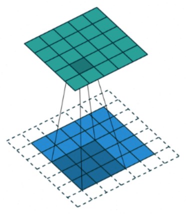
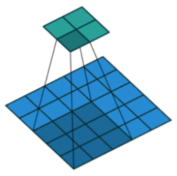
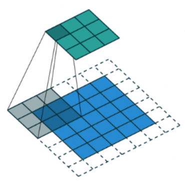
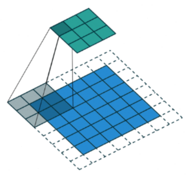
		
$P = 0, S = 1$	P bất kỳ, $S = 1$	$P = (f-1)/2, S = 1$
		
$P = 0, S = 2$	$P = (f-1)/2, S = 2$	$\frac{N+2P-f}{S} \notin \mathbb{N}$

Figure 7: Single-channel two-dimensional convolution illustration with (p, s) pairs

A more specific illustration of two-dimensional convolution with input image $X_{3\times3}$ and kernel $w_{2\times2}$:

$$\mathbf{X} = \begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix},$$

Transforming the output image into vector form with cross-correlation:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} w_1x_1 + w_2x_2 + w_3x_4 + w_4x_5 \\ w_1x_2 + w_2x_3 + w_3x_5 + w_4x_6 \\ w_1x_4 + w_2x_5 + w_3x_7 + w_4x_8 \\ w_1x_5 + w_2x_6 + w_3x_8 + w_4x_9 \end{bmatrix}$$

$$= \begin{bmatrix} w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 \\ 0 & 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} = \mathbf{Cx}$$

From the above example, we can see that two-dimensional convolution is equivalent to a linear transformation with a sparse weight matrix having shared weights (the number of non-zero elements per row is equal to the kernel size, w_1, w_2, w_3, w_4 appear in all rows of the matrix).

4.3 Conv2D Layer Calculations

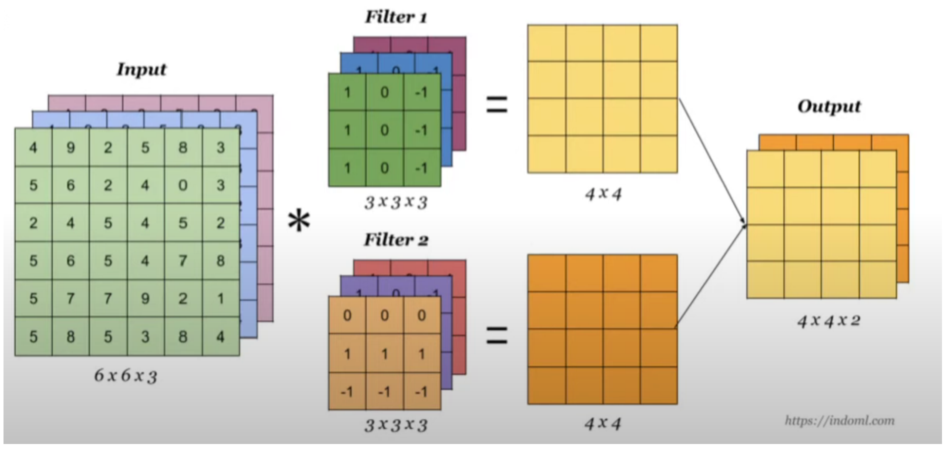


Figure 8: Conv2D Calculation Example

The Conv2D layer consists of n_f filters (of size $3 \times 3 \times n_c, 5 \times 5 \times n_c, \ldots$), where n_c is the number of input channels. The input is convolved with each filter and passed through the activation function σ_t to generate feature maps. By concatenating these feature maps, we obtain the output image of the Conv2D layer with n_f channels.

For instance, in Figure 8, the input image has $n_c = 3$ channels and there are $n_f = 2$ filters. The 16 neurons in the feature map share the weights in filter 1, and the 16 neurons in the lower feature map share the weights in filter 2.

Essentially, Conv2D is a layer in a Multi-Layer Perceptron (MLP) where the weights are shared among neurons (reducing the total number of parameters). Due to the nature of convolution and small filter sizes, the calculation results represent local properties of the input image, leading to the term feature map.

4.4 Activation Functions

After the convolution operation, the output is passed through an activation function, as illustrated in Figure 9. One reason for the breakthrough accuracy achieved by CNNs is the non-linearity introduced by the activation function.

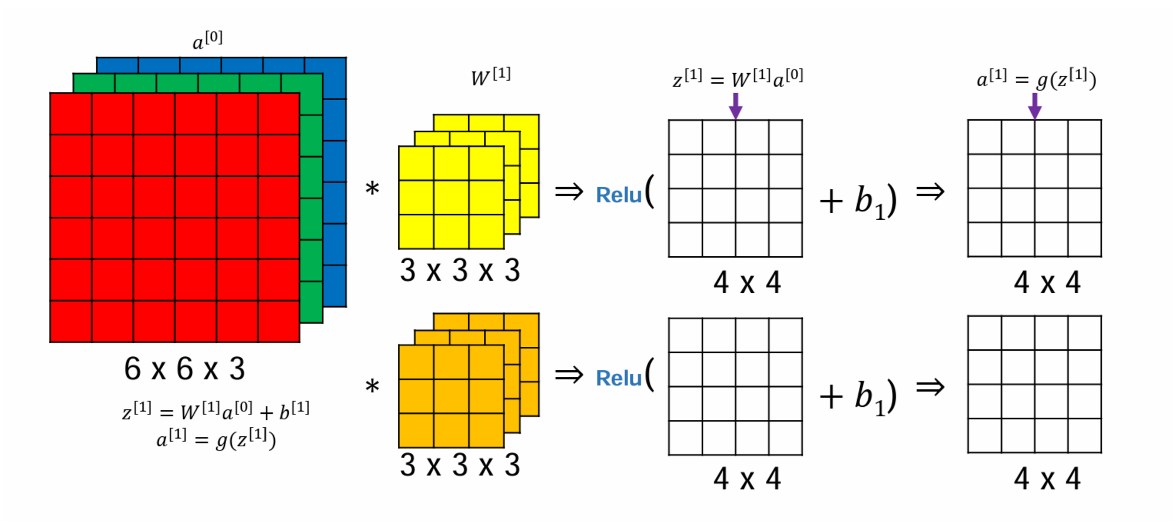


Figure 9: Activation Function in Conv2D

Non-linearity in the activation function is crucial for creating non-linear decision boundaries, so the output cannot be expressed as a linear combination of the inputs. Without a non-linear activation function, a deep CNN architecture would collapse into a single convolutional layer, failing to perform effectively.

ReLU Activation Function: The ReLU activation function is specifically used as a non-linear activation function instead of other non-linear functions like Sigmoid, as extensive experiments have shown that CNNs with ReLU activation train faster. This function is applied element-wise on every value from the input tensor.

$$\mathrm{ReLU}(x) = \max(0, x) \tag{12}$$

Softmax Activation Function: The Softmax function ensures the total output of a CNN sums to 1. Because of this, Softmax is used to convert the network's output into a probability distribution over classes (or choices).

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{13}$$

The class or choice with the highest probability after applying Softmax is usually selected as the model's prediction for the given input. Softmax helps convert results into probabilities, making it very useful in classification tasks where we want to know the likelihood of a sample belonging to each class. In CNNs, the final layer often uses Softmax to return probabilities for classification classes.

4.5 Pooling Layers

After the activation function in the convolutional layer, the output (feature map) is passed through a pooling layer to reduce the data size. Pooling helps increase invariance and stability to small input variations. A CNN typically combines multiple Conv2D and pooling layers. Pooling is applied to each sub-matrix of size $f \times f$ with a stride s.

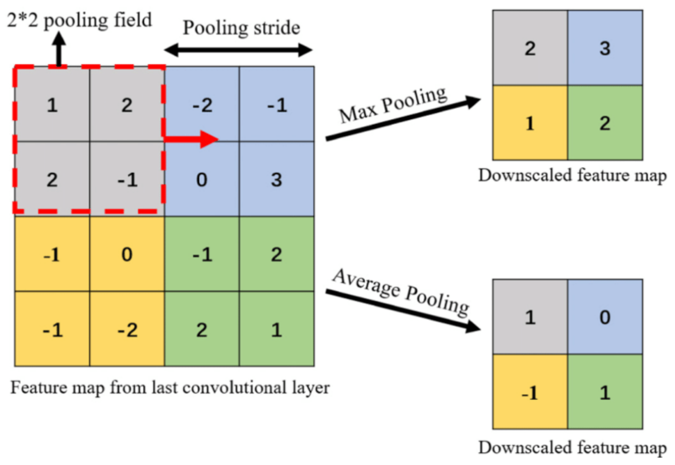


Figure 10: Average and Max Pooling with $f = 2, s = 2$

Common pooling layers include Max Pooling, Average Pooling, and Global Pooling. For example, Figure 10 illustrates Max Pooling and Average Pooling with $f = 2, s = 2$. Global Pooling simply applies pooling to the entire feature map.

4.6 Flatten Layer

The flatten layer converts the three-dimensional layer in the network into a one-dimensional vector.

For example, a tensor of $5 \times 5 \times 2$ will be converted into a vector of length $5 \times 5 \times 2 = 50$.

The preceding convolutional layers of the network have extracted features from the input image, and now it's time to classify these features. We use logistic/softmax functions to determine the confidence of labels, which requires a one-dimensional input. Hence, the flatten layer is necessary.

4.7 Fully Connected Layer

The fully connected layer (also known as the dense layer) is the final layer in a convolutional neural network. This layer is a combination of an affine function and a non-linear function:

- Affine function: $y = Wx + b$
- Non-linear function: Sigmoid, Tanh, ReLU

The fully connected layer takes input from the flatten layer. The data from the flatten layer is first passed to the affine function and then to the non-linear function. The combination of an affine function and a non-linear function is called a fully connected (or dense) layer. The fully connected layer can add multiple hidden layers based on the depth we want to apply for classification. This depends entirely on the training dataset.

The combination of the flatten layer with the fully connected layer and the softmax layer constitutes a deep neural network. If we look at the complete neural network, the initial layers of the convolutional neural network consist of convolutional and pooling layers. The output layer in a convolutional neural network is the flatten layer using the softmax or sigmoid function, calculating the loss with the cross-entropy function. Finally, we obtain the confidence values for each label, with the highest confidence label being the final label for the input image.

4.8 Backpropagation

4.8.1 Backpropagation in Fully Connected Layers

The first step is to find the loss function used to evaluate the output generated by the output layer of the dense layer. The choice of loss function depends on the task the network aims to perform or the nature of the problem, whether it is binary classification, multi-class classification, or regression. To illustrate, let's consider the Binary Cross-Entropy loss function:

$$L(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) \tag{14}$$

- **Derivative with respect to output $\hat{y}$:**

$$\begin{aligned}
\frac{\partial L}{\partial \hat{y}_i} &= \frac{\partial}{\partial \hat{y}_i} \left(-\frac{1}{n} \sum_{i=1}^{n} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) \right) \\
&= -\frac{1}{n} \left(\frac{\partial}{\partial \hat{y}_i} (y_i \log(\hat{y}_i)) + \frac{\partial}{\partial \hat{y}_i} ((1 - y_i) \log(1 - \hat{y}_i)) \right) \\
&= -\frac{1}{n} \left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i} \right)
\end{aligned}$$

- **Derivative with respect to weight w:**

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_{ij}} \tag{15}$$

  Knowing that $\hat{y}_i = x_j \cdot w_{ij} + b_j$, we have:

$$\frac{\partial \hat{y}_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} (x_j \cdot w_{ij} + b_j) = x_j$$

  Thus, $\frac{\partial L}{\partial w_{ij}}$ becomes:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial \hat{y}_i} \cdot x_j = -\frac{1}{n} \left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i} \right) \cdot x_j \tag{16}$$

The new weights are obtained using:

$$w_{ij} = w_{ij} - \eta \frac{\partial L}{\partial w_{ij}} \tag{17}$$

where η is the learning rate.

4.8.2 Backpropagation in Convolutional Layers

Backpropagation in convolutional layers involves calculating the gradient of the loss function with respect to each filter, which can be used to update the filter's weights. Here's a step-by-step outline:

Forward pass:

- Calculate the output of each convolutional layer by convolving the input with the filters and applying the activation function.
- Propagate the output through the network until the final output is obtained.
- Compute the loss using the final output and the ground truth labels.

Backward pass:

- Compute the gradient of the loss function with respect to the output of each layer using the chain rule.
- For each convolutional layer, calculate the gradient of the loss function with respect to the filter weights.
  - **Gradient with respect to filter weights:** Given the loss function L and the filter weights w, the gradient $\frac{\partial L}{\partial w}$ is computed by convolving the input with the gradient of the loss function with respect to the layer's output.
  - **Gradient with respect to input:** The gradient of the loss function with respect to the input of a convolutional layer is computed by convolving the gradient of the loss function with respect to the layer's output with the flipped filter weights.

Weight update:

- Update the filter weights using the computed gradients and the learning rate η:

$$w = w - \eta \frac{\partial L}{\partial w} \tag{18}$$

By repeating this process for each layer in the network, the weights of the convolutional layers are adjusted to minimize the loss function, allowing the network to learn from the training data.

In summary, backpropagation in convolutional layers involves computing the gradients of the loss function with respect to the filter weights and the input and then updating the filter weights using these gradients and the learning rate. This process is repeated

5 Experimental Results

5.1 Potential energy model

To evaluate our neural network (NN) for total energy prediction, we utilized the `2000K_rand.xyz` file to extract a dataset containing 2500 structures. We divided this dataset into training and testing sets, each consisting of 1250 structures. We employed a cutoff radius of 8 Å to determine the chemical environment of each atom.

We designed feature vectors to describe the pairwise interactions between atom i and atom j as follows:

$$\mathbf{b_{ij}} = \left(\mathbf{r_{ij} f_c(r_{ij})}, \frac{\mathbf{1}}{\mathbf{r_{ij}}} \mathbf{f_c(r_{ij})} \right), \tag{19}$$

where r_{ij} is the distance between atoms i and j. To implement this idea, we represented each atom by stacking all pairwise vectors of its interactions with neighboring atoms, sorted in descending order of distance from the center atom. We used padding to ensure all atoms were represented by matrices of the same dimensions. For the silicon system, we used 120×2 matrices for each silicon atom.

We then stacked the representation matrices of all constituent atoms to form descriptors for a structure as a 3D tensor. For the silicon system, a structure is represented as a $64 \times 120 \times 2$ tensor. The first dimension of the tensor is the number of atoms in the system, the second dimension is the maximum number of neighboring atoms in the chemical environment, and the third dimension is the number of dimensions of the pairwise vectors. This representation of the structure is similar to a color digital image but with two channels.

Our proposed model is as follows: the input ($64 \times 120 \times 2$) is processed through three Conv2D layers, each with 128 filters and a kernel size of (1, 1). Each Conv2D layer is followed by a ReLU activation function and L2 regularization. To obtain the feature vectors of the local structures, we summed the third dimension, yielding the output tensor $x_i = \sum_j a_{ij}$. Thus, we obtained a matrix $\mathbf{X}$ with rows representing the feature vectors of the local structures. We then used five Conv1D layers to predict the atomic energy and obtained the total energy by summing the atomic energies.

We trained the model using the Adam optimizer with a learning rate of 0.00005 for 2500 epochs, a batch size of 16, and the Mean Squared Error (MSE) loss function.

Due to limited computing resources, we will further develop and evaluate this section in the future when the necessary resources are available.

5.2 Atomic force

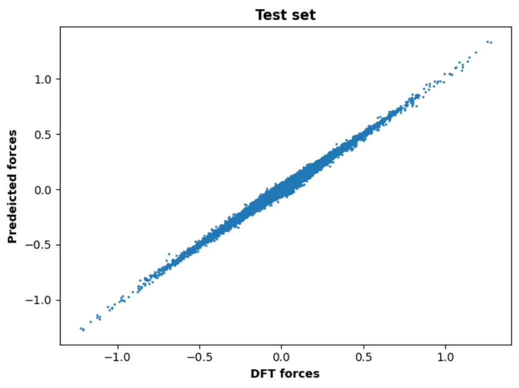
The performance of the atomic force model, calculated by obtaining the gradient of the energy, was relatively poor due to the gradient vanishing problem in the neural network (NN). To improve performance, we developed an NN to represent the force exerted on an atom directly. We used the ‘Si_64.xyz‘ file to extract a dataset containing 3507 structures. The DFT-calculated atomic forces were divided into training and testing sets consisting of 2805 and 702 structures, respectively. In summary, we had 179,520 and 44,928 force vectors associated with the atoms in the training and testing sets.

We used a cutoff radius of 8.0 Å to determine the neighboring atoms of each atom. The force exerted on an atom can be decomposed into the pairwise forces provided by the neighboring atoms. We also adopted the distance and its inverse as the features of the pairwise term as described in Eq. (19). Therefore, we obtained a matrix of $n \times 2$ for representing a silicon atom, where n is the number of neighboring atoms. For this silicon system, we used 120 as the maximum number of neighboring atoms. The unit vectors (in the Cartesian reference of the system) for all pairs were also acquired, resulting in N with dimensions of 120×3.

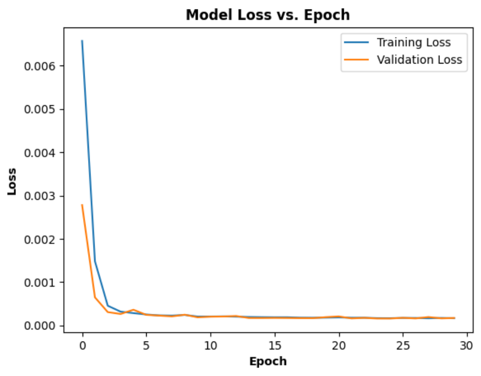
To implement the proposed NN model, we used two inputs: **B** (input_x) and **N** (input_n) with input shapes of 120×2 and 120×3, respectively. Five 1D convolutional neural network (Conv1D) layers with a kernel size of 1 were applied to extract the hidden pairwise force function. A residual connection was added between the first and last layers. A Conv1D layer with one neuron was used to represent the pairwise force function. Then, the unit vector (input_n) was multiplied by the pairwise force function to obtain the pairwise force. We summed the pairwise forces to obtain the force on an atom, as described in Eq. 5.

We trained the model using the Adam optimizer for 100 epochs, with a batch size of 2000, and using the MSE loss function. Additionally, we applied EarlyStopping with a patience of 5 epochs (if the loss did not improve after 5 epochs, the training would stop and restore the best model). We also investigated various activation functions for the feature extractor and network, including the "Rectified Linear Unit" (ReLU), "Tangent Hyperbolic" (Tanh), and "Sigmoid".

Table 1 summarizes the performance of our model using different activation functions (ReLU, Tanh, and Sigmoid) in the Conv1D layer. Among these, ReLU provides the best results, achieving the lowest RMSE (0.0127) and MAE (0.0096) with the fewest training epochs, making it the most suitable activation function for our model. Tanh achieves a low MAE but is less efficient and less accurate overall. Sigmoid performs the worst, with higher errors and longer training times. The experimental results indicate that our model with the ReLU activation function accurately predicts atomic forces using only the primitive information of atom pairs, as shown in Figure 11a. The typical learning curves of our model, demonstrating good learning behavior, are shown in Figure 11b.



(a) Forces predicted by the model and DFT for silicon atoms in the test set



(b) Learning curves (MSE) for the force model

Figure 11: Forces predicted by the machine learning and DFT for silicon atoms in the test set (a), and statistical error metrics (MSE) used to train the force model (learning curves)

Table 1: Model performance for atomic force prediction: RMSE (eV/Å) and MAE (eV/Å) on the test set

Activation	RMSE	MAE	Epoch
ReLU	0.0127	0.0096	25
Tanh	0.0129	0.0098	35
Sigmoid	0.0141	0.0108	80

In addition to these main results, we have included further detailed analyses and visualizations in Appendix A Visualization Analysis to provide a comprehensive understanding of our model's performance with various activation functions. For instance, Figure. 12a, 12b, 12c described a learning curve of model force on the test set. Figure. 13a, 13b, 13c described forces predicted by various models. Notably in Figure. 12a and 13a, we use a customized inverse R^2 loss as shown in Eq. 21 of B and ReLU activation function to train the force model.

6 Conclusion

This study has demonstrated the effectiveness of using deep neural networks to predict atomic forces and potential energies in molecular dynamics simulations. The models developed, particularly those employing the ReLU activation function, have shown high accuracy and efficiency, making them suitable for practical applications in computational material science.

For future work, we aim to finish training and evaluate the energy model from Section 5.1 Potential energy model. This will further enhance the precision and applicability of our approach, contributing to the development of more robust and scalable models for molecular simulations.

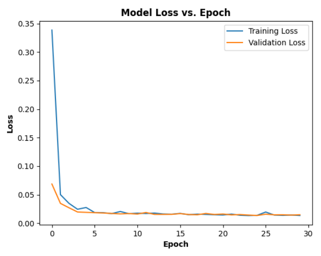
Data Availability

The data for experimental results can be accessed at https://drive.google.com/drive/folders/1Ldw4K0I5icuEqZwGJtRJ6vN_ee4AcQ9b?usp=sharing

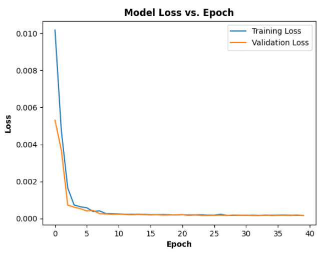
Code Availability

The code is available on GitHub at https://github.com/MHieutr/Pairwise_interactions_in_MD

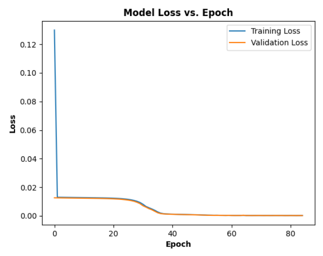
A Visualization Analysis



(a) Learning curve of model force using R^2 loss and ReLU activation function

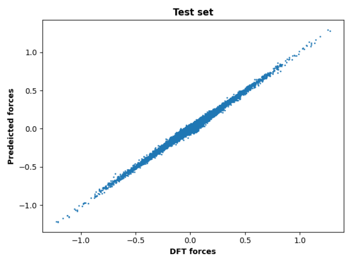


(b) Learning curve of model force using Tanh activation function

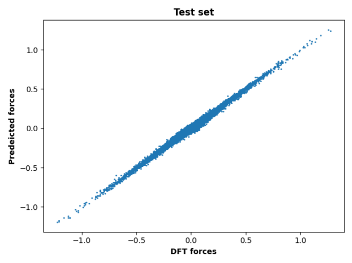


(c) Learning curve of model force using Sigmoid activation function

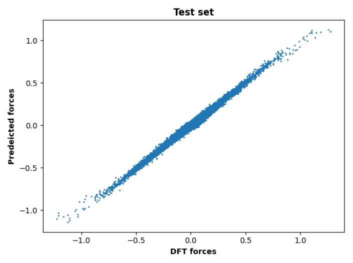
Figure 12: Statistical error metrics (R^2 loss) used to train the force model on the testing set(learning curves) (a), statistical error metrics (MSE) with Tanh and ReLU activation (b) and (c)



(a) Forces predicted by the model with R^2 loss, ReLU activation, and DFT for silicon atoms in the test set



(b) Forces predicted by the model with Tanh activation and DFT for silicon atoms in the test set



(c) Forces predicted by the model with Sigmoid activation and DFT for silicon atoms in the test set

Figure 13: Forces predicted by different models using various configurations of loss and activation functions.

B Loss Function

R^2 score measures the extent to which the predicted values ($\hat{y}$) match the actual values. It indicates the proportion of data variability that the model can explain. Therefore, I think using the inverse R^2 loss might result in better model performance. We obtained 0.0130 RMSE and 0.0097 MAE with 25 epochs. R^2 Score can be calculated:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{20}$$

By minimizing the inverse R^2, we are indirectly maximizing the $\mathrm{R}^{\mathbf{2}}$ value, which means we are working towards explaining as much of the variance in the target variable as possible. So that inverse R^2 loss can be expressed as follows:

$$R^2_{inversed} = 1 - R^2 = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{21}$$

References

[1] M. E. Tuckerman and G. J. Martyna, “Understanding modern molecular dynamics: Techniques and applications,” 2000.

[2] Y. Li, H. Li, F. C. Pickard IV, B. Narayanan, F. G. Sen, M. K. Chan, S. K. Sankaranarayanan, B. R. Brooks, and B. Roux, “Machine learning force field parameters from ab initio data,” *Journal of chemical theory and computation*, vol. 13, no. 9, pp. 4492–4503, 2017.

[3] R. Jinnouchi, F. Karsai, and G. Kresse, “On-the-fly machine learning force field generation: Application to melting points,” *Physical Review B*, vol. 100, no. 1, p. 014105, 2019.

[4] V. Botu, R. Batra, J. Chapman, and R. Ramprasad, “Machine learning force fields: construction, validation, and outlook,” *The Journal of Physical Chemistry C*, vol. 121, no. 1, pp. 511–522, 2017.

[5] I. Kruglov, O. Sergeev, A. Yanilkin, and A. R. Oganov, “Energy-free machine learning force field for aluminum,” *Scientific reports*, vol. 7, no. 1, p. 8512, 2017.

[6] C. Chen, Z. Deng, R. Tran, H. Tang, I.-H. Chu, and S. P. Ong, “Accurate force field for molybdenum by machine learning large materials data,” *Physical Review Materials*, vol. 1, no. 4, p. 043603, 2017.

[7] S. K. Mudedla, A. Braka, and S. Wu, “Quantum-based machine learning and ai models to generate force field parameters for drug-like small molecules,” *Frontiers in Molecular Biosciences*, vol. 9, p. 1002535, 2022.

[8] J.-Z. Xie, X.-Y. Zhou, D. Luan, and H. Jiang, “Machine learning force field aided cluster expansion approach to configurationally disordered materials: critical assessment of training set selection and size convergence,” *Journal of Chemical Theory and Computation*, vol. 18, no. 6, pp. 3795–3804, 2022.

[9] V.-Q. Nguyen, V.-C. Nguyen, T.-C. Nguyen, T.-L. Pham, *et al.*, “Pairwise interactions for potential energy surfaces and atomic forces using deep neural networks,” *Computational Materials Science*, vol. 209, p. 111379, 2022.

Task Assignment

The group consists of two members: Trần Minh Hiếu, Từ Lê Tú Uyên. Below is the detailed task assignment table for this project.

Member	Task	Contribution
Trần Minh Hiếu	Implement deep learning models, Visualization, Writing report	50 %
Từ Lê Tú Uyên	Conceptualization, Methodology, Draft revision, Supervision, Writing report	50 %

Hanoi, June 19, 2024

Group Members