

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INGEGNERIA DELL'ENERGIA ELETTRICA E
DELL'INFORMAZIONE
"Guglielmo Marconi"
DEI

CORSO DI LAUREA MAGISTRALE IN AUTOMATION
ENGINEERING

Tesi Di Laurea
in
Modeling and Simulation of Mechatronic Systems

Prediction of H_2 Adsorption Energy on Graphene Oxide Flakes Using Neural Networks

Relatore:
Chiar.mo Prof.
Alessandro Macchelli

Presentata da:
Giuseppe Bulgaro

Correlatori:
Dott. Francesco Mercuri
Dott. Rocco Libero Giossi

Ringraziamenti

La Regola delle tre L è un trattato che vede la sua origine in un paesino sperduto dell'Emilia Romagna, dov'è stata pensata e aggiornata fino ad arrivare alla trattazione completa dei giorni nostri. È dall'applicazione della stessa, insieme al grande aiuto di tutte le persone citate di seguito, che questo grande traguardo è stato raggiunto. A livello accademico vorrei cominciare ringraziando il Professore Alessandro Macchelli il quale, sebbene l'ambito non fosse propriamente il suo, ha comunque accettato la sfida in modo puntuale e professionale. Un altro grande Grazie va a Francesco Mercuri e Rocco Libero Giossi che con grande pazienza mi hanno seguito e guidato nei due tirocini svolti al Consiglio Nazionale delle Ricerche di Bologna.

Ed ora i ringraziamenti si spostano verso coloro che ci sono stati in questi tre anni, sia tra gli alti che tra i bassi. Alla mia fantastica famiglia, che mi ha supportato, spronato e guidato con i giusti valori. A Serena, per aver visto da vicino i famosi alti e bassi ed essermi sempre e comunque stata accanto, con le giuste parole e gesti, rendendo il viaggio meno difficile. Per ultimi ma non per importanza, voglio ringraziare tutti i miei amici, dai colleghi del GC a coloro che vanno veramente forti, passando dai compagni di corso, arrivando ai colleghi del leggendario Bar Bagno. Avete reso questi anni quelli che da vecchio dirò essere stati "i più belli della mia vita"

List of Figures

1.1	Graphene oxide flake. Carbon atoms are shown in green, oxygen in red, and hydrogen in white.	10
1.2	Rectified Linear Unit (ReLU) activation function.	12
1.3	Multi-Layer Perceptron (MLP) architecture. Adapted from [1].	12
1.4	Convolutional neural network (CNN) architecture. Adapted from [2]. . . .	13
1.5	Successive convolutional layers learn increasingly complex features, from edges to atomic motifs. Adapted from [3].	14
1.6	Illustration of padding, stride, and depth in a convolutional layer. Padding adds zero borders around the input, stride controls the step size of the kernel movement, and the depth corresponds to the number of filters producing multiple feature maps.	15
1.7	Overview of the ResNet18 architecture with residual connections. Adapted from [4].	17
1.8	Overview of the Graphenet framework [5]: (a) workflow schematic (Draw.io); (b) image representation of a nanographene sample (rendered with VMD and plotted with OpenCV/Pillow); (c) CNN architecture built upon ResNet18 (Draw.io). Figure adapted from [5].	18
3.1	Example of discarded configurations due to anomalous total energies. . . .	22
3.2	Bond-length constraint used to detect dissociated H_2 molecules.	23
3.3	Addition of four SK placeholder atoms at the corners of the bounding square to enforce uniform image dimensions.	24
3.4	Example of an image representation generated using one hydrogen atom as reference. The color channels (R, G, and B) correspond to different atomic species, as defined earlier in the methodology.	25

3.5	Example of an image representation generated using both hydrogen atoms as reference in a dual-frame superposition. The R, G, and B channels correspond to different atomic species.	26
3.6	Example of an image representation after applying Gaussian interpolation. The Gaussian spread reduces image sparsity and emphasizes local spatial structure. Colors correspond to atomic species as described earlier.	27
3.7	Rotation-based data-augmentation function from <code>lib_dataset_generation.py</code> , generating three augmented samples for each input image.	28
3.8	Augmented versions of the original configuration shown in Figure 3.4.	28
3.9	Distribution of raw adsorption energy differences extracted from the <code>flakes.json</code> file.	30
3.10	Rescaled target distribution after min-max normalization within the $[0, 1]$ range.	31
3.11	Distribution of target values after statistical standardization.	31
4.1	Header of the <code>matrix_discretization_fcn()</code> function.	33
4.2	Algorithmic flow of the <code>matrix_discretization_fcn()</code> function used to create the discretized channel matrices.	34
4.3	Header of the <code>save_input_sequence()</code> function and definition of the six tensor channels corresponding to different atomic species around the two hydrogen atoms.	35
4.4	Creation of the <code>per_hyd_pos_tensor</code> input tensor by combining the discretized matrices for each atomic species through calls to <code>matrix_discretization_fcn()</code>	35
4.5	Creation of the target tensor within the <code>process_flake()</code> function. Each adsorption configuration contributes a scalar energy value (<code>flake_energy_diff</code>), collected into the tensor <code>per_flake_energy_tensor</code>	36
4.6	Assembly of the complete dataset. The loop iterates over all flakes, generating and collecting the corresponding input and energy tensors, which are concatenated into the final tensors <code>complete_input_tensor</code> and <code>complete_energy_tensor</code>	37
4.7	Terminal output showing the shapes of the assembled input and target tensors, confirming the correct concatenation of all flake data.	38
4.8	Initial CNN architecture used for adsorption energy regression. The input tensor of shape $(B, 6, H, W)$ is processed through four convolutional blocks, followed by spatial pooling and a fully connected regression layer producing the final energy prediction.	40

4.9	Input tensor extended to 10 channels by including angular information. . .	43
4.10	Example of a raw input channel (Carbon), showing strong sparsity.	43
4.11	Same channel after Gaussian interpolation (kernel $\sigma = 2$), improving spatial continuity.	44
5.1	Accuracy plot for the single-hydrogen reference frame (no data augmentation).	45
5.2	Accuracy plot for the single-hydrogen reference frame with data augmentation.	46
5.3	Accuracy plot for the dual-frame representation (no data augmentation). .	46
5.4	Accuracy plot for the dual-frame representation with data augmentation. .	47
5.5	Accuracy plot for the Gaussian interpolation representation (no data augmentation).	47
5.6	Accuracy plot for the Gaussian interpolation representation with data augmentation.	48
5.7	Comparison of training behavior for the two preprocessing strategies. . . .	49
5.8	Regression results for the 10-channel model.	49
5.9	Regression results for the 6-channel model.	50
5.10	Regression performance after Gaussian interpolation.	50
5.11	Regression performance for the best CNN configuration.	51

Contents

Introduction	7
1 Theoretical background	9
1.1 Chemical and Nanoscale Structure of Materials	9
1.2 Graphene and Its Derivatives	10
1.3 Density Functional Tight Binding (DFTB) Simulations	11
1.4 Neural Networks for Modeling and Prediction	11
1.4.1 Convolutional Neural Networks (CNNs)	12
1.4.2 Graphenet (ResNet18) Architecture	17
2 Methodology	19
2.1 Overview	19
2.2 Rationale	20
3 Graphenet	21
3.1 Input Dataset Generation	21
3.1.1 Preprocessing Script: <code>sorted_c_then_o_h_xyz</code>	22
3.1.2 Coordinate Frame Variants	25
3.1.3 Data Augmentation.	27
3.2 Target Analysis	28
3.2.1 Preprocessing Script: <code>create_the_csv_file</code>	29
3.2.2 Processing of Target Values	29
4 Custom Convolutional neural network	32
4.1 Introduction	32

4.2	Dataset Generation	33
4.2.1	Input Tensor Generation	34
4.2.2	Target Tensor Generator	36
4.2.3	Complete Tensor Assembly	37
4.3	Convolutional Neural Network	38
4.3.1	CNN Training Pipeline	38
4.3.2	Neural Network Architecture	40
4.4	Model Optimization and Experimental Trials	42
4.4.1	Normalization vs. Standardization	42
4.4.2	Increasing the Number of Input Channels	42
4.4.3	Gaussian Interpolation of the Input Channels	43
4.4.4	Hyperparameter Space Exploration	44
5	Results	45
5.1	Graphenet	45
5.1.1	Single-Hydrogen Reference Frame	45
5.1.2	Dual-Frame Representation	46
5.1.3	Gaussian Interpolation	46
5.2	Custom CNN	48
5.2.1	Effect of Preprocessing: Normalization vs. Standardization	48
5.2.2	Effect of Increasing the Number of Input Channels	48
5.2.3	Effect of Gaussian Interpolation	49
5.2.4	Final Model Performance	50
5.2.5	Summary of Optimal Hyperparameters	51
6	Conclusions	52

Introduction

The transition to sustainable and low-emission energy systems is one of the main scientific and technological challenges of our time. Among the many options being studied, hydrogen is one of the most promising energy carriers. It is abundant, has a high energy density, and can produce clean energy when used in fuel cells. However, the large-scale use of hydrogen still faces several problems, especially in its safe and efficient storage and transport. Finding materials that can store and release hydrogen easily and safely is essential for the development of hydrogen-based technologies.

In this field, graphene oxide (GO) and similar nanostructured carbon materials have received great attention. Their chemical and structural properties can be tuned by introducing oxygen groups, which create active sites for hydrogen adsorption. Understanding how hydrogen molecules interact with graphene oxide surfaces at the atomic level is crucial for designing better materials for hydrogen storage.

To study these interactions, computational modeling is a powerful tool. Methods such as density functional theory (DFT), or more precisely Density Functional Tight Binding (DFTB), are simulations that allow scientists to study adsorption mechanisms and binding energies in great detail. However, these techniques are very demanding in terms of time and computational resources. This makes it difficult to apply them to large or complex systems, which limits their practical use.

In recent years, machine learning (ML) and artificial intelligence (AI) have provided new opportunities to overcome these limitations. By learning from data obtained through simulations or experiments, neural network models can predict physical and chemical properties with high accuracy and much lower computational cost. These models can act as fast predictive tools, allowing to explore many possible material configurations that would be too expensive to simulate with traditional methods.

This thesis is part of this growing research area. The main goal is to develop and test neural network models to predict hydrogen adsorption on nanostructured graphene oxide nanostructures. The models aim to achieve accuracy comparable to atomistic simulations, while being much faster and easier to scale. In this way, the trained network can estimate different graphene oxide properties directly, eliminating the need for further

computationally expensive simulations.

More broadly, this work contributes to the data-driven design of materials for sustainable energy applications. By combining detailed computational data with the predictive power of machine learning, it supports the development of efficient and reliable models for hydrogen storage. The approach presented here could also be applied to other materials, helping to speed up the discovery of advanced solutions for clean energy technologies.

1 Theoretical background

In this chapter, the main theoretical concepts behind this work are presented. It starts with a short overview of chemical bonding and the structure of materials at the nanoscale. Then, it explains the main ideas of Density Functional Tight Binding (DFTB) simulations and describes the properties of graphene and its derivatives. Finally, it introduces neural networks as computational tools for modeling and prediction, with a particular focus on convolutional neural networks (CNNs) and the Graphenet (ResNet18) architecture used in this study.

1.1 Chemical and Nanoscale Structure of Materials

All materials are made of atoms connected through different types of chemical bonds: these bonds determine how atoms interact, how strong the material is, and how it behaves under different conditions. The three main types of chemical bonds are ionic, covalent, and metallic ([6]). In covalent bonds, atoms share electrons, creating strong and stable structures typical of materials such as diamond or graphene.

When we move from the atomic to the nanoscale, materials start to show new and interesting properties ([7]). The nanoscale refers to dimensions between 1 and 100 nanometers, where the behavior of matter is influenced by both quantum effects and a high surface-to-volume ratio. At this scale, even small changes in the arrangement of atoms can strongly affect properties such as electrical conductivity, chemical reactivity, and mechanical strength.

Because of these unique characteristics, nanostructured materials are widely studied for applications in energy storage, catalysis, and electronic devices ([8]). In this context, graphene and its derivatives have become some of the most important materials in nanotechnology and materials science.

1.2 Graphene and Its Derivatives

Graphene is a two-dimensional material composed of a single layer of carbon atoms arranged in a hexagonal lattice ([9]). It is known for its extraordinary mechanical strength, high electrical conductivity, and large surface area. These properties make it an excellent candidate for many applications, including sensors, batteries, and hydrogen storage ([10]).

Graphene oxide (GO) is a derivative of graphene that contains oxygen functional groups, such as hydroxyl, epoxy, and carboxyl, attached to its surface ([11]). These groups disrupt the perfect carbon lattice of graphene, but they also introduce active sites that can interact with other molecules, including hydrogen. By modifying the type and number of these oxygen groups, it is possible to tune the chemical reactivity and adsorption properties of the material ([12]). An example of Graphene oxide nanostructure used in the thesis is the following Figure 1.1

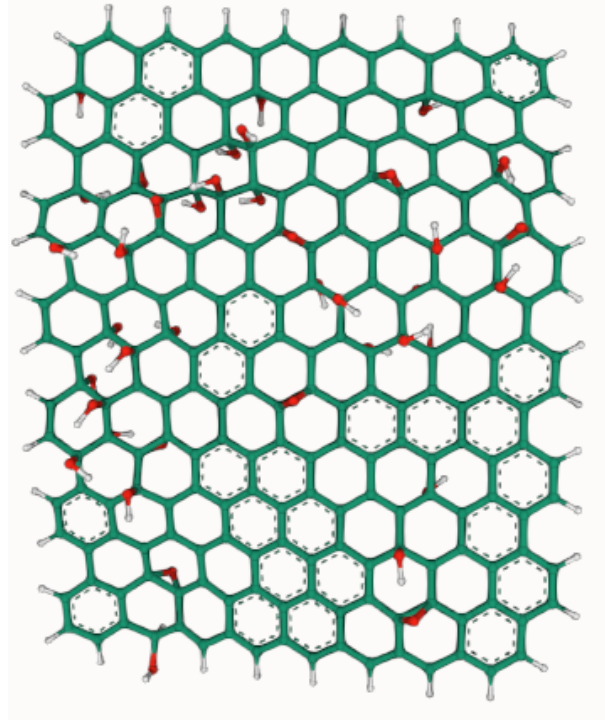


Figure 1.1: Graphene oxide flake. Carbon atoms are shown in green, oxygen in red, and hydrogen in white.

The interaction between hydrogen and GO depends on several factors, such as the distribution of oxygen groups, the surface curvature, and the local electronic environment ([2]). Understanding these factors requires atomistic modeling and simulation, which can provide detailed insight into the adsorption mechanisms and energy landscape of hydrogen storage in GO-based materials.

1.3 Density Functional Tight Binding (DFTB) Simulations

Understanding materials at the atomic level often requires quantum mechanical simulations, which describe how electrons and atoms interact. One of the most widely used methods is Density Functional Theory (DFT), which calculates the electronic structure of materials based on the electron density ([13]). However, DFT calculations are often very time-consuming and computationally expensive, especially for large or complex systems.

To reduce this cost, a simplified approach called Density Functional Tight Binding (DFTB) was developed ([14]). DFTB is an approximate version of DFT that maintains the essential physics while simplifying the mathematical formulation to make calculations faster.

In DFTB, the total energy of a system is calculated using precomputed parameters obtained from DFT reference calculations. This means that instead of solving complex equations for every new configuration, DFTB can use existing data to estimate electronic interactions ([1]). As a result, DFTB allows the study of larger systems and longer simulation times while maintaining good accuracy for many materials.

This makes DFTB an ideal method for studying hydrogen adsorption on graphene oxide (GO) substructures, as it provides a balance between computational efficiency and physical realism ([4]).

1.4 Neural Networks for Modeling and Prediction

Artificial neural networks (NNs) are computational models inspired by the structure and functioning of the human brain [1]. They consist of many interconnected units, called *neurons*, that process and transmit information through layers. Each neuron performs a simple mathematical operation, but together they can represent complex, nonlinear relationships between input and output data.

A typical neuron receives an input vector \mathbf{x} , multiplies it by a set of weights \mathbf{w} , adds a bias term b , and applies a nonlinear activation function f , resulting in an output:

$$y = f(\mathbf{w} \cdot \mathbf{x} + b)$$

Common activation functions include the sigmoid, hyperbolic tangent, and rectified linear unit (ReLU), which introduce nonlinearity and allow the network to learn complex mappings. Among these, the ReLU function is one of the most widely used due to its

simplicity and efficiency in mitigating the vanishing gradient problem [15].

As shown in Figure 1.2, the ReLU function, defined as $f(x) = \max(0, x)$, outputs zero for negative input values and a linear response for positive inputs, allowing networks to converge faster during training while maintaining sparse activations.

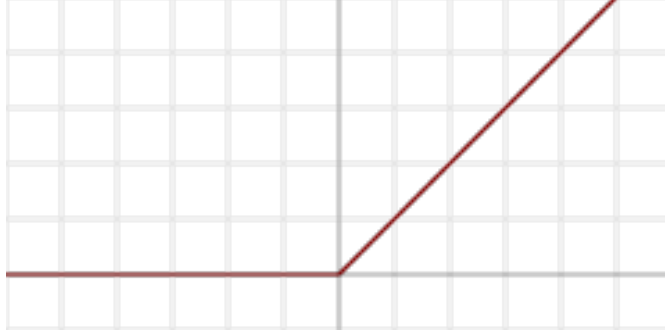


Figure 1.2: Rectified Linear Unit (ReLU) activation function.

Figure 1.3 illustrates a standard feedforward neural network, or Multi-Layer Perceptron (MLP). Information flows from the input layer through one or more hidden layers to the output layer. During training, the network adjusts its weights \mathbf{w} and biases b to minimize a loss function, typically using the *backpropagation* algorithm and stochastic gradient descent [1].

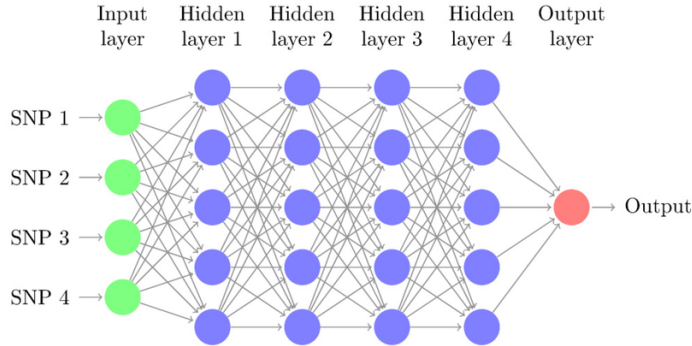


Figure 1.3: Multi-Layer Perceptron (MLP) architecture. Adapted from [1].

Neural networks are widely used in image recognition, natural language processing, and materials science [8]. In materials modeling, NNs can learn the relationship between atomic configurations and physical properties such as total energy, charge distribution, or adsorption capacity [16]. Once trained, these models can make fast and accurate predictions for new systems, eliminating the need for time-consuming quantum or atomistic simulations, making NNs powerful tools for accelerating materials discovery.

1.4.1 Convolutional Neural Networks (CNNs)

A specific class of neural networks used in this work is the *convolutional neural network* (CNN), designed to recognize spatial and structural patterns within data, which

makes them particularly suitable for analyzing images, grids, or maps with local correlations [2].

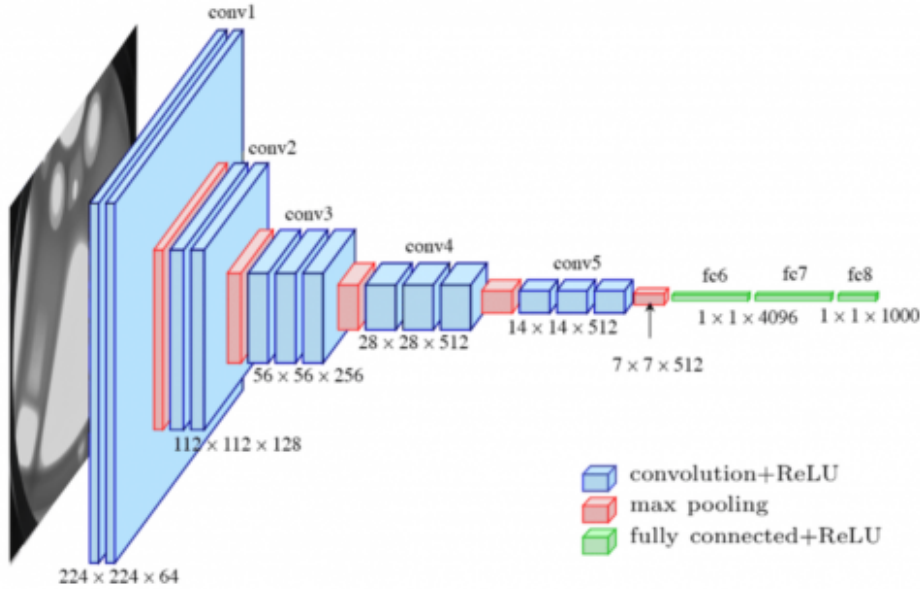


Figure 1.4: Convolutional neural network (CNN) architecture. Adapted from [2].

CNNs are typically composed of multiple layers (Figure 1.4):

- **Convolutional layers** extract local features.
- **Pooling layers** (e.g., max pooling) reduce spatial dimensions and computational cost.
- **Fully connected layers** combine high-level features for final predictions.

Figure 1.5 illustrates how successive convolutional layers learn increasingly complex features: initial layers detect edges, while deeper layers capture structural motifs or repeating atomic patterns [3].

In materials science, CNNs can process structured data such as atomic density maps or simulated energy surfaces. By identifying patterns in these representations, CNNs can predict material properties like adsorption energy or stability directly from structural data [17]. This approach enables efficient analysis of large datasets generated from simulations such as Density Functional Tight Binding (DFTB).

1.4.1.1 Convolutional Layers

Instead of connecting every neuron to all others (as in MLPs), CNNs use a mathematical operation called *convolution*, where small filters (or kernels) slide across the input data to detect features such as edges, textures, or repeating patterns. For an input image

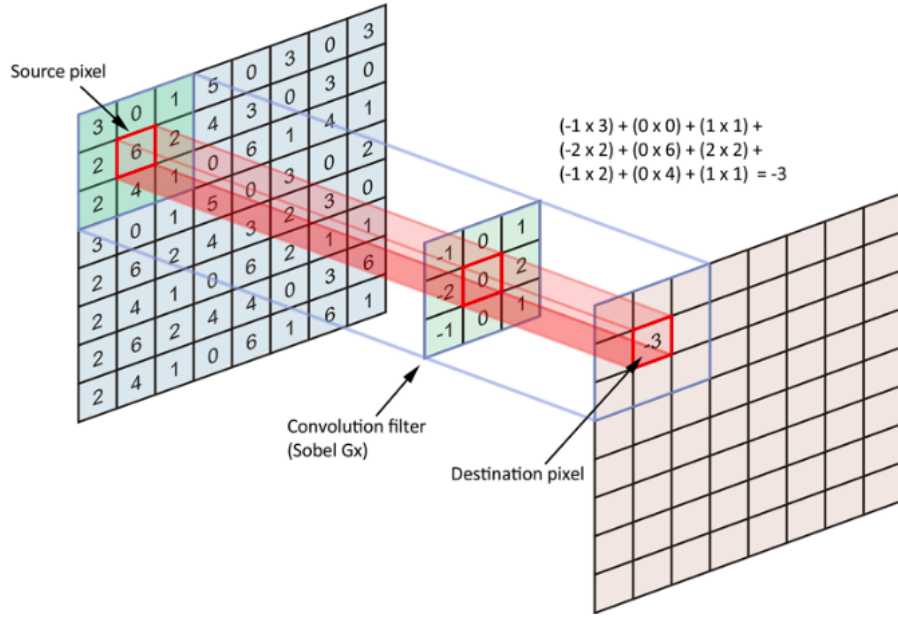


Figure 1.5: Successive convolutional layers learn increasingly complex features, from edges to atomic motifs. Adapted from [3].

I and a kernel K , the convolution operation is defined as:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n)$$

The resulting feature map $S(i, j)$ highlights local structures that the kernel detects.

In this equation:

- I is the **input image** or feature map (a 2D array of pixel values or activations),
- K is the **kernel** (or filter), typically a small matrix (e.g., 3×3 or 5×5) that scans across the input,
- i, j denote the spatial coordinates of the output feature map,
- m, n are indices that iterate over the kernel dimensions,
- $S(i, j)$ is the resulting **feature map** value at position (i, j) , representing how strongly the local region around that position matches the kernel pattern.

The **size of the output** feature map depends on the input dimensions, the kernel size, the amount of zero-padding, the stride, and the number of filters (depth). Let:

- H_{in}, W_{in} : input height and width,
- k_h, k_w : kernel (filter) height and width,

- p_h, p_w : padding applied to height and width,
- s_h, s_w : stride along height and width,
- D_{out} : number of filters applied in the convolution (output depth).

Then, the spatial output dimensions are computed as:

$$H_{out} = \left\lfloor \frac{H_{in} - k_h + 2p_h}{s_h} \right\rfloor + 1 \quad (1.1)$$

$$W_{out} = \left\lfloor \frac{W_{in} - k_w + 2p_w}{s_w} \right\rfloor + 1 \quad (1.2)$$

and the full output volume has size $H_{out} \times W_{out} \times D_{out}$.

Padding (p) adds a border of zeros around the input, allowing the convolution to be applied near the edges without shrinking the spatial dimensions. **Stride** (s) controls how far the kernel moves across the input at each step: a stride of 1 scans every pixel, while larger strides result in a lower-resolution output by skipping intermediate positions. Finally, the **depth** (D_{out}) corresponds to the number of different kernels applied in the layer, with each kernel producing its own feature map; these feature maps are stacked to form the output volume.

These concepts are illustrated in Figure 1.6, which shows how padding preserves the spatial size of the input, how stride affects the sampling of the kernel across the grid, and how using multiple kernels increases the depth of the resulting feature representation.

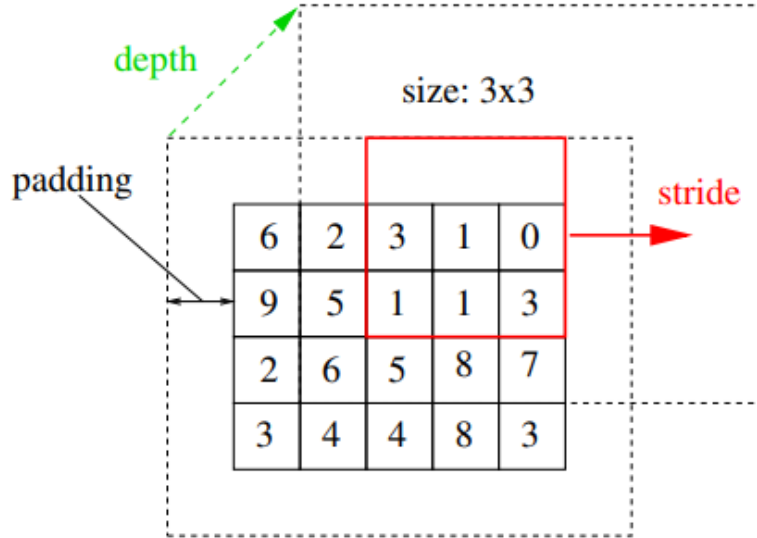


Figure 1.6: Illustration of padding, stride, and depth in a convolutional layer. Padding adds zero borders around the input, stride controls the step size of the kernel movement, and the depth corresponds to the number of filters producing multiple feature maps.

When padding and stride are chosen so that the output has the same spatial dimensions as the input (e.g., $p = \frac{k-1}{2}, s = 1$), the operation is referred to as a *same convolution*.

1.4.1.2 Feature Representation

Each kernel in a convolutional layer is trained to detect a specific **feature** of the input:

- In the first layers, kernels typically learn to detect simple features such as *edges*, *corners*, or *textures*.
- In deeper layers, kernels combine these lower-level patterns to form more complex and abstract features, such as *object parts* or *structural motifs*.

These hierarchical feature maps form a multi-level representation of the data, allowing CNNs to progressively extract meaningful information from raw inputs.

1.4.1.3 Pooling Layers

Pooling layers are used to reduce the spatial dimensions (H, W) of the feature maps while retaining the most relevant information. This operation decreases computational cost, controls overfitting, and provides translation invariance.

The most common pooling operation is **max pooling**, defined as:

$$S_{pool}(i, j) = \max_{(m, n) \in \text{window}} S(i + m, j + n)$$

where the pooling window (e.g., 2×2) slides across the feature map with a defined stride. Other variants include *average pooling*, which takes the mean value instead of the maximum.

1.4.1.4 Fully Connected Layers

After several convolutional and pooling layers, the extracted features are flattened and passed to one or more **fully connected (dense) layers**. These layers perform high-level reasoning by combining the learned features to make final predictions. In this stage, each neuron is connected to all activations from the previous layer, similar to traditional multilayer perceptrons (MLPs). The final output layer often uses activation functions such as *softmax* or *sigmoid*, depending on whether the task is classification or regression.

1.4.1.5 Application in Materials Science

In materials science, CNNs can process structured data such as atomic density maps or simulated energy surfaces. By identifying patterns in these representations, the CNNs

enable efficient analysis of large datasets generated from simulations such as Density Functional Tight Binding (DFTB).

1.4.2 Graphenet (ResNet18) Architecture

In this work, the chosen neural network model is *Graphenet*, based on the well-known ResNet18 architecture [4]. ResNet18 is a deep CNN composed of multiple layers connected through *residual blocks*, which introduce shortcut connections that allow information to bypass one or more layers during training, facilitating deeper network learning.

A residual block can be expressed mathematically as:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x}$$

where \mathcal{F} represents the transformations (convolution, batch normalization, and ReLU activation) applied to input \mathbf{x} . The addition of \mathbf{x} creates an identity shortcut, stabilizing training and allowing the network to learn residual mappings rather than direct transformations, mitigating vanishing gradients.

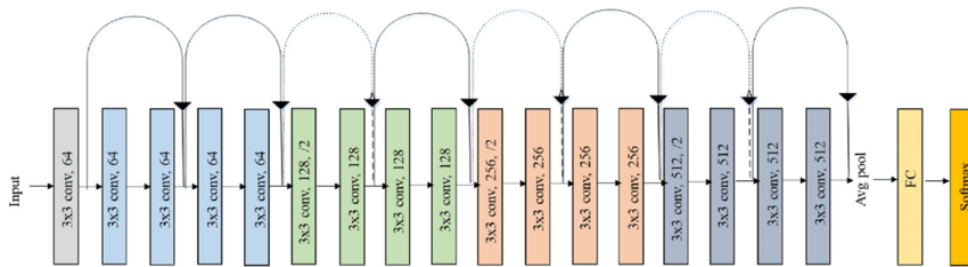


Figure 1.7: Overview of the ResNet18 architecture with residual connections. Adapted from [4].

The ResNet18 architecture consists of a series of convolutional and pooling layers organized into residual blocks, followed by fully connected layers. Skip connections preserve important features and improve gradient propagation during backpropagation.

Building upon ResNet18, the *Graphenet* model incorporates domain-specific adaptations for predicting nanographene properties. Figure 1.8 illustrates the Graphenet framework [5], divided into three parts: (a) schematic overview of the Graphenet workflow, showing data flow from nanographene images to predicted physical and electronic properties; (b) example of encoding a nanographene sample into an image representation using the proposed approach; (c) specific CNN architecture employed in Graphenet, based on ResNet18.

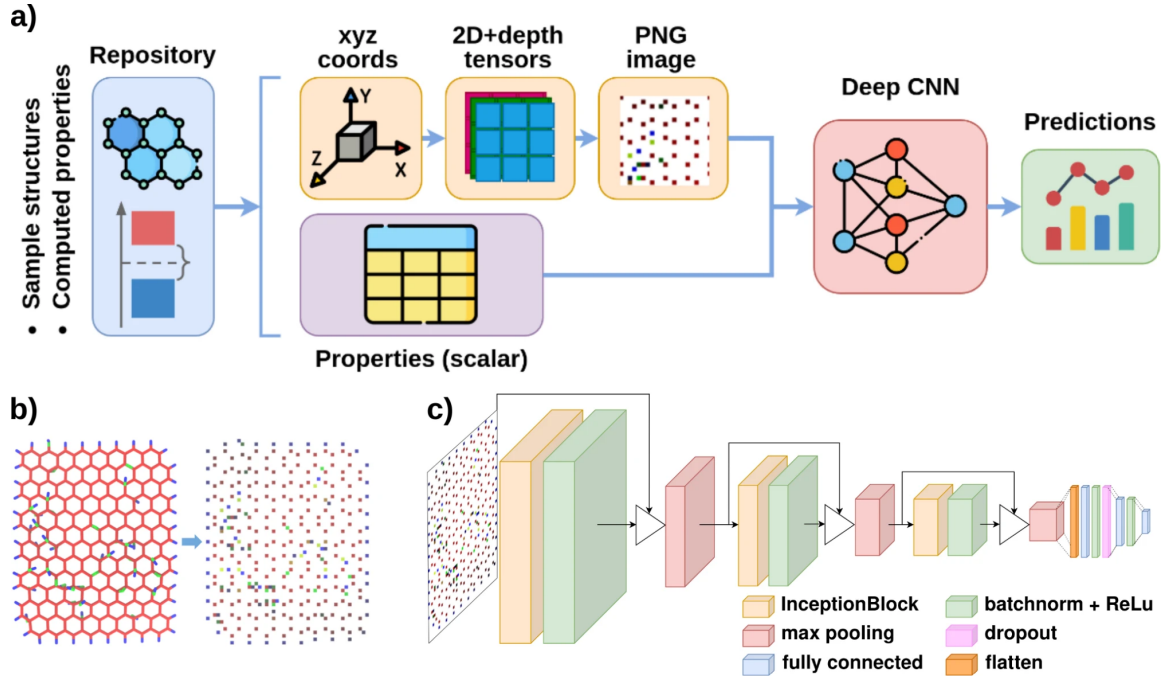


Figure 1.8: Overview of the Graphenet framework [5]: (a) workflow schematic (Draw.io); (b) image representation of a nanographene sample (rendered with VMD and plotted with OpenCV/Pillow); (c) CNN architecture built upon ResNet18 (Draw.io). Figure adapted from [5].

Graphenet leverages ResNet18 to efficiently process image-encoded nanographene structures. Residual connections enable stable gradient flow, while convolutional layers capture both local (atomic-scale) and global (structural and electronic) features. By combining DFTB-derived data with this CNN-based model, Graphenet accurately predicts key physical and electronic properties, providing a computationally efficient framework for accelerating the design and optimization of nanographene-based materials and devices [18].

2 Methodology

2.1 Overview

The present work builds upon the Graphenet framework, introduced in [5], which encodes nanographene structures as three-channel images and processes them through a CNN architecture derived from ResNet18. While Graphenet was originally developed to predict physical and electronic properties of graphene-oxide nanostructures (e.g., formation energy, ionization potential, electron affinity), the objective of this thesis is to adapt and extend this approach to a different prediction task and to a second, more flexible neural-network model.

The first part of the methodology focuses on using Graphenet to predict the **Hydrogen Adsorption Energy**, referred in the formula as HAE, defined as:

$$\text{HAE} = E_{\text{flake}+\text{wH}_2} - E_{\text{flake}} - E_{\text{H}_2}, \quad (2.1)$$

where $E_{\text{flake}+\text{wH}_2}$ is the total energy of the flake with an adsorbed hydrogen molecule, E_{flake} the energy of the flake, and E_{H_2} the reference energy of an isolated H_2 molecule. This quantity measures the adsorption energy associated with hydrogen attachment on the nanographene surface.

In the second part of the study, while the target quantity to be predicted remains exactly the same Hydrogen Adsorption Energy defined in Equation (2.1), the dataset is reprocessed into higher-dimensional, multi-channel tensors that encode a richer set of structural descriptors. These tensors serve as input to a custom convolutional neural network developed during the internship. Unlike Graphenet, which is constrained to RGB-like 3-channel images due to its ResNet-based architecture, the custom CNN accepts an arbitrary number of channels and is specifically designed to learn directly from this extended feature representation. This allows a complementary analysis between a transfer-learning model (Graphenet) and a purpose-built CNN tailored to the encoded local environment.

Thus, this chapter presents the two methodological pipelines used in the thesis:

- preprocessing and encoding nanographene–hydrogen configurations into image tensors or multi-channel arrays,
- adapting and retraining the Graphenet architecture to predict adsorption energies,
- constructing and training a custom multi-channel CNN from scratch on the same dataset.

2.2 Rationale

Graphenet represents graphene-based structures as 2D pixel grids in which atoms are mapped to three separate image channels corresponding to different chemical species. This representation, originally used for predicting formation and electronic energies [5], is maintained here to train Graphenet on adsorption-related energies. Since the dataset and target quantity differ from the original implementation, additional preprocessing is required.

At the same time, the 3-channel RGB constraint imposed by the ResNet18 backbone does not allow the direct incorporation of additional atomic descriptors beyond spatial location and species. For this reason, a second methodological path is introduced. The same raw structural data are transformed into multi-channel tensors encoding a richer set of geometric quantities, and a custom CNN is developed to process these higher-dimensional arrays. Designed and trained from scratch, this architecture provides full control over the convolutional depth, channel dimensionality, and regularization components, enabling a more flexible representation of the local adsorption environment.

By developing both a ResNet-based approach and a custom multi-channel CNN trained on aligned datasets, the methodology allows a controlled architectural comparison while isolating the influence of input representation and model complexity. The following sections detail the preprocessing, encoding, and training procedures for both neural-network frameworks.

3 Graphenet

3.1 Input Dataset Generation

The Graphenet framework relies on the function `generate_png`, which converts atomic structures provided in `.xyz` format into multi-channel images used as input to the neural network. Therefore, the first step in preparing the dataset is the generation of the `.xyz` files corresponding to each nanographene configuration, both with and without adsorbed hydrogen.

All structural and energetic information is initially stored in the `flakes.json` file. This dataset contains, for each nanographene flake, the Cartesian coordinates of all atoms, the total energy and, for each possible adsorption site, the geometry and total energy of the corresponding hydrogenated structure. A simplified representation of the data format is:

```
flake_n:
  XYZ:                # flake without H2
    atoms: [...]
    x: [...]
    y: [...]
    z: [...]
  flake_energy_wo_H2: float
  hydrogens:
    position_id:
      XYZ:            # flake + H2 adsorbed
        atoms: [...]
        x: [...]
        y: [...]
        z: [...]
      flake_energy_w_H2: float
      flake_energy_diff: float
```

Two constraints must be satisfied to produce valid `.xyz` input files for Graphenet:

- the list of atoms must be sorted by element type, with carbon atoms first and heteroatoms (oxygen, hydrogen) afterwards;
- only atoms located within a specified cutoff distance from the adsorption site must be included, ensuring that the encoded structure reflects the local adsorption environment.

To accomplish this, a dedicated preprocessing pipeline was developed, described in the following subsection. This pipeline converts the raw data stored in `flakes.json` into properly ordered and spatially filtered `.xyz` structures ready for the image encoding stage.

3.1.1 Preprocessing Script: `sorted_c_then_o_h_xyz`

The preprocessing script `sorted_c_then_o_h_xyz` generates the structural input data used to train Graphenet. Although it begins with standard steps such as file loading and basic physical filtering, a key aspect of this work is the construction of the `.xyz` files themselves, which required several methodological iterations during the project.

1. **Input Validation and Physical Filtering.** Each adsorption configuration from `flakes.json` is screened to exclude unphysical or numerically unreliable cases. The following criteria correspond to the checks implemented in the preprocessing script:

- *Unstable total energies.* Configurations with anomalously high values of $E_{\text{flake}+\text{H}_2}$ (above a chosen numerical threshold) are discarded, as they typically arise from unconverged or distorted geometries, as shown in the first *IF Condition* of Fig. 3.1.

```

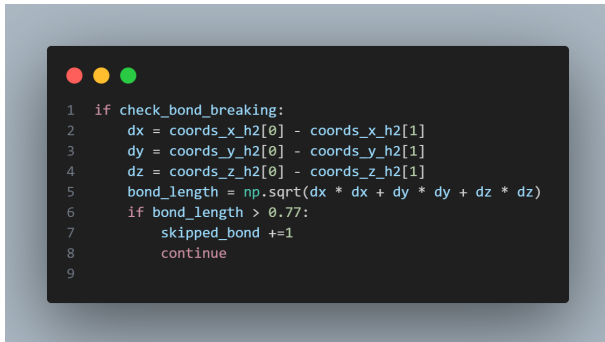
1  if value["flake_energy_w_H2"] > -15160:
2      skipped_energy +=1
3      continue
4  if energy_h2 >= 0 or energy_h2 < -0.05:
5      skipped_energy +=1
6      continue

```

Figure 3.1: Example of discarded configurations due to anomalous total energies.

- *Adsorption-energy range.* The adsorption energy difference E_{diff} is required to fall within the adsorption energy window. Configurations with $E_{\text{diff}} \geq 0$ (non-binding) or $E_{\text{diff}} < -0.05$ eV (too strongly bound) are excluded, as shown in the second *IF condition* of Fig. 3.1.

- *Preservation of the H_2 bond.* The $H-H$ distance must remain below 0.77 \AA to ensure that the molecule remains intact. As reported in Fig. 3.2, larger distances indicating dissociation are removed from the dataset.



```

1  if check_bond_breaking:
2      dx = coords_x_h2[0] - coords_x_h2[1]
3      dy = coords_y_h2[0] - coords_y_h2[1]
4      dz = coords_z_h2[0] - coords_z_h2[1]
5      bond_length = np.sqrt(dx * dx + dy * dy + dz * dz)
6      if bond_length > 0.77:
7          skipped_bond += 1
8          continue
9

```

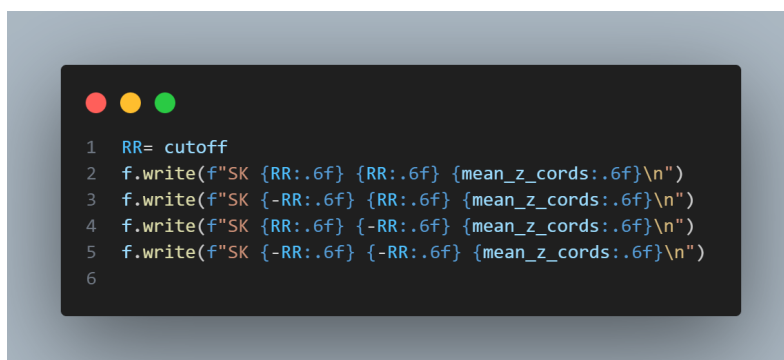
Figure 3.2: Bond-length constraint used to detect dissociated H_2 molecules.

2. **Local Reference Frame Definition.** To represent each adsorption environment consistently, a local coordinate system is constructed using the position of the adsorbed H_2 molecule. All atoms within a chosen cutoff radius from the molecule are selected and their coordinates are translated so that the center of the H_2 molecule becomes the origin. This ensures that every subflake is described relative to the same physically meaningful reference point. Over the course of the project, several reference definitions were tested (see Sec. 3.1.2), ranging from using the H_2 center to referencing each hydrogen atom individually. From the resulting relative coordinates, pairwise distances and spherical angles are computed to characterize the spatial neighborhood.
3. **Feature Extraction.** The closest atoms (ranked by distance) are selected and stored together with:
 - their chemical species,
 - distances to one or both hydrogen atoms,
 - spherical angular descriptors (inclination and azimuth),
 - their original Cartesian coordinates.

These collective descriptors encode the relevant geometric information surrounding the adsorption site.

4. **Addition of Placeholder Atoms for Uniform Image Dimensions.** To guarantee that every rendered subflake image has identical spatial extent and framing, four artificial atoms labelled SK are added at the corners of a square of side $\pm RR$, where RR corresponds to the chosen cutoff radius. These atoms do not represent

physical species; they serve exclusively as fixed reference points so that all generated pictures share the same width, height, and orientation. The code snippet in Figure 3.3 shows how these SK positions are written to the output.



```
1  RR= cutoff
2  f.write(f"SK {RR:.6f} {RR:.6f} {mean_z_cords:.6f}\n")
3  f.write(f"SK {-RR:.6f} {RR:.6f} {mean_z_cords:.6f}\n")
4  f.write(f"SK {RR:.6f} {-RR:.6f} {mean_z_cords:.6f}\n")
5  f.write(f"SK {-RR:.6f} {-RR:.6f} {mean_z_cords:.6f}\n")
6
```

Figure 3.3: Addition of four SK placeholder atoms at the corners of the bounding square to enforce uniform image dimensions.

This pipeline produces standardized structural representations that preserve the local environment responsible for adsorption energetics. The resulting `.xyz` dataset is the input used to train Graphenet to predict the energy difference quantity *flake_energy_diff*.

3.1.2 Coordinate Frame Variants

A central aspect of this work was determining how to represent the local environment surrounding the adsorbed H_2 molecule in a way that both preserves the structural physics of the system and provides sufficient detail for machine learning. Several coordinate-reference strategies were tested during the development of the preprocessing script, each offering different advantages.

For clarity and direct comparison, all example images shown in this section refer to the *same physical configuration*: adsorption *position 1* of *flake 3*. Only the coordinate transformation method changes, while the underlying atomic geometry remains identical. This allows the differences among the various representations to be visualized and analyzed without altering the underlying adsorption structure.

1. Reference to one of the two hydrogen atoms.

In this representation, the coordinate shift is performed with respect to a single hydrogen atom. This introduces a fixed direction in the local reference frame, but at the cost of breaking the intrinsic structure between the two hydrogen atoms. An example of a generated image using this convention is shown in Figure 3.4.

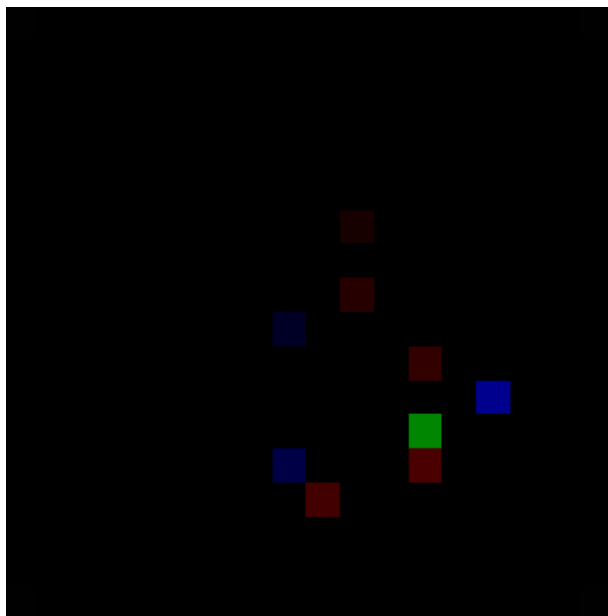


Figure 3.4: Example of an image representation generated using one hydrogen atom as reference. The color channels (R, G, and B) correspond to different atomic species, as defined earlier in the methodology.

2. Reference to the geometric midpoint of H_2 .

In this alternative, the midpoint between the two hydrogen atoms is used as the origin of the local coordinate frame. This representation preserves symmetry, because

both hydrogen atoms are treated equivalently; however, it does not encode the orientation of the H_2 bond. As a result, the resulting images are very similar to those obtained with the single-hydrogen reference, but without directional information.

3. Dual-frame representation.

To retain symmetry while still capturing orientation, the coordinates are computed twice: once with respect to each hydrogen atom. The two resulting representations are then combined into a single image. This produces a superposition of the two local environments, allowing bond orientation information to be implicitly encoded while avoiding bias toward one hydrogen atom. An example of such a dual-frame image is shown in Figure 3.5.

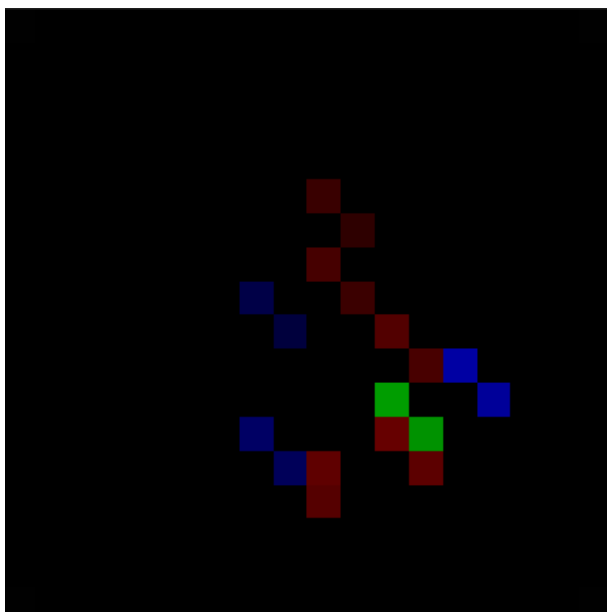


Figure 3.5: Example of an image representation generated using both hydrogen atoms as reference in a dual-frame superposition. The R, G, and B channels correspond to different atomic species.

4. Gaussian interpolation.

Pixel-based representations of atomic environments are typically sparse, since only a small number of pixels correspond to atom positions while the remaining area is empty. Such sparsity is not ideal for convolutional neural networks, which tend to learn more effectively from continuous spatial patterns. To address this, each atom is represented not as a single pixel, but as a two-dimensional Gaussian distribution centered at its projected coordinates. This smooths the image and spreads the information over neighboring pixels, making spatial features easier for the network to detect. An example of an image processed with Gaussian interpolation (using the same atomic coordinates as in the midpoint-based representation) is shown in Figure 3.6.

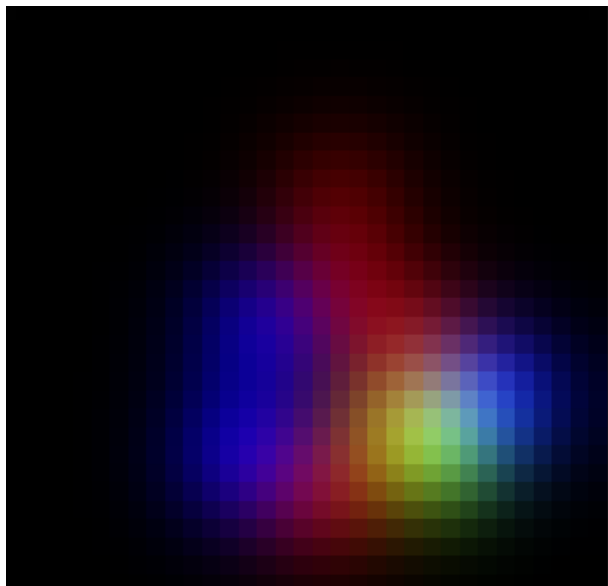


Figure 3.6: Example of an image representation after applying Gaussian interpolation. The Gaussian spread reduces image sparsity and emphasizes local spatial structure. Colors correspond to atomic species as described earlier.

These iterative refinements were essential to arrive at a structural encoding that is sufficiently descriptive for downstream learning.

3.1.3 Data Augmentation.

The Graphenet preprocessing pipeline includes a data-augmentation routine, implemented in the `lib_dataset_generation.py` file, which generates three additional samples for each input image by applying 90° clockwise, 90° counterclockwise, and 180° rotations. A snapshot of the corresponding function is provided in Figure 3.7.

These rotations generate alternative views of the same atomic configuration without modifying its physical meaning. The main benefit is an effective enlargement of the training dataset: by increasing the number of available samples, the neural network is exposed to more variations of the same underlying structure, which reduces overfitting and generally improves the stability and accuracy of the learning process.

Figures 3.8a, 3.8b, and 3.8c show examples of the data-augmentation procedure applied to the original image 3.4.

```

1 def data_augmentation(self):
2     for dir in ["train", "val", "test"]:
3         samples = [
4             f
5             for f in self.dpath.joinpath(dir).iterdir()
6             if (f.suffix == ".png" and not "R" in f.stem)
7         ]
8
9         for sample in tqdm(samples):
10            img = cv2.imread(str(sample), -1)
11
12            for angle in [1, 2, 3]:
13                if angle == 1:
14                    rotated_image = cv2.rotate(img, cv2.ROTATE_90_CLOCKWISE)
15                elif angle == 2:
16                    rotated_image = cv2.rotate(img, cv2.ROTATE_90_COUNTERCLOCKWISE)
17                elif angle == 3:
18                    rotated_image = cv2.rotate(img, cv2.ROTATE_180)
19
20                cropped_image = np.array(Utils.crop_image(rotated_image))
21
22                # Save the rotated image
23                cv2.imwrite(
24                    str(sample.with_stem(sample.stem + f"_R{angle}")), cropped_image
25                )

```

Figure 3.7: Rotation-based data-augmentation function from `lib_dataset_generation.py`, generating three augmented samples for each input image.

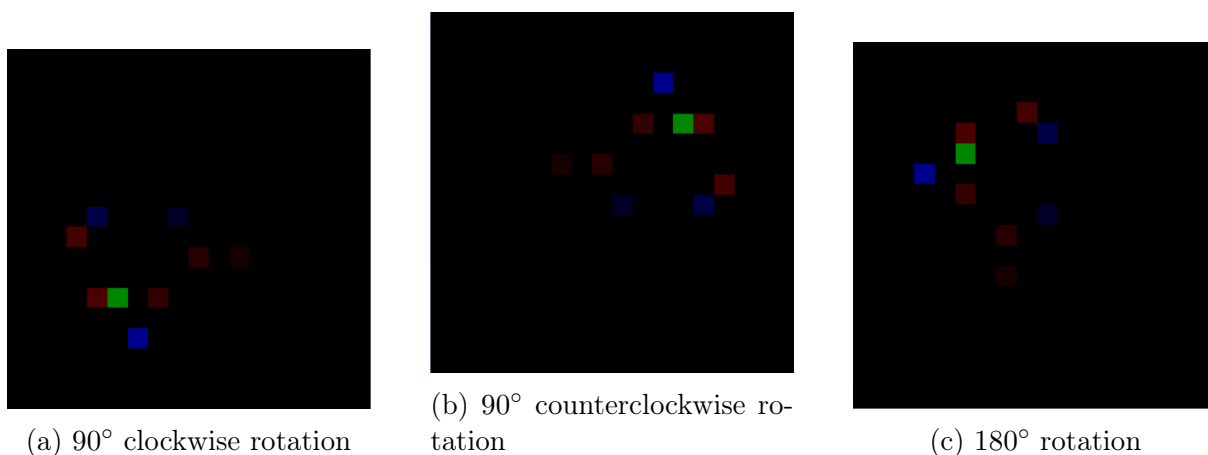


Figure 3.8: Augmented versions of the original configuration shown in Figure 3.4.

3.2 Target Analysis

Graphenet requires that the target information associated with each image input be provided in tabular form, specifically as a `.csv` file. During training, the network reads the target values directly from this file to compute the loss function and optimize its parameters. In the present work, the predicted property is the adsorption-related energy difference HAE , referred to in the code as *flake_energy_diff*, defined according to Equation (2.1) as the difference between the total energy of the graphene-oxide flake and the reference energy of the isolated hydrogen molecule. Consequently, a dedicated script named `create_the_csv_file.py` was implemented to generate the dataset of target values in a format compatible with Graphenet's training pipeline. This file, produced in

parallel with the structural images, establishes a one-to-one correspondence between each image (representing a specific flake configuration) and its associated scalar target.

3.2.1 Preprocessing Script: `create_the_csv_file`

The script `create_the_csv_file.py` is responsible for generating the `.csv` file that contains the energetic target values required by Graphenet during training. Its main purpose is to extract and organize the adsorption-related energy quantities stored in the hierarchical `flakes.json` dataset, and to associate each of them with the corresponding structural configuration represented by an `.xyz` file.

The `flakes.json` file is organized as a nested dictionary, where each entry (`flake_n`) includes the atomic coordinates of both the graphene flake and its hydrogen-adsorbed variants. For every adsorption site, the data include the total energy of the pure nanostructure (`flake_energy_wo_H2`), the total energy of the flake with an adsorbed hydrogen molecule (`flake_energy_w_H2`), and the corresponding adsorption energy difference (`flake_energy_diff`).

The script first loads the JSON dataset and iterates through the directory containing all preprocessed `.xyz` structures. Each `.xyz` file name encodes the flake identifier and adsorption site index, which are used to locate the corresponding entries within the JSON data structure. For each configuration, the following information is retrieved:

- the total number of atoms in the `.xyz` file,
- the total energy of the pure nanostructure (`flake_energy_wo_H2`),
- the total energy of the hydrogenated nanostructure (`flake_energy_w_H2`),
- and the computed adsorption energy difference (`flake_energy_diff`).

All retrieved quantities are appended to a Pandas `DataFrame`, where each row corresponds to a single adsorption configuration. Once the complete dataset is collected, different preprocessing strategies are applied to the target column in order to ensure numerical consistency and facilitate model convergence. These strategies include direct use of the raw energy values, min-max normalization, and statistical standardization. The resulting processed dataset is then exported as `flake_dataset_summary.csv`, which constitutes the definitive target file used by Graphenet to map each image-based input to its corresponding adsorption energy descriptor.

3.2.2 Processing of Target Values

During the preparation of the `flake_dataset_summary.csv` file, several approaches were tested to define the most suitable form of the target quantity to be used during

training. The purpose of these tests was to ensure that the target values were numerically stable and within a range that facilitates the convergence of the neural network optimizer. The following subsections describe the different preprocessing strategies applied to the adsorption energy difference values before determining the final configuration adopted in this work.

1. **Raw Target Data.** In the initial implementation, the target column was directly populated with the raw adsorption–energy values stored in the `flakes.json` file. These correspond to the quantity defined earlier in Eq. 2.1, namely the Hydrogen Adsorption Energy (HAE). In the dataset, however, this same physical quantity is stored under the variable name `flake_energy_diff`. Thus, although the notation differs between the theoretical definition (HAE) and the code implementation (`flake_energy_diff`), both refer to the same adsorption–energy difference used as the learning target.

This representation preserves the physical units of the adsorption energies (expressed in electronvolts) and directly reflects the energetic stability associated with each adsorption configuration. The resulting numerical values typically span a narrow range of a few electronvolts, as illustrated in Figure 3.9.

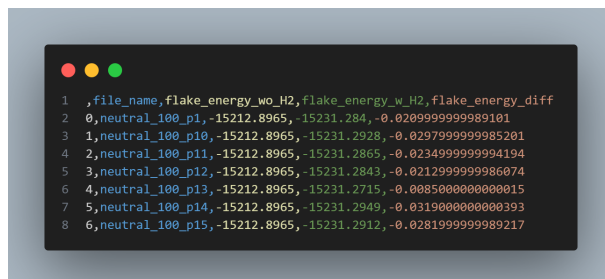


Figure 3.9: Distribution of raw adsorption energy differences extracted from the `flakes.json` file.

2. **Normalization.** A further transformation consisted in applying min–max normalization to rescale the energy differences within a fixed interval, in our case between 0 and 1:

$$E_{\text{norm}} = \frac{E_{\text{diff}} - E_{\text{min}}}{E_{\text{max}} - E_{\text{min}}}.$$

This operation preserves the relative differences among samples while constraining the target range to a uniform scale. Normalized values are dimensionless and facilitate consistent input–output scaling across datasets. The resulting distribution is shown in Figure 3.10.

3. **Standardization.** Finally, a statistical standardization procedure was implemented, centering and scaling each target value according to the mean (μ) and standard de-



Figure 3.10: Rescaled target distribution after min-max normalization within the $[0, 1]$ range.

viation (σ) of the dataset:

$$E_{\text{std}} = \frac{E_{\text{diff}} - \mu}{\sigma}.$$

This transformation produces a zero-centered distribution with unit variance, expressed in dimensionless standardized units. Any missing or undefined values were replaced by a small constant (10^{-4}) to avoid numerical instabilities. The resulting standardized targets are illustrated in Figure 3.11.

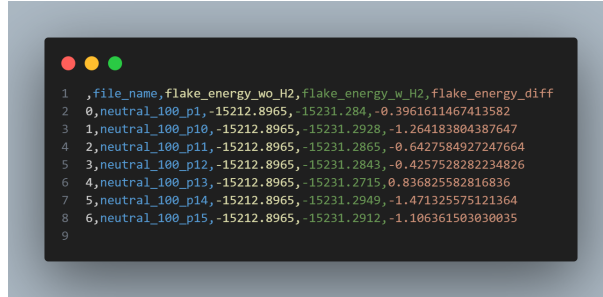


Figure 3.11: Distribution of target values after statistical standardization.

The final processed dataset is stored as a single `.csv` file, containing one row per adsorption configuration and one column corresponding to the target value `flake_energy_diff`. This file serves as the input required by Graphenet to associate each image-based flake representation with its corresponding adsorption energy descriptor.

4 Custom Convolutional neural network

4.1 Introduction

Although GrapheneNet performs well in its original setting, its accuracy decreases when applied to the local-environment dataset developed in this work. This may indicate that the architecture is not ideally suited for this representation. The central objective of this chapter is therefore the development of a **custom Convolutional Neural Network (CNN)** specifically designed to process multi-channel local-neighbourhood tensors and predict adsorption energies

The key contribution of this work is the design and evaluation of a custom CNN architecture, which:

- removes the 3-channel constraint of GrapheneNet,
- allows flexible incorporation of multiple structural and chemical descriptors,
- aims to improve predictive performance on the local-environment dataset.

The goal is to achieve accurate and efficient adsorption-energy prediction while enabling scalable screening of graphene-based hydrogen-storage materials.

The design of the custom CNN stems from the need to overcome a fundamental limitation of the original GrapheneNet framework, which relies on RGB image representations of atomic structures. This approach restricts the input to only three feature channels, which is insufficient to fully describe the multi-faceted chemical and geometric information of graphene oxide systems.

The proposed network architecture generalises this representation by allowing an arbitrary number of input channels

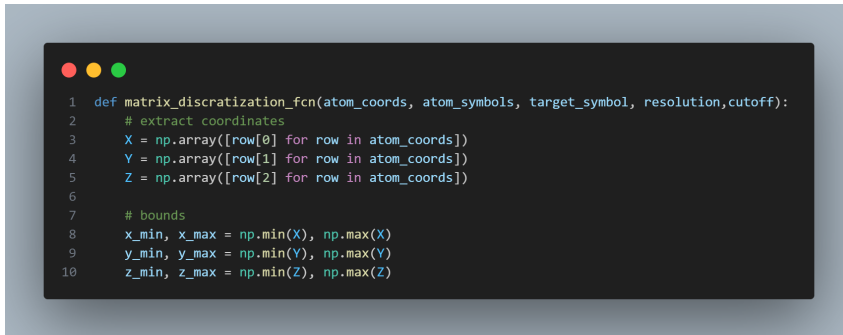
This flexibility enables the model to integrate a richer and more physically meaningful representation of the local adsorption environment, ultimately aiming to improve accuracy in predicting hydrogen adsorption energies.

4.2 Dataset Generation

As in the GrapheneNet methodology discussed in the previous chapter, the construction of the dataset relies on the `flakes.json` file. This file provides the complete structural description of each nanographene flake together with the corresponding adsorption energies. However, unlike GrapheneNet, which encodes the entire flake into a 3-channel (RGB) image, the present approach extracts only the local atomic environment around each adsorption site and represents it with an extended, multi-channel tensor. The `flakes.json` file therefore serves as the central data source, supplying both the atomic coordinates needed to assemble the input tensors and the target energy values used for supervised training.

As a first approach, the H_2 molecule to be adsorbed is treated by splitting it into its two hydrogen atoms. For each hydrogen atom, we compute its distance from every atom in the nanostructured flake (hereafter referred to simply as the *flake*). Since the flake atoms can be carbon, oxygen, or hydrogen, this results in six channels in total: three channels corresponding to the atomic distributions around the first hydrogen atom $(H_2)_1$, and three channels for the second one $(H_2)_2$.


Each channel is generated using the function `matrix_discretization_fcn()`, whose header and algorithm are shown in Figures 4.1 and 4.2, respectively.



```
1 def matrix_discretization_fcn(atom_coords, atom_symbols, target_symbol, resolution, cutoff):
2     # extract coordinates
3     X = np.array([row[0] for row in atom_coords])
4     Y = np.array([row[1] for row in atom_coords])
5     Z = np.array([row[2] for row in atom_coords])
6
7     # bounds
8     x_min, x_max = np.min(X), np.max(X)
9     y_min, y_max = np.min(Y), np.max(Y)
10    z_min, z_max = np.min(Z), np.max(Z)
```

Figure 4.1: Header of the `matrix_discretization_fcn()` function.

The function `matrix_discretization_fcn` converts the three-dimensional atomic coordinates of a selected element into a two-dimensional grid representation. It filters atoms within a defined cutoff region, subdivides the xy -plane into discrete bins, and assigns each cell the corresponding atomic height (z coordinate). The resulting matrix encodes the spatial distribution of that atomic species around the adsorption site, serving as one input channel for the CNN.



```

1  # bin edges
2  x_bins = np.linspace(0, 2 * cutoff, resolution + 1)
3  y_bins = np.linspace(0, 2 * cutoff, resolution + 1)
4
5  # discretization matrix
6  DISC_MATRIX = np.zeros((resolution, resolution))
7
8
9  for i in range(len(X)):
10     if atom_symbols[i] != target_symbol:
11         continue
12
13     for x_counter in range(len(x_bins)-1):
14         for y_counter in range(len(y_bins)-1):
15             if (X[i] >= x_bins[x_counter] and X[i] < x_bins[x_counter+1] and
16                 Y[i] >= y_bins[y_counter] and Y[i] < y_bins[y_counter+1]):
17                 DISC_MATRIX[y_counter, x_counter] = Z[i]

```

Figure 4.2: Algorithmic flow of the `matrix_discretization_fcn()` function used to create the discretized channel matrices.

4.2.1 Input Tensor Generation

Once the discretization matrices are obtained, the `save_input_sequence()` function is used to assemble them into the complete CNN input tensor. This function is called within the higher-level routine `process_flake()`, which iterates over all adsorption configurations and manages the full dataset preprocessing workflow.

The `save_input_sequence()` function generates a six-channel tensor representing the local atomic environment around the adsorption site. Each of the six channels corresponds to a different atomic species (C, O, H) for both hydrogen atoms of the H_2 molecule:

$$[C, O, H]_{H_1} + [C, O, H]_{H_2}.$$

The first three channels (`channel_0`, `channel_1`, `channel_2`) encode the spatial distribution of carbon, oxygen, and hydrogen atoms around the first hydrogen atom $(H_2)_1$, while the remaining three channels (`channel_3`, `channel_4`, `channel_5`) represent the same atomic distributions relative to the second hydrogen atom $(H_2)_2$. The resulting structure is thus a 4D tensor with dimensions:

$$(1, 6, \text{resolution}, \text{resolution}),$$

where `resolution` depends on the chosen spatial discretization step (0.5 Å in this case).

The header of the function and the definition of the channel indices are shown in Figure 4.3. The second part of the function, responsible for creating the tensor through successive calls to `matrix_discretization_fcn()`, is reported in Figure 4.4.

```

1 def save_input_sequence(atom_symbols_h1, atom_coords_h1, inclination_h1,azimut_h1,
2                         atom_symbols_h2, atom_coords_h2, inclination_h2,azimut_h2,
3                         cutoff, suffix="_mean",
4
5                         ):
6
7
8     """ saves the input data in a tensor with shapes (N_flakes x h2_positions , channels , H , W)"""
9     channels = 6
10    channel_0 = 0 # C
11    channel_1 = 1 # O
12    channel_2 = 2 # H
13    channel_3 = 3 # C
14    channel_4 = 4 # O
15    channel_5 = 5 # H

```

Figure 4.3: Header of the `save_input_sequence()` function and definition of the six tensor channels corresponding to different atomic species around the two hydrogen atoms.

```

1 resolution = int(2*cutoff/0.5)
2
3 # x = torch.zeros(1, 6, 40, 40)
4 per_hyd_pos_tensor= torch.zeros(1, channels, resolution, resolution) #per_hyd_pos_tensor is a tensor of shape (1,6,18,18)
5
6 # per_hyd_pos_tensor= np.array(1, channels, 18, 18) #per_hyd_pos_tensor is a tensor of shape (1,6,18,18)
7
8 # ### H1
9 per_hyd_pos_tensor[0,channel_0] = torch.from_numpy(matrix_discretization_fcn(atom_coords_h1, atom_symbols_h1,"C",resolution,cutoff))
10 per_hyd_pos_tensor[0,channel_1] = torch.from_numpy(matrix_discretization_fcn(atom_coords_h1, atom_symbols_h1,"O",resolution,cutoff))
11 per_hyd_pos_tensor[0,channel_2] = torch.from_numpy(matrix_discretization_fcn(atom_coords_h1, atom_symbols_h1,"H",resolution,cutoff))
12
13 # ### H2
14 per_hyd_pos_tensor[0,channel_3] = torch.from_numpy(matrix_discretization_fcn(atom_coords_h2, atom_symbols_h2,"C",resolution,cutoff))
15 per_hyd_pos_tensor[0,channel_4] = torch.from_numpy(matrix_discretization_fcn(atom_coords_h2, atom_symbols_h2,"O",resolution,cutoff))
16 per_hyd_pos_tensor[0,channel_5] = torch.from_numpy(matrix_discretization_fcn(atom_coords_h2, atom_symbols_h2,"H",resolution,cutoff))
17

```

Figure 4.4: Creation of the `per_hyd_pos_tensor` input tensor by combining the discretized matrices for each atomic species through calls to `matrix_discretization_fcn()`.

The resulting tensor, `per_hyd_pos_tensor`, encodes both chemical composition and geometric structure within the cutoff region. These per-hydrogen tensors are later collected and stacked by the `process_flake()` function to form the complete per-flake dataset, used as input for CNN training.

4.2.2 Target Tensor Generator

As described in the Target Analysis of the Graphenet chapter, the adsorption-related energy difference (HAE , see Equation (2.1)) constitutes the scalar target value associated with each hydrogen adsorption configuration. While the previous chapter employed a `.csv`-based target file compatible with Graphenet’s training pipeline, the present work integrates the same energetic information directly into the tensor-generation routine of the custom multi-channel CNN.

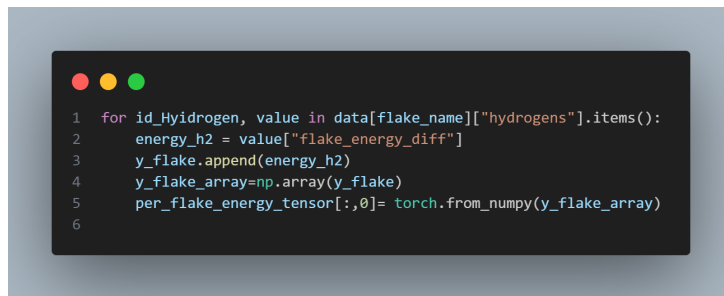
In this framework, the target tensor is constructed within the `process_flake()` function, which pairs each generated input tensor with its corresponding adsorption energy. For every flake, the atomic coordinates and chemical symbols are extracted together with the positions of the hydrogen atoms belonging to the H_2 molecule.

After validation, the adsorption energy difference stored in the dataset as `flake_energy_diff` is retrieved for each configuration and appended to the list `y_flake`. The collected values are then converted into a NumPy array and finally into a PyTorch tensor of shape

$$\text{per_flake_energy_tensor} \in \mathbb{R}^{(N,1)},$$

where N represents the number of adsorption sites associated with the considered flake. Each entry of this tensor corresponds to the adsorption energy of a single hydrogen configuration.

The code excerpt responsible for building the target tensor is reported in Figure 4.5. Within the loop over all adsorption sites, each `flake_energy_diff` value is extracted and inserted into the final energy tensor, ensuring perfect alignment with the set of generated input tensors.

A screenshot of a code editor window with a dark background and light-colored text. The code is written in Python and is enclosed in a light gray border. The code is as follows:

```
1 for id_Hydrogen, value in data[flake_name]["hydrogens"].items():
2     energy_h2 = value["flake_energy_diff"]
3     y_flake.append(energy_h2)
4     y_flake_array=np.array(y_flake)
5     per_flake_energy_tensor[:,0]= torch.from_numpy(y_flake_array)
6
```

Figure 4.5: Creation of the target tensor within the `process_flake()` function. Each adsorption configuration contributes a scalar energy value (`flake_energy_diff`), collected into the tensor `per_flake_energy_tensor`.

The `process_flake()` function therefore returns both the structured spatial input tensor and its associated target tensor, which together form the fundamental building blocks for training the present convolutional architecture.

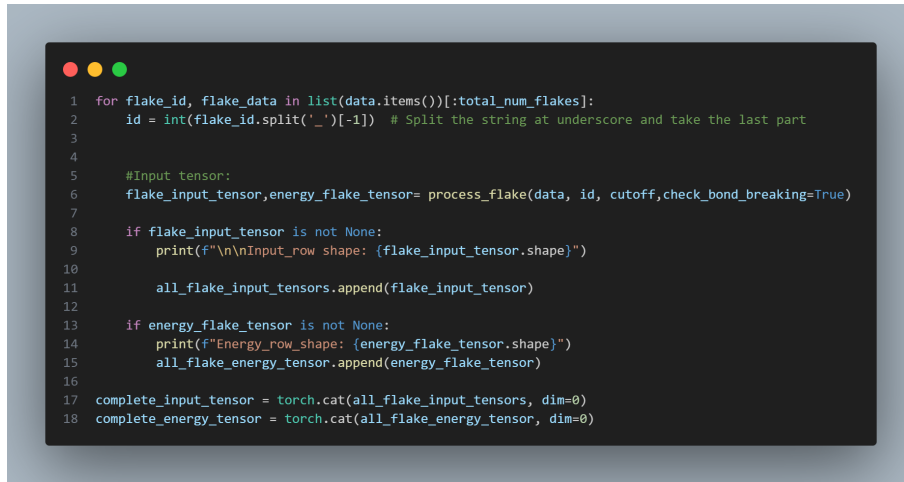
4.2.3 Complete Tensor Assembly

Once the input and target tensors have been generated for each flake, the final dataset is assembled by concatenating these tensors across all flakes. This operation is performed within a loop that iterates through the complete dataset, as illustrated in Figure 4.6. For every flake identifier, the routine calls `process_flake()` and retrieves both the input tensor (`flake_input_tensor`) and the corresponding energy tensor (`energy_flake_tensor`).

When successfully produced, the tensors are appended to `all_flake_input_tensors` and `all_flake_energy_tensor`, two lists that gather all processed configurations. After all flakes have been evaluated, the complete datasets are constructed via concatenation along the batch dimension:

```
complete_input_tensor = torch.cat(all_flake_input_tensors, dim=0),  
  
complete_energy_tensor = torch.cat(all_flake_energy_tensor, dim=0).
```

These tensors constitute the full training dataset for the custom multi-channel CNN model, in which each sample consists of a spatial representation of an adsorption configuration paired with its adsorption energy, already introduced and justified in the previous chapter.



```
1 for flake_id, flake_data in list(data.items())[:total_num_flakes]:  
2     id = int(flake_id.split('_')[-1]) # Split the string at underscore and take the last part  
3  
4  
5     #Input tensor:  
6     flake_input_tensor, energy_flake_tensor = process_flake(data, id, cutoff, check_bond_breaking=True)  
7  
8     if flake_input_tensor is not None:  
9         print(f"\nInput_row shape: {flake_input_tensor.shape}")  
10  
11         all_flake_input_tensors.append(flake_input_tensor)  
12  
13     if energy_flake_tensor is not None:  
14         print(f"Energy_row shape: {energy_flake_tensor.shape}")  
15         all_flake_energy_tensor.append(energy_flake_tensor)  
16  
17 complete_input_tensor = torch.cat(all_flake_input_tensors, dim=0)  
18 complete_energy_tensor = torch.cat(all_flake_energy_tensor, dim=0)
```

Figure 4.6: Assembly of the complete dataset. The loop iterates over all flakes, generating and collecting the corresponding input and energy tensors, which are concatenated into the final tensors `complete_input_tensor` and `complete_energy_tensor`.

To verify the correctness of the concatenation process, the tensor shapes are printed in the terminal, ensuring that both the input and target tensors have the expected dimensions, as shown in Figure 4.7.

```

Input_row shape: torch.Size([78, 6, 18, 18])
Energy_row_shape: torch.Size([78, 1])

Input_row shape: torch.Size([80, 6, 18, 18])
Energy_row_shape: torch.Size([80, 1])

Input_row shape: torch.Size([79, 6, 18, 18])
Energy_row_shape: torch.Size([79, 1])

Input_row shape: torch.Size([76, 6, 18, 18])
Energy_row_shape: torch.Size([76, 1])

```

Figure 4.7: Terminal output showing the shapes of the assembled input and target tensors, confirming the correct concatenation of all flake data.

4.3 Convolutional Neural Network

4.3.1 CNN Training Pipeline

Once the complete input and target tensors are assembled, the training of the convolutional neural network (CNN) is performed through the `CNN_launcher()` routine. This function is responsible for loading the dataset, preparing the train/validation split, normalizing the data, constructing the CNN model, and executing the full training procedure.

The dataset is first loaded using `sequence_builder_graphenet()`, which returns the input tensor

$$X \in \mathbb{R}^{(N, C, H, W)}$$

and the corresponding target tensor

$$Y \in \mathbb{R}^{(N, 1)}.$$

A random split divides the data into training and validation subsets according to a user-defined ratio.

Dataset Normalization

To ensure stable and consistent learning, both the input representation and the target energy values are **normalized based only on the training set**. In this work, normalization refers to *min-max scaling*, which rescales each variable to the range $[0, 1]$ using:

$$x_{\text{norm}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}.$$

This is handled by the `NormalizedCNNDataset` class, which:

- computes the minimum x_{\min} and maximum x_{\max} values of X and Y from the training subset,
- applies the same $(x_{\min}, x_{\max})_{\text{train}}$ parameters to the validation and test subsets,
- exposes normalized samples via `__getitem__()` for use in PyTorch `DataLoaders`.

The normalization workflow is schematically illustrated below:

Raw Train Data $\rightarrow (x_{\min}, x_{\max})_{\text{train}} \rightarrow \text{Normalize} \rightarrow \text{Train CNN}$

Raw Validation/Test Data $\rightarrow \text{Use } (x_{\min}, x_{\max})_{\text{train}} \rightarrow \text{Normalize} \rightarrow \text{Evaluate CNN}$

Dataset Standardization

An alternative to min–max normalization is **standardization**, where each variable is rescaled to have zero mean and unit variance:

$$x_{\text{std}} = \frac{x - \mu}{\sigma},$$

with μ and σ computed exclusively from the **training set**, mirroring the same data-leakage precautions described above.

Standardization is implemented in this work for comparative experiments, and is particularly useful when the distribution of input values is approximately Gaussian or when the relative scale of features carries less meaningful information than their variation patterns. In practice, the standardization procedure follows the same data pipeline structure:

- compute the mean μ_{train} and standard deviation σ_{train} from the training subset,
- apply these parameters to both validation and test subsets,
- return standardized samples through the dataset’s `__getitem__()` method.

This provides a controlled way to evaluate how the learning performance depends on the choice of input scaling strategy. In subsequent sections, we compare the effects of min–max normalization and standardization on model convergence and prediction accuracy.

This ensures that no information from the validation set leaks into the training process, preserving the integrity of model evaluation.

4.3.2 Neural Network Architecture

The model adopted in this work is a two-dimensional convolutional neural network designed to process the six-channel input tensor generated for each adsorption configuration. A schematic representation of the initial network architecture is shown in Figure 4.8, where the overall structure of the model is outlined.

```
class CNN2dRegression(nn.Module):
    def __init__(self, in_channels=6, hidden_channels=64, num_layers=4):
        super(CNN2dRegression, self).__init__()
        layers = []
        channels = in_channels
        for i in range(num_layers):
            layers.append(nn.Conv2d(channels, hidden_channels, kernel_size=3, padding=1))
            layers.append(nn.ReLU())
            channels = hidden_channels
        self.conv_layers = nn.Sequential(*layers)
        # Final linear regression head
        # Flatten and reduce to a single output
        self.regression_head = nn.Sequential(
            nn.AdaptiveAvgPool2d(1), # shape becomes (B, C, 1, 1)
            nn.Flatten(), # shape becomes (B, C)
            nn.Linear(hidden_channels, 1) # output: (B, 1)
        )
    def forward(self, x):
        x = self.conv_layers(x) # shape: (B, hidden_channels, 18, 18)
        out = self.regression_head(x) # shape: (B, 1)
        return out
```

Figure 4.8: Initial CNN architecture used for adsorption energy regression. The input tensor of shape $(B, 6, H, W)$ is processed through four convolutional blocks, followed by spatial pooling and a fully connected regression layer producing the final energy prediction.

The network is composed of two main blocks: a feature extraction module, implemented as a stack of convolutional layers, and a regression head that reduces the resulting feature maps to a single scalar quantity. Conceptually, the forward pass of the model can be summarised through the following schematic flow:

$$\begin{aligned}
 \text{Input (6 channels)} &\rightarrow (\text{Conv2D} + \text{ReLU})^{\times 4} \\
 &\rightarrow \text{AdaptiveAvgPool2D} \\
 &\rightarrow \text{Flatten} \\
 &\rightarrow \text{Linear} \\
 &\rightarrow \hat{y}.
 \end{aligned}$$

The convolutional block is responsible for identifying spatial patterns and local correlations in the multi-channel input representation. Each of the four convolutional layers applies a 3×3 kernel with padding, ensuring that the spatial dimensions are preserved throughout the feature extraction process. After each convolution, a ReLU activation introduces nonlinearity, allowing the network to capture increasingly complex features. This sequence of transformations progressively maps the original input into a richer representation:

$$(6, H, W) \rightarrow (64, H, W) \rightarrow (64, H, W) \rightarrow (64, H, W) \rightarrow (64, H, W).$$

Once the feature extraction stage is complete, the regression head condenses this spatial information into a compact vector through an adaptive average pooling operation:

$$(64, H, W) \rightarrow (64, 1, 1).$$

This pooled representation is then flattened into a 64-dimensional feature vector, which is finally mapped to a single scalar output through a fully connected linear layer:

$$(64) \rightarrow (1).$$

Overall, the model implements the following conceptual pipeline:

Local feature extraction \longrightarrow Global feature summarisation \longrightarrow Scalar regression.

Model Training

The model is trained using the **AdamW** optimizer, while a cosine learning rate scheduler is used to progressively reduce the learning rate during training, improving convergence stability. The loss minimized during training is a regression loss (Mean Squared Error), computed between the predicted adsorption energies and the reference values obtained from the dataset.

Mini-batch sampling, shuffling, and GPU data transfer are handled by PyTorch **DataLoader** objects, ensuring efficient training. The validation loss is monitored throughout training to evaluate generalization and prevent overfitting.

Finally, both the trained model weights and the preprocessing parameters are saved, ensuring that the model can be reliably used for inference and comparison with future models.

4.4 Model Optimization and Experimental Trials

To determine the most suitable configuration of the convolutional neural network, several controlled experimental trials were performed. Each trial consisted of modifying one or more architectural or preprocessing choices and observing their effect during training. The performance and quantitative outcomes of these tests are presented separately in Chapter 5. Here, we focus exclusively on the methodological aspects and the motivation behind each modification.

4.4.1 Normalization vs. Standardization

As discussed in Chapter 3, preprocessing plays a crucial role in stabilizing neural network training. For the present CNN architecture, two strategies were implemented and tested:

- **Min–Max Normalization:**

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}, \quad y' = \frac{y - y_{\min}}{y_{\max} - y_{\min}}$$

- **Standardization:**

$$x' = \frac{x - \mu_x}{\sigma_x}, \quad y' = \frac{y - \mu_y}{\sigma_y}$$

Both approaches were integrated into the data pipeline to evaluate their influence on numerical stability and training behavior. The comparative performance of each preprocessing technique is presented in Chapter 5.

4.4.2 Increasing the Number of Input Channels

The baseline model used six input channels generated from local structural descriptors. Since the dataset also provides spherical-coordinate information (azimuth and inclination) for each atom, four additional channels were incorporated, increasing the dimensionality of the input representation from 6 to 10 channels (Fig. 4.9).

These new channels encode the angular environment of the system. For each of the two hydrogen atoms involved in the adsorption complex, one channel stores the inclination angles and another stores the azimuth angles, aggregating the corresponding values of all neighbouring C, O, and H atoms. This extension enriches the spatial representation by integrating directional information and allowing the CNN to capture orientation-dependent interactions.

The effect of this modification is analyzed in Chapter 5.

```

#### H1
per_hyd_pos_tensor[0,channel_0] = torch.from_numpy(matrix_discretization_fn(atom_coords_h1, atom_symbols_h1,"C",resolution,cutoff))
per_hyd_pos_tensor[0,channel_1] = torch.from_numpy(matrix_discretization_fn(atom_coords_h1, atom_symbols_h1,"O",resolution,cutoff))
per_hyd_pos_tensor[0,channel_2] = torch.from_numpy(matrix_discretization_fn(atom_coords_h1, atom_symbols_h1,"H",resolution,cutoff))
per_hyd_pos_tensor[0,channel_3] = torch.from_numpy(azimuth_inclin_matrix_fn(atom_coords_h1,resolution,cutoff,azimuth_h1))
per_hyd_pos_tensor[0,channel_4] = torch.from_numpy(azimuth_inclin_matrix_fn(atom_coords_h1,resolution,cutoff,inclination_h1))

#### H2
per_hyd_pos_tensor[0,channel_5] = torch.from_numpy(matrix_discretization_fn(atom_coords_h2, atom_symbols_h2,"C",resolution,cutoff))
per_hyd_pos_tensor[0,channel_6] = torch.from_numpy(matrix_discretization_fn(atom_coords_h2, atom_symbols_h2,"O",resolution,cutoff))
per_hyd_pos_tensor[0,channel_7] = torch.from_numpy(matrix_discretization_fn(atom_coords_h2, atom_symbols_h2,"H",resolution,cutoff))
per_hyd_pos_tensor[0,channel_8] = torch.from_numpy(azimuth_inclin_matrix_fn(atom_coords_h2,resolution,cutoff,azimuth_h2))
per_hyd_pos_tensor[0,channel_9] = torch.from_numpy(azimuth_inclin_matrix_fn(atom_coords_h2,resolution,cutoff,inclination_h2))

```

Figure 4.9: Input tensor extended to 10 channels by including angular information.

4.4.3 Gaussian Interpolation of the Input Channels

The raw input channels are highly sparse, containing non-zero values only at atomic coordinates (Fig. 4.10). CNNs struggle to extract informative patterns from extremely sparse spatial data due to the limited receptive overlap between convolution kernels and isolated non-zero points.

To mitigate sparsity, a Gaussian interpolation was applied, distributing atomic contributions smoothly across neighboring grid points (Fig. 4.11). This operation converts the inputs from nearly binary masks into continuous fields, allowing kernels to form richer spatial representations. The resulting performance changes are reported in Chapter 5.

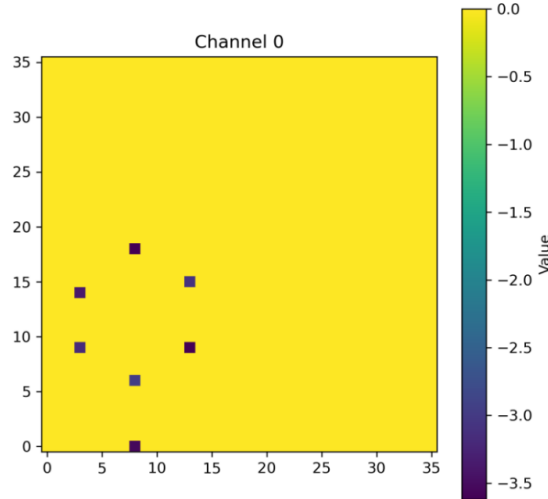


Figure 4.10: Example of a raw input channel (Carbon), showing strong sparsity.

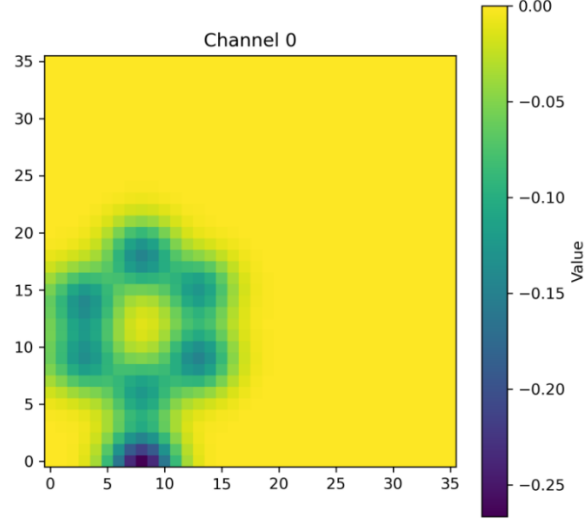


Figure 4.11: Same channel after Gaussian interpolation (kernel $\sigma = 2$), improving spatial continuity.

4.4.4 Hyperparameter Space Exploration

A series of controlled experiments was carried out to explore the effect of key architectural and training hyperparameters. The following components were varied independently or in combination:

- network depth (number of feature channels),
- cutoff radius for grid construction,
- spatial grid resolution,
- dropout rate,
- learning rate,
- number of epochs,
- Gaussian interpolation width (σ),
- batch size.

The optimal configuration identified through these trials is reported in Chapter 5, together with the final performance comparison between the custom CNN and Graphenet.

5 Results

5.1 Graphenet

The different coordinate-frame strategies introduced in Section 3.1.2 were evaluated to assess their influence on the predictive performance of the neural network. In this section, we analyze how each choice affects the learning dynamics of the model and the quality of the resulting predictions, first without and then with the data augmentation described in Section 3.1.3. .

5.1.1 Single-Hydrogen Reference Frame

Using either one hydrogen atom or the midpoint between the two H_2 atoms as the coordinate origin produces an extremely sparse image: each atom occupies essentially a single pixel in the input tensor (Fig. 5.1). As most of the grid remains empty, the convolutional filters receive very limited spatial information, hindering the extraction of meaningful patterns. Although physically consistent, this representation yields poor predictive performance, with a best result of $R^2 = 0.650$.

With the application of data augmentation, the result improves slightly, as shown in Figure 5.2, with $R^2 = 0.714$.

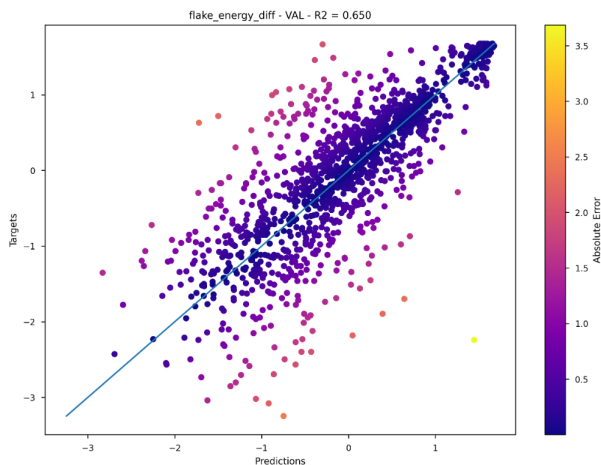


Figure 5.1: Accuracy plot for the single-hydrogen reference frame (no data augmentation).

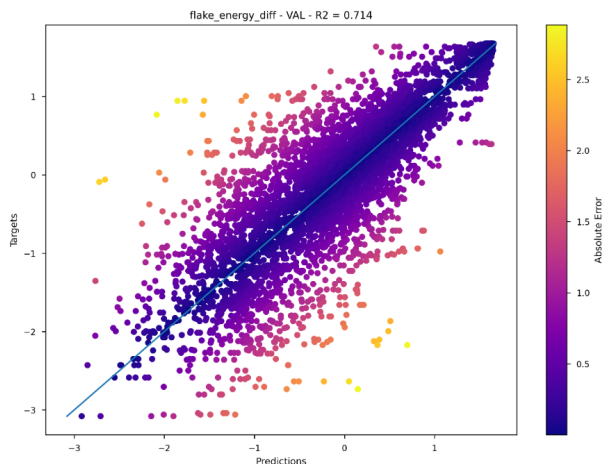


Figure 5.2: Accuracy plot for the single-hydrogen reference frame with data augmentation.

5.1.2 Dual-Frame Representation

The dual-frame approach offers a richer representation through the superposition of two coordinate systems. By overlaying the two atomic reference frames, the resulting image contains more information about the orientation of the H_2 molecule and a significantly larger number of non-zero pixels. This more informative encoding provides the network with improved predictive accuracy.

The best result achieved, shown in Figure 5.3, is $R^2 = 0.743$.

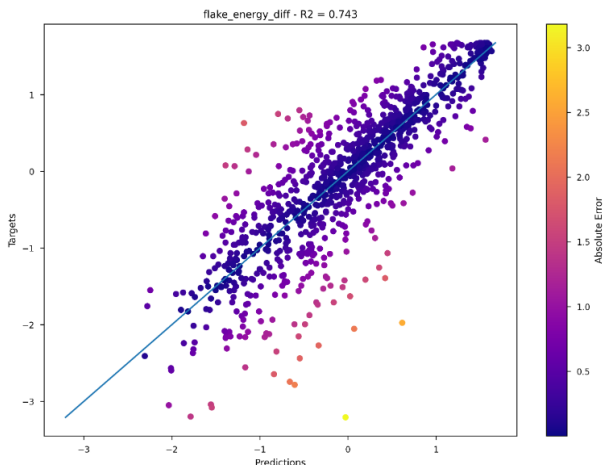


Figure 5.3: Accuracy plot for the dual-frame representation (no data augmentation).

With data augmentation, the result increases further reaching $R^2 = 0.791$, as shown in Figure 5.4.

5.1.3 Gaussian Interpolation

The application of Gaussian smoothing to the pixel-based atomic images has a remarkably positive impact on learning. By replacing atomic pixels with Gaussian dis-

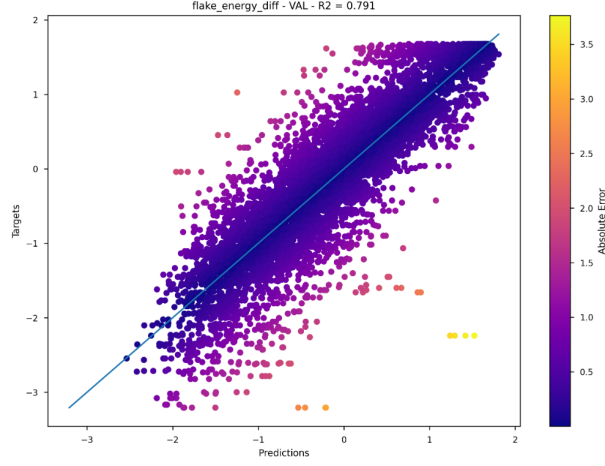


Figure 5.4: Accuracy plot for the dual-frame representation with data augmentation.

tributions, the input becomes a continuous field in which neighborhood information is naturally encoded. Training becomes more stable, convergence is significantly faster, and both training and validation performance show a substantial improvement, reaching $R^2 = 0.795$.

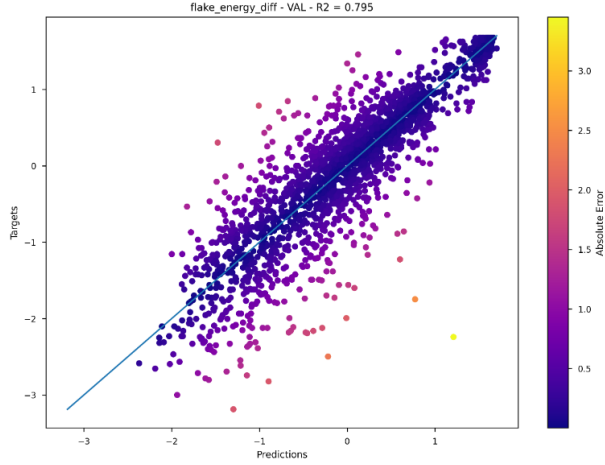


Figure 5.5: Accuracy plot for the Gaussian interpolation representation (no data augmentation).

With data augmentation, we reach the highest accuracy value obtained: $R^2 = 0.818$, as shown in Figure 5.6.

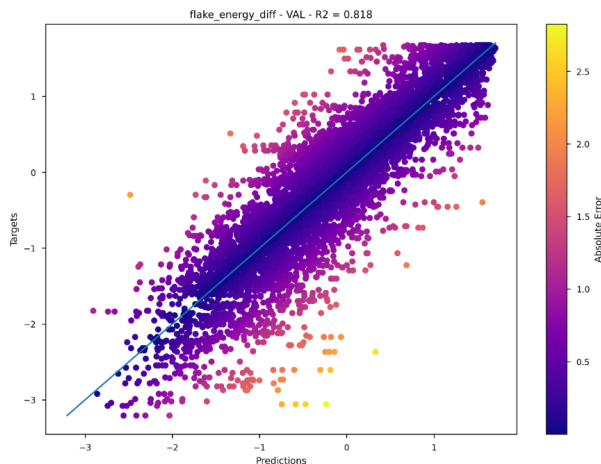


Figure 5.6: Accuracy plot for the Gaussian interpolation representation with data augmentation.

5.2 Custom CNN

This chapter presents the quantitative results obtained from the experimental trials described in Chapter 4.4. All experiments discussed in the following sections are conducted using the data augmentation strategy outlined previously. Each section evaluates the impact of a specific methodological modification on model performance, with the goal of identifying the most effective configuration for predicting hydrogen adsorption energies. .

5.2.1 Effect of Preprocessing: Normalization vs. Standardization

A first investigation focused on the impact of input preprocessing. Although both normalization and standardization map the data into comparable numerical ranges, their effect on the optimization process proved substantially different. When normalization was employed, the learning curves exhibited noticeable fluctuations and slower convergence, suggesting difficulties in stabilizing gradient magnitudes across layers. In contrast, standardization provided a markedly smoother descent of the loss, with fewer oscillations and a more predictable training trajectory. This improvement is visible in Fig. 5.7, where the standardized inputs lead to a steadier and more coherent reduction of both training and validation losses. It was therefore adopted for all subsequent experiments.

5.2.2 Effect of Increasing the Number of Input Channels

The dataset originally supplied six channels encoding distances and atom-type information. However, by including inclination and azimuth, the input tensor expanded to ten channels (Fig. 4.9), thus enriching the geometric information provided to the CNN.

As illustrated by the regression plots in Fig. 5.8 and Fig. 5.9, the model trained with ten channels produces predictions that align more closely with the target energies,

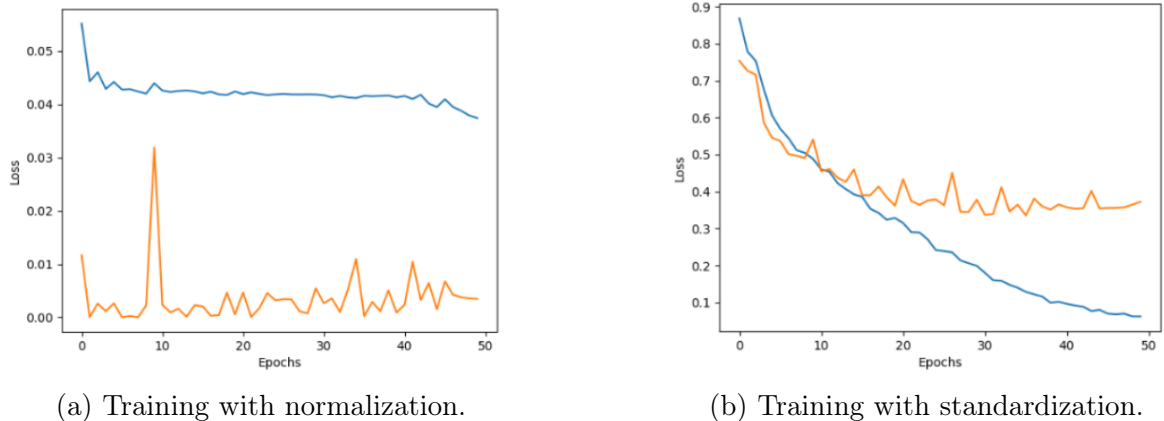


Figure 5.7: Comparison of training behavior for the two preprocessing strategies.

increasing the accuracy factor from $R_{6\text{ch}}^2 = 0.6729$ to $R_{10\text{ch}}^2 = 0.7023$.

The scatter of the validation points is visibly reduced, showing that the inclusion of angular descriptors leads to a more faithful representation of the underlying physical trends.

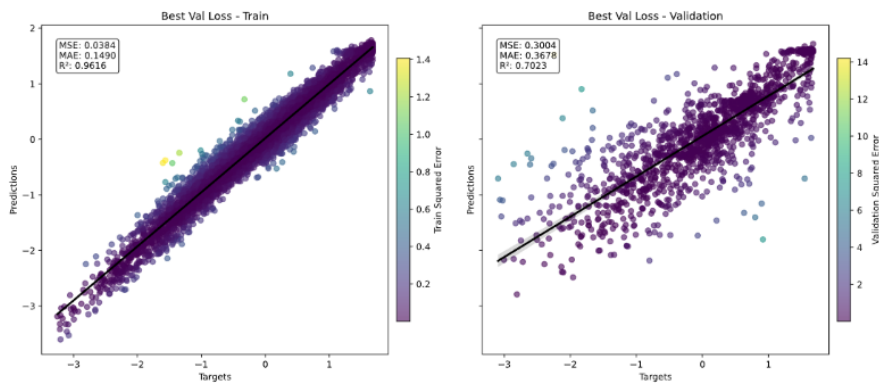


Figure 5.8: Regression results for the 10-channel model.

5.2.3 Effect of Gaussian Interpolation

Another challenge encountered in the modeling process was the extreme sparsity of the raw input tensors. Each channel contained meaningful values only at atom positions, while the rest of the grid remained zero. Such sparsity severely limited the network’s ability to extract patterns, especially in the initial convolutional layers that depend on local spatial continuity.

To mitigate this effect, Gaussian interpolation was applied to each channel. This procedure spreads the contribution of an atom over neighboring grid points, transforming the input from a nearly binary mask into a smooth, continuous density map. The contrast between the raw and interpolated channels is evident in Fig. 4.10 and Fig. 4.11.

This smoothing significantly enhanced the learning capability of the CNN. The re-

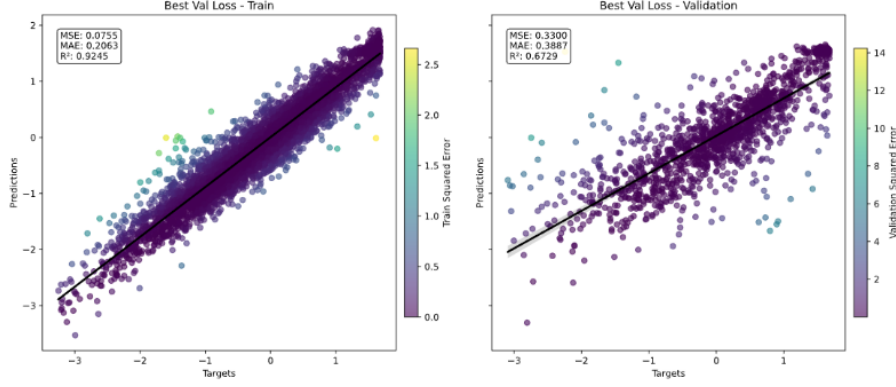


Figure 5.9: Regression results for the 6-channel model.

gression results after interpolation (Fig. 5.10) reveal a stronger linear correlation between predicted and target values, and a clear improvement in generalization. These findings confirm that reducing sparsity is essential when using CNNs to process spatially discretized atomistic data.

This results in further performance improvement, with $R^2_{\text{interp}} = 0.7449$

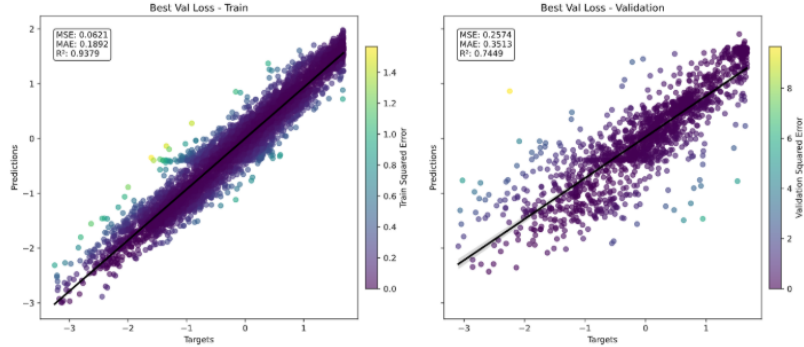


Figure 5.10: Regression performance after Gaussian interpolation.

5.2.4 Final Model Performance

The model's performance was further improved by systematically exploring various hyperparameters, including dropout, learning rate, network depth, cutoff radius, interpolation width, spatial resolution, and number of training epochs. Through this iterative search, a configuration emerged that provided the best trade-off between accuracy and stability.

Under this optimized setup, the CNN achieved an accuracy score of $R^2 = 0.7745$

The regression trends in Fig. 5.11 show well-aligned training and validation distributions, indicating that the model captures the main physical dependencies of the adsorption energy without excessive overfitting. The narrow dispersion around the ideal line demon-

strates that the combination of angular channels, Gaussian interpolation, and refined hyperparameters yields a robust and reliable model.

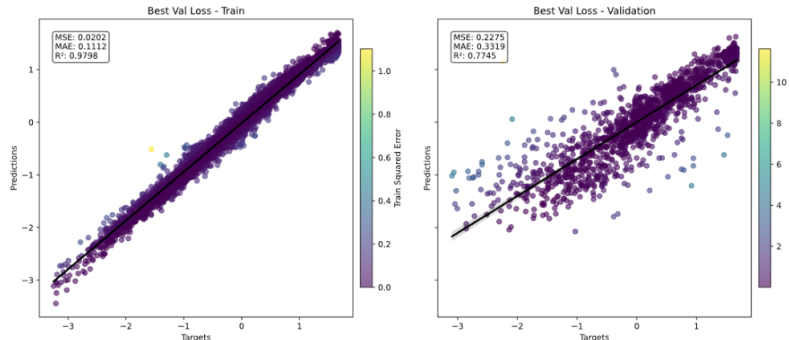


Figure 5.11: Regression performance for the best CNN configuration.

5.2.5 Summary of Optimal Hyperparameters

After extensive experimentation and hyperparameter tuning, the optimal configuration for the custom CNN was identified. The key parameters of the final model are summarized below:

- **Network depth:** 256 channels, providing sufficient representational capacity to capture complex spatial patterns in the input.
- **Cutoff radius:** 4.6 Å, defining the local neighbourhood considered around each atom.
- **Spatial resolution:** 0.2 Å, ensuring a fine-grained discretization of the input space.
- **Dropout rate:** 0.45, effectively regularizing the network and reducing the risk of overfitting.
- **Training epochs:** 1000, allowing the model to converge while preventing excessive training.
- **Learning rate:** 1×10^{-4} , balancing stable convergence and efficient gradient updates.
- **Batch size:** 32, providing a compromise between gradient estimation accuracy and memory efficiency.

This combination of hyperparameters yielded the best-performing CNN in this study, achieving a balance between predictive accuracy and model generalization.

6 Conclusions

This thesis investigated the development of image-based neural models for predicting hydrogen adsorption energies on graphenic materials. Two complementary pipelines were explored: the original *Graphenet* framework, based on pixelized atomic projections, and a custom convolutional neural network (CNN) architecture designed to overcome the intrinsic limitations of the initial representation.

Summary of Findings

A first set of experiments evaluated how different coordinate-frame choices influence the structure of the input tensors and the corresponding predictive performance. The single-hydrogen reference frame yielded highly sparse images, offering insufficient spatial continuity for effective feature extraction. The dual-frame strategy alleviated this issue by increasing the number of non-zero pixels and providing richer information about molecular orientation, leading to better predictive accuracy. The most significant improvement, however, came from the introduction of Gaussian interpolation, which transformed discrete atomic pixels into smooth density fields. This approach not only improved spatial coherence but also enabled the network to capture more meaningful geometric relationships, ultimately achieving the best performance within the original Graphenet framework, with a final accuracy of

$$R^2 = 0.818.$$

Building on these findings, the second part of the study introduced a custom-designed CNN. A comprehensive analysis of preprocessing methods demonstrated that standardization consistently stabilized the optimization trajectory and led to more reliable convergence than simple normalization. Enriching the input channels with angular descriptors (inclination and azimuth) further enhanced the representation, improving accuracy by capturing essential geometric features of the adsorption sites. Gaussian interpolation again proved crucial, confirming that reducing sparsity is essential for convolution-based architectures.

Finally, a systematic hyperparameter search, including dropout, learning rate, grid

resolution, interpolation width, and network depth, yielded an optimized configuration achieving an accuracy of

$$R^2 = 0.7745,$$

showing that the combination of dense geometric encoding, angular information, and carefully tuned architecture results in a robust and reliable predictive model.

Future Work

While the results achieved in this study are encouraging, several promising research directions remain open and warrant further exploration:

- **Expansion of the Dataset.** The performance of neural networks strongly depends on the quantity and diversity of the training data. Increasing the size of the dataset, both in terms of additional adsorption sites and a broader variety of graphenic configurations, would likely lead to further improvements in accuracy and generalization.
- **Additional Model Simulations.** Due to time constraints, the number of training simulations exploring different hyperparameters (such as cutoff radius, number of epochs, spatial resolution, interpolation width, and network depth) was limited. Running a more extensive hyperparameter search would likely uncover configurations that further improve accuracy and model stability.
- **Exploration of Alternative Neural Architectures.** While the CNN proved effective, other architectures may capture long-range interactions or rotational symmetries more naturally. Promising candidates include Graph Neural Networks (GNNs), attention-based models, or hybrid architectures combining convolutional layers with transformer blocks.

Final Remarks

Overall, this work demonstrates that image-based neural models can successfully capture the complex physical trends governing hydrogen adsorption on carbon-based surfaces. By progressively enriching the input representation and refining the architecture, significant improvements were achieved over the baseline approaches. The methods developed here lay the groundwork for more advanced and scalable machine-learning tools for atomistic simulations, and they open the door to future investigations aimed at improving performance, interpretability, and physical reliability.

Bibliography

- [1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, 2016. Figure of MLP adapted from this reference.
- [2] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. Figure of CNN structure adapted from this reference.
- [3] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision (ECCV)*, volume 8689 of *Lecture Notes in Computer Science*, pages 818–833. Springer, 2014. Figure of CNN filters adapted from this reference.
- [4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. Figures of residual blocks and ResNet18 architecture adapted from this reference.
- [5] Tommaso Forni, Matteo Baldoni, Fabio Le Piane, and Francesco Mercuri. Graphenet: a deep learning framework for predicting the physical and electronic properties of nanographenes using images. *Scientific Reports*, 14:24576, 2024. Source for Graphenet framework and figure 1.
- [6] Peter Atkins and Julio de Paula. *Atkins’ Physical Chemistry*. Oxford University Press, 10th edition, 2014.
- [7] Alexander A. Balandin. Thermal properties of graphene and nanostructured carbon materials. *Nature Materials*, 10(8):569–581, 2011.
- [8] Keith T. Butler, Donald W. Davies, Hugh Cartwright, Olexandr Isayev, and Aron Walsh. Machine learning for molecular and materials science. *Nature*, 559(7715):547–555, 2018.
- [9] Ben Hourahine, Bálint Aradi, Volker Blum, et al. Dftb+, a software package for efficient approximate dft simulations. *The Journal of Chemical Physics*, 152(12):124101, 2020.

- [10] Kamil Jastrzębski et al. Emerging technologies for green, sustainable energy: carbon-based materials for hydrogen storage. *Materials*, 14(23):6826, 2021.
- [11] Walter Kohn and Lu Jeu Sham. Self-consistent equations including exchange and correlation effects. *Physical Review*, 140(4A):A1133–A1138, 1965.
- [12] Pekka Koskinen and Ville Mäkinen. Density-functional tight-binding for beginners. *Computational Materials Science*, 47(1):237–253, 2009.
- [13] Guozhong Cao. *Nanostructures and Nanomaterials: Synthesis, Properties and Applications*. Imperial College Press, 2004.
- [14] Daniel R. Dreyer, Sungjin Park, Christopher W. Bielawski, and Rodney S. Ruoff. The chemistry of graphene oxide. *Chemical Society Reviews*, 39(1):228–240, 2010.
- [15] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010.
- [16] Matthias Rupp, Alexandre Tkatchenko, Klaus-Robert Müller, and O. Anatole von Lilienfeld. Fast and accurate modeling of molecular atomization energies with machine learning. *Physical Review Letters*, 108(5):058301, 2012.
- [17] Tian Xie and Jeffrey C. Grossman. Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties. *Physical Review Letters*, 120(14):145301, 2018.
- [18] Valentina Tozzini and Vittorio Pellegrini. Graphene-based hydrogen storage for fuel cell applications. *Physical Chemistry Chemical Physics*, 15(1):80–89, 2013.