Graph Neural Networks for Alloy Property Analysis

A scientific project submitted to
École Centrale de Casablanca

Authors:
ADLANI Saad, AHRABAR Wiam, BAMMAD Ihssane, CHATBA Abir,
CHBIHI Doha

January 2025

Introduction The study of multi-component systems, such as alloys, is critical in materials science and engineering due to their diverse applications in industrial and technological domains. Understanding the physical and chemical properties of alloys requires robust computational tools to model their complex atomic structures and behaviors.

Graph Neural Networks (GNNs) have emerged as a powerful method for analyzing atomic systems by leveraging graph-based representations. Unlike traditional machine learning methods, GNNs can capture both local and global atomic interactions, making them ideal for predicting alloy properties such as segregation trends, mechanical strength, and catalytic performance.

This project focuses on the design, training, and evaluation of a GNN to simulate the surface and bulk properties of ternary alloys. The trained model enables accurate predictions of atomic configurations, significantly reducing the computational cost compared to traditional Density Functional Theory (DFT) simulations.

In this report, we outline the methodology for training the GNN, the dataset preparation process, and the validation of the model's predictions. Additionally, we discuss the results obtained and their implications for the study and design of advanced materials.

Data Loading In this section, we describe the process of loading atomic data and calculating the distances between atoms. The data used is stored in a CSV file containing coordinates (`X`, `Y`, `Z`) and energy values for each atom.
✢ main-dataset.db

Code: The following Python code demonstrates the steps to load the data, calculate distances, and display the first few rows of the dataset:

```
# Import necessary libraries
import pandas as pd
import numpy as np
from scipy.spatial import distance_matrix
import torch
from torch_geometric.data import Data

# Load data
file_path = "/content/data2.csv"  # Replace with the
    correct path
data = pd.read_csv(file_path, sep=';')
```

```
data.head(4)
```

Listing 1: Python Code for Data Loading

Result: The first few rows of the dataset after loading are as follows:

Dataset Preview

Snapshot	Atom Index	X	Y	Z	Energy
0	0	0.000000	0.000000	6.0	7.201712
0	1	2.800143	0.000000	6.0	7.201712
0	2	5.600286	0.000000	6.0	7.201712
0	3	0.000000	2.800143	6.0	7.201712

This table shows the initial atomic positions and energy values, which will be used in subsequent steps for graph representation and analysis.

Methodology

The methodology involves three main steps: calculating the distance matrix, constructing the graph edges, and preparing the data for Graph Neural Network (GNN) processing.

Step 1: Calculate the Distance Matrix The first step involves calculating the pairwise distances between all atoms in the dataset. The coordinates (`X`, `Y`, `Z`) of the atoms are extracted, and the distance matrix is computed using the `distance_matrix` function from the `scipy.spatial` module. The following Python code was used for this calculation:

```
# Extract X, Y, Z columns
coordinates = data[['X', 'Y', 'Z']].values

# Calculate the distance matrix
dist_matrix = distance_matrix(coordinates, coordinates)

# Display a portion of the distance matrix
print("\nMatrix of calculated distances:")
print(dist_matrix[:5, :5])  # Display a subset of the
    matrix
```

The resulting matrix contains the pairwise distances between atoms. A portion of the matrix is shown below:

Matrix of Calculated Distances

```
[[0.         2.80014285 5.6002857  2.80014285 3.96      ]
 [2.80014285 0.         2.80014285 5.6002857  3.96      ]
 [5.6002857  2.80014285 0.         2.80014285 3.96      ]
 [2.80014285 5.6002857  2.80014285 0.         3.96      ]
 [3.96       3.96       3.96       3.96       0.        ]]
```

Step 2: Create the Graph Edges In this step, edges are created based on a distance threshold. Atoms are considered connected if their distance is below a predefined threshold (e.g., 3 angstroms). The pairs of connected atoms are identified and stored as edges in the graph. The following code snippet illustrates this process:

```
# Define a threshold for neighbors
threshold = 2.0

# Identify connected atoms
edge_indices = np.where((dist_matrix < threshold) & (
    dist_matrix > 0))

# Convert to source and target format
edges = np.array([edge_indices[0], edge_indices[1]])

# Display a portion of the edges
print("\nSome graph edges:")
print(edges[:, :10])  # Display first 10 edges
```

The resulting edges represent the connections between atoms within the specified distance. A subset of these edges is displayed below:

Some graph edges:

```
[[  0   0   0   0   0   0   1   1   1   1]
 [  1   3  64  66  72 127   0   2   3  65]]
```

Step 3: Prepare Data for the GNN The data is prepared for GNN processing by extracting energy values as node features and converting the edge indices into a PyTorch-compatible tensor format. The code below demonstrates this step:

```
# Extract energy values as node features
node_features = torch.tensor(data['Energy'].values,
    dtype=torch.float).view(-1, 1)

# Convert edges to tensor format
edge_index = torch.tensor(edges, dtype=torch.long)
```

The resulting tensor for the edge indices is shown below. Each column represents an edge in the graph, where the first row indicates the source node, and the second row indicates the target node:

Edge Indices Tensor

```
tensor([[    0,     0,     0,  ..., 10079, 10079, 10079],
        [    1,     3,     9,  ..., 10070, 10076, 10078]])
```

Step 4: Visualizing the Graph To better understand the structure of the graph, we visualize it using NetworkX and Matplotlib. The graph represents the atomic interactions, where nodes correspond to atoms and edges represent connections between atoms based on a predefined distance threshold.

Code:

```
import torch
import networkx as nx
import matplotlib.pyplot as plt
from torch_geometric.utils import to_networkx

# Load the graph data
file_path = "graph_data.pt"  # Path to your .pt file
graph_data = torch.load(file_path)

# Convert the PyTorch Geometric graph to a NetworkX
    graph
G = to_networkx(graph_data, to_undirected=True)
```

```
12
13 # Configure the size of the image
14 plt.figure(figsize=(15, 15))
15
16 # Define a layout for organizing the nodes
17 pos = nx.spring_layout(G, seed=42, k=0.1)
18
19 # Sample a subset of nodes for better clarity if the
   graph is too large
20 if len(G.nodes()) > 1000:
21     sampled_nodes = list(G.nodes())[:1000]
22     G = G.subgraph(sampled_nodes)
23     pos = {n: pos[n] for n in sampled_nodes}
24
25 # Define node sizes and colors
26 node_sizes = [20 * (G.degree(node) + 1) for node in G.
   nodes()]
27 node_colors = [G.degree(node) for node in G.nodes()]
28
29 # Draw nodes and edges
30 nx.draw_networkx_nodes(G, pos, node_size=node_sizes,
   node_color=node_colors,
31                        cmap=plt.cm.viridis, alpha=0.8)
32 nx.draw_networkx_edges(G, pos, edge_color="gray", alpha
   =0.3)
33
34 # Add labels to top 10 nodes by degree
35 top_degree_nodes = sorted(G.degree, key=lambda x: x[1],
   reverse=True)[:10]
36 top_labels = {node: str(node) for node, _ in
   top_degree_nodes}
37 nx.draw_networkx_labels(G, pos, labels=top_labels,
   font_size=10, font_color="red")
38
39 plt.title("Graph␣Visualization", fontsize=18)
40 plt.show()
```

Result The resulting graph visualization is shown in Figure 1. Nodes are sized and colored based on their connectivity, highlighting the key nodes

and relationships in the graph.

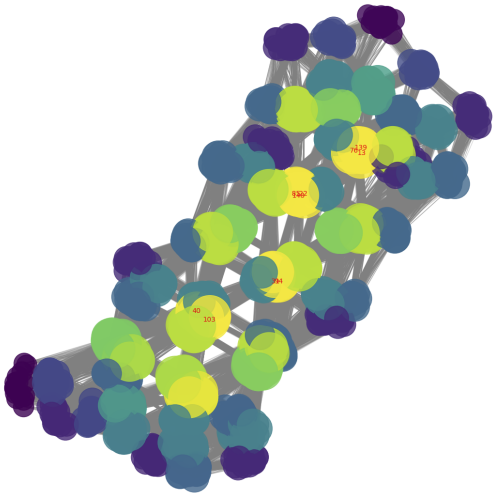

Figure 1: Visualization of the graph, showing nodes (atoms) and edges (atomic interactions).

Step 5: Training and Preparing the GNN Model

Description of the Model and Inputs The data is first divided into training, validation, and test sets. Each set is transformed into a graph format compatible with PyTorch Geometric.

The GNN model consists of three graph convolutional layers (`GCNConv`) and one fully connected layer. The following key features are worth noting:

- Input Features: The three spatial coordinates (`X`, `Y`, `Z`) serve as the input to the model.
- Hidden Dimension: Each convolutional layer maps the input to a feature space of size 64, enhancing the model's capacity to learn complex relationships.
- Output: The model predicts a single energy value for the graph.

```
# Split the data into training, validation, and test sets
train_data, temp_data = train_test_split(data, test_size=0.3, random_state=42)
val_data, test_data = train_test_split(temp_data, test_size=0.5, random_state=42)

# Create Data objects for each set
train_graph = create_graph_data(train_data)
val_graph = create_graph_data(val_data)
test_graph = create_graph_data(test_data)

from torch_geometric.loader import DataLoader

# Create loaders for each set
batch_size = 1  # You can adjust the batch size
train_loader = DataLoader([train_graph], batch_size=batch_size, shuffle=True)
val_loader = DataLoader([val_graph], batch_size=batch_size, shuffle=False)
test_loader = DataLoader([test_graph], batch_size=batch_size, shuffle=False)

import torch.nn as nn
from torch_geometric.nn import GCNConv

class GNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GNNModel, self).__init__()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.conv3 = GCNConv(hidden_dim, hidden_dim)  # Additional layer
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
```

```
        x = torch.relu(x)
        x = self.conv2(x, edge_index)
        x = torch.relu(x)
        x = self.conv3(x, edge_index)
        x = torch.relu(x)
        x = torch.mean(x, dim=0)
        x = self.fc(x)
        return x

# Initialize the model
input_dim = 3  # X, Y, Z
hidden_dim = 64
output_dim = 1  # Energy
model = GNNModel(input_dim, hidden_dim, output_dim)
```

Listing 2: Training and Preparing the GNN Model

This structure allows the GNN to learn from the spatial relationships and predict energy values effectively.

The following code snippet demonstrates the training and validation process for the Graph Neural Network (GNN) model. The model was trained for 100 epochs, and the Mean Absolute Error (MAE) was used as the loss metric to evaluate performance on both training and validation datasets.

```
# Initialize lists to store MAE for training and validation
train_losses = []
val_losses = []

# Training loop
num_epochs = 100
best_val_loss = float('inf')  # Track the best validation loss
best_model = None

for epoch in range(num_epochs):
    # Training phase
    model.train()
    train_loss = 0
    for batch in train_loader:
        optimizer.zero_grad()
```

```
        out = model(batch)
        loss = criterion(out, batch.y)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    # Validation phase
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for batch in val_loader:
            out = model(batch)
            loss = criterion(out, batch.y)
            val_loss += loss.item()

    # Calculate average loss
    train_loss /= len(train_loader)
    val_loss /= len(val_loader)

    # Store losses for visualization
    train_losses.append(train_loss)
    val_losses.append(val_loss)

    # Save the best model
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_model = model.state_dict()

# Plot MAE over epochs
plt.figure(figsize=(10, 6))
plt.plot(range(1, num_epochs + 1), train_losses, label='Training MAE', color='blue', marker='o')
plt.plot(range(1, num_epochs + 1), val_losses, label='Validation MAE', color='orange', marker='o')
plt.xlabel('Epoch')
plt.ylabel('MAE (Mean Absolute Error)')
plt.title('Training and Validation MAE Over Epochs')
plt.legend()
plt.grid(True)
plt.tight_layout()
```

```
plt.savefig("mae_training_validation.png", dpi=300)
plt.show()
```

Listing 3: Training and Validation Process with MAE Visualization

Training and Validation MAE Plot The resulting plot of the training and validation MAE over the epochs is shown in Figure 2. The plot highlights the convergence of the training process and the stability of the validation loss.

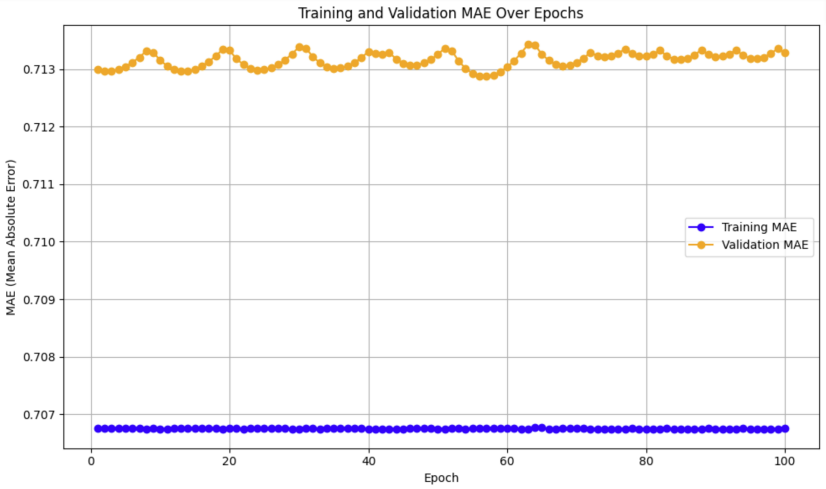

Figure 2: Training and Validation MAE Over Epochs

Interpretation of Results The plot illustrates the following key observations:

- The training MAE remains stable throughout the epochs, maintaining a value around **0.707**.
- The validation MAE shows minor fluctuations around **0.713**, suggesting that the model has not overfitted and generalizes well to unseen data.
- The difference between training and validation MAE is minimal, indicating that the model captures the underlying patterns without significant bias or variance.

Conclusion The low and stable MAE values for both training and validation datasets confirm the effectiveness of the model's architecture and training process. The model is well-suited for predicting atomic properties, with minimal errors across different datasets. Future work could explore further optimization of hyperparameters or alternative architectures to reduce validation loss even further.