ORNL/TM-2023/3203

User Manual - HydraGNN: Distributed PyTorch Implementation of Multi-Headed Graph Convolutional Neural Networks

Massimiliano Lupo Pasini
Jong Youl Choi
Pei Zhang
Justin Baker

November 2023

ORNL/TM-2023/3203

ORNL Artificial Intelligence Initiative

User Manual - HydraGNN: Distributed PyTorch Implementation of Multi-Headed Graph Convolutional Neural Networks

Massimiliano Lupo Pasini
Jong Youl Choi
Pei Zhang
Justin Baker

November 2023

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831
managed by
UT-Battelle LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

AI	Artificial Intelligence
ASE	Atomic Simulation Environment
CI	Continuous Integration
DDP	Distributed Data Parallelism
DL	Deep Learning
FC	Fully Connected
GAT	Global Attention
GIN	Graph Isomorphism Network
GNN	Graph Neural Network
HPO	Hyperparameter Optimization
MLP	Multi-Layer Perceptron
MPNN	Message Passing Neural Network
MTL	Multi-task Learning
NN	Neural Network
ORNL	Oak Ridge National Laboratory
PNA	Principal Neighborhood Aggregation
PyG	PyTorch-Geometric
PyPI	Python Package Index
US	United States
DoE	Department of Energy

ACKNOWLEDGEMENTS

This work was supported in part by the Office of Science of the Department of Energy and by the Artificial Intelligence Initiative as part of the Laboratory Directed Research and Development (LDRD) Program of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC, for the US Department of Energy under contract DE-AC05-00OR22725. This work was supported in part by the US-DOE Advanced Scientific Computing Research (ASCR) under contract DE-SC0023490.

ABSTRACT

This document serves as user manual for HydraGNN, a scalable graph neural network (GNN) architecture that allows for a simultaneous prediction of multiple target properties using multi-task learning (MTL).

The HydraGNN architecture is constructed by successive superposition of three different sets of layers. The first set is made of message-passing layers to exchange information across nodes in the graph and use this to update the nodal features. The second set is made of global pooling layers that aggregate information from all the nodes in the graph and map it into a scalar, and is needed only for global target properties that are related to the entire graph. The third set of layers is dedicated to the implementation of MTL, which is enabled by forking of the architecture into separate heads, each one of them dedicated to the predictive task of one specific target property.

Through an object-oriented programming paradigm, HydraGNN is templated over different message-passing policies, which allows for a user-friendly hyperparameter study to assess the sensitivity of the predictive performance of the HydraGNN architecture on a specific dataset with respect to the choice of the message-passing policy. The object-oriented paradigm used by HydraGNN also allows for a user-friendly inclusion of newly developed message passing policies within the existing framework.

HydraGNN supports distributed computing capabilties for scalable data reading and scalable training on leadership-class supercomputers.

MOTIVATION AND BACKGROUND

1. INTRODUCTION

Deep learning (DL) has emerged as a transformative field within artificial intelligence (AI), revolutionizing various domains with its remarkable ability to learn intricate patterns and representations from vast amounts of data. At its core, DL relies on neural networks (Nns), a class of models inspired by the human brain's interconnected neurons. These networks consist of multiple layers, each containing neurons that perform computations on the input data and progressively learn higher-level features. By leveraging large-scale labeled datasets and powerful computational resources, DL models have achieved unprecedented success in tasks such as image recognition, natural language processing, and speech recognition.

While traditional DL has predominantly focused on structured data like images and sequences, the need for analyzing complex and interconnected data has given rise to Graph Neural Networks (GNNs). GNNs extend the capabilities of deep learning to data represented as graphs, where entities (nodes) and their relationships (edges) form intricate network structures. GNNs allow the incorporation of graph topology, enabling the models to understand and reason about the underlying graph connectivity. This unique feature makes GNNs well-suited for a wide range of applications, including social network analysis, molecule property prediction, and recommendation systems, where data is naturally represented as graphs. By applying message passing and aggregation techniques across graph nodes and edges, GNNs can effectively capture local and global dependencies, making them powerful tools for learning expressive representations and performing tasks on graph-structured data.

2. RELEVANCE TO THE UNITED STATES - DEPARTMENT OF ENERGY (DOE)

GNNs have shown significant promise in various scientific domains, particularly those that involve complex relationships and structures, such as materials science, physics, chemistry, and biology. Some specific ways in which GNNs could be important in accelerating scientific applications at the United States (US) Department of Energy (DoE) include:

- Materials Discovery: GNNs can be used to predict and optimize material properties, accelerating the search for new materials with desired characteristics for energy storage, catalysis, and other applications.
- Drug Discovery: In the field of computational biology and pharmaceuticals, GNNs can help predict molecular properties and interactions, streamlining the drug discovery process.
- Climate Science: GNNs can aid in understanding complex climate models and predicting climate patterns, contributing to more accurate climate simulations and predictions.
- Particle Physics: In high-energy physics, GNNs can analyze complex detector data and assist in particle identification, event classification, and anomaly detection.
- Energy Efficiency: GNNs can be applied to optimize energy systems, such as power grids, energy consumption, and load balancing.
- Network Analysis: GNNs can enhance the understanding of energy distribution networks and improve the management and efficiency of energy resources.

It is essential to note that the successful implementation of GNN in scientific applications depends on several factors, including data availability, computational resources, and domain-specific challenges.

3. DISTINCT CAPABILITIES OF HYDRAGNN

Open-source GNN libraries lack crucial features for atomistic modeling, including multi-task learning (MTL), user freedom to seamlessly switch between different MPNN layers, hyperparameter control, distributed data parallelism (DDP), and regular maintenance.

HydraGNN addresses these gaps because:

- it supports MTL, is open-source https://github.com/ORNL/HydraGNN,
- it offers an object-oriented interface for message passing policies,
- it provides HPC capabilities for scaling with DDP on supercomputers, and
- it maintains compatibility with required software packages through continuous integration (CI) tests.

4. MATHEMATICAL BACKGROUND

4.1 GRAPH NEURAL NETWORKS (GNNS)

GNNs are a class of DL models designed to process and analyze data structured as graphs. Graphs are mathematical structures that consist of nodes (vertices) connected by edges (links). GNNs aim to learn representations of nodes and edges in a graph, allowing them to perform tasks such as node classification, link prediction, graph classification, and more.

The main idea behind GNNs is to leverage information from neighboring nodes to update the node representations iteratively. This process allows GNNs to capture the structural information of the graph and incorporate it into the learning process. GNNs are well-suited for various real-world applications where data can be represented as graphs, such as social networks, citation networks, molecule structures, knowledge graphs, recommendation systems, and more.

A graph G is usually represented in mathematical terms as

$$G = (V, \mathcal{E}) \tag{1}$$

where V represents the set of nodes and $\mathcal{E}$ represents the set of edges between these nodes (1). An edge $(u, v) \in \mathcal{E}$ connects nodes u and v, where $u, v \in V$, $\mathcal{E} \in V \times V$. The topology of a graph can be described through the *adjacency matrix*, A, an $N \times N$ square matrix where N is the number of nodes in the graph, whose entries are associated with edges of the graph according to the following rule:

$$\begin{cases} A[u, v] = 1 & \text{iff} \quad (u, v) \in \mathcal{E} \\ A[u, v] = 0 & \text{otherwise.} \end{cases} \tag{2}$$

The degree of a node $u \in V$ is defined as:

$$d_u = \sum_{v \in V} A[u, v] \tag{3}$$

and represents the number of edges connected to a node. Every node u is represented by a a-dimensional feature vector $\mathbf{x} \in \mathbb{R}^a$ containing the embedded nodal properties and also a label vector $\mathbf{y} \in \mathbb{R}^b$ in tasks related to node-level predictions. In order to take advantage of the topology of the graph, many DL models include both the number of neighbors per node, as well as the length of each edge between nodes. Larger sizes of the local neighborhood lead to a higher computational cost to train the GNN model, as the number of regression coefficients to train at each hidden convolutional layer increases proportional to the number of neighbors.

The general architecture of a GNN model involves the following components:

- Input Layer: Nodes and edges in the graph are initially represented with feature vectors. For node-centric tasks, each node is associated with a feature vector, while edges may have their own features as well.
- Message Passing Layer: The core operation in GNNs is message passing, where information from neighboring nodes is aggregated and combined to update the node representations. This process is typically performed iteratively for multiple layers. A particular example of message passing layers is the graph convolutional layer (GCL), which allow each node to gather information from its neighbors and update its feature representation accordingly.

- Global Pooling Layer: After several iterations of message passing, a global pooling layer aggregates the final node representations to produce a graph-level representation. This representation can be used for graph-level tasks like graph classification or regression tasks where the target property is a global property of the entire graph.
- Fully Connected Layer: a stack of fully connected (FC) layers can be optionally added at the end of the GNN architecture to increase the expressivity of the GNN model in predicting both nodal outputs as well as global outputs.

GNN models that use GCLs for message passing are called graph convolutional neural networks (GCNNs), which extract information from local interactions between nodes of a graph, and transfer the learnt interactions from one local neighborhood to another, to alleviate the computational burden of DL training. Currently, GCNN models are extensively used in material science to predict material properties from atomic information by directly mapping the atomic structure input to graphs, with atoms as graph nodes and chemical bonds as edges (2; 3). Bond angles (4) and crystallographic information (5) have also been directly included in GCNN models to improve predictive accuracy. GCNNs not only reduce the cumbersome and expensive data pre-processing, but can also naturally transfer the learning across lattices of different structures and sizes.

Differently from GCLs, message passing layers are a more general concept used in GNNs. In message passing, each node sends and receives messages to and from its neighbors, and these messages are computed based on the node's features and the features of its neighbors. The aggregation function is not limited to weighted sums as it happens in GCL, allowing for greater flexibility. In fact, message passing neural networks (MPNNs) often rely on attention mechanisms and/or skip connections to establish more complex relations between embedding features constructed by the stack of hidden layers.

4.1.1 Input layer

In a GNN model, the input layer is responsible for encoding the initial node and edge features of the input graph.

For each node $v_i \in V$, the input layer takes its initial feature vector $\mathbf{X}_i^{(0)} \in \mathbb{R}^{d^{(0)}}$ and encodes it as follows:

$$\mathbf{X}_i^{(0)} = \text{Encoding}(\text{Node Features of } v_i), \tag{4}$$

where $\mathbf{X}_i^{(0)}$ is the encoded feature vector of node v_i at the input layer, and $d^{(0)}$ represents the dimensionality of the initial node features.

Similarly, for each edge $e_{ij} \in E$ between nodes v_i and v_j, the input layer takes its initial feature vector $\mathbf{E}_{ij}^{(0)} \in \mathbb{R}^{d_e^{(0)}}$ and encodes it as follows:

$$\mathbf{E}_{ij}^{(0)} = \text{Encoding}(\text{Edge Features of } e_{ij}), \tag{5}$$

where $\mathbf{E}_{ij}^{(0)}$ is the encoded feature vector of edge e_{ij} at the input layer, and $d_e^{(0)}$ represents the dimensionality of the initial edge features.

The encoding process can involve various techniques such as embedding layers, feature transformations, or even direct use of raw features, depending on the specific GNN architecture and the characteristics of the input data.

Additional position information may be given for each node $v_i \in V$, $\mathbf{P}_i^{(0)} \in \mathbb{R}^n$ where n is the spatial dimension.

After the input layer, the GNN proceeds to perform message aggregation and node update steps in subsequent layers to propagate information and learn expressive representations throughout the graph.

4.1.2 Message Passing Layer

In a GNN model, a message passing layer is responsible for aggregating and passing information from neighboring nodes and edges to update the node representations.

At the l-th layer of the GNN architecture, for each node $v_i \in V$, the message aggregation step computes the aggregated messages $\mathbf{M}_i^{(l)}$ from its neighboring nodes and edges. It can be formulated as:

$$\mathbf{M}_i^{(l)} = \text{Aggregate}_{\mathcal{N}(v_i)}^{(l)}\left(\mathbf{X}_i^{(l)}, \mathbf{X}_j^{(l)}, \mathbf{E}_{ij}^{(l)}\right), \tag{6}$$

where $\mathbf{M}_i^{(l)}$ is the aggregated message for node v_i at the l-th layer, $\mathcal{N}(v_i)$ represents the set of neighboring nodes of v_i, and $\mathbf{X}_i^{(l)}$ and $\mathbf{X}_j^{(l)}$ are the node representations of v_i and v_j at layer l, respectively. Additionally, $\mathbf{E}_{ij}^{(l)}$ denotes the edge representation between nodes v_i and v_j at layer l.

After aggregating messages, the node update step incorporates the aggregated messages to update the node representations. It can be expressed as:

$$\mathbf{X}_i^{(l+1)} = \text{Update}^{(l)}(\mathbf{X}_i^{(l)}, \mathbf{M}_i^{(l)}), \tag{7}$$

where $\mathbf{X}_i^{(l+1)}$ is the updated representation of node v_i at the $(l+1)$-th layer, and $\text{Update}^{(l)}(\cdot)$ is a function that takes the current node representation $\mathbf{X}_i^{(l)}$ and the aggregated message $\mathbf{M}_i^{(l)}$ as inputs and produces the updated representation.

By iterating the message passing and node update steps for multiple layers, the GNN can propagate information throughout the graph, capturing complex dependencies and structural patterns to learn expressive node representations for various graph-related tasks.

4.1.3 Equivariant Layer

If an equivariant architecture is selected, the message passing layer incorporates an $E(n)$-equivariant update to the positional information. This process assumes that the node representations $\mathbf{X}_i$ are $E(n)$-invariant. The message aggregation for the equivariant architecture incorporates the distance between nodes as an invariant feature as follows

$$\mathbf{M}_i^{(l)} = \text{Aggregate}_{\mathcal{N}(v_i)}^{(l)}\left(\mathbf{X}_i^{(l)}, \mathbf{X}_j^{(l)}, \|\mathbf{P}_i^{(l)} - \mathbf{P}_j^{(l)}\|, \mathbf{E}_{ij}^{(l)}\right), \tag{8}$$

where $\mathbf{P}_i^{(l)}$ is the position of node v_i at the l-th layer. The position update at each layer can be expressed as

$$\mathbf{P}_i^{(l+1)} = \text{Equivariant}_{\mathcal{N}(v_i)}^{(l)}(\mathbf{P}_i^{(l)}, \mathbf{P}_j^{(l)}, \mathbf{X}_i^{(l)}, \mathbf{X}_j^{(l)}), \tag{9}$$

where $\text{Equivariant}_{\mathcal{N}(v_i)}^{(l)}(\cdot)$ is an $E(n)$-equivariant function that takes the current node representation $\mathbf{X}_i^{(l)}$ and positions $\mathbf{P}_i^{(l)}$ as inputs and produces the updated position.

4.1.4 Global Pooling Layer

In a GNN model, a global pooling layer is used to aggregate information from all nodes in the graph to create a graph-level representation. It summarizes the node-level features into a fixed-size representation that can be used for graph-level tasks. Let's denote the input graph as G = (V, E) with node set V and edge set E.

At the final layer of the GNN, the global pooling layer aggregates the node representations $\mathbf{X}_i^{(L)}$ at the final layer L (where L is the total number of layers) to produce a graph-level representation $\mathbf{Z}$:

$$\mathbf{Z} = \text{GlobalPooling}(\mathbf{X}_i^{(L)}), \tag{10}$$

where $\mathbf{X}_i^{(L)}$ represents the set of all node representations at the final layer, and $\mathbf{Z}$ is the graph-level representation.

The aggregation function GlobalPooling$(\cdot)$ can take various forms, such as mean pooling, max pooling, or attention-based pooling, depending on the specific task and requirements. For example, mean pooling computes the element-wise mean of all node representations, while max pooling takes the element-wise maximum. Attention-based pooling can be learned weights that assign importance to different node representations during aggregation.

The resulting graph-level representation $\mathbf{Z}$ captures essential information from all nodes in the graph and can be used for tasks like graph classification or regression.

Overall, the global pooling layer allows the GNN to summarize the node-level information into a fixed-size representation, enabling it to handle graphs of varying sizes and produce outputs at the graph level.

4.1.5 Fully Connected Layer

In a GNN model, an FC layer at the end of the stack of message passing layers (and optionally global pooling layers) is used to map the graph-level representation to the final output. The FC layer typically takes the graph-level representation as input and produces the final prediction or output for the specific task at hand.

Let's denote the graph-level representation as $\mathbf{Z}$, which may be the output of a global pooling layer or the direct result of the final message passing layer.

The fully connected layer can be described as follows:

$$\mathbf{O} = \text{FullyConnected}(\mathbf{Z}), \tag{11}$$

where $\mathbf{O}$ is the output of the fully connected layer.

The FC layer involves a linear transformation followed by a non-linear activation function. The linear transformation can be represented as a matrix multiplication:

$$\mathbf{O}' = \mathbf{Z} \cdot \mathbf{W} + \mathbf{b}, \tag{12}$$

where $\mathbf{W}$ is the weight matrix and $\mathbf{b}$ is the bias vector of the FC layer.

After the linear transformation, a non-linear activation function is applied element-wise to introduce non-linearity to the output:

$$\mathbf{O} = \text{Activation}(\mathbf{O}'), \tag{13}$$

where Activation$(\cdot)$ is a non-linear activation function, such as ReLU (Rectified Linear Unit), sigmoid, or softmax, depending on the specific task and model design.

The FC layer allows the GNN to map the graph-level representation to the appropriate output space, making it suitable for various node-related and graph-related predictive tasks.

Parameter sharing of FC layers for node-level predictions

For node-level predictions, the same FC layer is reused (recycled) by the GNN model to make node-level predictions on different nodes of the graph. The key idea is parameter sharing, which allows the model to use the same set of weights and biases in the fully connected layer for all nodes in the graph.

When a GNN processes a graph, it applies the same set of message passing layers to each node in the graph, resulting in node representations that capture the local neighborhood information for each node. After the message passing layers, the node representations $\mathbf{X}_i^{(L)}$ for all nodes v_i in the graph are obtained at the final layer L.

Subsequently, the FC layer is applied to each node representation to make node-level predictions:

$$\mathbf{O}_i = \text{FullyConnected}(\mathbf{X}_i^{(L)}), \tag{14}$$

where $\mathbf{O}_i$ represents the prediction for node v_i.

Since the FC layer shares the same weights and biases across all nodes, it uses the learned information from the entire graph to make predictions for each individual node. This parameter sharing significantly reduces the number of parameters in the model and enables the GNN to generalize well to new, unseen nodes during inference. Moreover, this parameter sharing mechanism is one of the strengths of GNNs that allows them to handle graphs of varying sizes efficiently.

4.2 MULTI-TASK LEARNING (MTL)

MTL involves training a single model to perform multiple related tasks simultaneously, rather than training separate models for each task, thereby offering the following benefits:

- *exploiting task relationships*: scientific data data often consists of related tasks or subtasks that share some underlying structure or dependencies due to physics correlations. MTL allows to leverage these task relationships by jointly learning from multiple predictive tasks. The shared representation learned by the GNN model can capture common patterns and knowledge across tasks, leading to improved generalization and performance.
- *regularization and generalization*: MTL acts as a form of regularization by encouraging the model to learn shared representations across tasks. This regularization can help prevent overfitting. The shared representations learned can capture important features and patterns that are common across tasks, leading to improved generalization performance.

The improvement of an MTL model depends on how strongly the quantities to be predicted are mutually correlated in a particular application. This type of field specific inductive bias has been defined as *knowledge-based*, for which the training of a quantity can benefit from the information contained in the training signal for other quantities. Indeed, other tasks can provide additional domain knowledge that would otherwise be absent if the quantities were trained independently from each other. Therefore, the mutual correlation between quantities is used to treat every single task as an enrichment of the knowledge available to improve the training of other tasks. This mechanism allows for the explicitly incorporation of physics knowledge to improve model quality. Ultimately, MTL allows a direct and automated incorporation of physics knowledge into the model by extracting correlations between multiple quantities, with manual intervention by a domain expert only needed in determining which quantities to use.

Each predicted quantity is associated with a separate loss function and the global objective function minimized during NN training is a linear combination of these individual loss functions. Formally, let T be the total number of physical quantities, or tasks, we want to predict. A single task identified by index i focuses on reconstructing a function $f_i : \mathbb{R}^a \to \mathbb{R}^{b_i}$ defined as

$$\mathbf{y}_i = f_i(\mathbf{x}), \quad i = 1, \dots, T, \tag{15}$$

where $\mathbf{x} \in \mathbb{R}^a$, $\mathbf{y}_i \in \mathbb{R}^{b_i}$. The MTL makes use of the correlation between the quantities $\mathbf{y}_i$, where the functions f_i in (15) are replaced by a single function $\hat{f} : \mathbb{R}^a \to \mathbb{R}^{\sum_i^T b_i}$ that can model all the relations between inputs and outputs as follows:

$$\begin{bmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_T \end{bmatrix} = \hat{f}_{\mathbf{W}_{\text{MTL}}}(\mathbf{x}), \tag{16}$$

where $\mathbf{W}_{\text{MTL}}$ represents the weights to be learned.

The global loss function $\ell_{\text{MTL}} : \mathbb{R}^{N_{\text{MTL}}} \to \mathbb{R}^+$ to be minimized in MTL is a linear combination of the loss functions for the single tasks:

$$\ell_{\text{MTL}}(\mathbf{W}_{\text{MTL}}) = \sum_{i=1}^{T} \alpha_i \|\mathbf{y}_{\text{predict},i} - \mathbf{y}_i\|_2^2, \tag{17}$$

where $\mathbf{y}_{\text{predict},i} = \hat{f}_{\mathbf{W}_{\text{MTL}},i}(\mathbf{x})$ is the vector of predictions for the i^{th} quantity of interest and α_i (for $i = 1, \dots, T$) are the mixing weights for the loss functions associated with each single quantity. The values of the α_i's in Equation (17) are hyperparameters of the surrogate model and thus can be tuned. In this work we assigned an equal weight to each property being predicted because we are equally interested in all of them; however, this definition of the loss function enables one to modify the values of the α_i to purposely favor the training towards one property of interest.

As mentioned above, the multiple quantities in MTL can be interpreted as mutual inductive biases because the error of a single quantity acts as a regularizer with respect to the loss functions of other quantities. In order to clarify why MTL can be seen as a regularizer, let us start with the formulation of a regularized training as a constrained optimization problem:

$$\begin{cases} \underset{\mathbf{w}}{\operatorname{argmin}} \quad \|\mathbf{y}_{\text{predicted}}(\mathbf{w}) - \mathbf{y}\|_2^2 \\ \mathbf{c}(\mathbf{w}) = \mathbf{g} \end{cases}, \tag{18}$$

where $\mathbf{w} \in \mathbb{R}^N$ is the vector of regression coefficients of the model, $\mathbf{y}$ are the target values, $\mathbf{y}_{\text{predict}}(\mathbf{w})$ are the predictions produced by the DL model, $\mathbf{c}(\mathbf{w}) \in \mathbb{R}^c$ and $\mathbf{g} \in \mathbb{R}^c$ are quantities used to express a constraint. The formulation in (18) imposes the constraint $\mathbf{c}(\mathbf{w}) = \mathbf{g}$ in a strong form. The values of $\mathbf{c}(\mathbf{w})$ depend on the model configuration identified by the parameter vector $\mathbf{w}$, whereas the reference values $\mathbf{g}$ are assumed to be given. The constrained optimization problem in (18) can be reformulated so that the constraint is incorporated in the definition of the objective function itself as follows:

$$\underset{\mathbf{w}}{\operatorname{argmin}} \quad \left\{ ||\mathbf{y}_{\text{predicted}}(\mathbf{w}) - \mathbf{y}||_2^2 + \lambda ||\mathbf{c}(\mathbf{w}) - \mathbf{g}||_* \right\}, \tag{19}$$

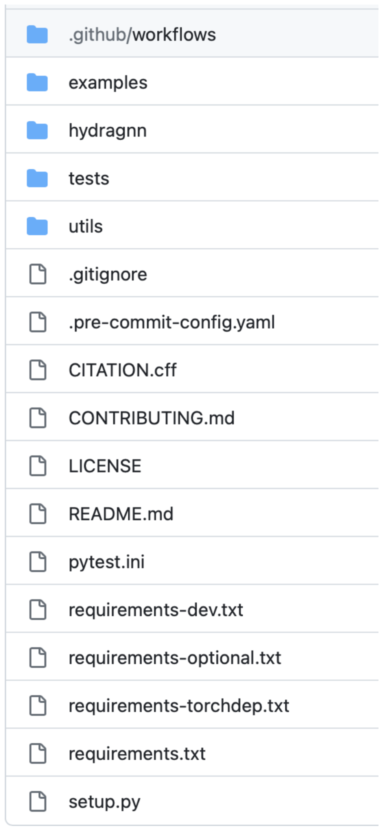
where $||\cdot||_*$ may denote the ℓ^1-norm or ℓ^2-norm as explained below. The objective function to be minimized in (19) interprets the constraint as a penalization term with the penalization multiplier λ. This is a weak formulation of the constraint added to the original objective function. Standard ℓ^1 or ℓ^2 regularizations correspond to choosing $\mathbf{c}(\mathbf{w}) = \mathbf{w}$ and $\mathbf{g} = \mathbf{0}$, and they recast the constraint term in (19) from the strong formulation to the weak formulation using the ℓ^1-norm and the ℓ^2-norm, respectively. ***In MTL,*** $\mathbf{c}(\mathbf{w})$ ***are the predictions of additional target properties whose target values are stored in*** $\mathbf{g}$***, which allows to recast the global objective function used in Equation*** (19) ***as the global loss function for MTL defined in Equation*** (17)**.**

DESCRIPTION OF THE LIBRARY

The HydraGNN software is available open source at the GitHub repository https://github.com/ORNL/HydraGNN.

The top level of the repository (also illustrated in Figure 1) contains the following:

- `requirements-dev.txt`, `requirements.txt`, `requirements-torchdep.txt`, `requirements-optional.txt` : each one of these files contains the list of packages that needs to be installed on the computing environment to utilize basic capabilities, optional capabilities, or advanced capabilities for developers. More details about these files are provided in sub-section 5.
- `setup.py`: script that allows to locally install the `hydragnn` package on the computing environment
- `hydragnn`: a module that contains the core capabilities of HydraGNN, organized in sub-modules
- `tests`: script with Python tests that are executed by the continuous integration (CI) workflow
- `utils`: contains auxiliary utilities that are specific to some applications and that are not needed by the core capabilities of HydraGNN
- `examples`: contains a few examples that describe different capabilities to run HydraGNN



.github/workflows
examples
hydragnn
tests
utils
.gitignore
.pre-commit-config.yaml
CITATION.cff
CONTRIBUTING.md
LICENSE
README.md
pytest.ini
requirements-dev.txt
requirements-optional.txt
requirements-torchdep.txt
requirements.txt
setup.py

Figure 1. Structure of top level of the HydraGNN GitHub repository

5. INSTALLATION REQUIREMENTS

All the installation requirements can be installed using the Python Package Index (PyPI).

5.1 INSTALLATION REQUIREMENTS FOR UTILITIES THAT SUPPORT HYDRAGNN

The following list of python packages need to be installed to support the execution of HydraGNN:

- Command Line Interface Creation Kit (Click) (must be exactly version 8.0.0)
- pickle5
- matplotlib
- PyTorch (must be at least version 1.13)
- Atomic Simulation Environment (ASE)
- tqdm
- tensorboard
- psutil
- sympy

All these requirements are contained in the file `requirements.txt` in the upper level and can be installed all at once by running:

```
'pip install -r requirements.txt'
```

5.2 INSTALLATION REQUIREMENTS SPECIFIC TO HYDRAGNN

training GNN models in HydraGNN requires installing the following packages to perform message passing operations:

- PyTorch Scatter [`'pip install torch-scatter'`]
- PyTorch Sparse [`'pip install torch-sparse'`]
- PyTorch Cluster [`'pip install torch-cluster'`]
- PyG (must be at least version 1.7.2) [`'pip install torch-geometric>=1.7.2'`]

All these requirements are contained in the file `requirements-torchdep.txt` in the upper level and can be installed all at once by running:

```
'pip install -r requirements-torchdep.txt'
```

5.3 OPTIONAL INSTALLATION REQUIREMENTS SPECIFIC TO RUN HYDRAGNN

Optional capabilities can be used after installing the requirements contained in the file:

```
'pip install -r requirements-optional.txt'
```

Currently, `requirements-optional.txt` contains only the installation requirements to allow the use of stochastic optimizers developed by DeepSpeed.

5.3.1 Installation requirements for contributing developers of the HydraGNN library

The installation requirements needed to run sanity checks on newly developed capabilities to expand the HydraGNN libraries are:

```
‘pip install -r requirements-dev.txt’
```

5.4 INSTRUCTIONS TO SET UP THE COMPUTATIONAL ENVIRONMENT IN A `UNIX` OPERATING SYSTEM

In a `UNIX` operating system, there are two options available to ensure that the computational environment is set up to search for the Python scripts within the different modules of HydraGNN:

- set up the `PYTHONPATH` environment variable
- locally install the `hydragnn` package

In the following subsections 5.4.1 and 5.5.1 we provide the details to complete either option.

5.4.1 Set up the environment variable `‘PYTHONPATH’`

Once the `UNIX` shell is located at the top level of the HydraGNN `GitHub` repository, the following command must be executed:

```
‘export PYTHONPATH=$PWD:$PYTHONPATH’
```

As follows, we provide the semantic description of each component of the command above:

- `‘export’` is the command used in `UNIX`-like operating systems (including `Linux`) to set environment variables. In this case, it is used to set the `‘PYTHONPATH’` variable.
- `‘PYTHONPATH=$PWD:$PYTHONPATH’` is the assignment part of the command. It sets the `‘PYTHONPATH’` variable to a new value.
- `‘$PWD:’` This is an environment variable that represents the current working directory, i.e., the directory you are currently in.
- `‘:$PYTHONPATH:’` appends the current value of `‘PYTHONPATH’` to the end of the new value, separated by a colon (:). It’s done to preserve any existing paths in `‘PYTHONPATH’` while adding the current directory (`‘$PWD`’) to it.

5.5 LOCALLY INSTALL THE HYDRAGNN PACKAGE

5.5.1 Using `‘setup.py’`

Alternatively to updating the `‘:$PYTHONPATH:’` variable, the user can also run the following command with the shell located at the top level of the HydraGNN repository:

```
‘python setup.py install’
```

The `‘setup.py’` script contains instructions on how to install the `hydragnn` module, including its dependencies, where to place the package files, and any other installation-specific configurations. This information is specified using the `‘setuptools’` library, which is commonly used for packaging and distribution of Python projects. During the installation process, the code contained in the `hydragnn` module is copied to the appropriate location on the system, typically in the site-packages directory of the Python installation.

5.5.2 Using `‘pip’`

HydraGNN can be installed with the standard package manager for Python, ‘pip‘. Users can use the following command:

```
‘pip install HydraGNN’
```

The above command locates the HydraGNN package within the Python Package Index (PyPI), a public repository, and proceeds to download and install it in the user’s Python environment. For this process, there is no need for users to set the PYTHONPATH. Please note that the examples in the HydraGNN repository are not part of this PIP installation. It will install only the HydraGNN module as a library.

6. DEFINITION OF THE GNN MODEL

The definition of the GNN model is provided in the section `'NeuralNetwork'` in the JSON file. This section is made of subsection that cover the main components that fully characterize the GNN training. The subsection are the following:

- `'Architecture'`: defines the hyperparameters of the GNN architecture
- `'Variables_of_interest'`: defines the input and output features used by the model for supervised learning
- `'training'`: defines the hyperparameters of the stochastic optimizer used for training

6.1 DEFINITION OF THE ARCHITECTURE

The subsection `'Architecture'` contains the following parameters:

- `'model_type'`: categorical parameter that defines the type of GCN layer used for message passing that updates the input features on nodes (and possibly edges) of the graph sample
- `'radius'`: real valued parameter that defines the radius cut-off used to establish the connectivity among nodes of the graph. In situations where the connectivity among nodes of the graph is prescribed a priori, this parameter is not needed
- `'max_neighbours'`: integer parameter that provides the upper bound to the degree of a node (maximum number of neighbours) to retain a tuneable degree of sparsity in the graph connectivity
- `'periodic_boundary_conditions'`: Boolean parameter that allows to wrap the graph structure to establish additional connections between the nodes in proximity of the boundaries through periodic boundary conditions (PBCs). This must be compatible with (1) the size of the graph and (2) the size of the radius cut-off. If (1) is too small and/or (2) is too large, then PBCs cannot be applied and the code returns an error.
- `'hidden_dim'`: integer parameter that establishes the number of neurons (or channels) in the hidden GCN layers
- `'num_conv_layers'`: integer parameter that establishes the number of hidden GCN layers
- `'output_heads'`: categorical variable that can be either set to `'graph'` or `'node'` depending on the type of target output that the head of the GNN architecture has to predict, namely a graph-level output (one target property associated with the entire graph) or node-level output (the target property is predicted for each node within the graph sample). For graph-level outputs, the dimension of the output must be fixed across all the graph samples.
- `'task_weights'`: list of scalar values α_i that are used as multiplying factors in Equation (17) of sub-ection 4.2 for the loss function of each predictive task to define the global loss function of MTL

6.2 DEFINITION OF THE VARIABLES OF INTEREST

The subsection `'Variables_of_interest'` is used to provide indexing information of the nodal input features and nodal output features within the attribute `data.x`, and global output features within the attribute `data.y` for each data sample. The indices of the nodal input features must be provided in the key `‘input_node_features’`, and the output indices must be provided in the key `‘output_index’`. The indexing for `data.x` and the indexing for `data.y` are independent, and both start from 0.

A list of categorical values for the key `‘type’` must also be provided to describe the type of output, which can be set either to `‘graph’` or to `‘node’`.

The list of integers inside `‘output_dim’` must be used to specify the number of dimensions for each nodal and global feature.

The key `‘denormalize_output’` is used to define the Boolean variable to map the predictions back to the original domain during the post-processing. This Boolean variable is used only for specific data loaders described in Section 7.1.1.

The key `‘output_names’` is optional and is used only during the post-processing to assign titles to plots created for diagnostics of the predictive performance of the model on each target output.

6.3 DEFINITION OF THE TRAINING

The `‘training’` section is used to characterize the following parameters:

- the key `‘Checkpoint’` is used to set up a Boolean variable that decides whether checkpoint will be applied or not. If set to `true`, the model is saved on a `pickle` file named `‘saved_model.pk’` inside a log folder which is automatically created at the beginning of each training run.
- `‘checkpoint_warmup’` is used to set up an integer that provides the number of preliminary epochs to wait before activating the checkpointing. This value is used only if `‘Checkpoint’` is set to `true`
- `‘num_epoch’` is used to set up an integer variable that describes the number of epochs for the training
- `‘batch_size’` is used to set up an integer variable that describes the size of the local batch size for each process. The effective global batch size can be calculated by multiplying this integer value by the number of processes used to distribute the training with DDP
- `‘continue’` is used to set a Boolean variable. If set to true, it loads a pre-existing model to start a new training
- `‘startfrom’` is used to provide the path where the pre-existing model can be found and loaded to start a new training. This parameter is used only if the value associated with the key `‘continue’` is set to `true`.
- `‘perc_train’` is used to set a scalar value between 0.0 and 1.0 that prescribes the fraction of the dataset that must be used for the training. The remaining fraction is equally split between validation and testing.
- `‘loss_function_type’` is used to set a string value that prescribes the type of loss function to minimize during the training of the neural network
- `‘Optimizer’` is used to create a subsection of the JSON parsing file that provides hyperparameter values for the training, such as the type of optimizer (e.g, SGD, Adam, AdamW) and the learning rate.

Listing 1. Example of JSON parsing file

```
{
  "Verbosity": {
    "level": 2
  },
  "NeuralNetwork": {
    "Architecture": {
      "model_type": "PNA",
      "edge_features": ["bond_length"],
      "radius": 5.0,
      "hidden_dim": 200,
      "num_conv_layers": 6,
      "output_heads": {
        "graph": {
          "num_sharedlayers": 2,
          "dim_sharedlayers": 200,
          "num_headlayers": 2,
          "dim_headlayers": [1000,1000]
        },
        "node": {
          "num_headlayers": 2,
          "dim_headlayers": [1000,1000],
          "type": "mlp"
        }
      },
      "task_weights": [1.0, 1.0, 1.0, 1.0, 1.0]
    },
    "Variables_of_interest": {
      "input_node_features": [0],
      "output_index": [0, 1, 2, 3, 4],
      "type": ["graph", "node", "node", "node", "node"],
      "output_dim": [1, 3, 1, 1, 1],
      "output_names": ["HLGAP", "forces", "hCHG", "hVDIP", "hRAT"],
      "denormalize_output": false
    },
    "training": {
      "Checkpoint": true,
      "num_epoch": 3,
      "batch_size": 32,
      "continue": 0,
      "startfrom": "existing_model",
      "Optimizer": {
        "learning_rate": 0.001
      }
    }
  }
}
```

6.4 VERBOSITY

The key `‘verbosity’` is used to tune the amount of information included in the output of print statements. A higher degree of verbosity may provide meaningful information in debugging and logging. Adjusting the verbosity level can help developers diagnose issues, track the flow of a program, or provide varying levels of detail in log files. The output is redirected to the output channel used through the `Python` package `log`.

The utilities to print outputs based on the level of verbosity are implemented in the file `HydraGNN/hydragnn/utils/print_utils.py`

The value of verbosity can be set to any integer between 0 and 4. The meaning of each verbosity level is the following:

- `‘level: 0’`: nothing is printed on the screen
- `‘level: 1’`: only the process with rank 0 prints output at the end of each training epoch
- `‘level: 2’`: only the process with rank 0 prints output at each batched gradient update, showing the stage of the training on each epoch using a progression bar
- `‘level: 3’`: every process prints output at the end of each training epoch
- `‘level: 4’`: every process prints output at each batched gradient update, showing the stage of the training on each epoch using a progression bar

7. MODULES

7.1 THE ‘HYDRAGNN’ MODULE

The ‘hydragnn’ module is composed of the following sub-modules:

- ‘preprocess’: contains capabilities to read data from files and generate ‘torch.geometric.data’ objects, as well as rescale input and output features
- ‘postprocess’: contains capabilities to apply rescaling transformations on the output to map the predictions back to the original domain
- ‘models’: contains the list of MPNN layers that implements different message passing
- ‘train’: contains the capabilities to perform the distributed training of a model using DDP
- ‘utils’: contains auxiliary capabilities to enable the execution of functionalities contained in all the other sub-modules of ‘hydragnn’

7.1.1 The ‘preprocess’ sub-module

The ‘preprocess’ sub-module contains functionalities that (i) perform pre-processing transformations on the data samples before feeding them to the HydraGNN model for training and (ii) generate serialized dataloaders for specific formatting of the data. The transformations are contained in the file ‘utils.py’. The serialized data loaders for specific data formats are included in the files:

- ‘raw_dataset_loader.py’
- ‘lsms_raw_dataset_loader.py’ and the atomic descriptors needed for this format are contained in the file ‘dataset_descriptors.py’
- ‘cfg_raw_dataset_loader.py’

7.1.2 The ‘models’ sub-module

The ‘models’ sub-module provides the set of MPNN layers that can be used to perform message passing in the HydraGNN architecture. The MPNN layers are organized according to an object-oriented framework to attain:

- Abstraction. Complex implementation details are hidden and only necessary features of an object are exposed. This simplifies the interface for using the object, making it easier to work with.
- Inheritance. New MPNN layer classes can be created by inheriting properties and behaviors from existing classes. This promotes code reuse and helps establish a hierarchical relationship between classes.
- Reusability. Each MPNN class can be reused in different parts of an application or even in different applications altogether. This reusability can save development time and reduce the likelihood of errors.

Figure 2 details the object-oriented programming structure utilized by each of the MPNN models in HydraGNN. The inheritance of the Base class for each MPNN enables the seamless change between message passing policies. It also enables a direct comparison between MPNNs in a unified framework of the Base class. This comprehensive setup makes HydraGNN a distinct GNN architecture that enables to seamlessly switch between different MPNN layers, thereby allowing them to be treated as hyperparameters.

The `‘Base’` class is used to define the fundamental methods of a message passing layer, and every class that implements a new message passing policy is requested to inherit from this class. Since the core methods implemented in the `‘Base’` class are core capabilities common to all the MPNN architecture, their implementation may not change based on the specific message policy. If a new message passing policy that is implemented by a user requires a new re-implementation of one of these methods, the user can re-implement such method in the child class to override the original implementation.

The choice of the MPNN layer used is specified by the `‘model_type’` key inside the `‘Architecture’` section of the JSON file.

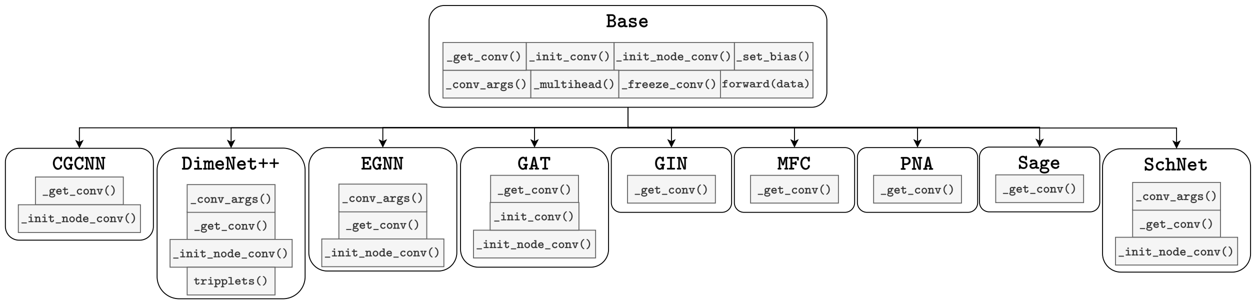


Figure 2. The object oriented programming structure of HydraGNN, in which each class inherits all of the functions from the `Base` class, and over-ride the selected functions that allow for the full functionality of the MPNN.

7.2 THE `‘EXAMPLES’` MODULE

This module provides a set of examples that instantiate HydraGNN models and train them on some open-source datasets. The user is recommended to create a new folder inside this module when a new HydraGNN model must be trained on a new user-specific dataset to support an application of interest. More details about some specific examples will be provided in Sections 14. and 15.9.

7.3 THE `‘TESTS’` MODULE

The Python tests provided in this module aim at testing the different capabilities supported by HydraGNN, and make sure that new changes made to the existing library do not compromise the correct functioning of the other existing capabilities.

As follows, we provide a description of the existing python tests:

- `‘test_config.py’`: checks that the syntax of a JSON parsing file is correct
- `‘test_graphs.py’`: uses the script `‘deterministic_graph_data.py’` to generate a dataset with random graphs (varying number of nodes and randomized node feature), serializes the dataset into files in `pickle` format, and trains different MPNN models. The test fails of at least of the models does not reach the prescribed accuracy. For consecutive runs of this test, the test automatically identifies the presence of pre-existing serialized `pickle` files and pre-loads the data from these files to avoid performing again time-consuming process of regenerating the data from scratch.

- ‘test_examples.py’: runs the two examples available at ‘HydraGNN/examples/md17’ and ‘HydraGNN/examples/qm9’. These examples create a HydraGNN model and train it on the two open-source molecular datasets MD17 and QM9, respectively.
- ‘test_atomicdescriptors.py’: checks that an object from the class ‘atomicdescriptors’ implemented in ‘HydraGNN/hydragnn/utils/atomicdescriptors.py’ can be instantiated to extract relevant atomic descriptors that provide tabulated knowledge about the natural element
- ‘test_loss.py’: checks that all the training loss function supported in HydraGNN can be used to perform the training of a model
- ‘test_model_loadpred.py’: checks that a pre-trained model can be loaded and used for inference
- ‘test_optimizer.py’: checks that all the stochastic optimization numerical schemes supported in HydraGNN can be used to perform the training of a model
- ‘test_periodic_boundary_conditions.py’: checks that periodic boundary conditions are correctly applied to a graph structure that models a crystal with body-centered cubic (BCC). More specifically, the script checks that (1) the graph with PBC applied has more edges than the equivalent version without PBC, (2) every additional edge included as a result of applying PBC has a reasonable length (the length of the edge should be shorter than the radius cut-off)
- ‘test_rotational_invariance.py’: checks that the pre-processing correctly maps two graphs that originally differ only for rotations and translations into the same pre-processed graph

7.4 THE ‘UTILS’ SUB-MODULE

The ‘utils’ sub-module contains auxiliary functionalities that can be categorized as follows:

- data loading and pre-processing (details about these functionalities will be provided in Section 11.):
  - abstractbasedataset.py
  - abstractrawdataset.py
  - adiosdataset.py
  - cfgdataset.py
  - distdataset.py
  - lsmsdataset.py
  - pickledataset.py
  - serializeddataset.py
  - xyzdataset.py
- inclusion of atomic descriptors (details about these functionalities are provided in Section 12.):
  - atomicdescriptors.py

- print details of a model, save trained model on a file, load trained model from a file (details about these functionalities are provided in Section 6.4)
  - `model.py`
- profiling the performance of the training with details of computational time spent in different sections of the execution:
  - `profile.py`
  - `time_utils.py`
  - `tracer.py`

DISTRIBUTED ENVIRONMENT

8. DISTRIBUTED DATA PARALLELISM

Distributed Data Parallelism (DDP) is a technique used in parallel computing and distributed systems to train machine learning models across multiple devices or nodes (e.g., GPUs, CPUs, or different machines) in a distributed and efficient manner. It is commonly used in deep learning to accelerate the training process for large models on extensive datasets.

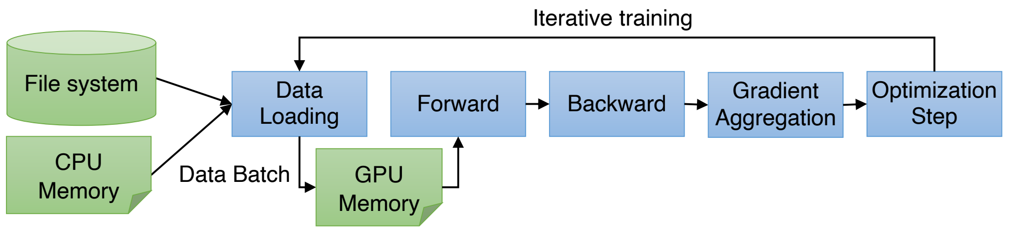


Figure 3. Iterative training with distributed data parallel (DDP).

DDP is one of the parallel methods commonly used in AI/DL to train models using multiple processors or machines. Each process handles only a subset of the data (or graphs) and executes functions in parallel with others. When information needs to be consolidated, such as for aggregated gradient updates, processes communicate data using *all-gather* operations. Message Passing Interface (MPI) (6; 7) and NVIDIA Collective Communications Library (NCCL) (8) are well-known communication backends in multi-node, multi-GPU HPC environments.

DDP provides several benefits, such as efficient use of computing resources by providing fault tolerance, and enabling training on large datasets through concurrent processing of batch data. DDP is widely implemented in many DL frameworks such as PyTorch (9) and TensorFlow (10) and is commonly used for applications such as image recognition, natural language processing, and speech recognition. HydraGNN extends the PyTorch's DDP implementation with enhancements targeting HPC environments.

Generally, DDP for model training involves the following five steps (Fig. 3): i) The dataset is divided into smaller, non-overlapping subsets called "batches". A single batch consisting of *N* samples is loaded into each process (data loading). ii) Each process generates predictions for the samples in the batch using local model (forward). iii) Each process then assesses the loss between these predictions and the actual values and subsequently computes gradients for the model parameters based on the loss (backward). iv) Gradients from all processes are aggregated, producing a consolidated gradient (gradient aggregation). v) Each process updates its local model parameters using the aggregated gradients (optimization).

9. SCALABLE INPUT/OUTPUT DATA MANAGEMENT

Various methods have been developed to reduce overhead in executing these training steps and ensure seamless integration for optimal performance, such as data loading optimizations with latency hiding on HPC systems. An example of these efforts is the use of multi-threaded data loading and overlapping I/O and computation implemented in PyTorch's parallel data loading module (9). NVIDIA's Data Loading Library (DALI) has been specifically developed to offload the data loading from the CPU to the GPU, effectively minimizing overhead and enhancing training efficiency (11).

In DDP, data shuffling is another primary factor for optimal performance. Data sharding with local shuffling (12) is one of the common techniques used with DDP, and consists of splitting the datasaset into chunks that are individually processed by each GPU. Once loaded, shuffling is done within the local chunk. However, it carries two serious performance implications when used with large-scale DL models. First, it is important that the training data stored in partitions on different nodes needs to be shuffled across successive epochs of the DL training to maintain model generality and avoid overfitting (13). Secondly, in situations where the number of GPUs changes after a training session, like during hyper-parameter optimization, the shared dataset must be restructured to align with the updated GPU count, which is time consuming. On the other hand, transferring data between distributed nodes for global shuffling emerges as the optimal solution to enhance the model's generalizability. However, exchanging vast amounts of data during each epoch poses scalability challenges for DL applications. DL training becomes I/O and communication bound and thus incurs significant overhead due to data movement, which serves as the primary motivation for our paper. A comprehensive overview of these approaches is provided in the related work section.

In the version HydraGNN 3.0, the choice of the file formats is not manageable through the `JSON` file. However, once the dataset has been imported using customized dataset classes (please see Section 11. for more details), the process to convert the data into different formats is the same across all examples. Currently, the choice of the file format is passed in input as argument of the parser `argparse`. To familiarize with the use of parsing arguments to switch between different file formats, we refer the user to the example available at https://github.com/ORNL/HydraGNN/tree/main/examples/csce.

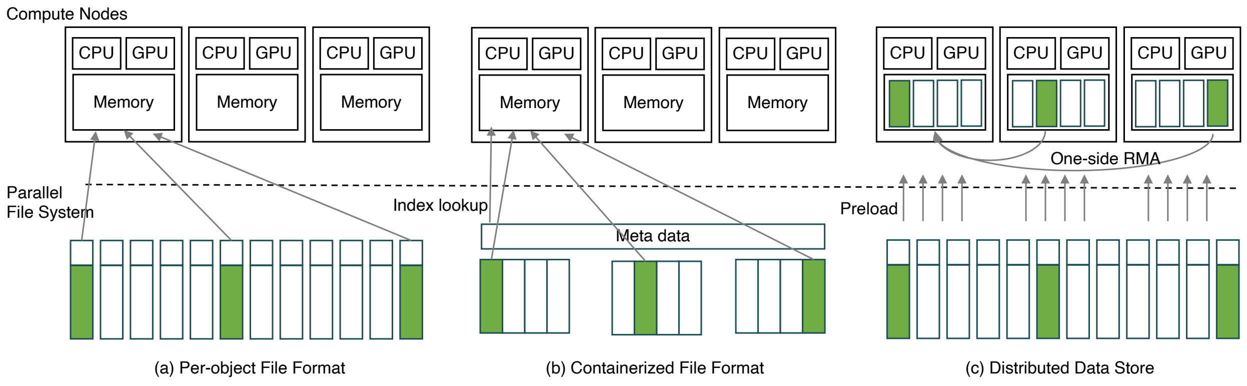


Figure 4. Data storing strategy and its common I/O access patterns in deep learning (a) Per-object file format (b) Containerized file format (c) Distributed data store.

9.1 PER-OBJECT FILE FORMAT

The per-object file format serializes the graph object structures and stores one sample per file. This is one of the simpler approaches for storing graph data, but exerts significant overhead on the underlying parallel file system as the number of samples becomes large. Additionally, as training is performed using tens of thousands of processes, concurrent access to the large number of files from all processes causes a severe I/O bottleneck.

9.2 CONTAINERIZED FILE FORMAT

Fig. 4 (b) shows an alternative method of storing data using containerized file format libraries such as HDF5 (14), ADIOS (15), WebDataset (16) and TFRecord (10). In this approach, multiple samples are stored in a single file which reduces the overhead on the file system. The data management library manages metadata on behalf of the user and provides the API to store and retrieve particular data samples. However, DL training requires reading samples in a random or shuffled order to enable unbiased training. Frequent, random, non-sequential I/O accesses to CFF data can lead to a large number of accesses to the file system, which is very inefficient. Additionally, concurrent I/O access from multiple processes trying to read samples from the same containerized file leads to congestion and high I/O times.

9.3 IN-MEMORY DATA CACHING

Irrespective of the file format used for storing training data, some solutions involve reading the entire dataset into device memory or node-local storage systems such as NVMe devices. However, these options may not be feasible for large datasets, which are the primary concern of this paper. The size of these datasets can surpass the capacity of a single node's memory or its node-local storage. Moreover, several HPC resources (including some of the existing US-DoE supercomputers) are not endowed with NVMe devices yet.

For those HPC architectures that cannot rely on node-local storage, the only state-of-the-art methodologies that enable to scale the DL training on large volumes of data are PFF and CFF techniques. Both methodologies heavily rely on a frequent data movement between file system and volatile node memory, which results in inefficient data reading and causes severe I/O bottlenecks.

9.4 DISTRIBUTED DATA STORE

To address challenges associated with efficient reading of data for large-scale DL/GNN, we have designed DDStore, specifically addresses random, read-oriented, global shuffle operations.

9.4.1 Design

Our design of DDStore is driven by two main performance considerations: *1) Can we minimize access to the file system during the shuffling steps and make in-memory data accessible to other nodes? 2) How do we design fast, efficient, and portable communication mechanisms to provide high-performance shuffling operations for DL?*

These considerations stem from the fact that parallel file systems are shared resources on HPC clusters and are the slowest component of the system. Applications that are I/O bound due to poor performance of the file system typically experience severe challenges with scalability. At large scale, the communication overhead can also become high due to contention in the network, which is also a shared resource on HPC systems.

To address the first objective, DDStore splits data into chunks and stores them in the device memory of compute nodes similarly to data sharding. The dataset is read from the file system and distributed across the device memories of compute nodes. All subsequent accesses to samples in the dataset are made via in-memory read transactions. Secondly, to reduce communication overhead over the network during read operations, DDStore uses data replication to maintain multiple copies of the dataset in memory. This is the novel component of DDStore with respect to all existing state-of-the-art scalable data management methodologies, which allows DDStore to internally partition application processes into groups that are assigned a replica. This hierarchical design prevents situations in which all processes access a single process's memory to obtain the next batch of data which can lead to communication bottlenecks. Finally, DDStore uses low-latency communication functions such as the MPI Remote Memory Access (RMA) (17), known as one-sided communication, to provide fast, non-blocking read operations that are portable across different systems and architectures (Fig. 4c).

We formally define DDStore as

$$DS = (c, w, f)$$

where 'c' is the number of chunks that a dataset is striped into, 'w' represents the store *width* that controls the degree of replication of data, and 'f' represents the communication framework used for transferring data between processes.

Chunking

To effectively utilize the memory available on compute nodes of HPC systems, similarly to data sharding, DDStore uses chunking to split a dataset and distribute the chunks evenly amongst nodes. This avoids performing expensive accesses to the file system to retrieve data during every shuffle operation. The number of chunks 'c' depends on the total size of the dataset 'T' and the width 'w' of DDStore.

$$c = T/w$$

The description of 'w' below will clarify how the chunk size is calculated. By default, for a training using N processes, DDStore stripes the data into N chunks and places one chunk on each process.

Replication

The width 'w' controls the degree of replication of data chunks, which makes DDstore distinguishable from all the other state-of-the-art scalable data management methodologies and reduces the communication bottlenecks due to data shuffling across consecutive epochs of DL training. DDStore divides application processes internally into sub-groups where each sub-group holds a full replica of the dataset. The width represents the cardinality of these groups. To demonstrate how the width influences the replication strategy, let $N = 1024$ and $w = 128$. DDStore creates 8 groups (1024/128) of 128 processes each. Every process group holds a full replica of the dataset. Processes within a group communicate only with each other to exchange data. The number of replicas r is same as the number of process groups, and is represented by

$$r = N/w$$

In our example, there are 8 replicas of the dataset. The processes within a group each hold $T/128$ chunks of the data. By default, $w = N$, which creates a single replica of the dataset striped evenly over all processes. The degree of replication is inversely proportional to the width of the store; as we increase the width, we reduce the number of replicas maintained in the system. As a result, larger width values consume less memory as compared to smaller width values. On the other hand, smaller values for width can help reduce communication bottlenecks as increasing the number of process groups can help distribute communication requests more evenly. The width is configurable so that a user can tune.

Communication

The communication framework 'f' provides the data plane and control mechanism for fetching data from the memory of other nodes. For the data plane, we considered several state-of-the-art options such as 1) ZeroMQ (18), an asynchronous message passing library for distributed applications, and 2) MPI's blocking, non-blocking, and one-sided communication functions. For the control plane, design options included 1) developing a message-broker framework in which additional message brokers on each node facilitate data exchange amongst themselves during the shuffle operation, and 2) fully de-coupled and asynchronous communication without dedicated message brokers. To provide a scalable, high-performant, and portable communication framework over our use of MPI for parallelization, we selected MPI's RMA one-sided library functions for DDStore. The communication layer in the DDStore Architecture discusses our use of the MPI RMA routines.

9.4.2 Architecture

We describe the main components of DDStore in details that allow to efficiently perform frequent random shuffling of data distributed across nodes to reduce communication bottlenecks.

DDStore is composed of four components: 1) a data preloader, which reads data in various formats from a parallel file system and loads it into the memory of deep learning applications, 2) a data registry that manages the index of data chunks, 3) the data loader that reads the next batch of data from other processes, and 4) the communication layer that leverages one-sided RMA to asynchronously fetch data from remote processes. We delve into the details of each component below.

Data Preloader

The data preloader loads a dataset from the file system and initializes the distributed store. Data may be stored in per-sample or containerized file formats. DDStore provides plugins for reading different data formats. Data is split into chunks depending on the user-provided width or the number of replicas.

Data Registry

After data is loaded into memory, each process registers its chunks. DDStore maintains a global registry of chunks on each process. In order to read another data chunk from a remote process, a process consults its registry to determine the location of the data item and issues a read operation to fetch the data object.

Data Loader

The data loader performs the main task of reading data in-memory during the data loading operation (See Fig. 3).

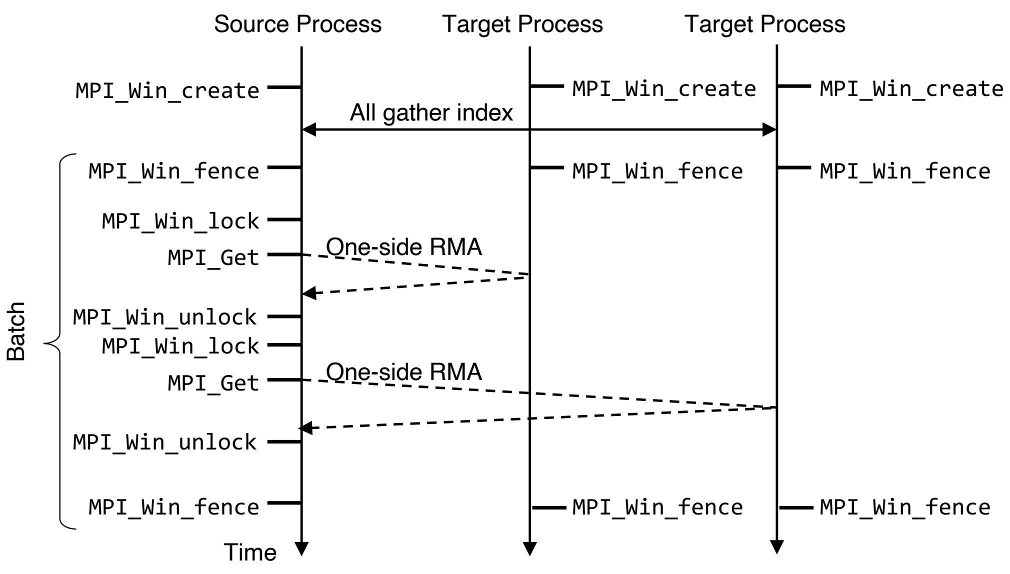


Figure 5. Example walk-through of DDStore's RMA operations for a batch size of 2.

Data samples in a batch can be a random distribution of samples spread across processes. These form the set of data that are input to the model during the forward step. To get the next set of data items not in the process's local memory, a process generates a list of data items to be retrieved from remote locations and passes it to DDStore. DDStore performs a lookup for the data items in its internal sub-group of processes that the calling process belongs to. It then starts initiating one-sided `MPI_Get` operations to fetch data from remote processes. DDStore returns control to the application when all requested data items have been read.

One-sided RMA Communication Layer

The communication layer performs the actual MPI RMA registration and read operations for fetching remote data. During the registration step, each process registers its memory region containing the data by calling MPI's `MPI_Win_create` function. In contrast to the tightly coupled MPI send-receive paradigm used in two-sided MPI communications, MPI's RMA minimizes the target process's involvement, helping to increase throughput. However, it still requires a non-blocking lock-unlock synchronization (19) to avoid data inconsistency arising from contention between multiple processes. Among MPI RMA's multiple synchronization mechanisms (19), we employ a read lock (`MPI_Win_lock` with `MPI_LOCK_SHARED`) and fence synchronization (`MPI_Win_fence`) as a lightweight set of contention-avoiding methods. The sequence of operations involved in MPI RMA are shown in Figure 5.

We have integrated DDStore into PyTorch's data library by creating a set of subclasses that inherit from `torch.utils.data.Dataset`. This enables PyTorch's data loader `torch.utils.data.DataLoader` to directly interact with our customized dataset. Our dataset classes extend basic data readers for commonly used data formats and integrate them with DDStore for preloading, as well as MPI RMA registration and fetching.

10. PROFILING

10.1 TIME CLASS

The Python script “`hydragnn\utils\time_utils.py`” provides an implementation of a “`Timer`” class that measures the time spe

nt in different sections of the code. Some timers to measure the performance of HydraGNN during the training are already implemented. The `Timer` allows the user to implement new timers with a customized name to measure additional section of the code that are of interest to the user. The timers instantiated with the `Timer` class collects the minimum, the maximum, and the average time across all processes when the code is executed in a distributed computing manner. The amount of printouts used to describe the performance of HydraGNN across multiple processes can be tuned using the verbosity level described in Section 6.4.

10.2 PROFILER CLASS

The Python script “`hydragnn\utils\profile.py`” implements a “`Profiler`” class that inherits from “`torch.profiler.profile`”. Its use can be activate by including the “`Profile`” sub-section in the “`NeuralNetwork`” section of the JSON file as shown in the code snippet 2.

Listing 2. Example of JSON parsing file

```
"NeuralNetwork": {
        "Profile": {"enable": 1},
        "Architecture": {
            ...
        },
        "Variables_of_interest": {
            ...
        },
        "Training": {
            ...
        }
    },
```

For more details about this type of profilers, we refer the user to https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

DATA CLASSES

The dataset classes are separated in two categories:

- dataset classes that read data from raw input files and converts it into standardinzed formats accepted by HydraGNN
- dataset classes that load pre-standardized data and make it ready for the HydraGNN training

The HydraGNN routines for both categories of dataset classes are structured according to an object oriented programming framework. An illustration of the object-oriented programming framework for data reading and loading routines is provided in Figure 6. Dataset classes for reading of data from raw files are colored in orange, and dataset classes for loading pre-standardized data are colored in green.

Th class inheritance for both categories of dataset classes stem from the same common parenting abstract class called `AbstractBaseDataset`. This class provides the backbone structure of methods for all the derived classes. The user can customize the level if inheritance layers before fully specifying the complete set of methods for the data class. For example, many methods for the classes `LSMSDataset` and `CFGDataset` are the the same, and we thus conveniently introduced and intermediate layer of abstract class called `AbstractRawDataset` that partially specifies the methods that are independent of the specifics of each raw input.

HydraGNN 3.0 provides two different methodologies to convert raw files into formats that can be readily passed to the HydraGNN model for training. The first methodology automates the read of data from specific raw data formats and creates `pickle` files with `.pkl` format. This methodology uses the class `SimplePickleWriter` to create `pickle` files and uses the class `SimplePickleDataset` to load the pre-standardized data. The second methodology automates the read of data from specific raw data formats, and creates `ADIOS` binary files. This methodology iuses the class `AdiosWriter` to create an `ADIOS` binary file and uses the class `AdiosDataset` to load the pre-standardized data. This is also illustrated in Figure 7.

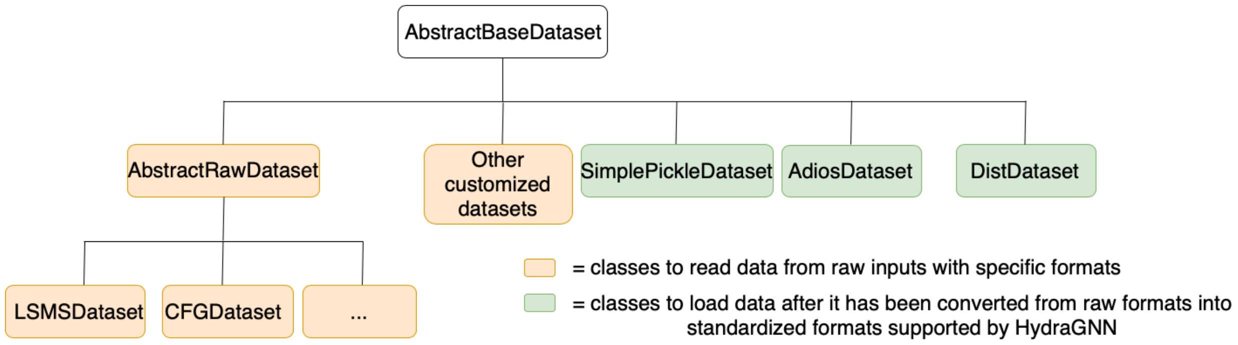

AbstractBaseDataset
AbstractRawDataset
Other customized datasets
SimplePickleDataset
AdiosDataset
DistDataset
LSMSDataset
CFGDataset
...
= classes to read data from raw inputs with specific formats
= classes to load data after it has been converted from raw formats into standardized formats supported by HydraGNN

Figure 6. Object oriented organization of data classes to load data from raw input files and after they have already been standardized into readable formats for HydraGNN.

11. OBJECT ORIENTED PROGRAMMING ORGANIZATION OF DATASET CLASSES

HydraGNN provides capabilities to load customized data from files with different formats. Once loaded, the file is converted into `pickle` or `adios` formats that are easier to manage on HPC resources. Once the

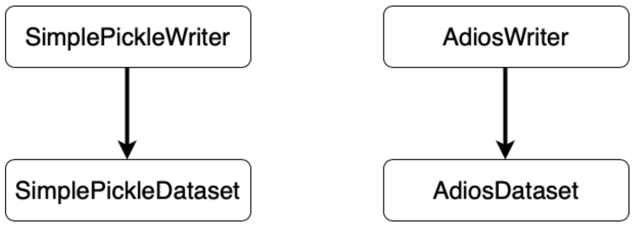


Figure 7. Sequence of data classes to use to convert data in pickle or adios format, and load it in the respective pre-standardized formats.

datasets are converted either in `pickle` or `adios` formats, data loader capabilities are used to feed the data into the HydraGNN model for training.

11.1 ABSTRACTBASEDATASET

The `‘AbstractBaseDataset’` class defines the signature of fundamental abstract methods that all subclasses must implement to provide data import functionalities for specific data formats and to adopt specific I/O data management techniques. The fundamental abstract methods whose signature is provided inside this class are the following:

- `‘get’` method to fetch data samples from the dataset and feed it to HydraGNN for training
- `‘len’` method to retrieve the number of samples contained in the dataset
- `‘apply’` method to execute a generic transformation on every data sample contained in the dataset
- `‘__len__’`
- `‘__getitem__’`
- `‘__iter__’`

11.2 ABSTRACTRAWDATASET

The `‘AbstractRawDataset’` class implements some methods that are common to all the datasets independently of their specific format and that can be used for data pre-processing. These methods are:

- `‘__load_raw_data’` allows to iterate over the raw files that describe every sample of a dataset, and for each data sample calls the function `‘transform_input_to_data_object_base’` to create import the data samples into a `torch.geometric.dataset.data` object. This method assumes that the information for each data sample is either stored in a separate file within one global directory called `‘dataset’`, or stored in a sub-directory within `‘dataset’`
- `‘__normalize_dataset’` method normalizes the output features within the range [0,1] to help stabilize the training of HydraGNN
- `‘__scale_features_by_num_nodes’` method rescales the value of the output features (both global and nodal) based on the number of nodes in the graph
- `‘__build_edge’` method establishes the connectivity among nodes of the graph and builds edge features

The implementation of the method `'transform_input_to_data_object_base'` depends on the specific data format, and is thus delegated to any child class that inherits from `'AbstractRawDataset'`.

11.3 ADIOSDATASET

The `'AdiosDataset'` class inherits from the `'AbstractBaseDataset'` class. It is a class to read information written in a Adios file and provide graph objects of the class `'torch_geometric.data.Data'`. This class is designed to provide graph objects in multiple different ways, such as caching, shared memory, and DDStore.

11.4 CFGDATASET

The `'CFGDataset'` class inherits from the `'AbstractRawDataset'` class and implements the method `'__transform_ASE_object_to_data_object'` to import data saved in files with `.cfg` format and converts it into objects of the class `'torch_geometric.data.Data'`.

11.5 DISTDATASET

This class allows to distribute datasets according to the DDStore technique. The `'DistDataset'` class inherits from the `'AbstractBaseDataset'` class, and implements the `'len'` and `'get'` methods.

11.6 LSMSDATASET

The `'LSMSDataset'` class inherits from the `'AbstractRawDataset'` class and implements the method `'__transform_LSMS_input_to_data_object_base'` to import data from files generated as output by the quantum mechanics code LSMS-3 and converts it into objects of the class `'torch_geometric.data.Data'`.

11.6.1 LSMS output - `info_evec_out`

LSMS generates a file called `info_evec_out`. The first line of this file has three columns and lists:

1. total energy
2. band energy
3. Fermi energy

Per Atom Lines

The rest of `info_evec_out`, beginning with the second line, contains one line for each site in the system. Each line has 18 columns with the following information:

- Atomic number
- Site index (starting with 0 for the first site)
- x component of the site coordinate
- y component of the site coordinate

- z component of the site coordinate
- electron charge associated with this site (note that the
- electron charge is counted here with a positive sign)
- magnetic moment associated with this site in μB
- x component of the magnetic moment direction
- y component of the magnetic moment direction
- z component of the magnetic moment direction
- site dependent magnetic moment direction mixing parameter (Currently always -1)
- x component of the magnetic constraining field
- y component of the magnetic constraining field
- z component of the magnetic constraining field shift of spin-up and spin-down potential
- x component of the new magnetic moment direction
- y component of the new magnetic moment direction
- z component of the new magnetic moment direction

Local Atom Data File - `localAtomData`

If the input files for LSMS specifies a filename for `localAtomData` this file is written in a format similar to `info_evec_out`, but with somewhat different contents.

Line 1

The first line has four columns and lists

1. total energy
2. band energy
3. Fermi energy
4. electrostatic energy

Per Atom Lines

The rest of `info_evec_out`, beginning with the second line, contains one line for each site in the system. Each line has 16 columns with the following information:

- Atomic number
- Site index (starting with 0 for the first site)
- x component of the site coordinate
- y component of the site coordinate

- z component of the site coordinate
- electron charge associated with this site (note that the
- electron charge is counted here with a positive sign)
- magnetic moment associated with this site in μB
- x component of the magnetic moment direction
- y component of the magnetic moment direction
- z component of the magnetic moment direction
- x component of the magnetic constraining field
- y component of the magnetic constraining field
- z component of the magnetic constraining field
- shift of spin-up and spin-down potential
- local site volume (in units of a_0^3)
- local site energy

Currently, the `‘LSMSDataset’` class follows the format explained in Section 11.6.1. However, if needed, it can be easily customized to follow also the format explained in Section 11.6.1.

12. ATOMIC AND EDGE DESCRIPTORS

The atomic and edge descriptors help discriminate different types of nodes and this in turn could improve the expressivity of the HydraGNN model. The atomic descriptors are included as columns in the two-dimensional tensor stores in `data.x` whose number of rows is equal to the number of nodes in the graph. The edge descriptors are added as rows to the tensor `data.edge_attr`, which contains the descriptors to discriminate every edge that connects two nodes of the graph.

The atomic descriptors available in HydraGNN are provided through the `mendeleev` library (20). The capabilities to extract atomic descriptors are available within the following file `HydraGNN/hydragnn/utils/atomicdescriptors.py`

The list of atomic descriptors are:

- atomic number
- group
- period
- covalent radius
- electron affinity
- atomic volume
- atomic weight
- electronegativity
- valence electrons
- ion energies

Each property can also be converted into a one-hot representation. For integer-valued quantities, the length of the one-hot encoding vector is dictated by the maximum attainable value. For real-valued quantities, the conversion into one-hot encoding is performed by splitting the range of values into discrete bins, and use this partition for a one-hot encoding conversion. In this case, the number of bins used to discretize the range is arbitrarily chosen by the user.

In order to extract tabulated information about each atom element and use it as descriptors, first an object of the `atomicdescriptors` class must be instantiated. An example of this is provided in Listing 3.

The first argument of the `atomicdescriptors` class must provide the name of the `JSON` file from which the atomic descriptors can be extracted under the request of the user. If a `JSON` file with the specified name already exists, the atomic descriptors are loaded from the existing file. If instead the `JSON` file does not exist, it is generated within the call to the constructor of the `atomicdescriptors` class.

Listing 3. Example of JSON parsing file

```
atomicdescriptor = atomicdescriptors(
        "./embedding.json", overwritten=True, element_types=["C", "H", "S"]
    )
    atomicdescriptor_onehot = atomicdescriptors(
        "./embedding_onehot.json",
        overwritten=True,
        element_types=["C", "H", "S"],
        one_hot=True,
    )
```

CONSTRUCTION OF THE HYDRAGNN ARCHITECTURE

13. MPNN LAYERS AND FC LAYERS

The HydraGNN architecture is made of the following main components:

- MPNN layers
- optional FC layers shared across all heads for all predictive tasks
- FC layers specific to each head for a specific predictive task

13.1 MPNN LAYERS

The MPNN layers are located at the beginning of the architecture, and are shared across all the predictive tasks. Their goal is to learn interactions between nodes to create a feature context that can contribute to all the downstream predictive tasks.

13.2 FC LAYERS

After the stack of message passing layers, the architecture bifurcates to create subsequent stacks of FC layers that are specific to each target property. In Listing 4, we provide an example of the section `Architecture` of the `JSON` file that defines the construction of a multi-headed HydraGNN architecture. The list of parameters to define inside this section are:

- `"model_type"`: type of MPNN layers
- `"radius"`: radius cut-off used to establish the graph connectivity
- `"max_neighbours"`: maximum number of connections that a node can have with its neighbors
- `"hidden_dim"`: number of channels for the MPNN layers
- `"num_conv_layers"`: number of MPNN layers
- `"output_heads"`: type of head, can either be `"graph"` or `"node"`
  - `"num_sharedlayers"`: number of FC layers shared across all heads of the architecture
  - `"num_headlayers"`: number of FC layers specific to one head of the architecture
- `"task_weights"`: list of scalars that provide the scaling coefficients used to multiply the loss functions of each head in order to define the global loss function minimized during the MTL training

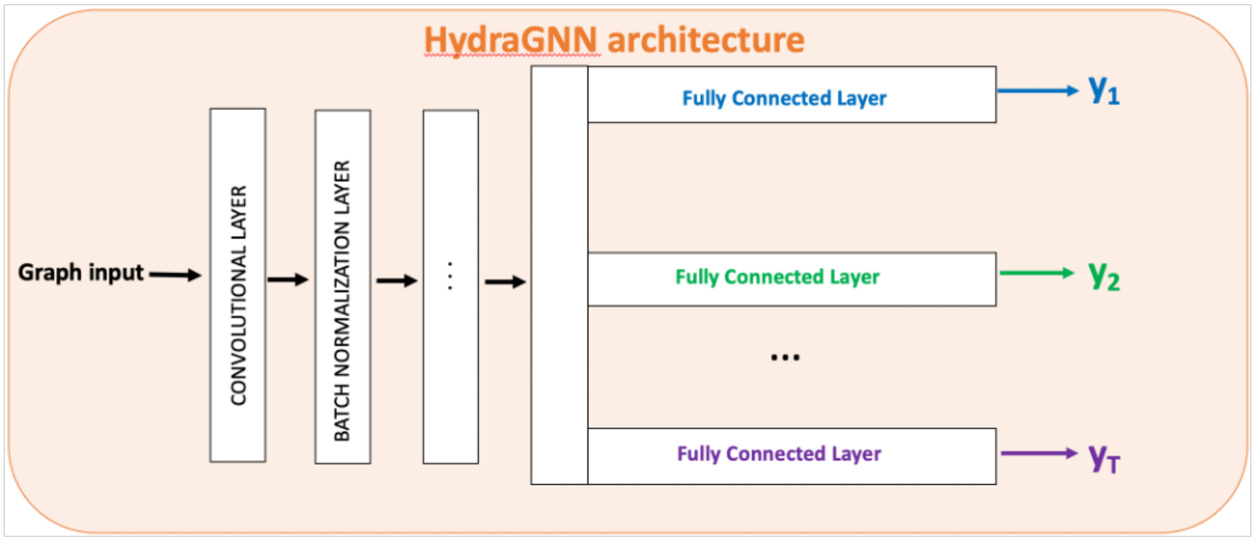


Figure 8. HydraGNN architecture

Listing 4. Example of JSON parsing file

```
"NeuralNetwork": {
        "Profile": {"enable": 1},
        "Architecture": {
            "model_type": "PNA",
            "radius": 7,
            "max_neighbours": 100,
            "hidden_dim": 5,
            "num_conv_layers": 6,
            "output_heads": {
                "graph":{
                    "num_sharedlayers": 2,
                    "dim_sharedlayers": 5,
                    "num_headlayers": 2,
                    "dim_headlayers": [50,25]
                },
                "node": {
                    "num_headlayers": 2,
                    "dim_headlayers": [50,25],
                    "type": "mlp"
                }
            },
            "task_weights": [1.0, 1.0, 1.0]
        },
        "Variables_of_interest": {
            ...
        },
        "Training": {
            ...
        }
    },
```

COMPLETE EXAMPLES

14. QM9

QM9 is an open-source quantum chemistry dataset that consists of organic molecules with up to nine heavy atoms (C, O, N, and F). The original dataset (21) has about 134K molecules and 15 properties for each. The one provided by PyTorch Geometric (PyG), i.e., `torch_geometric.datasets.qm9`, is a processed version (22) containing about 130K molecules and 19 properties for each. All the 19 properties are graph-level properties. To demonstrate the nodal-level regression, we introduce the partial charge from (21) as the 20th target (as shown in Table 1). In the following, we are going to show two HydraGNN examples for regression tasks of the 20 properties in Table 1. The script and configuration files for the QM9 examples can be found at https://github.com/ORNL/HydraGNN/tree/LoG2023_tutorial/examples/qm9/.

Table 1. Properties of QM9 with with 20 properties[*]

Target	Property	Description	Unit
0	μ	Dipole moment	D
1	α	Isotropic polarizability	a_0^3
2	ϵ_{HOMO}	Highest occupied molecular orbital energy	eV
3	ϵ_{LUMO}	Lowest unoccupied molecular orbital energy	eV
4	$\Delta\epsilon$	Gap between ϵ_{HOMO} and ϵ_{LUMO}	eV
5	$\langle R^2 \rangle$	Electronic spatial extent	a_0^2
6	ZPVE	Zero point vibrational energy	eV
7	U_0	Internal energy at 0K	V
8	U	Internal energy at 298.15K	eV
9	H	Enthalpy at 298.15K	eV
10	G	Free energy at 298.15K	eV
11	c_{V}	Heat capavity at 298.15K	$\frac{\mathrm{cal}}{\mathrm{mol\ K}}$
12	U_0^{ATOM}	Atomization energy at 0 K	eV
13	U^{ATOM}	Atomization energy at 298.15 K	eV
14	H^{ATOM}	Atomization enthalpy at 298.15 K	eV
15	G^{ATOM}	Atomization free energy at 298.15 K	eV
16	A	Rotational constant	GHz
17	B	Rotational constant	GHz
18	C	Rotational constant	GHz
19	E	Mulliken partial charge	e

[*]Table is modified from (23). Targets 0-18 are from the original `torch_geometric.datasets.qm9`, and target 19 is from (21).

14.1 GENERAL HYDRAGNN WORKFLOW

The general procedure to train a HydraGNN model is as follows:

1. Load the input configuration file (into the dictionary variable `config`): the file should have the information for your

`Dataset` and the architecture and training setup for your `NeuralNetwork` model. Readers can find a complete example in section 15.. The `config` will be used in the next steps.

2. Prepare the graph datasets for model training/validation/testing: the datasets can be customized (e.g., LSMS in 11.6) or built-in (e.g., QM9 in this section).
3. Create the dataloaders: the dataloaders are created by using **torch_geometric.loader.DataLoader** with the graph datasets for convenient mini-batch sampling with a user-specified batch size.
4. Create the model: the model is created by calling the function `hydragnn.models.create_model_config` with `NeuralNetwork` information in `config`.
5. Specify optimization method: an optimizer is defined with the Optimizer type and the learning rate specified in `config["NeuralNetwork"]["Training"]["Optimizer"]`.
6. Train adn test the model: the model is trained in the function `hydragnn.train.train_validate_test` with the dataloaders and Optimizer created in the previous steps. The trained model will be saved along with the loss history in `./logs/` directory. A few results like the parity plot and histogram of the prediction error of the trained model on the test set will be saved as well if `config["Visualization"]["create_plots"]` is specified to be `True`.

In the next, we will show a few HydraGNN application examples with the critical information in these steps presented. The complete script and input files are available in https://github.com/ORNL/HydraGNN/tree/LoG2023_tutorial/examples/.

14.2 SINGLE-TASKING EXAMPLE

We set up HydraGNN to predict the free energy, G. The example script is `qm9.py` and the input configuration file is `qm9.json`. The dataset is loaded from the built-in `torch_geometric.datasets.QM9` as shown in Listing 5. The `qm9_pre_transform` function in is used to select G (scaled by the number of atoms) as the regression target and the atomic number `data.z` as the node feature.

Listing 5. `qm9_pre_transform` function used to select regression targets

```python
# Update each sample prior to loading.
def qm9_pre_transform(data):
    # Set descriptor as element type.
    data.x = data.z.float().view(-1, 1)
    # Only predict free energy
    data.y = data.y[:, 10] / len(data.x)
    return data

dataset = torch_geometric.datasets.QM9(
    root="dataset/qm9", pre_transform=qm9_pre_transform
)
```

As for the model architecture, we use 5 graph convolution layers with `PNA` message passing method at a hidden dimension of 5. These PNA convolution layers are followed by 4 fully connected hidden layers with 50, 50, 50, and 25 neurons, respectively, and the output layer with 1 neuron for free energy. The listing below shows the model architecture setting in the input file: `qm9.json`.

```json
"Architecture": {
    "model_type": "PNA",
    "hidden_dim": 5,
    "num_conv_layers": 6,
    "output_heads": {
        "graph":{
            "num_sharedlayers": 2,
            "dim_sharedlayers": 50,
            "num_headlayers": 2,
            "dim_headlayers": [50,25]
        }
    },
    "task_weights": [1.0]
},
```

We use the `AdamW` optimizer with a learning rate at 1×10^{-3}. The model is trained on 70% (i.e., `perc_train`) of the total samples for 200 epochs with a batch size of 64 and the mean square error (`MSE`) loss function, as shown in the following listing.

```
 "Training": {
    "num_epoch": 200,
    "perc_train": 0.7,
    "loss_function_type": "mse",
    "batch_size": 64,
    "Optimizer": {
        "type": "AdamW",
        "learning_rate": 1e-3
    }
}
```

By default, the remaining data samples (excluding the ones used for training) are split equally for validation and testing. The splitting is implemented in the function `hydragnn.preprocess.split_dataset` and the returned three datasets (train/val/test) are then used to build three data loaders used for model training, validation, and testing.

```
train, val, test = hydragnn.preprocess.split_dataset(
    dataset, config["NeuralNetwork"]["Training"]["perc_train"], False
)
(train_loader, val_loader, test_loader,) = hydragnn.preprocess.
   create_dataloaders(
    train, val, test, config["NeuralNetwork"]["Training"]["batch_size"]
)
```

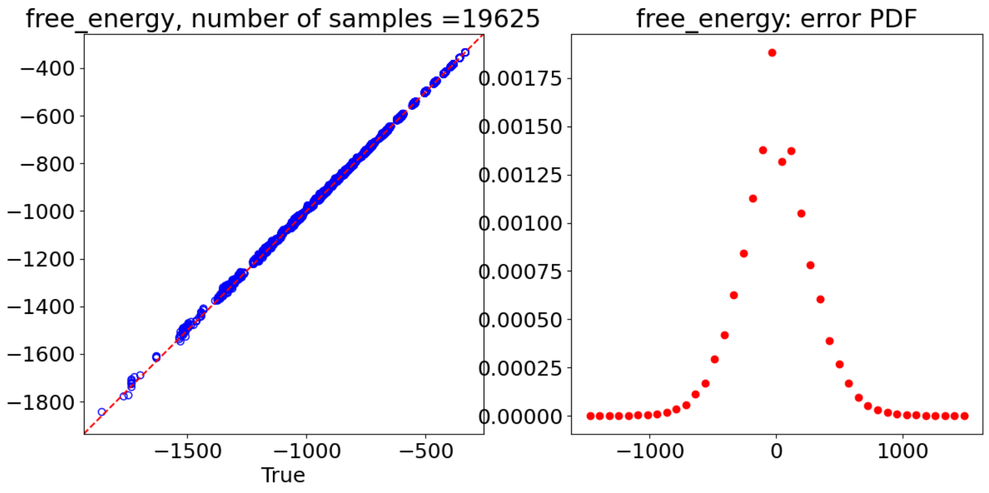


Figure 9. HydraGNN predicted free energy for molecules in the test set: scatter plot of predicted VS. true values (left) and probability density function (PDF) of prediction errors.

Figure 9 shows the scatter plot of the predicted values by the trained model against the true values on the left subplot and the probability density function (PDF) of prediction errors on the right subplot for the test set.

14.3 MULTITASKING EXAMPLE

For the multitasking example, we set up HydraGNN for the prediction of all 20 properties in Table 1. The script and the json input files are `qm9_custom20.py` and `qm9_all20.json`, respectively. The dataset is the customized `QM9_custom` (in file `qm9_custom20.py`) from `torch_geometric.datasets.QM9` with the partial charge (21) added as the 20*th* and the only nodal property.

For the graph objects in `QM9_custom`, the 12-dimension node features, which are atom type (i.e., "atomH", "atomC", "atomN", "atomO", "atomF"), atomic number, aromatic [or not], hybridization types (i.e., sp, sp^2, or sp^3), Hprop (i.e., number of hydrogen neighbors are used as features for each node), and the partial charge. The first 11 node features are used in the training of HydraGNN (`data.x`), and the last node feature–the partial charge–and the 19 graph properties are the 20 regression targets (`data.y`). The specification in `qm9_all20.json` is shown as follows,

```
"Variables_of_interest": {
           "input_node_features": [0,1,2,3,4,5,6,7,8,9,10],
           "output_index":
              [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,11],
           "type": ["graph","graph","graph","graph","graph","graph","graph","
              graph","graph","graph","graph","graph","graph","graph","graph"
              ,"graph","graph","graph","graph","node"]
       },
}
```

, where the `input_node_features` shows the indices of node features used in training, and `output_index` together with `type` show the regression targets, i.e., 19 graph property and the 12th node property (i.e., partial charge).

We use 6 `PNA` graph convolution layers at a hidden dimension of 30. These PNA convolution layers are followed by 20 MLP heads for the 20 regression tasks. Different weights, i.e., `task_weights`, can be used to prioritize tasks (the higher the more important) and by default equal weights are used for each. For the 19 graph-level properties, the MLP heads consist of two shared layers with 50 and 50 neurons and then two private layers with 50 and 25 neurons each. As for the partial charge, a 3-layer MLP head with 50, 50, and 25 neurons is used.

```
"Architecture": {
           "model_type": "PNA",
           "hidden_dim": 30,
           "num_conv_layers": 6,
           "output_heads": {
               "graph":{
                   "num_sharedlayers": 2,
                   "dim_sharedlayers": 50,
                   "num_headlayers": 2,
                   "dim_headlayers": [50,25]
               },
               "node": {
                   "num_headlayers": 3,
                   "dim_headlayers": [50,50,25],
                   "type": "mlp"
               }
           },
           "task_weights": [1.0,1.0,1.0,1.0,1.0,1.0,1.0,
           1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0]
       },
```

We use the `AdamW` optimizer with a learning rate at 1×10^{-3}. The model is trained on 70% (i.e., `perc_train`) of the total samples for 200 epochs with a batch size of 64 and the MSE loss function, as shown in the following listing. Figure 10 shows the scatter plot of the predicted values by the trained model against the true values for the test set.

```
"Training": {
            "num_epoch": 150,
            "perc_train": 0.7,
            "loss_function_type": "mse",
            "batch_size": 64,
            "Optimizer": {
                "type": "AdamW",
                "learning_rate": 1e-3
            }
```

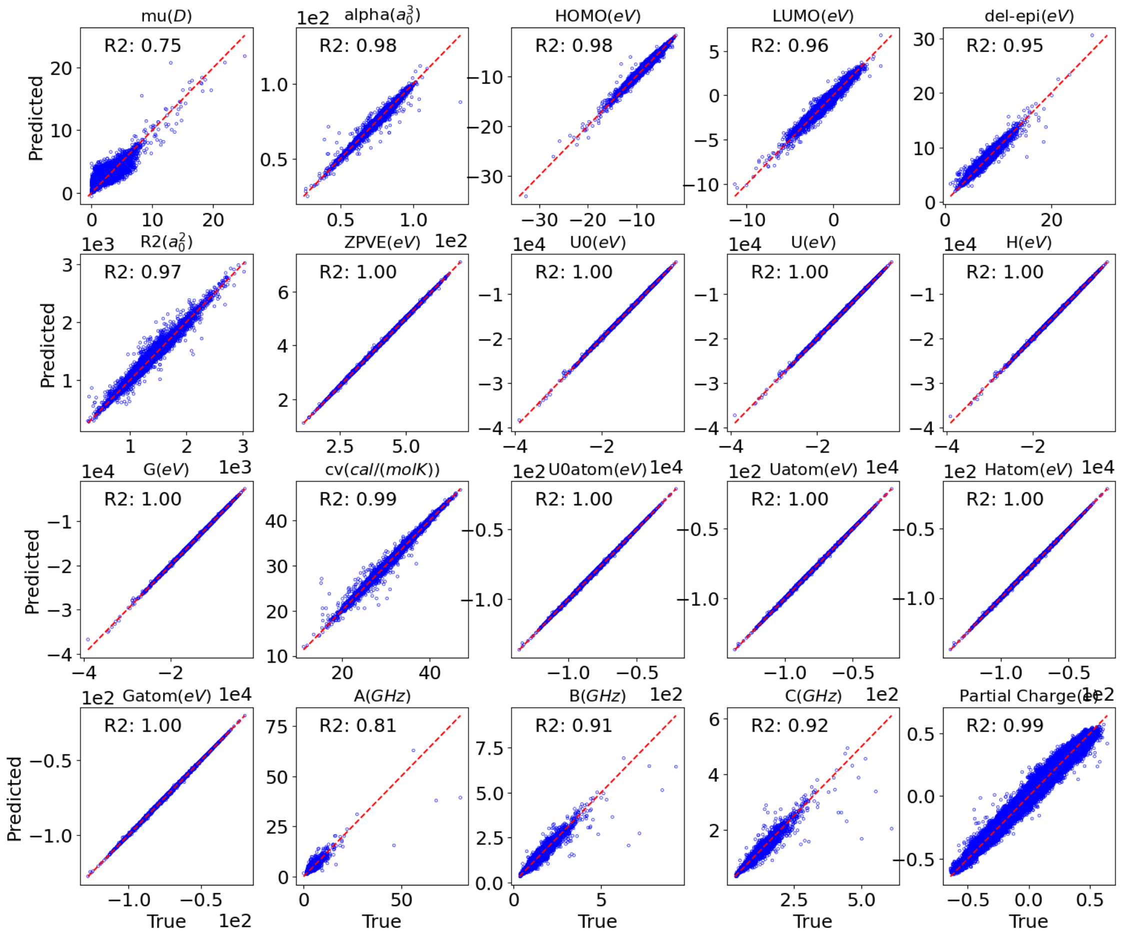


Figure 10. HydraGNN predicted values VS. true values for the 20 properties in Table 1 for the test set.

15. FEPT BINARY ALLOY WITH 32 ATOMS - LSMS-3 DATA

This open-source dataset describes a solid solution binary iron-platinum (FePt) alloy (24), where two constituent elements are randomly placed on an underlying crystal lattice. The dataset provides the total enthalpy, atomic charge transfer, and atomic magnetic moment. Each atomic sample has a body centered tetragonal (BCT) structure with a $2 \times 2 \times 4$ supercell. The dataset was computed with LSMS-3 (25), a locally self-consistent multiple scattering (LSMS) DFT application (26; 27). The dataset was created with fixed volume in order to isolate the effects of graph interactions and graph positions for models such as GCNN. This produces non-equilibrium alloy samples, with non-zero pressure and positive mixing enthalpy, shown as a function of composition in Figure 11. Specific information about the output format of LSMS-3 is provided in Section 11.6.

15.1 LOCATION OF THE EXAMPLE

The example is available on the branch https://github.com/ORNL/HydraGNN/tree/LoG2023_tutorial of the GitHub repository at the path `'examples/lsms'`. This folder contains five files:

- `compute_enthalpy.py`
- `lsms.py`
- `lsms_single_tasking.json`
- `lsms_multi_tasking.json`
- `inference.py`

The script `compute_enthalpy.py` computes the mixing enthalpy from the total energy for each atomic structure in the dataset. Moreover, since the original dataset is imbalanced in terms of number of atomic structures across chemical compositions, the script `compute_enthalpy.py` performs an histogram cutoff on the number of atomic structures per chemical compositions to ensure balanced representation of different chemical compositions across the dataset.

The JSON file `lsms_single_tasking.json` defines the hyperparameters to create a single-tasking HydraGNN model for predictions of the mixing enthalpy.

The JSON file `lsms_multi_tasking.json` defines the hyperparameters to create a multi-tasking HydraGNN model for simultaneous predictions of the mixing enthalpy, atomic charge transfer, and atomic magnetic moment.

The switching between single tasking and multi-tasking for the HydraGNN can be performed by specifying the name of the correct JSON file as

The script `lsms.py` loads the data from raw files and converts it into data samples, each with `'torch.geometric.data'` format, and trains the HydraGNN model using the hyperparameter configuration specified in the JSON file.

The script `inference.py` analyses the performance of the trained HydraGNN model and generates parity plots for each target property using the testing portion of the dataset.

15.2 INSTRUCTIONS TO DOWNLOAD THE DATASET

The FePt dataset is available open-source through the OLCF Data Constellation Facility and can be downloaded using the application Globus. If you do not have a Globus account, instructions on how to create one are available the website https://www.globus.org. The Globus application requires the user to specify the name of the source and destination endpoints among which the data transfer must be established. The name of the source destination from which the dataset can be downloaded is **OLCF-DOI-DOWNLOADS** and the path is **/~/OLCF/202102/10.13139_OLCF_1762742/**.

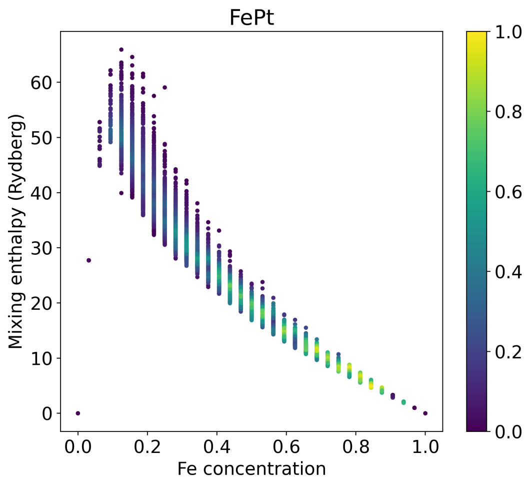


Figure 11. Configurational mixing enthalpy of solid solution binary alloy FePt with BCT structure as a function of Fe concentration. The color map indicates the relative frequency of data.

15.3 DESCRIPTION OF THE DATASET IN THE JSON FILE

```
{
    "Dataset": {
        "name": "FePt_32atoms",
        "path": {"total": "./dataset/FePt_enthalpy"},
        "format": "LSMS",
        "compositional_stratified_splitting": true,
        "rotational_invariance": false,
        "node_features": {
            "name": ["num_of_protons","charge_density", "magnetic_moment"],
            "dim": [1,1,1],
            "column_index": [0,5,6]
        },
        "graph_features":{
            "name": [ "free_energy_scaled_num_nodes"],
            "dim": [1],
            "column_index": [0]
        }
    },
    ...
}
```

15.4 DESCRIPTION OF HYDRAGNN ARCHITECTURE FOR SINGLE-TASKING

We set up HydraGNN to predict the mixing enthalpy. The example script is `lsms.py` and the input configuration file is `lsms_single_tasking.json`.

```
{
    "NeuralNetwork": {
        "Architecture": {
            "model_type": "PNA",
            "radius": 7,
            "max_neighbours": 100,
            "periodic_boundary_conditions": false,
            "hidden_dim": 100,
            "num_conv_layers": 6,
            "output_heads": {
                "graph":{
                    "num_sharedlayers": 2,
                    "dim_sharedlayers": 5,
                    "num_headlayers": 2,
                    "dim_headlayers": [50,25]
                }
            },
        "task_weights": [1.0]
        },
        "Variables_of_interest": {
            "input_node_features": [0],
            "output_names": ["free_energy_scaled_num_nodes"],
            "output_index": [0],
            "type": ["graph"],
            "denormalize_output": true
        },
        "Training": {
            ...
            }
        }
    },
    ...
}
```

15.5 DESCRIPTION OF HYDRAGNN ARCHITECTURE FOR MULTI-TASKING

We set up HydraGNN to simultaneously predict the mixing enthalpy, the atomic charge transfer, and the atomic magnetic moment with multi-task learning. The example script is the same as the one we used for the single-tasking example, `lsms.py`, and the input configuration file is `lsms_multi_tasking.json`.

```
{
    "NeuralNetwork": {
        "Architecture": {
            "model_type": "PNA",
            "radius": 7,
            "max_neighbours": 100,
            "periodic_boundary_conditions": false,
            "hidden_dim": 100,
            "num_conv_layers": 6,
            "output_heads": {
                "graph":{
                    "num_sharedlayers": 2,
                    "dim_sharedlayers": 5,
                    "num_headlayers": 2,
                    "dim_headlayers": [50,25]
                },
                "node": {
                    "num_headlayers": 2,
                    "dim_headlayers": [50,25],
                    "type": "mlp"
                }
            },
            "task_weights": [1.0, 1.0, 1.0]
        },
        "Variables_of_interest": {
            "input_node_features": [0],
            "output_names": ["free_energy_scaled_num_nodes","charge_density",
               "magnetic_moment"],
            "output_index": [0, 1, 2],
            "type": ["graph","node","node"],
            "denormalize_output": true
        },
        "Training": {
            ...
        }
    },
    ...
}
```

15.6 PYTHON SCRIPT FOR DATA LOAD AND HYDRAGNN TRAINING

The script used to load the dataset and train the HydraGNN model is called `lsms.py`. The scripts includes successive sections to:

- read data from raw LSMS output files using the `LSMSDataset` class
- write data in pickle format using the `SimplePickleWriter` class or in ADIOS format using the `AdiosWriter` class
- load pre-standardized `pickle` data using `SimplePickleDataset` or pre-standardized `ADIOS` data using `AdiosDataset`
- train the HydraGNN model
- save the trained model in a file with `.pk` format

15.7 PYTHON SCRIPT TO LOAD TRAINED HYDRAGNN MODEL AND USE IT FOR INFERENCE

The Python script `inference.py` loads a previously trained HydraGNN model and the test data, and performs post-training analysis of the trained HydraGNN model on the test data by measuring the mean absolute error (MAE) on each target property and creating the respective parity plots.

15.8 RESULTS FROM SINGLE-TASKING HYDRAGNN TRAINING

When the `inference.py` script is run on the single-tasking HydraGNN model after being trained with the hyperparameter configurations provided in the example online, the MAE for the predictions of the mixing enthalpy on the test dataset is 0.01 Rydberg.

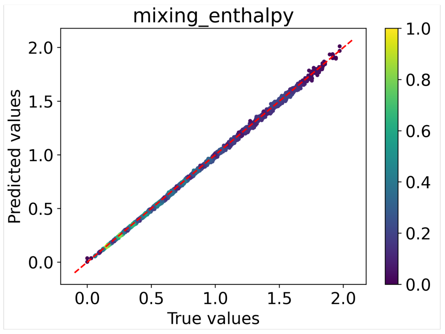


Figure 12. Parity plot of mixing enthalpy measured in Rydberg for atomistic structures of the test dataset using predictions of a single-tasking HydraGNN model.

15.9 RESULTS FROM MULTI-TASKING HYDRAGNN TRAINING

When the `inference.py` script is run on the multi-tasking HydraGNN model after being trained with the hyperparameter configurations provided in the example online, the MAE for the predictions of the mixing enthalpy on the test dataset is 0.01 Rydberg, the MAE for the predictions of the charge transfer on the test dataset is 0.66 electron charges, and the MAE for the predictions of the magnetic moment on the test dataset is 0.98 magnetons.

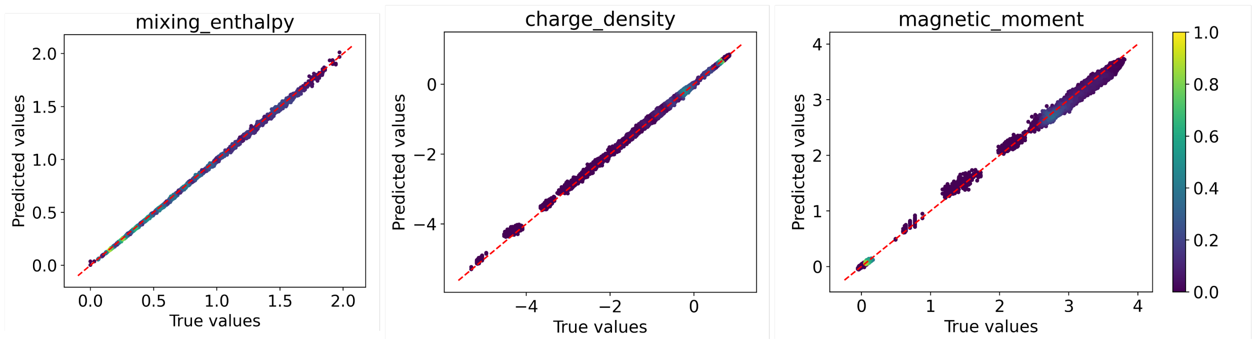


Figure 13. Parity plot of mixing enthalpy measured in Rydberg (left), charge transfer measured in electron charges (center), and magnetic moment measured in magnetons (right) for atomistic structures of the test dataset using predictions of a multi-tasking HydraGNN model.

REFERENCES

[1] J. A. Bondy and U. S. R. Murty, "Graphs and subgraphs," in *Graph theory with applications*. North-Holland.

[2] T. Xie and J. C. Grossman, "Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties," *Phys. Rev. Lett.*, vol. 120, no. 14, p. 145301, Apr. 2018. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.120.145301

[3] C. Chen, W. Ye, Y. Zuo, C. Zheng, and S. P. Ong, "Graph networks as a universal machine learning framework for molecules and crystals," *Chem. Mater.*, vol. 31, no. 9, pp. 3564–3572, May 2019. [Online]. Available: https://doi.org/10.1021/acs.chemmater.9b01294

[4] K. Choudhary and B. DeCost, "Atomistic line graph neural network for improved materials property predictions," *npj Computational Materials*, vol. 7, no. 1, pp. 1–8, 2021.

[5] C. W. Park and C. Wolverton, "Developing an improved crystal graph convolutional neural network framework for accelerated materials discovery," *Phys. Rev. Materials*, vol. 4, p. 063801, Jun 2020. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevMaterials.4.063801

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[7] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 2004, pp. 97–104.

[8] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[10] M. Abadi, "Tensorflow: learning functions at scale," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 2016, pp. 1–1.

[11] M. Zolnouri, X. Li, and V. P. Nia, "Importance of data loading pipeline in training deep neural networks," *arXiv preprint arXiv:2005.02130*, 2020.

[12] A. Aizman, G. Maltby, and T. Breuel, "High performance i/o for large scale deep learning," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 5965–5967.

[13] T. T. Nguyen, F. Trahay, J. Domke, A. Drozd, E. Vatai, J. Liao, M. Wahib, and B. Gerofi, "Why globally re-shuffle? revisiting data shuffling in large scale deep learning," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 1085–1096.

[14] Q. Koziol, D. Robinson, and U. O. of Science, "Hdf5," 3 2018.

[15] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[16] "Webdataset library," https://github.com/webdataset/webdataset, 2020, accessed: 2023/07.

[17] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "An implementation and evaluation of the mpi 3.0 one-sided communication interface," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 17, pp. 4385–4404, 2016.

[18] P. Hintjens, *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[19] R. Thakur, W. Gropp, and B. Toonen, “Optimizing the synchronization operations in message passing interface one-sided communication,” *The International Journal of High Performance Computing Applications*, vol. 19, no. 2, pp. 119–128, 2005.

[20] “mendeleev,” https://mendeleev.readthedocs.io/en/stable/.

[21] R. Ramakrishnan, P. O. Dral, M. Rupp, and O. A. Von Lilienfeld, “Quantum chemistry structures and properties of 134 kilo molecules,” *Scientific data*, vol. 1, no. 1, pp. 1–7, 2014.

[22] Z. Wu, B. Ramsundar, E. N. Feinberg, J. Gomes, C. Geniesse, A. S. Pappu, K. Leswing, and V. Pande, “Moleculenet: a benchmark for molecular machine learning,” *Chemical science*, vol. 9, no. 2, pp. 513–530, 2018.

[23] “torch_geometric.datasets.QM9 — pytorch_geometric documentation.” [Online]. Available: https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.datasets.QM9.html?highlight=QM9

[24] M. Lupo Pasini and M. Eisenbach, “FePt binary alloy with 32 atoms - LSMS-3 data - DOI:10.13139/OLCF/1762742.” [Online]. Available: https://www.osti.gov/dataexplorer/biblio/dataset/1762742

[25] M. Eisenbach, Y. W. Li, O. K. Odbadrakh, Z. Pei, G. M. Stocks, and J. Yin, “LSMS. https://github.com/mstsuite/lsms.” [Online]. Available: https://www.osti.gov//servlets/purl/1420087

[26] M. Eisenbach, J. Larkin, J. Lutjens, S. Rennich, and J. H. Rogers, “GPU acceleration of the locally self-consistent multiple scattering code for first principles calculation of the ground state and statistical physics of materials,” *Comput. Phys. Commun.*, vol. 211, pp. 2–7, 2017.

[27] Y. Wang, G. M. Stocks, W. A. Shelton, D. M. C. Nicholson, Z. Szotek, and W. M. Temmerman, “Order-N multiple scattering approach to electronic structure calculations,” *Phys. Rev. Lett.*, vol. 75, pp. 2867–2870, 1995.