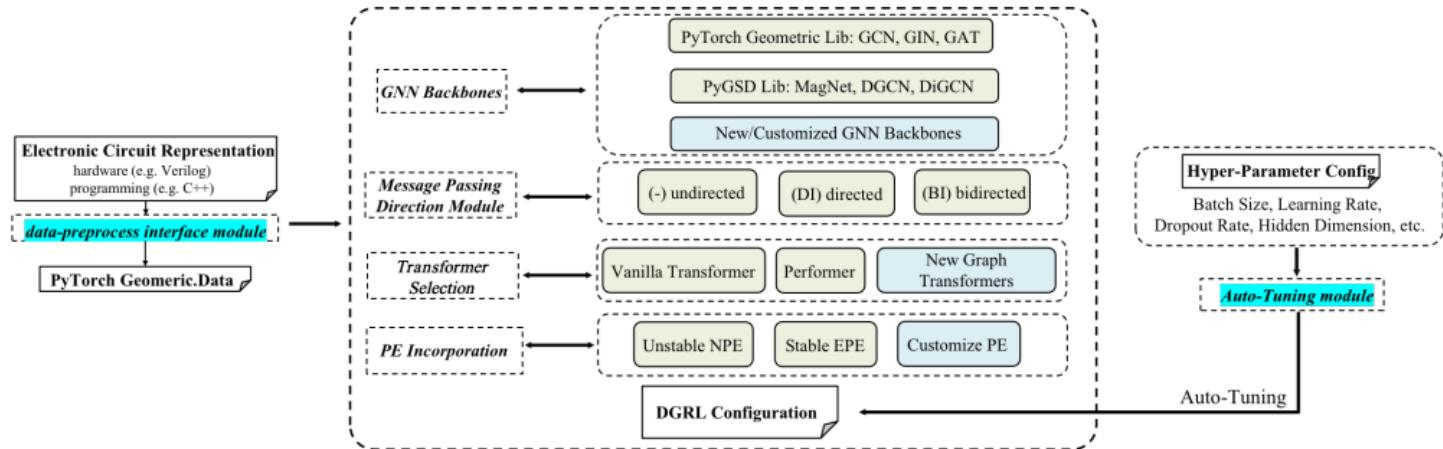


🏠 Welcome to DGRL-Hardware's documentation!

# Welcome to DGRL-Hardware's documentation!



DGRL-Hardware is a toolbox for evaluating **directed graph representation learning (DGRL)** methods. DGRL aims to encode and make inferences for directed graphs, which finds applications in various disciplines. Notably, DGRL holds significant importance in accelerating hardware design iterations by serving as surrogate models for predicting hardware performance. This significance stems from the wide use of directed graph representations in hardware data.

DGRL-Hardware provides the implementation of multiple DGRL techniques with backbone models including spectral GNNs, spatial GNNs, and graph transformers (GT), design options such as message passing directions and positional encodings, and incorporates several hardware design datasets as testbeds to evaluate different combinations of DGRL methods. The toolbox also offers hyper-parameter auto-tuning and evaluation pipelines. Users are encouraged to introduce new directed graph datasets for DGRL evaluation/selection or to develop new methods to evaluate on the selected datasets.

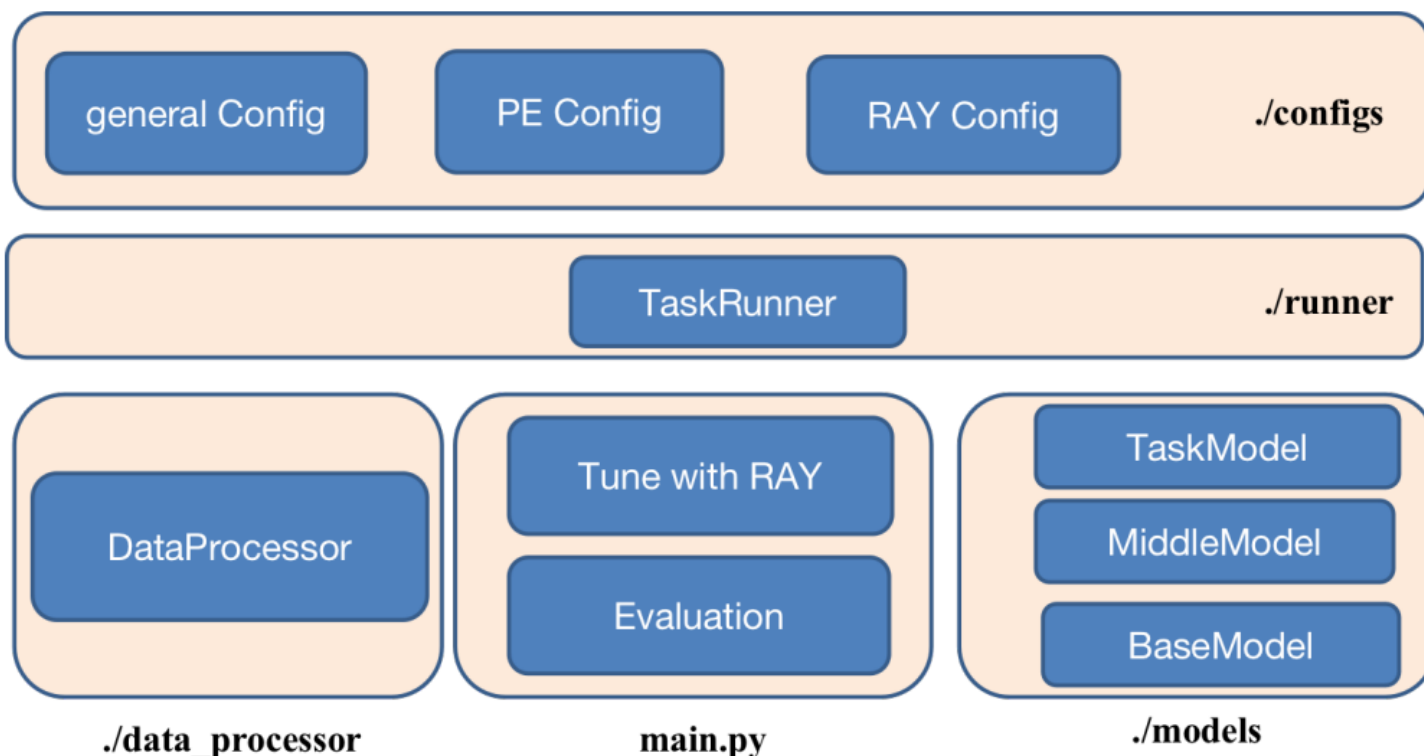
DGRL-hardware is built based on [Pytorch 2.0](#), [PyTorch Geometric](#), [PyTorch Geometric Signed and Directed](#), [RAY Tune](#) .

The Github repository page is available [here](#).

### ! Note

This project is under active development.

## Get Started



DGRL-Hardware is built and controlled by three configurations, user could configure a DGRL method with the **general config**, which defines the GNN backbone or the graph transformer to use along with MPNN layer message passing direction and their hyper-parameters. **RAY config** defines the hyper-parameter search space when conducting auto hyper-parameter tuning on the models. The toolbox also provides the implementation of two kinds of positional encodings (PE) which could be flexibly combined with the backbones and further improve the model expressiveness, users can configure the incorporation of positional encodings in the **PE config**. With the configuration, one could call a TaskRunner (as shown in the middle layer of the figure) for either hyper-parameter tuning or model evaluation. Three key components in the toolbox are connected with the TaskRunner, namely the dataset processor, the tuning/evaluation pipeline and the model implementation.

To get started, one may first set-up the environment, then configure DGRL methods (select an existing method or design a novel method) and config datasets (select an existing dataset or introduce a new dataset). After the DGRL method is configured, one may run RAY-tune for hyper-parameter tuning and then conduct performance evaluation.

## 1. Environment Requirement

This section illustrates the basic environment requirement to run the toolbox.

## 2. Config a method

This section introduces how to config a DGRL method in the toolbox. One may select from an existing method or customize a novel method.

- [Select from existing methods](#)
- [Customize new backbone/PE/message passing methods](#)

## 3. Config a dataset

This section describes how to config a dataset. One may select from the existing dataset or customize a novel dataset.

- [Select from existing datasets](#)
- [Customize new datasets](#)

## 4. Tune with RAY

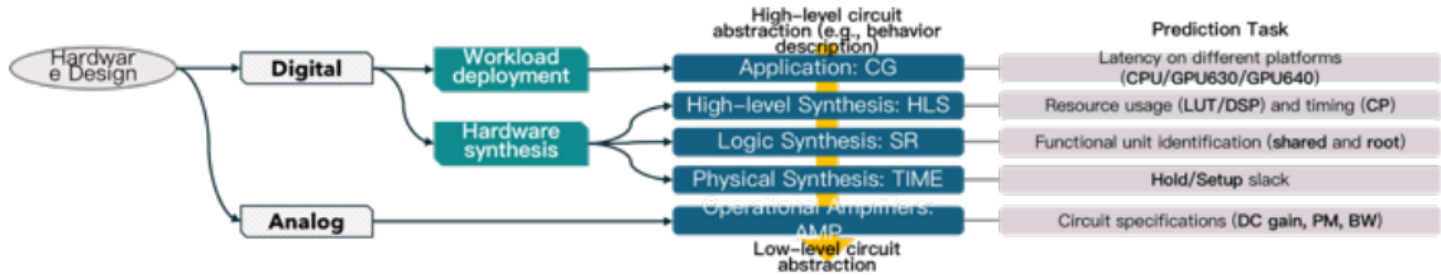
This section shows the interface on how to do hyper-parameter with the help of RAY, and how to config the search space.

## 5. Evaluation on Existing Datasets

This section introduces how to evaluate method on datasets with configuration.

# Toolbox Reference

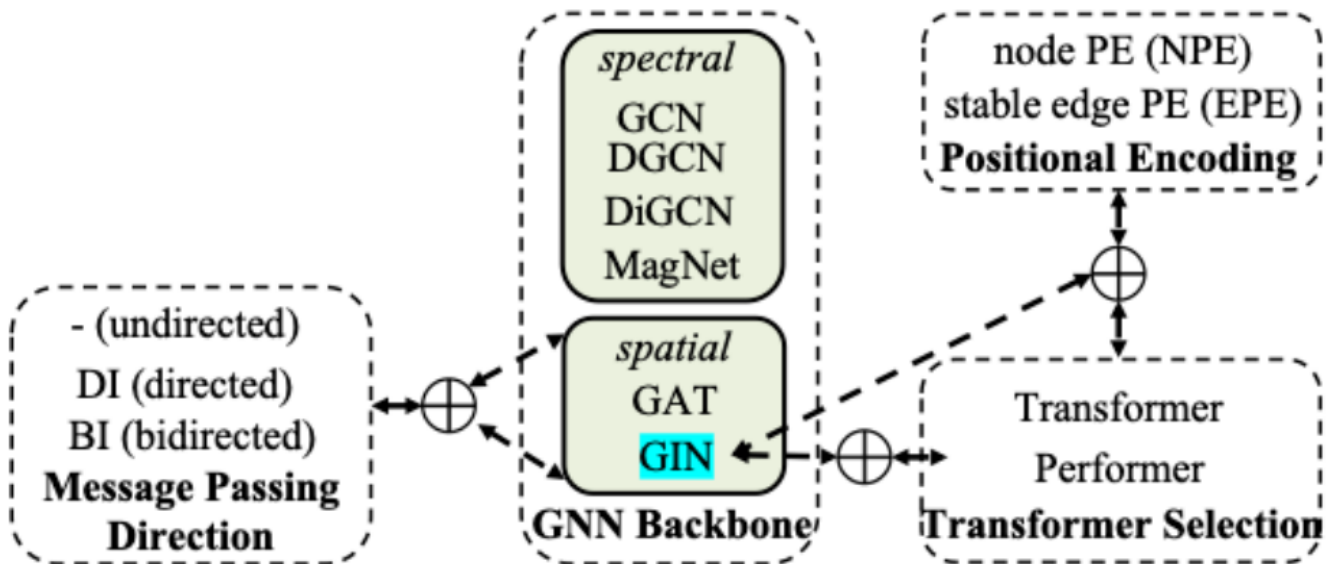
## Covered Datasets



- High-Level Synthesis (HLS)
- Symbolic Reasoning (SR)
- Pre-Routing Time Prediction (TIME)
- Computational Graph (CG)
- Operational Amplifiers (AMP)

## DGRL Methods

- Base Models: GNN Backbones, Graph Transformers and MPNN Layer Direction



- Positional Encoding (PE)

- [Positional Encodings, an Overview](#)
- [Obtain Magnetic Laplician PE for Directed Graphs](#)
- [Incorporate Magnetic Laplician PE for Directed Graphs](#)

# Environment

## Installation

We recommend environment management with [Conda](#). The supports below are based on Conda.

To config a DGRL method and train the model on existing datasets, the following packages are required:

```
conda create -n dgrl python==3.10
conda activate dgrl
# torch
conda install pytorch==2.1.2 torchvision==0.16.2 torchaudio==2.1.2 pytorch-
cuda=12.1 -c pytorch -c nvidia
# PyG (>=2.4.0)
pip install torch_geometric==2.4.0
# PyG dependencies
pip install torch_scatter torch_sparse torch_cluster torch_spline_conv -f
https://data.pyg.org/whl/torch-2.1.0+cu121.html
# PyGSD
pip install torch-geometric-signed-directed
# RAY
pip install -U "ray[data,train,tune,serve]"
# RAY dependencies
pip install hyperopt
# other dependencies
pip install prettytable
pip install torchmetrics
pip install hyperopt
pip install easydict
```



## Select from an Existing Method from Configuration

# Select from an Existing Method from Configuration

The DGRL-Hardware toolbox provides the implementation of 21 existing combinations of GNN, GT backbone, message passing direction and positional encoding (PE) incorporation. One can call one of these methods by editing the general configuration (listed in [./configs/general/](#)) and the PE configuration (listed in [./configs/pe/](#)).

An example of the genral config can be as follows:

```
# general_config.yaml
model:
  name: BIGAT
  hidden_dim: 192
  num_layers: 3
  node_input_dim: 7
  edge_input_dim: 2
  dropout: 0.2
  mlp_out:
    num_layer: 4
```

The name of all the implemented backbones are listed in the table below, for a detailed introduction on the interface and implementation of each base model, please refer to [Introduction on base DGRL methods](#):

GNN backbone/transformer	mesage passing	name in config
DGCN	directed	DGCN
DiGCN	directed	DiiGCN
MagNet	directed	MSGNN
GCN	undirected	GCN
GCN	directed	DIGCN
GCN	bidirected	BIGCN
GIN(E)	undirected	GIN(E)

GIN(E)	directed	DIGIN(E)
GIN(E)	bidirected	BIGIN(E)
GAT	undirected	GAT
GAT	directed	DIGAT
GAT	bidirected	BIGAT
GPS-T	undirected	GPS
GPS-T	directed	DIGPS
GPS-T	bidirected	BIGPS
GPS-P	undirected	PERFORMER
GPS-P	directed	DIPERFORMER
GPS-P	bidirected	BIPERFORMER

An example to configure PE is shown as follows:

```
# pe_config.yaml
model:
  pe_file_name: maglap_1q_spe
  pe_type: maglap
  q: 0.1
  q_dim: 1
  pe_strategy: invariant_fixed
  pe_encoder: spe
  mag_pe_dim_input: 10
  mag_pe_dim_output: 10
  se_pe_dim_input: 0
  se_pe_dim_output: 0

  eigval_encoder:
    in: 1
    hidden: 32
    out: 8
    num_layer: 3
```

The table below show the configuration to use for magnetic Laplacian PE with NPE or EPE:

stable	potential q	pe_type	pe_strategy	pe_embedder	example
NPE	q=0	lap	variant	naive naive	./configs/pe/lap
NPE	q>0	maglap	variant		./configs/pe/mag
EPE	q=0	lap	invariant_fixed		./configs/pe/lap
EPE	q>0	maglap	invariant_fixed		./configs/pe/mag



The `eigval_encoder` is used to configure the hyper-parameters of stable PE.

 To customize a new method

---

## To customize a new method

To customize a new methods other than existing base models, one should give the name and implementation in [./models/base\\_model.py](#), which controls the use of backbone models.

Specifically, one gives the initialization:

```
class BaseModel(torch.nn.Module):
    def __init__(self, **kwargs):
        if self.base_model == $New_method:
            # define the init of new method
            self.conv = #New_conv

        # an example could be:
        if self.base_model in ['GINE', 'DIGINE']:
            nn = Sequential(
                Linear(self.hidden_dim, self.hidden_dim),
                BatchNorm1d(self.hidden_dim),
                ReLU(),
                Dropout(self.dropout),)
            self.conv = GINEConv(nn)
```

And then gives the implementation:

```
class BaseModel(torch.nn.Module):
    def forward(self, x, edge_index, batch, **kwargs):
        if self.base_model == $New_method:
            x = self.conv(x, edge_index, kwargs['edge_attr'])
        # an example could be:
        if self.base_model in ['GINE', 'DIGINE']:
            x = self.conv(x, edge_index, kwargs['edge_attr'])
```

Once the `base_model.py` is edited, the base model could be called by a single line in the `general_config` as described in [Select from an Existing Method](#), and flexibly combine with the Positional Encoding (PE) methods with PE configurations.

## Select from an existing dataset

---

# Select from an existing dataset

To select from one of the existing datasets and tasks, one may configure the selected dataset in general configs, an example could be:

```
task:
  name: 'HLS'
  type: cdfg # type: cdfg or dfg
  target: dsp # different target prediction
  processed_folder: '~/DGRL_Hardware/data_processed/'
  divide_seed: default #set as default or a seed
  raw_data_path: '~/DGRL_Hardware/data_raw/HLS/'
  data_processor: HLSDataProcessor
```

here the name gives the name of the dataset, type and target determines the task, the processed\_folder defines the path to save the processed PyG format data, raw\_data\_path provides the path of the original data, the data\_processor defines the name of the data processor to process the data.

The data processor for the existing datasets are implemented in [./data\\_processor/](#). For more details on how to customize the dataset including the data processor, please refer to [customize new datasets](#).

 Customize new datasets and tasks

---

## Customize new datasets and tasks

To handle the new datasets, one needs to 1) process the data into either standard raw-data format or the PyG compatible format, and 2) customize a dataprocessor 3) customize a runner for the dataset.

## The Data Format that DGRL-Hardware asscpets

### Raw Data

DGRL accepts the following csv format to store the raw data:

file name	description
edge.csv	saves all the edges
node-feat.csv	saves all the edge feature
num-edge-list.csv	saves the number of edges in each graph
num-node-list.csv	saves the number of nodes in each graph
edge-feat.csv	saves the edge features of each graph
flexible	may save the labels

Examples of the raw data can be found at [./data\\_raw/](#).

### DataProcessor to handle the raw data

One need to customize a Data processor to process the raw data into PyG compatible data. The file name should be NEWDATA\_data\_processor.py (e.g. AMP\_data\_processor), saved in the folder [./data\\_processor](#).

A tutorial to customize such data processor is as follows:

```
class [$NEWDATA]DataProcessor(InMemoryDataset):
    def __init__(self, config, mode):
        # one may directly follow/copy the implementation of the initialization of
        data processors in existing datasets
    def process(self):
        # here to process the raw data into the PyG compatible data format

    def read_csv_graph_raw(self, raw_dir, check_repeat_edge):
        # this is the key function to process .csv files into the PyG data,
        # for setails please see https://github.com/Graph-COM/Benchmark\_for\_DGRL\_in\_Hardwares/tree/main/DGRL\_Hardware/data\_processor.
```

## Runner to Run with the New Datasets

One also needs to customize a Runner to run with the new dataset. The file name should be NEWDATA\_runner.py (e.g. AMP\_runner.py). saved in the folder `./runner`.

A tutorial to customize such runner is as follows:

```
class [$NewData]Runner():
    def __init__(self, config):
        # one may follow/copy the implementation of any existing DatasetRunners.
    def train_ray(self, tune_parameter_config):
        # the function that tuning with RAY would call for training, one may refer
        to the implementation of existing datasets
        # All the datasets/tasks share almost the same implementation

    def train(self):
        # the function that evaluation would call for training, one may refer to
        the implementation of existing datasets
        # All the datasets/tasks share almost the same implementation

    def raytune(self):
        # the function to load hyper-parameter design space
        # All the datasets/tasks share almost the same implementation

    def train_one_epoch(self, data_loader, mode, epoch_idx):
        # One may need to customize due to the difference in evaluation metrics

    def test(self, load_statedict = True, test_num_idx = 0):
        # One may need to customize due to the difference in evaluation metrics
```



The other functions in the Runner class share the same implementation across the datasets.

## Tune with RAY after Model Configuration

---

# Tune with RAY after Model Configuration

Users could design the hyper-parameter search space simply with a config, for the config we used for our benchmark, please refer to [./configs/ray/](#).

A sample config of RAY to search the hyper-parameter space is as follows:

```
name: BIGINE
hidden_dim: [2,7]
num_layers: [3,8]
# for HLS lr: [1e-4, 5e-3]
lr: [1e-4, 1e-2]
batch_size: [64, 128, 256, 512, 1024]
dropout: [0, 0.1, 0.2, 0.3]
# for HLS node_input_dim: 7
pe_dim_input: 20
pe_dim_output: 8

mlp_out:
  num_layer: [2,5]
```

An example command to tune with ray could be:

```
nohup python -u main.py --mode tune --general_config amp/gain/bigine --pe_config
lap10/lap_spe \
--ray_config BIGINE --device '1,3,4,5' --num_gpu_per_trial 0.24 \
>./ray_amp_gain_bigine_lap10_lap_spe.log 2>&1 </dev/null &
```

The scripts to tune the methods from our benchmark is provided in [./tune.sh](#).

In each Runner, we use similar functions to call RAY for hyper-parameter search, an example of the function is as follows:

```

def raytune(self, tune_config, num_samples, num_cpu, num_gpu_per_trial):
    reporter = CLIReporter(parameter_columns=['hidden_dim'], metric_columns=
['loss', 'mse', 'r2'])
    # init ray tune
    dropout_p = hp.choice('dropout_p', tune_config['dropout'])
    if self.config['model'].get('pe_file_name') in ['lap_naive',
'maglap_lq_naive'] and self.config['model']['name'] in ['GPS', 'GPSSE', 'DIGPS',
'BIGPS']:
        hidden_dim = 14 + 28 * hp.randint('hidden_dim',
int(tune_config['hidden_dim'][0]), int(tune_config['hidden_dim'][1]))
    else:
        hidden_dim = 28 * hp.randint('hidden_dim', int(tune_config['hidden_dim']
[0]), int(tune_config['hidden_dim'][1]))
    tune_parameter_config = {
        'name': tune_config['name'],
        'batch_size': hp.choice('batch_size', tune_config['batch_size']),
        'hidden_dim': hidden_dim,
        'num_layers': hp.randint('num_layers', int(tune_config['num_layers'][0]),
int(tune_config['num_layers'][1])),
        'lr': hp.uniform('lr', float(tune_config['lr'][0]), float(tune_config['lr']
[1])),
        'dropout': dropout_p,
        'mlp_out': {'num_layer': hp.randint('mlp_out', int(tune_config['mlp_out']
['num_layer'][0]),
int(tune_config['mlp_out']['num_layer']
[1]))},
        'node_input_dim': self.config['model']['node_input_dim'],
        'edge_input_dim': self.config['model']['edge_input_dim'],
        'pe_dim_input': tune_config['pe_dim_input'],
        'pe_dim_output': tune_config['pe_dim_output'],
        'criterion': 'MSE',
        'attn_type': 'multihead',
        'attn_kwargs': {'dropout': dropout_p},
    }
    tune_parameter_config = {**self.config['model'], **tune_parameter_config}
    scheduler = ASHAScheduler(
        max_t=800,
        grace_period=80,
        reduction_factor=2)

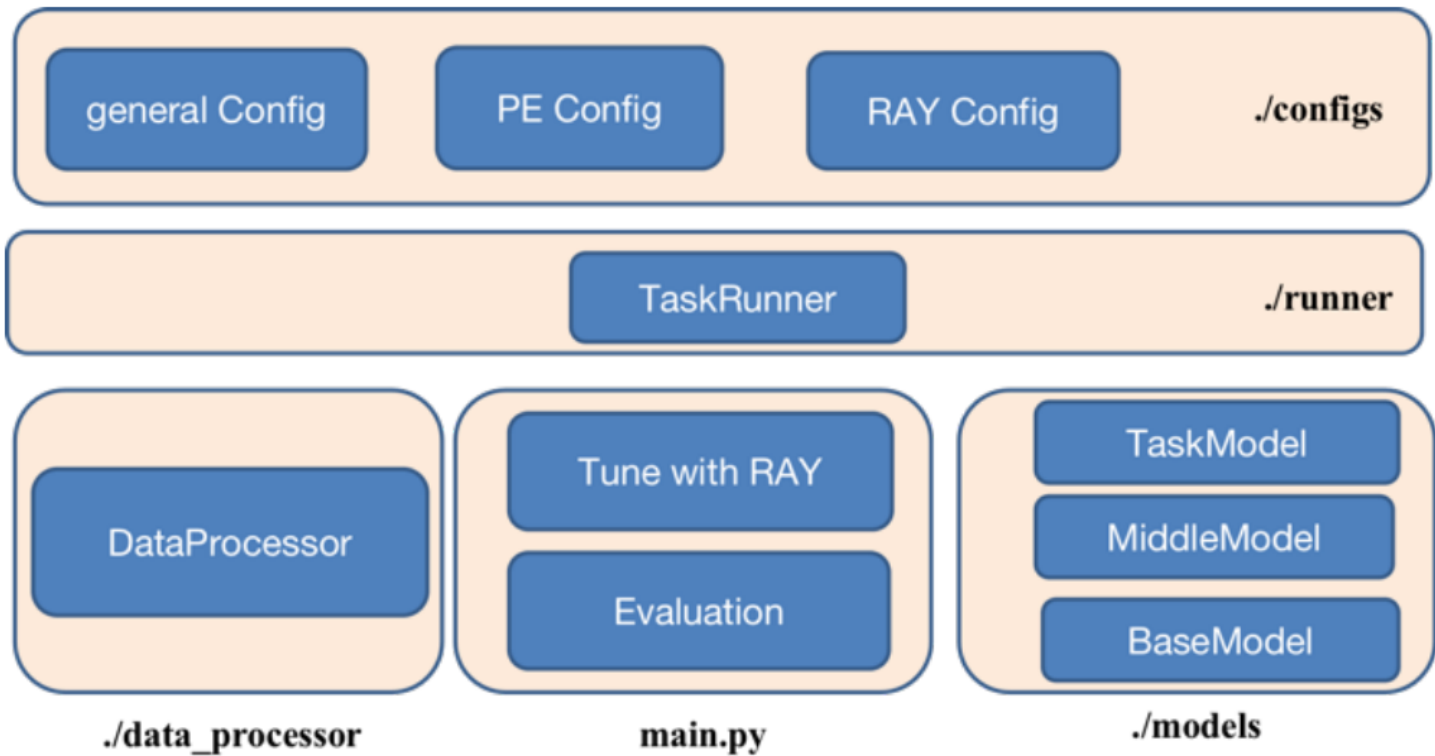
    hyperopt_search = HyperOptSearch(tune_parameter_config, metric='mse',
mode='min')

    tuner = tune.Tuner(
        tune.with_resources(
            tune.with_parameters(self.train_ray),
            resources={'cpu': num_cpu, 'gpu': num_gpu_per_trial}
        ),
        tune_config=tune.TuneConfig(
            metric='mse',
            mode='min',
            scheduler=scheduler,

```

```
        num_samples=num_samples,  
        search_alg=hyperopt_search,  
    ),  
    run_config=RunConfig(progress_reporter=reporter),  
)  
results = tuner.fit()  
  
best_result = results.get_best_result('mse', 'min')
```

## Model Evaluation with Existing Methods



After configuring a DGRL method and specify a dataset, one could perform evaluation by training and testing with different seeds, an example command to evaluate could be:

```
nohup python -u main.py --mode get_result --general_config amp/gain/bigine_lap \  
--pe_config lap10/lap_spe --device 1 \  
>./amp_gain_bigin_lap_spe_final_results_origin.log 2>&1 </dev/null &
```

The scripts to tune the methods from our benchmark is provided in [./main.sh](#).

# High-Level Synthesis (HLS)

## Overview

HLS is originally from [High-Level Synthesis Performance Prediction using GNNs: Benchmarking, Modeling, and Advancing](#).

After HLS front-end compilation, six node features are extracted, as summarized in the table below:

Feature	Description	Values
Node type	General node type	operation nodes, block
Bitwidth	Bitwidth of the node	0-256, misc
Opcode type	Opcode categories based on LLVM	binary_unary, bitwise, l
Opcode	Opcode of the node	load, add, xor, icmp, et
Is start of path	Whether the node is the starting node of a path	0, 1, misc
Cluster group	Cluster number of the node	-1 - 256, misc

Each edge has two features, the edge type represented in integers, and a binary value indicating whether this edge is a back edge. Each graph is labeled based on its post-implementation performance metrics, which are synthesized by [Vitis HLS](#) and implemented by [Vivado](#). Three metrics are used for regression: DSP, LUT, and CP. The first two are integer numbers indicating the number of resources used in the final implementation; the last one is CP timing in fractional number, determining the maximum working frequency of FPGA. The DFG and CDFG datasets consists of 19,120 and 18,570 C programs, respectively. The figure below shows an example C program from the CDFG dataset, with the corresponding control dataflow graph shown in the right. More information can be found in the original paper.

--	--





```
#include <stdio.h>
unsigned int fn1(char p, float p_4, double p_6, unsigned int p_8)
{
    long v;
    unsigned int result;
    v = (long)p_6;
    if ((unsigned long)(- v) < 192523741UL)
        result = - ((unsigned int)((float)p - p_4) - 48507U);
    else {
        result = 4294963506U;
        result /= (unsigned int)(- p_6) * ! p_8 + 876U;
    }
    return result;
}
```

## Interface

## Runner

```
class HLSRunner():
    def __init__(self, config):
        # init takes a config
    def train_ray(self, tune_parameter_config):
        # function to implement training when tuning with ray
    def train(self):
        # function to implement training when evaluation
    def train_one_epoch(self, data_loader, mode, epoch_idx):
        # function that do back propogation for one epoch
    def test(self, load_statedict = True, test_num_idx = 0):
        # function for testing
    def raytune(self, tune_config, num_samples, num_cpu, num_gpu_per_trial):
        # main function to take the hyper-parameter search space in RAY
```

Details are in [./runner/HLS\\_runner.py](#).

## DataProcessor

```
class HLSDataProcessor(InMemoryDataset):
    def __init__(self, config, mode):
        # init takes a config, mode takes from `tune` for tuning, `get_result` for
        # evaluation
    def process(self):
        # key functions to implement HLS data processing
    def read_csv_graph_raw(self, raw_dir, check_repeat_edge):
        # key function to process raw data into PyG data
```

Details are in [./data\\_processor/HLS\\_data\\_processor.py](#).

# Symbolic Regression (SR)

## Overview

SR is originally from [Gamora: Graph Learning based Symbolic Reasoning for Large-Scale Boolean Networks](#).

In this dataset, all the circuit designs are represented as and-inverter graphs (AIGs), a concise and uniform representation of BNs consisting of inverters and two-input AND gates, which allows rewriting, simulation, technology mapping, placement, and verification to share the same data structure. In an AIG, each node has at most two incoming edges; a node without incoming edges is a primary input (PI); primary outputs (POs) are denoted by special output nodes; each internal node represents a two-input AND function. Based on De Morgan's laws, any combinational BN can be converted into an AIG in a fast and scalable manner.

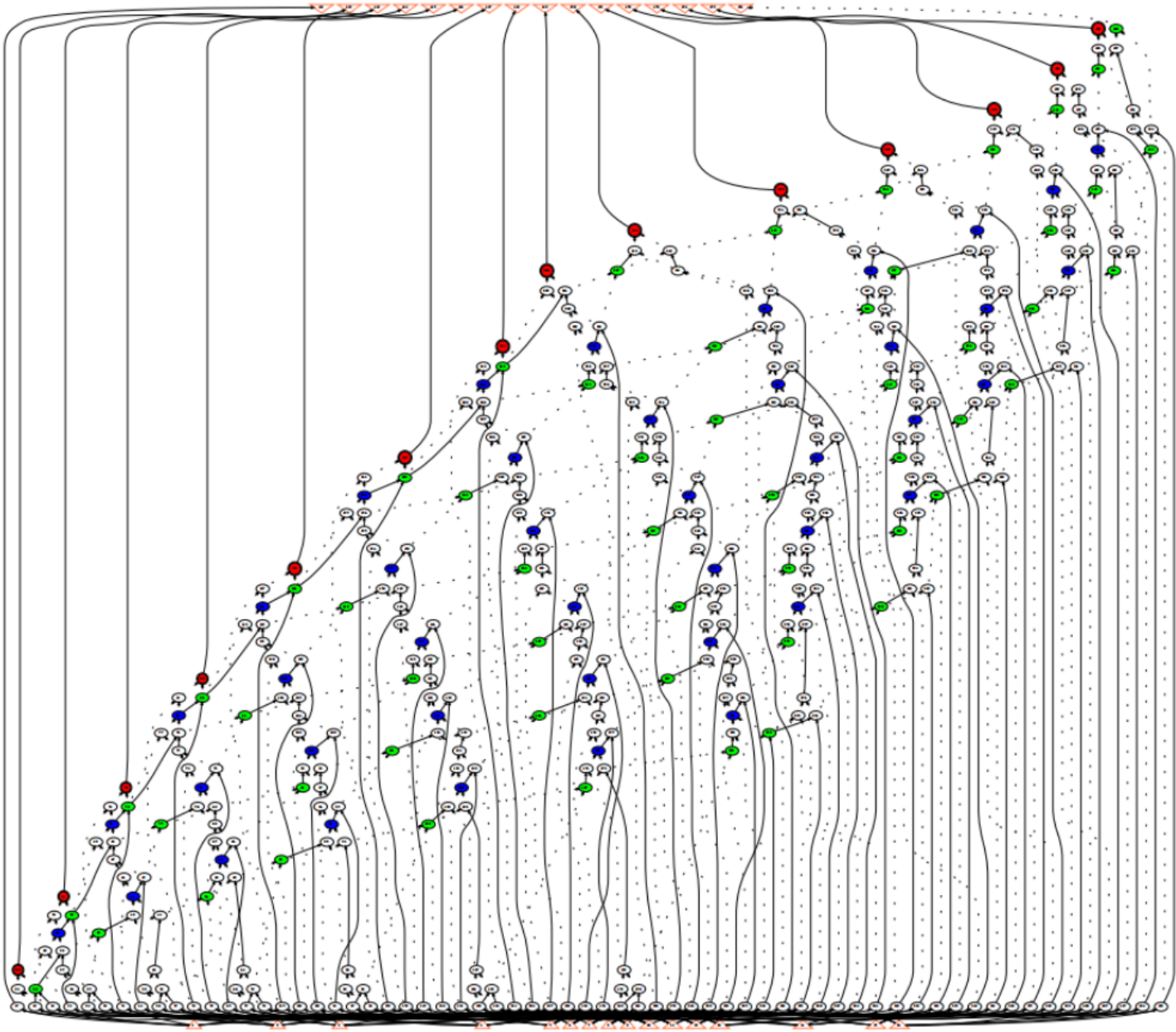
For each node, there are three node features represented in binary values denoting node types and Boolean functionality. The first node feature indicates whether this node is a PI/PO or intermediate node (i.e., AND gate). The second and the third node features indicate whether each input edge is inverted or not, such that AIGs can be represented as homogeneous graphs without additional edge features.

Node Feature Description	Size
(Net edge) Distances along x/y direction	2
(Cell edge) LUT is valid or no	8
(Cell) LUT indices	$8 * (7+7)$
(Cell) LUT value matrices	$8 * (7 * 7)$

Edge Feature Description	Size
Is primary I/O pin or not	1

Is fan-in or fan-out	1
Distance to the 4 die area boundaries	4
Pin capacitance	4 (EL/RF)

This dataset aims to leverage graph learning based approaches to accelerate the adder tree extraction in (integer) multiplier verification, which involves two reasoning steps: (1) detecting XOR/MAJ functions to construct adders, and then (2) identifying their boundaries. Thus, there are two sets of node labels, i.e., two node-level classification tasks. One task provides labels specifying whether a node (i.e., a gate) in the AIG belongs to MAJ, XOR, or is shared by both MAJ and XOR. The other task provides labels specifying whether a node is the root node of an adder. These AIGs and ground truth labels are generated by the logic synthesis tool [ABC](#). Figure below shows the AIG of an 8-bit multiplier: the blue and red nodes are the root nodes of XOR functions, with the red nodes directly connecting to the POs; the green nodes are the root nodes of MAJ functions. By pairing one XOR function with one MAJ function sharing the same set of inputs, we can extract the adder tree.



**Interface**

**Runner**

```

class SRRunner():
    def __init__(self, config):
        # init takes a config
    def train_ray(self, tune_parameter_config):
        # function to implement training when tuning with ray
    def train(self):
        # function to implement training when evaluation
    def train_one_epoch(self, data_loader, mode, epoch_idx):
        # function that do back propogation for one epoch
    def test(self, load_statedict = True, test_num_idx = 0):
        # function for testing
    def raytune(self, tune_config, num_samples, num_cpu, num_gpu_per_trial):
        # main function to take the hyper-parameter search space in RAY

```

Details are in [./runner/SR\\_runner.py](#).

## DataProcessor

```

class SRDataProcessor(InMemoryDataset):
    def __init__(self, config, mode):
        # init takes a config, mode takes from `tune` for tuning, `get_result` for
        # evaluation
    def process(self):
        # key functions to implement SR data processing
    def read_csv_graph_raw(self, raw_dir, check_repeat_edge):
        # key function to process raw data into PyG data

```

Details are in [./data\\_processor/SR\\_data\\_processor.py](#).



 [Pre-routing Timing Prediction \(TIME\)](#)

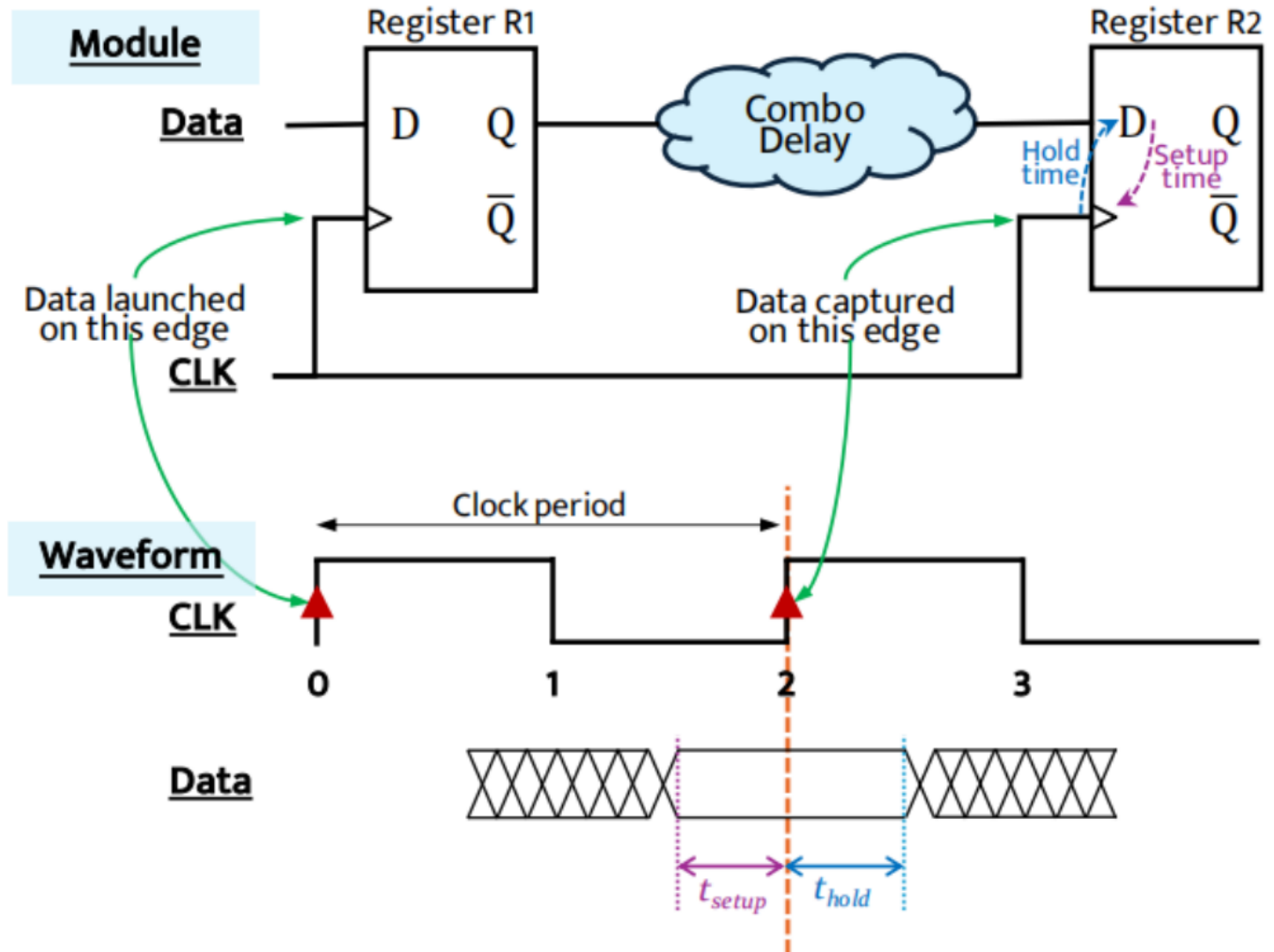
---

# Pre-routing Timing Prediction (TIME)

## Overview

TIME is originally from [A Timing Engine Inspired Graph Neural Network Model for Pre-Routing Slack Prediction](#).

Similar to timing analysis tools, circuits in this dataset are represented as heterogeneous graphs consisting of two types of edges: net edges and cell edges. The nodes in graphs denote pins in circuits. The TIME dataset collects 21 real-world benchmark circuits from [OpenCores](#) with [OpenROAD](#) on [SkyWater](#) 130nm technology (i.e. blabla, usb\_cdc\_core, BM64, salsa20, aes128, aes192, aes256, wbqspiflash, cic\_decimator, des, aes\_cipher, picorv32a, zipdiv, genericfir, usb, jpeg\_encoder, usbf\_device, xtea, spm, y\_huff, and synth\_ram). More information can be found in the original paper.



## Interface

## Runner

```
class TIMERunner():
    def __init__(self, config):
        # init takes a config
    def train_ray(self, tune_parameter_config):
        # function to implement training when tuning with ray
    def train(self):
        # function to implement training when evaluation
    def train_one_epoch(self, data_loader, mode, epoch_idx):
        # function that do back propagation for one epoch
    def test(self, load_statedict = True, test_num_idx = 0):
        # function for testing
    def raytune(self, tune_config, num_samples, num_cpu, num_gpu_per_trial):
        # main function to take the hyper-parameter search space in RAY
```

Details are in [./runner/TIME\\_runner.py](#).

## DataProcessor

```
class TIMEDataProcessor(InMemoryDataset):
    def __init__(self, config, mode):
        # init takes a config, mode takes from `tune` for tuning, `get_result` for
        # evaluation
    def process(self):
        # key functions to implement TIME data processing
    def read_csv_graph_raw(self, raw_dir, check_repeat_edge):
        # key function to process raw data into PyG data
```

Details are in [./data\\_processor/TIME\\_data\\_processor.py](#).

# Computational Graph (CG)

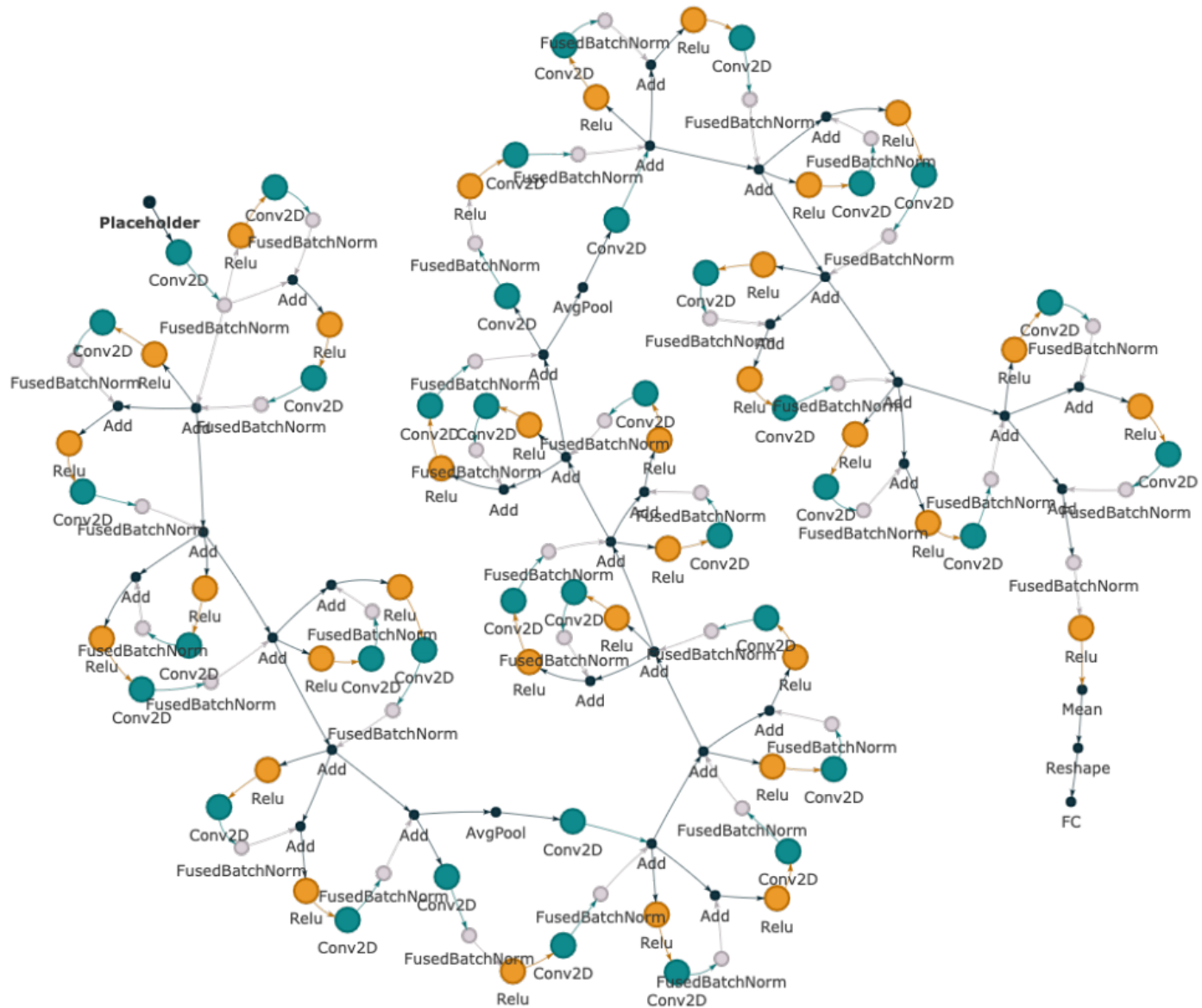
## Overview

CG is originally from [nn-Meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices](#).

This dataset includes

1. 12 state-of-the-art CNN models for the ImageNet2012 classification task (i.e., AlexNet, VGG, DenseNet, ResNet, SqueezeNet, GoogleNet, MobileNetv1, MobileNetv2, MobileNetv3, ShuffleNetv2, MnasNet, and ProxylessNas), each with 2,000 variants that differ in output channel number and kernel size per layer, and
2. 2,000 models from NASBench201 with the highest test accuracy on CIFAR10, each featuring a unique set of edge connections.

In total, this dataset contains 26,000 models with different operators and configurations. Figure below shows an example of the computational graph of a model in [NASBench201](#).



Node features include input shape (5 dimensions), kernel/weight shape (padding to 4 dimensions), strides (2 dimensions), and output shape (5 dimensions). Each computational graph is labeled with the inference latency on three edge devices (i.e., Cortex A76 CPU, Adreno 630 GPU, Adreno 640 GPU).

There is no edge feature in this dataset.

More information can be found in the original paper.

## Interface

## Runner

```

class CGRunner():
    def __init__(self, config):
        # init takes a config
    def train_ray(self, tune_parameter_config):
        # function to implement training when tuning with ray
    def train(self):
        # function to implement training when evaluation
    def train_one_epoch(self, data_loader, mode, epoch_idx):
        # function that do back propogation for one epoch
    def test(self, load_statedict = True, test_num_idx = 0):
        # function for testing
    def raytune(self, tune_config, num_samples, num_cpu, num_gpu_per_trial):
        # main function to take the hyper-parameter search space in RAY

```

Details are in [./runner/CG\\_runner.py](#).

## DataProcessor

```

class CGDataProcessor(InMemoryDataset):
    def __init__(self, config, mode):
        # init takes a config, mode takes from 'tune' for tuning, 'get_result' for
        # evaluation
    def process(self):
        # key functions to implement CG data processing
    def read_csv_graph_raw(self, raw_dir, check_repeat_edge):
        # key function to process raw data into PyG data

```

Details are in [./data\\_processor/CG\\_data\\_processor.py](#).



# Multi-Stage Amplifiers (AMP)

## Overview

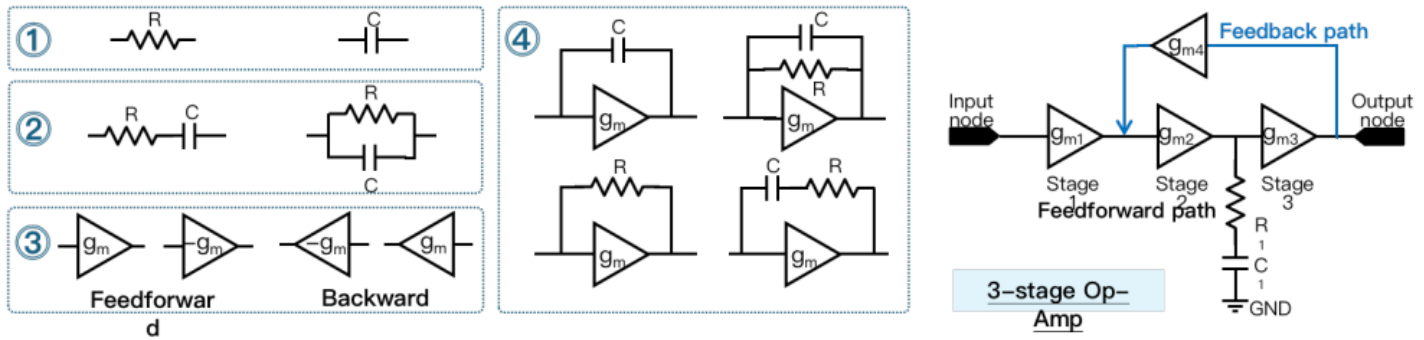
AMP is originally from [CktGNN: Circuit Graph Neural Network for Electronic Design Automation](#).

This dataset focuses on predicting circuit specifications (e.g., DC gain, bandwidth (BW), phase margin (PM)) of 2/3-stage operational amplifiers (Op-Amps), which are simulated by the [circuit simulator Cadence Spectre](#). A 2/3-stage Op-Amp consists of (1) two/three single-stage Op-Amps on the main feedforward path and (2) several feedback paths, with one example shown in the right part of Figure.

To make multi-stage Op-Amps more stable, feedforward and feedback paths are used to achieve different compensation schemes, each of which is implemented with a sub-circuit, e.g., single-stage Op-Amps, resistors, and capacitors. Due to the different topologies of single-stage Op-Amps and various compensation schemes, each sub-circuit is built as a subgraph.

There are 24 potential sub-circuits in the considered 2/3-stage Op-Amps:

- Single R or C ① in Figure, 2 types.
  - R and C connected in parallel or serial ② in Figure, 2 types.
  - A single-stage Op-Amp (gm) with different polarities (positive, +gm, or negative, -gm) and directions (feedforward or feedback) (③ in Figure, 4 types).
  - A single-stage Op-Amp (gm) with R or C connected in parallel or serial (16 types).
- Note that we use the single-stage Op-Amp with feedforward direction and positive polarities as an example for ④ in Figure.



## Interface

### Runner

```
class AMPRunner():
    def __init__(self, config):
        # init takes a config
    def train_ray(self, tune_parameter_config):
        # function to implement training when tuning with ray
    def train(self):
        # function to implement training when evaluation
    def train_one_epoch(self, data_loader, mode, epoch_idx):
        # function that do back propogation for one epoch
    def test(self, load_statedict = True, test_num_idx = 0):
        # function for testing
    def raytune(self, tune_config, num_samples, num_cpu, num_gpu_per_trial):
        # main function to take the hyper-parameter search space in RAY
```

Details are in [./runner/AMP\\_runner.py](#).

### DataProcessor

```
class AMPDataProcessor(InMemoryDataset):
    def __init__(self, config, mode):
        # init takes a config, mode takes from 'tune' for tuning, 'get_result' for evaluation
    def process(self):
        # key functions to implement AMP data processing
    def read_csv_graph_raw(self, raw_dir, check_repeat_edge):
        # key function to process raw data into PyG data
```

Details are in [./data\\_processor/AMP\\_data\\_processor.py](#).

# Base Models

The base models includes the GNN backbones/ graph transformers, and the MPNN message passing directions.

## GNN Backbones/ Graph Transformers

### DGCN (DGCN in config)

```
class DGCNConv(improved: bool = False, cached: bool = False, add_self_loops: bool = True, normalize: bool = True, **kwargs)

    def forward(x: torch.Tensor, edge_index: Union[torch.Tensor, torch_sparse.tensor.SparseTensor], edge_weight: Optional[torch.Tensor] = None) → torch.Tensor
```

The method is from [Directed Graph Convolutional Network](#). The implementation adopts the [PyGSD library](#).

### DiGCN (DiiGCN in config)

```
class DiGCNConv(in_channels: int, out_channels: int, improved: bool = False, cached: bool = True, bias: bool = True, **kwargs)

    def forward(x: torch.FloatTensor, edge_index: torch.LongTensor, edge_weight: Optional[torch.FloatTensor] = None) → torch.FloatTensor
```

The method is from [Digraph Inception Convolutional Networks](#). The implementation adopts the [PyGSD library](#).

## Magnet (MSGNN in config)

```
class MSConv(in_channels: int, out_channels: int, K: int, q: float, trainable_q:
bool, normalization: str = 'sym', bias: bool = True, cached: bool = False,
absolute_degree: bool = True, **kwargs)

    def forward(x_real: torch.FloatTensor, x_imag: torch.FloatTensor,
edge_index: torch.LongTensor, edge_weight: Optional[torch.Tensor] = None,
lambda_max: Optional[torch.Tensor] = None) → torch.FloatTensor
```

The method is from [MagNet: A Neural Network for Directed Graphs](#). The implementation adopts the [PyGSD library](#).

## GCN (GCN in config)

```
class GCNConv(in_channels: int, out_channels: int, improved: bool = False,
cached: bool = False, add_self_loops: Optional[bool] = None, normalize: bool =
True, bias: bool = True, **kwargs)

    def forward(x: Tensor, edge_index: Union[Tensor, SparseTensor],
edge_weight: Optional[Tensor] = None) → Tensor
```

The method is from [Semi-Supervised Classification with Graph Convolutional Networks](#). The implementation adopts the [PyG library](#).

## GIN(E) (GIN, GINE in config)

```
class GINConv(nn: Callable, eps: float = 0.0, train_eps: bool = False, **kwargs)

    def forward(x: Union[Tensor, Tuple[Tensor, Optional[Tensor]]], edge_index:
Union[Tensor, SparseTensor], size: Optional[Tuple[int, int]] = None) → Tensor

class GINEConv(nn: Module, eps: float = 0.0, train_eps: bool = False, edge_dim:
Optional[int] = None, **kwargs)

    def forward(x: Union[Tensor, Tuple[Tensor, Optional[Tensor]]], edge_index:
Union[Tensor, SparseTensor], edge_attr: Optional[Tensor] = None, size:
Optional[Tuple[int, int]] = None) → Tensor
```

The method is from [How Powerful are Graph Neural Networks?](#). The implementation adopts the [PyG library](#).

## GAT (GAT in config)

```
class GATConv(in_channels: Union[int, Tuple[int, int]], out_channels: int, heads:
int = 1, concat: bool = True, negative_slope: float = 0.2, dropout: float = 0.0,
add_self_loops: bool = True, edge_dim: Optional[int] = None, fill_value:
Union[float, Tensor, str] = 'mean', bias: bool = True, **kwargs)

    def forward(x: Union[Tensor, Tuple[Tensor, Optional[Tensor]]], edge_index:
Union[Tensor, SparseTensor], edge_attr: Optional[Tensor] = None, size:
Optional[Tuple[int, int]] = None, return_attention_weights: Optional[Tensor] =
None)→ Tensor
```

The method is from [Graph Attention Networks](#). The implementation adopts the [PyG library](#).

## GPS-T (GPS in config)

```
class GPSConv(channels: int, conv: Optional[MessagePassing], heads: int = 1,
dropout: float = 0.0, act: str = 'relu', act_kwargs: Optional[Dict[str, Any]] =
None, norm: Optional[str] = 'batch_norm', norm_kwargs: Optional[Dict[str, Any]] =
None, attn_type: str = 'multihead', attn_kwargs: Optional[Dict[str, Any]] = None)

    def forward(x: Tensor, edge_index: Union[Tensor, SparseTensor], batch:
Optional[Tensor] = None, **kwargs)→ Tensor
```

The method is from [Recipe for a General, Powerful, Scalable Graph Transformer](#). The implementation adopts the [PyG library](#).

## GPS-P (PERFORMER in config)



```
class GPSConv(channels: int, conv: Optional[MessagePassing], heads: int = 1,
              dropout: float = 0.0, act: str = 'relu', act_kwargs: Optional[Dict[str, Any]] =
              None, norm: Optional[str] = 'batch_norm', norm_kwargs: Optional[Dict[str, Any]] =
              None, attn_type: str = 'performer', attn_kwargs: Optional[Dict[str, Any]] = None)

    def forward(x: Tensor, edge_index: Union[Tensor, SparseTensor], batch:
              Optional[Tensor] = None, **kwargs)→ Tensor
```

The method is from [Rethinking Attention with Performers](#). The implementation adopts the [PyG library](#).

## Message Passing Directions

- undirected (-)

for undirected message passing, set directed=0 in the general config and implement the forward function with undirected message passing:

```
#general config.yaml
train:
    directed: 0
```

```
def __init__():
    self.conv = $model
def forward():
    x = self.conv(x, edge_index)
```

- directed (DI-)

```
#general config.yaml
train:
    directed: 1
```



```
def __init__():
    self.conv = $model
def forward():
    x = self.conv(x, edge_index)
```

- bidirected (BI-)

```
#general config.yaml
train:
    directed: 1
```

```
def __init__():
    self.forward_conv = $model
    self.backward_conv = $model
def forward():
    x1 = self.forward_conv(x, edge_index)
    x2 = self.backward_conv(x, edge_index)
    x = merge(x1 + x2)
```

The detailed implementation of each methods are in [./models/base\\_model.py](#).

# An Overview of Positional Encodings (PE)

Positional encodings (PE) for graphs are vectorized representations that can effectively describe the global position of nodes (absolute PE) or relative position of node pairs (relative PE). They provide crucial positional information and thus benefits many backbone models that is position-agnostic. For instance, on undirected graphs, PE can provably alleviate the limited expressive power of Message Passing Neural Networks [1], [2]; PE are also widely adopted in many graph transformers to incorporate positional information and break the identicalness of nodes in attention mechanism [3], [4]. As a result, the design and use of PE become one of the most important factors in building powerful graph encoders.

Likely, one can expect that direction-aware PE are also crucial when it comes to directed graph encoders. ‘Direction-aware’ implies that PE should be able to capture the directedness of graphs. A notable example is Magnetic Laplacian PE [5], which adopts the eigenvectors of Magnetic Laplacian as PE. Note that Magnetic Laplacian can encode the directedness via the sign of phase of  $\exp(\pm i 2\pi q)$ . Besides, when  $q=0$ , Magnetic Laplacian reduces to normal symmetric Laplacian. Thus, Magnetic Laplacian PE for directed graphs can be seen as a generalization of Laplacian PE for undirected graphs, and the latter is known to enjoy many nice spectral properties and be capable to capture many undirected graph distances. Therefore, Magnetic Laplacian appears to be a strong candidate for designing direction-aware PE. See [6] for a comprehensive introduction to Magnetic Laplacian.

Last, it is worth mentioning that there are also other PE for directed graphs, such as SVD of Adjacency matrix and directed random walk.

 Obtain Magnetic Laplacian PE via PyG Pre-transform

---

## Obtain Magnetic Laplacian PE via PyG Pre-transform

We provide a function that could obtain magnetic Laplacian PE based on [torch\\_geometric.transforms](#), our codes is built on [AddLaplacianEigenvectorPE](#).

An example to call our function for MagLap PE is as follows:

```
class DataProcessor(InMemoryDataset):
    def __init__(self, config):
        self.mag_pre_transform =
Compose([AddMagLaplacianEigenvectorPE(k=config['model']['mag_pe_dim_input'],
                                       q=config['model']['q'],
                                       multiple_q=config['model']['q_dim'],
                                       attr_name='mag_pe')])

    def process:
        if self.mag_pre_transform is not None:
            data = self.mag_pre_transform(data)
```

The class is located at [./maglap/get\\_mag\\_lap.py](#) and is as follows:

```
@functional_transform('add_mag_laplacian_eigenvector_pe')
class AddMagLaplacianEigenvectorPE(BaseTransform):
    """Adds the Magnetic Laplacian eigenvector positional encoding. The
    eigenvectors are
    complex number, so choosing k of them means there will be 2*k channels (k
    real parts and k imaginary parts)
    in total.

    Args:
        k (int): The number of non-trivial eigenvectors to consider.
        attr_name (str, optional): The attribute name of the data object to add
        positional encodings to. If set to :obj:`None`, will be
        concatenated to :obj:`data.x`.
        (default: :obj:`"laplacian_eigenvector_pe"`)
```

```

        **kwargs (optional): Additional arguments of
        :meth:`scipy.sparse.linalg.eigs` (when :attr:`is_undirected` is
        :obj:`False`) or :meth:`scipy.sparse.linalg.eigsh` (when
        :attr:`is_undirected` is :obj:`True`).
    """
    def __init__(
        self,
        k: int,
        q: float = 0.1,
        dynamic_q: bool = False,
        multiple_q: int = 1,
        attr_name: Optional[str] = 'laplacian_eigenvector_pe',
        **kwargs,
    ):
        self.k = k
        self.q = q
        self.dynamic_q = dynamic_q
        self.multiple_q = multiple_q
        self.attr_name = attr_name
        self.kwargs = kwargs

    def __call__(self, data: Data) -> Data:
        from scipy.sparse.linalg import eigs, eigsh
        eig_fn = eigsh # always use hermitian version

        num_nodes = data.num_nodes
        edge_index, edge_weight_list = get_mag_laplacian(
            data.edge_index,
            data.edge_weight,
            normalization='sym',
            num_nodes=num_nodes,
            q = self.q,
            dynamic_q=self.dynamic_q,
            multiple_q=self.multiple_q
        )

        pe_list = []
        eigvals_list = []
        for edge_weight in edge_weight_list:
            L = to_scipy_sparse_matrix(edge_index, edge_weight, num_nodes)

            #try:
            #    eig_vals, eig_vecs = eig_fn(
            #        L,
            #        k=self.k,
            #        which='SA',
            #        return_eigenvectors=True,
            #        **self.kwargs,
            #    )
            #    sort = eig_vals.argsort()
            #    eig_vals = eig_vals[sort]
            #    eig_vecs = eig_vecs[:, sort]
            #except:

```

```

        #from scipy.linalg import eigh
        #eig_vals, eig_vecs = eigh(L.toarray())
        #sort = eig_vals.argsort()[0:self.k]
        #eig_vals = eig_vals[sort]
        #eig_vecs = eig_vecs[:, sort]
        #eig_vals = eig_vals[0:self.k]
        #eig_vecs = eig_vecs[:, 0:self.k]

        #if np.isnan(eig_vecs).any() or np.isnan(eig_vals).any():
        eig_vals, eig_vecs = np.linalg.eigh(L.toarray())
        sort = eig_vals.argsort()[0:self.k]
        eig_vals = eig_vals[sort]
        eig_vecs = eig_vecs[:, sort]

        # padding zeros if num of nodes less than desired pe dimension
        if len(eig_vals) < self.k:
            eig_vals = np.pad(eig_vals, (0, self.k - len(eig_vals)))
            eig_vecs = np.pad(eig_vecs, ((0, 0), (0, self.k -
eig_vecs.shape[-1])))

        #pe = np.concatenate( (np.expand_dims(np.real(eig_vecs[:,
eig_vals.argsort()[0:self.k]], -1),
        #
            np.expand_dims(np.imag(eig_vecs[:,
eig_vals.argsort()[0:self.k]], -1)), axis=-1)
        #pe = np.concatenate( (np.expand_dims(np.real(eig_vecs), -1),
        #
            np.expand_dims(np.imag(eig_vecs), -1)),
axis=-1)

        # pe = torch.from_numpy(pe) # [N, pe_dim, 2]
        #sign = -1 + 2 * torch.randint(0, 2, (self.k, ))
        #sign = torch.unsqueeze(torch.unsqueeze(sign, dim=-1), dim=0)
        #pe = sign * pe

        #pe = pe.flatten(1, 2) # [N, pe_dim * 2]

        pe = torch.from_numpy(np.expand_dims(eig_vecs, 1))
        eig_vals = np.expand_dims(np.expand_dims(eig_vals, 0), 0)
        pe_list.append(pe)
        eigvals_list.append(torch.from_numpy(eig_vals))
        #pe = torch.cat(pe_list, dim=-1)
        #eig_vals = torch.cat(eigvals_list, dim=-1)
        pe = torch.cat(pe_list, dim=1)
        eig_vals = torch.cat(eigvals_list, dim=1)
        data = add_node_attr(data, pe, attr_name=self.attr_name)
        #data = add_node_attr(data, eig_vals.reshape(1, -1), attr_name='Lambda')
        data = add_node_attr(data, eig_vals, attr_name='Lambda')
        return data

```

# Positional Encoding Usage

We provide two ways of incorporating PEs, node PE (NPE) and edge PE (EPE), by simply adding a configuration of PE.

The configurations can be found in [./configs/pe/](#), and the implementations can be found in [./models/middle\\_model.py](#).

A sample config can be like:

```
model:
  pe_file_name: lap_naive
  pe_type: lap
  pe_strategy: variant
  lap_pe_dim_input: 10
  lap_pe_dim_output: 10
  se_pe_dim_input: 0
  se_pe_dim_output: 0

  eigval_encoder:
    in: 1
    hidden: 32
    out: 8
    num_layer: 3

  pe_embedder:
    name: naive
```

The table below show the configuration to use for magnetic Laplacian PE with NPE or EPE:

stable	potential q	pe_type	pe_strategy	pe_embedder	example
NPE	q=0	lap	variant		./configs/pe/lap
NPE	q>0	maglap	variant	naive	./configs/pe/maglap
EPE	q=0	lap	invariant_fixed	naive	./configs/pe/lap



EPE	q>0	maglap	invariant_fixed		./configs/pe/m:
-----	-----	--------	-----------------	--	-----------------

The eigval\_encoder is used to configure the hyper-parameters of stable PE.

Note that NPE directly concatenate PE with node feature, while EPE processes PE with [stable PE](#) and concatenates PE on edge features.