AutoGL

Release v0.2.0rc0

THUMNLab/aglteam

Jul 11, 2021

TUTORIAL

CHAPTER ONE

AUTOGL

Actively under development by @THUMNLab

AutoGL is developed for researchers and developers to quickly conduct autoML on the graph datasets & tasks.

The workflow below shows the overall framework of AutoGL.

AutoGL uses `AutoGL Dataset` to maintain datasets for graph-based machine learning, which is based on the dataset in PyTorch Geometric with some support added to corporate with the auto solver framework.

Different graph-based machine learning tasks are solved by different `AutoGL Solvers` , which make use of four main modules to automatically solve given tasks, namely `Auto Feature Engineer`, `Auto Model`, `Neural Architecture Search`, `HyperParameter Optimization`, and `Auto Ensemble`.

CHAPTER

TWO

INSTALLATION

2.1 Requirements

Please make sure you meet the following requirements before installing AutoGL.

1. Python >= 3.6.0
2. PyTorch (>=1.6.0)

   see PyTorch for installation.
3. PyTorch Geometric (>=1.7.0)

   see PyTorch Geometric for installation.

2.2 Installation

2.2.1 Install from pip & conda

Run the following command to install this package through pip.

```
pip install autogl
```

2.2.2 Install from source

Run the following command to install this package from the source.

```
git clone https://github.com/THUMNLab/AutoGL.git
cd AutoGL
python setup.py install
```

2.2.3 Install for development

If you are a developer of the AutoGL project, please use the following command to create a soft link, then you can modify the local package without installation again.

```
pip install -e .
```

CHAPTER

THREE

MODULES

In AutoGL, the tasks are solved by corresponding solvers, which in general do the following things:

1. Preprocess and feature engineer the given datasets. This is done by the module named **auto feature engineer**, which can automatically add/delete useful/useless attributes in the given datasets. Some topological features may also be extracted & combined to form stronger features for current tasks.
2. Find the best suitable model architectures through neural architecture search. This is done by modules named **nas**. AutoGL provides several search spaces, algorithms and estimators for finding the best architectures.
2. Automatically train and tune popular models specified by users. This is done by modules named **auto model** and **hyperparameter optimization**. In the auto model, several commonly used graph deep models are provided, together with their hyperparameter spaces. These kinds of models can be tuned using **hyperparameter optimization** module to find the best hyperparameter for the current task.
3. Find the best way to ensemble models found and trained in the last step. This is done by the module named **auto ensemble**. The suitable models available are ensembled here to form a more powerful learner.

3.1 Quick Start

This tutorial will help you quickly go through the concepts and usages of important classes in AutoGL. In this tutorial, you will conduct a quick auto graph learning on dataset Cora.

3.1.1 AutoGL Learning

Based on the concept of autoML, auto graph learning aims at **automatically** solve tasks with data represented by graphs. Unlike conventional learning frameworks, auto graph learning, like autoML, does not need humans inside the experiment loop. You only need to provide the datasets and tasks to the AutoGL solver. This framework will automatically find suitable methods and hyperparameters for you.

The diagram below describes the workflow of AutoGL framework.

To reach the aim of autoML, our proposed auto graph learning framework is organized as follows. We have `dataset` to maintain the graph datasets given by users. A `solver` object needs to be built for specifying the target tasks. Inside `solver`, there are five submodules to help complete the auto graph tasks, namely `auto feature engineer`, `auto model`, `neural architecture search`, `hyperparameter optimization` and `auto ensemble`, which will automatically preprocess/enhance your data, choose and optimize deep models and ensemble them in the best way for you.

Let's say you want to conduct an auto graph learning on dataset `Cora`. First, you can easily get the `Cora` dataset using the `dataset` module:

```
from autogl.datasets import build_dataset_from_name
cora_dataset = build_dataset_from_name('cora')
```

The dataset will be automatically downloaded for you. Please refer to *AutoGL Dataset* or *autogl.datasets* for more details of dataset constructions, available datasets, add local datasets, etc.

After deriving the dataset, you can build a `node classification solver` to handle auto training process:

```
import torch
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
from autogl.solver import AutoNodeClassifier
solver = AutoNodeClassifier(
    feature_module='deepgl',
    graph_models=['gcn', 'gat'],
    hpo_module='anneal',
    ensemble_module='voting',
    device=device
)
```

In this way, we build a `node classification solver`, which will use `deepgl` as its feature engineer, and use `anneal` hyperparameter optimizer to optimize the given three models `['gcn','gat']`. The derived models will then be ensembled using `voting` ensembler. Please refer to the corresponding tutorials or documentation to see the definition and usages of available submodules.

Then, you can fit the solver and then check the leaderboard:

```
solver.fit(cora_dataset, time_limit=3600)
solver.get_leaderboard().show()
```

The `time_limit` is set to 3600 so that the whole auto graph process will not exceed 1 hour. `solver.show()` will present the models maintained by `solver`, with their performances on the validation dataset.

Then, you can make the predictions and evaluate the results using the evaluation functions provided:

```
from autogl.module.train import Acc
predicted = solver.predict_proba()
print('Test accuracy: ', Acc.evaluate(predicted,
    cora_dataset.data.y[cora_dataset.data.test_mask].cpu().numpy()))
```

Note: You don't need to pass the `cora_dataset` again when predicting, since the dataset is **remembered** by the `solver` and will be reused when no dataset is passed at predicting. However, you can also pass a new dataset when predicting, and the new dataset will be used instead of the remembered one. Please refer to *AutoGL Solver* or *autogl.solver* for more details.

3.2 AutoGL Dataset

We import the module of datasets from *CogDL* and *PyTorch Geometric* and add support for datasets from *OGB*. One can refer to the usage of creating and building datasets via the tutorial of CogDL, PyTorch Geometric, and OGB.

3.2.1 Supporting datasets

AutoGL now supports the following benchmarks for different tasks:

Semi-supervised node classification: Cora, Citeseer, Pubmed, Amazon Computers*, Amazon Photo*, Coauthor CS*, Coauthor Physics*, Reddit *: using *utils.random_splits_mask_class* for splitting dataset is recommended.). For detailed information for supporting datasets, please kindly refer to PyTorch Geometric Dataset.

Dataset	PyG	CogDL	x	y	edge_index	edge_attr | train/val/test node		train/val/test mask
Cora	✓		✓	✓	✓	✓		✓
Citeseer	✓		✓	✓	✓	✓		✓
Pubmed	✓		✓	✓	✓	✓		✓
Amazon Computers	✓		✓	✓	✓	✓		
Amazon Photo	✓		✓	✓	✓	✓		
Coauthor CS	✓		✓	✓	✓	✓		
Coauthor Physics	✓		✓	✓	✓	✓		
Reddit	✓		✓	✓	✓	✓		✓

Graph classification: MUTAG, IMDB-B, IMDB-M, PROTEINS, COLLAB

Dataset	PyG	CogDL	x	y	edge_index	edge_attr
MUTAG	✓		✓	✓	✓	✓
IMDB-B	✓			✓	✓	
IMDB-M	✓			✓	✓	
PROTEINS	✓		✓	✓	✓	
COLLAB	✓			✓	✓	

TODO: Supporting all datasets from *PyTorch Geometric*.

3.2.2 OGB datasets

AutoGL also supports the popular benchmark on *OGB* for node classification and graph classification tasks. For the summary of *OGB* datasets, please kindly refer to the their docs.

Since the loss and evaluation metric used for *OGB* datasets vary among different tasks, we also add *string* properties of datasets for identification:

Dataset	dataset.metric	datasets.loss
ogbn-products	Accuracy	nll_loss
ogbn-proteins	ROC-AUC	BCEWithLogitsLoss
ogbn-arxiv	Accuracy	nll_loss
ogbn-papers100M	Accuracy	nll_loss
ogbn-mag	Accuracy	nll_loss
ogbg-molhiv	ROC-AUC	BCEWithLogitsLoss
ogbg-molpcba	AP	BCEWithLogitsLoss
ogbg-ppa	Accuracy	CrossEntropyLoss
ogbg-code	F1 score	CrossEntropyLoss

3.2.3 Create a dataset via URL

If your dataset is the same as the 'ppi' dataset, which contains two matrices: 'network' and 'group', you can register your dataset directly use the above code. The default root for downloading dataset is *~/.cache-autogl*, you can also specify the root by passing the string to the *path* in *build_dataset(args, path)* or *build_dataset_from_name(dataset, path)*.

```
# following code-snippet is from autogl/datasets/matlab_matrix.py

@register_dataset("ppi")
class PPIDataset(MatlabMatrix):
    def __init__(self, path):
        dataset, filename = "ppi", "Homo_sapiens"
        url = "http://snap.stanford.edu/node2vec/"
        super(PPIDataset, self).__init__(path, filename, url)
```

You should declare the name of the dataset, the name of the file, and the URL, where our script can download the resource. Then you can use either *build_dataset(args, path)* or *build_dataset_from_name(dataset, path)* in your task to build a dataset with corresponding parameters.

3.2.4 Create a dataset locally

If you want to test your local dataset, we recommend you to refer to the docs on creating PyTorch Geometric dataset.

You can simply inherit from *torch_geometric.data.InMemoryDataset* to create an empty *dataset*, then create some *torch_geometric.data.Data* objects for your data and pass a regular python list holding them, then pass them to *torch_geometric.data.Dataset* or *torch_geometric.data.DataLoader*. Let's see this process in a simplified example:

```
from typing import Iterable
from torch_geometric.data.data import Data
from autogl.datasets import build_dataset_from_name
from torch_geometric.data import InMemoryDataset

class MyDataset(InMemoryDataset):
    def __init__(self, datalist) -> None:
        super().__init__()
        self.data, self.slices = self.collate(datalist)

# Create your own Data objects

```

(continues on next page)

(continued from previous page)

```
# for example, if you have edge_index, features and labels
# you can create a Data as follows
# See pytorch geometric more info of Data
data = Data()
data.edge_index = edge_index
data.x = features
data.y = labels

# create a list of Data object
data_list = [data, Data(...), ..., Data(...)]

# Initialize AutoGL Dataset with your own data
myData = MyDataset(data_list)
```

3.3 AutoGL Feature Engineering

We provide a series of node and graph feature engineers for you to compose within a feature engineering pipeline. An automatic feature engineering algorithm is also provided.

3.3.1 Quick Start

```
# 1. Choose a dataset.
from autogl.datasets import build_dataset_from_name
data = build_dataset_from_name('cora')

# 2. Compose a feature engineering pipeline
from autogl.module.feature import BaseFeature,AutoFeatureEngineer
from autogl.module.feature.generators import GeEigen
from autogl.module.feature.selectors import SeGBDT
from autogl.module.feature.graph import SgNetLSD
# you may compose feature engineering bases through BaseFeature.compose
fe = BaseFeature.compose([
GeEigen(size=32) ,
SeGBDT(fixlen=100),
SgNetLSD()
])
# or just through '&' operator
fe = fe & AutoFeatureEngineer(fixlen=200,max_epoch=3)

# 3. Fit and transform the data
fe.fit(data)
data1=fe.transform(data,inplace=False)
```

3.3.2 List of FE base names

Now three kinds of feature engineering bases are supported,namely generators, selectors , graph.You can import bases from according module as is mentioned in the Quick Start part. Or you may want to just list names of bases in configurations or as arguments of the autogl solver.

1. generators

Base	Description
graphlet	concatenate local graphlet numbers as features.
eigen	concatenate Eigen features.
pagerank	concatenate Pagerank scores.
PYGLocalDegreeProfile	concatenate Local Degree Profile features.
PYGNormalizeFeatures	Normalize all node features
PYGOneHotDegree	concatenate degree one-hot encoding.
onehot	concatenate node id one-hot encoding.

2. selectors

Base	Description
SeFilterConstant	delete all constant and one-hot encoding node features.
gbdt	select top-k important node features ranked by Gradient Descent Decision Tree.

3. graph

netlsd is a graph feature generation method. please refer to the according document.

A set of graph feature extractors implemented in NetworkX are wrapped, please refer to NetworkX for details. (NxLargeCliqueSize, NxAverageClusteringApproximate, NxDegreeAssortativityCoefficient, NxDegreePearsonCorrelationCoefficient, NxHasBridge ,``NxGraphCliqueNumber``, NxGraphNumberOfCliques, NxTransitivity, NxAverageClustering, NxIsConnected, NxNumberConnectedComponents, NxIsDistanceRegular, NxLocalEfficiency, NxGlobalEfficiency, NxIsEulerian)

The taxonomy of base types is based on the way of transforming features. generators concatenate the original features with ones newly generated or just overwrite the original ones. Instead of generating new features , selectors try to select useful features and keep learned selecting methods in the base itself. The former two types of bases can be exploited in node or edge level (modification upon each node or edge feature) ,while graph focuses on feature engineering in graph level (modification upon each graph feature). For your convenience in further development,you may want to design a new item by inheriting one of them. Of course, you can directly inherit the BaseFeature as well.

3.3.3 Create Your Own FE

You can create your own feature engineering object by simply inheriting one of feature engineering base types ,namely generators, selectors , graph, and overloading methods _fit and _transform.

```python
# for example : create a node one-hot feature.
from autogl.module.feature.generators.base import BaseGenerator
import numpy as np
class GeOnehot(BaseGenerator):
    def __init__(self):
        super(GeOnehot,self).__init__(data_t='np',multigraph=True,subgraph=False)
        # data type in mid is 'numpy',
        # and it can be used for multigraph,
```

(continues on next page)

(continued from previous page)

```
        # but not suitable for subgraph feature extraction.

    def _fit(self):
        pass # nothing to train or memorize

    def _transform(self, data):
        fe=np.eye(data.x.shape[0])
        data.x=np.concatenate([data.x,fe],axis=1)
        return data
```

3.4 AutoGL Model

In AutoGL, we use `model` and `automodel` to define the logic of graph nerual networks and make it compatible with hyper parameter optimization. Currently we support the following models for given tasks.

Tasks	Models
Node Classification	`gcn`, `gat`, `sage`
Graph Classification	`gin`, `topk`
Link Prediction	`gcn`, `gat`, `sage`

3.4.1 Lazy Initialization

In current AutoGL pipeline, some important hyper-parameters related with model cannot be set outside before the pipeline (e.g. input dimensions, which can only be caluclated during running after feature engineered). Therefore, in `automodel`, we use lazy initialization to initialize the core `model`. When the `automodel` initialization method `__init__()` is called with argument `init` be `False`, only (part of) the hyper-parameters will be set. The `automodel` will have its core `model` only after `initialize()` is explicitly called, which will be done automatically in `solver` and `from_hyper_parameter()`, after all the hyper-parameters are set properly.

3.4.2 Define your own model and automodel

We highly recommend you to define both `model` and `automodel`, although you only need your `automodel` to communicate with `solver` and `trainer`. The `model` will be responsible for the parameters initialization and forward logic declaration, while the `automodel` will be responsible for the hyper-parameter definiton and organization.

General customization

Let's say you want to implement a simple MLP for node classification and want to let AutoGL find the best hyper-parameters for you. You can first define the logics assuming all the hyper-parameters are given.

```
import torch

# define mlp model, need to inherit from torch.nn.Module
class MyMLP(torch.nn.Module):
    # assume you already get all the hyper-parameters
    def __init__(self, in_channels, num_classes, layer_num, dim):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        if layer_num == 1:
            ops = [torch.nn.Linear(in_channels, num_classes)]
        else:
            ops = [torch.nn.Linear(in_channels, dim)]
            for i in range(layer_num - 2):
                ops.append(torch.nn.Linear(dim, dim))
            ops.append(torch.nn.Linear(dim, num_classes))

        self.core = torch.nn.Sequential(*ops)

    # this method is required
    def forward(self, data):
        # data: torch_geometric.data.Data
        assert hasattr(data, 'x'), 'MLP only support graph data with features'
        x = data.x
        return torch.nn.functional.log_softmax(self.core(x))
```

After you define the logic of `model`, you can now define your `automodel` to manage the hyper-parameters.

```
from autogl.module.model import BaseModel

# define your automodel, need to inherit from BaseModel
class MyAutoMLP(BaseModel):
    def __init__(self):
        # (required) make sure you call __init__ of super with init argument properly␣
↪set.
        # if you do not want to initialize inside __init__, please pass False.
        super().__init__(init=False)

        # (required) define the search space
        self.space = [
            {'parameterName': 'layer_num', 'type': 'INTEGER', 'minValue': 1, 'maxValue':␣
↪5, 'scalingType': 'LINEAR'},
            {'parameterName': 'dim', 'type': 'INTEGER', 'minValue': 64, 'maxValue': 128,
↪'scalingType': 'LINEAR'}
        ]

        # set default hyper-parameters
        self.layer_num = 2
        self.dim = 72

        # for the hyper-parameters that are related with dataset, you can just set them␣
↪to None
        self.num_classes = None
        self.num_features = None

        # (required) since we don't know the num_classes and num_features until we see␣
↪the dataset,
        # we cannot initialize the models when instantiated. the initialized will be set␣
↪to False.
        self.initialized = False

```

(continues on next page)

(continued from previous page)

```
        # (required) set the device of current auto model
        self.device = torch.device('cuda')

    # (required) get current hyper-parameters of this automodel
    # need to return a dictionary whose keys are the same with self.space
    def get_hyper_parameter(self):
        return {
            'layer_num': self.layer_num,
            'dim': self.dim
        }

    # (required) override to interact with num_classes
    def get_num_classes(self):
        return self.num_classes

    # (required) override to interact with num_classes
    def set_num_classes(self, n_classes):
        self.num_classes = n_classes

    # (required) override to interact with num_features
    def get_num_features(self):
        return self.num_features

    # (required) override to interact with num_features
    def set_num_features(self, n_features):
        self.num_features = n_features

    # (required) instantiate the core MLP model using corresponding hyper-parameters
    def initialize(self):
        # (required) you need to make sure the core model is named as `self.model`
        self.model = MyMLP(
            in_channels = self.num_features,
            num_classes = self.num_classes,
            layer_num = self.layer_num,
            dim = self.dim
        ).to(self.device)

        self.initialized = True

    # (required) override to create a copy of model using provided hyper-parameters
    def from_hyper_parameter(self, hp):
        # hp is a dictionary that contains keys and values corrsponding to your self.
↪space
        # in this case, it will be in form {'layer_num': XX, 'dim': XX}

        # create a new instance
        ret = self.__class__()

        # set the hyper-parameters related to dataset and device
        ret.num_classes = self.num_classes
        ret.num_features = self.num_features
        ret.device = self.device
```

(continues on next page)

(continued from previous page)

```
        # set the hyper-parameters according to hp
        ret.layer_num = hp['layer_num']
        ret.dim = hp['dim']

        # initialize it before returning
        ret.initialize()

        return ret
```

Then, you can use this node classification model as part of AutoNodeClassifier `solver`.

```
from autogl.solver import AutoNodeClassifier

solver = AutoNodeClassifier(graph_models=(MyAutoMLP(),))
```

The model for graph classification is generally the same, except that you can now also receive the `num_graph_features` (the dimension of the graph-level feature) through overriding `set_num_graph_features(self, n_graph_features)` of `BaseModel`. Also, please remember to return graph-level logits instead of node-level one in `forward()` of `model`.

Model for link prediction

For link prediction, the definition of model is a bit different with the common forward definition. You need to implement the `lp_encode(self, data)` and `lp_decode(self, x, pos_edge_index, neg_edge_index)` to interact with `LinkPredictionTrainer` and `AutoLinkPredictor`. Taking the class `MyMLP` defined above for example, if you want to perform link prediction:

```
class MyMLPForLP(torch.nn.Module):
    # num_classes is removed since it is invalid for link prediction
    def __init__(self, in_channels, layer_num, dim):
        super().__init__()
        ops = [torch.nn.Linear(in_channels, dim)]
        for i in range(layer_num - 1):
            ops.append(torch.nn.Linear(dim, dim))

        self.core = torch.nn.Sequential(*ops)

    # (required) for interaction with link prediction trainer and solver
    def lp_encode(self, data):
        return self.core(data.x)

    # (required) for interaction with link prediction trainer and solver
    def lp_decode(self, x, pos_edge_index, neg_edge_index):
        # first, get all the edge_index need calculated
        edge_index = torch.cat([pos_edge_index, neg_edge_index], dim=-1)
        # then, use dot-products to calculate logits, you can use whatever decode method␣
↪you want
        logits = (x[edge_index[0]] * x[edge_index[1]]).sum(dim=-1)
        return logits

class MyAutoMLPForLP(MyAutoMLP):
```

(continues on next page)

(continued from previous page)

```
    def initialize(self):
        # init MyMLPForLP instead of MyMLP
        self.model = MyMLPForLP(
            in_channels = self.num_features,
            layer_num = self.layer_num,
            dim = self.dim
        ).to(self.device)

        self.initialized = True
```

Model with sampling support

Towards efficient representation learning on large-scale graph, AutoGL currently support node classification using sampling techniques including node-wise sampling, layer-wise sampling, and graph-wise sampling. See more about sampling in *AutoGL Trainer*.

In order to conduct node classification using sampling technique with your custom model, further adaptation and modification are generally required. According to the Message Passing mechanism of Graph Neural Network (GNN), numerous nodes in the multi-hop neighborhood of evaluation set or test set are potentially involved to evaluate the GNN model on large-scale graph dataset. As the representations for those numerous nodes are likely to occupy large amount of computational resource, the common forwarding process is generally infeasible for model evaluation on large-scale graph. An iterative representation learning mechanism is a practical and feasible way to evaluate **Sequential Model**, which only consists of multiple sequential layers, with each layer taking a `Data` aggregate as input. The input `Data` has the same functionality with `torch_geometric.data.Data`, which conventionally provides properties `x`, `edge_index`, and optional `edge_weight`. If your custom model is composed of concatenated layers, you would better make your model inherit `ClassificationSupportedSequentialModel` to utilize the layer-wise representation learning mechanism to efficiently conduct representation learning for your custom sequential model.

```
import autogl
from autogl.module.model.base import ClassificationSupportedSequentialModel

# override Linear so that it can take graph data as input
class Linear(torch.nn.Linear):
    def forward(self, data):
        return super().forward(data.x)

class MyMLPSampling(ClassificationSupportedSequentialModel):
    def __init__(self, in_channels, num_classes, layer_num, dim):
        super().__init__()
        if layer_num == 1:
            ops = [Linear(in_channels, num_classes)]
        else:
            ops = [Linear(in_channels, dim)]
            for i in range(layer_num - 2):
                ops.append(Linear(dim, dim))
            ops.append(Linear(dim, num_classes))

        self.core = torch.nn.ModuleList(ops)

    # (required) override sequential_encoding_layers property to interact with sampling
    @property
```

(continues on next page)

(continued from previous page)

```
    def sequential_encoding_layers(self) -> torch.nn.ModuleList:
        return self.core

    # (required) define the encode logic of classification for sampling
    def cls_encode(self, data):
        # if you use sampling, the data will be passed in two possible ways,
        # you can judge it use following rules
        if hasattr(data, 'edge_indexes'):
            # the edge_indexes are a list of edge_index, one for each layer
            edge_indexes = data.edge_indexes
            edge_weights = [None] * len(self.core) if getattr(data, 'edge_weights',␣
↪None) is None else data.edge_weights
        else:
            # the edge_index and edge_weight will stay the same as default
            edge_indexes = [data.edge_index] * len(self.core)
            edge_weights = [getattr(data, 'edge_weight', None)] * len(self.core)

        x = data.x
        for i in range(len(self.core)):
            data = autogl.data.Data(x=x, edge_index=edge_indexes[i])
            data.edge_weight = edge_weights[i]
            x = self.sequential_encoding_layers[i](data)
        return x

    # (required) define the decode logic of classification for sampling
    def cls_decode(self, x):
        return torch.nn.functional.log_softmax(x)
```

3.5 AutoGL Trainer

AutoGL project use `trainer` to handle the auto-training of tasks. Currently, we support the following tasks:

- `NodeClassificationTrainer` for semi-supervised node classification
- `GraphClassificationTrainer` for supervised graph classification
- `LinkPredictionTrainer` for link prediction

3.5.1 Lazy Initialization

Similar reason to :ref:model, we also use lazy initialization for all trainers. Only (part of) the hyper-parameters will be set when `__init__()` is called. The `trainer` will have its core `model` only after `initialize()` is explicitly called, which will be done automatically in `solver` and `duplicate_from_hyper_parameter()`, after all the hyper-parameters are set properly.

3.5.2 Train and Predict

After initializing a trainer, you can train it on the given datasets.

We have given the training and testing functions for the tasks of node classification, graph classification, and link prediction up to now. You can also create your tasks following the similar patterns with ours. For training, you need to define `train_only()` and use it in `train()`. For testing, you need to define `predict_proba()` and use it in `predict()`.

The evaluation function is defined in `evaluate()`, you can use your our evaluation metrics and methods.

3.5.3 Node Classification with Sampling

According to various present studies, training with spatial sampling has been demonstrated as an efficient technique for representation learning on large-scale graph. We provide implementations for various representative sampling mechanisms including Neighbor Sampling, Layer Dependent Importance Sampling (LADIES), and GraphSAINT. With the leverage of various efficient sampling mechanisms, users can utilize this library on large-scale graph dataset, e.g. Reddit.

Specifically, as various sampling techniques generally require model to support some layer-wise processing in forwarding, now only the provided GCN and GraphSAGE models are ready for Node-wise Sampling (Neighbor Sampling) and Layer-wise Sampling (LADIES). More models and more tasks are scheduled to support sampling in future version.

- **Node-wise Sampling (GraphSAGE)** Both `GCN` and `GraphSAGE` models are supported.
- **Layer-wise Sampling (Layer Dependent Importance Sampling)** Only the `GCN` model is supported in current version.
- **Subgraph-wise Sampling (GraphSAINT)** As The GraphSAINT sampling technique have no specific requirements for model to adopt, most of the available models are feasible for adopting GraphSAINT technique. However, the prediction process is a potential bottleneck or even obstacle when the GraphSAINT technique is actually applied on large-scale graph, thus the the model to adopt is better to support layer-wise prediction, and the provided `GCN` model already meet that enhanced requirement. According to empirical experiments, the implementation of GraphSAINT now has the leverage to support an integral graph smaller than the *Flickr* graph data.

The sampling techniques can be utilized by adopting corresponding trainer `NodeClassificationGraphSAINTTrainer`, `NodeClassificationLayerDependentImportanceSamplingTrainer`, and `NodeClassificationNeighborSamplingTrainer`. You can either specify the corresponding name of trainer in YAML configuration file or instantiate the solver `AutoNodeClassifier` with the instance of specific trainer. However, please make sure to manange some key hyper-paramters properly inside the hyper-parameter space. Specifically:

For `NodeClassificationLayerDependentImportanceSamplingTrainer`, you need to set the hyper-parameter `sampled_node_sizes` properly. The space of `sampled_node_sizes` should be a list of the same size with your **Sequential Model**. For example, if you have a model with layer number 4, you need to pass the hyper-parameter space properly:

```
solver = AutoNodeClassifier(
    graph_models=(A_MODEL_WITH_4_LAYERS,),
    default_trainer='NodeClassificationLayerDependentImportanceSamplingTrainer',
    trainer_hp_space=[
        # (required) you need to set the trainer_hp_space properly.
        {
            'parameterName': 'sampled_node_sizes',
            'type': 'NUMERICAL_LIST',
```

(continues on next page)

(continued from previous page)

```
            "numericalType": "INTEGER",
            "length": 4,                    # same with the layer number of your model
            "minValue": [200,200,200,200],
            "maxValue": [1000,1000,1000,1000],
            "scalingType": "LOG"
        },
        ...
    ]
)
```

If the layer number of your model is a searchable hyper-parameters, you can also set the `cutPara` and `cutFunc` properly, to make it connected with your layer number hyper-parameters of model.

```
'''
Suppose the layer number of your model is of the following forms:
{
    'parameterName': 'layer_number',
    'type': 'INTEGER',
    'minValue': 2,
    'maxValue': 4,
    'scalingType': 'LOG'
}
'''

solver = AutoNodeClassifier(
    graph_models=(A_MODEL_WITH_DYNAMIC_LAYERS,),
    default_trainer='NodeClassificationLayerDependentImportanceSamplingTrainer',
    trainer_hp_space=[
        # (required) you need to set the trainer_hp_space properly.
        {
            'parameterName': 'sampled_node_sizes',
            'type': 'NUMERICAL_LIST',
            "numericalType": "INTEGER",
            "length": 4,                    # max length
            "cutPara": ("layer_number", ),  # link with layer_number
            "cutFunc": lambda x:x[0],       # link with layer_number
            "minValue": [200,200,200,200],
            "maxValue": [1000,1000,1000,1000],
            "scalingType": "LOG"
        },
        ...
    ]
)
```

Similarly, if you want to use `NodeClassificationNeighborSamplingTrainer`, you need to make sure setting the hyper-parameter `sampling_sizes` the same length as the layer number of your model. For example:

```
'''
Suppose the layer number of your model is of the following forms:
{
    'parameterName': 'layer_number',
    'type': 'INTEGER',
```

(continues on next page)

(continued from previous page)

```
    'minValue': 2,
    'maxValue': 4,
    'scalingType': 'LOG'
}
'''

solver = AutoNodeClassifier(
    graph_models=(A_MODEL_WITH_DYNAMIC_LAYERS,),
    default_trainer='NodeClassificationNeighborSamplingTrainer',
    trainer_hp_space=[
        # (required) you need to set the trainer_hp_space properly.
        {
            'parameterName': 'sampling_sizes',
            'type': 'NUMERICAL_LIST',
            "numericalType": "INTEGER",
            "length": 4,                    # max length
            "cutPara": ("layer_number", ),  # link with layer_number
            "cutFunc": lambda x:x[0],       # link with layer_number
            "minValue": [20,20,20,20],
            "maxValue": [100,100,100,100],
            "scalingType": "LOG"
        },
        ...
    ]
)
```

You can also pass a trainer inside model list directly. A brief example is demonstrated as follows:

```
ladies_sampling_trainer = NodeClassificationLayerDependentImportanceSamplingTrainer(
    model='gcn', num_features=dataset.num_features, num_classes=dataset.num_classes, ...
)

ladies_sampling_trainer.hyper_parameter_space = [
    # (required) you need to set the trainer_hp_space properly.
    {
        'parameterName': 'sampled_node_sizes',
        'type': 'NUMERICAL_LIST',
        "numericalType": "INTEGER",
        "length": 4,                    # max length
        "cutPara": ("num_layers", ),    # link with layer_number
        "cutFunc": lambda x:x[0],       # link with layer_number
        "minValue": [200,200,200,200],
        "maxValue": [1000,1000,1000,1000],
        "scalingType": "LOG"
    },
    ...
]

AutoNodeClassifier(graph_models=(ladies_sampling_trainer,), ...)
```

3.6 Hyper Parameter Optimization

We support black box hyper parameter optimization in variant search space.

3.6.1 Search Space

Three types of search space are supported, use `dict` in python to define your search space. For numerical list search space. You can either assign a fixed length for the list, if so, you need not provide `cutPara` and `cutFunc`. Or you can let HPO cut the list to a certain length which is dependent on other parameters. You should provide those parameters' names in `curPara` and the function to calculate the cut length in "cutFunc".

```
# numerical search space:
{
    "parameterName": "xxx",
    "type": "DOUBLE" / "INTEGER",
    "minValue": xx,
    "maxValue": xx,
    "scalingType": "LINEAR" / "LOG"
}

# numerical list search space:
{
    "parameterName": "xxx",
    "type": "NUMERICAL_LIST",
    "numericalType": "DOUBLE" / "INTEGER",
    "length": 3,
    "cutPara": ("para_a", "para_b"),
    "cutFunc": lambda x: x[0] - 1,
    "minValue": [xx,xx,xx],
    "maxValue": [xx,xx,xx],
    "scalingType": "LINEAR" / "LOG"
}

# categorical search space:
{
    "parameterName": xxx,
    "type": "CATEGORICAL"
    "feasiblePoints": [a,b,c]
}

# fixed parameter as search space:
{
    "parameterName": xxx,
    "type": "FIXED",
    "value": xxx
}
```

How given HPO algorithms support search space is listed as follows:

Algorithm	numerical	numerical list	categorical	fixed
Grid			✓	✓
Random	✓	✓	✓	✓
Anneal	✓	✓	✓	✓
Bayes	✓	✓	✓	✓
TPE[1]	✓	✓	✓	✓
CMAES[2]	✓	✓	✓	✓
MOCMAES[3]	✓	✓	✓	✓
Quasi random[4]	✓	✓	✓	✓
AutoNE[5]	✓	✓	✓	✓

3.6.2 Add Your HPOptimizer

If you want to add your own HPOptimizer, the only thing you should do is finishing `optimize` function in you HPOptimizer:

```python
# For example, create a random HPO by yourself
import random
from autogl.module.hpo.base import BaseHPOptimizer
class RandomOptimizer(BaseHPOptimizer):
    # Get essential parameters at initialization
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.max_evals = kwargs.get("max_evals", 2)

    # The most important thing you should do is completing optimization function
    def optimize(self, trainer, dataset, time_limit=None, memory_limit=None):
        # 1. Get the search space from trainer.
        space = trainer.hyper_parameter_space + trainer.model.hyper_parameter_space
        # optional: use self._encode_para (in BaseOptimizer) to pretreat the space
        # If you use _encode_para, the NUMERICAL_LIST will be spread to DOUBLE or INTEGER, LOG scaling type will be changed to LINEAR, feasible points in CATEGORICAL will be changed to discrete numbers.
        # You should also use _decode_para to transform the types of parameters back.
        current_space = self._encode_para(space)

        # 2. Define your function to get the performance.
        def fn(dset, para):
            current_trainer = trainer.duplicate_from_hyper_parameter(para)
            current_trainer.train(dset)
            loss, self.is_higher_better = current_trainer.get_valid_score(dset)
            # For convenience, we change the score which is higher better to negative, then we should only minimize the score.
```

(continues on next page)

[1] Bergstra, James S., et al. "Algorithms for hyper-parameter optimization." Advances in neural information processing systems. 2011.

[2] Arnold, Dirk V., and Nikolaus Hansen. "Active covariance matrix adaptation for the (1+ 1)-CMA-ES." Proceedings of the 12th annual conference on Genetic and evolutionary computation. 2010.

[3] Voß, Thomas, Nikolaus Hansen, and Christian Igel. "Improved step size adaptation for the MO-CMA-ES." Proceedings of the 12th annual conference on Genetic and evolutionary computation. 2010.

[4] Bratley, Paul, Bennett L. Fox, and Harald Niederreiter. "Programs to generate Niederreiter's low-discrepancy sequences." ACM Transactions on Mathematical Software (TOMS) 20.4 (1994): 494-495.

[5] Tu, Ke, et al. "Autone: Hyperparameter optimization for massive network embedding." Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2019.

(continued from previous page)

```
            if self.is_higher_better:
                loss = -loss
            return current_trainer, loss

        # 3. Define the how to get HP suggestions, it should return a parameter dict.␣
↪You can use history trials to give new suggestions
        def get_random(history_trials):
            hps = {}
            for para in current_space:
                # Because we use _encode_para function before, we should only deal with␣
↪DOUBLE, INTEGER and DISCRETE
                if para["type"] == "DOUBLE" or para["type"] == "INTEGER":
                    hp = random.random() * (para["maxValue"] - para["minValue"]) + para[
↪"minValue"]
                    if para["type"] == "INTEGER":
                        hp = round(hp)
                    hps[para["parameterName"]] = hp
                elif para["type"] == "DISCRETE":
                    feasible_points = para["feasiblePoints"].split(",")
                    hps[para["parameterName"]] = random.choice(feasible_points)
            return hps

        # 4. Run your algorithm. For each turn, get a set of parameters according to␣
↪history information and evaluate it.
        best_trainer, best_para, best_perf = None, None, None
        self.trials = []
        for i in range(self.max_evals):
            # in this example, we don't need history trails. Since we pass None to␣
↪history_trails
            new_hp = get_random(None)
            # optional: if you use _encode_para, use _decode_para as well. para_for_
↪trainer undos all transformation in _encode_para, and turns double parameter to␣
↪interger if needed. para_for_hpo only turns double parameter to interger.
            para_for_trainer, para_for_hpo = self._decode_para(new_hp)
            current_trainer, perf = fn(dataset, para_for_trainer)
            self.trials.append((para_for_hpo, perf))
            if not best_perf or perf < best_perf:
                best_perf = perf
                best_trainer = current_trainer
                best_para = para_for_trainer

        # 5. Return the best trainer and parameter.
        return best_trainer, best_para
```

3.7 Neural Architecture Search

We support different neural architecture search algorithm in variant search space. To be more flexible, we modulize NAS process with three part: algorithm, space and estimator. Different models in different parts can be composed in some certain constrains. If you want to design your own NAS process, you can change any of those parts according to your demand.

3.7.1 Usage

You can directly enable architecture search for node classification tasks by passing the algorithms, spaces and estimators to solver. Following shows an example:

```
# Use graphnas to solve cora
from autogl.datasets import build_dataset_from_name
from autogl.solver import AutoNodeClassifier

solver = AutoNodeClassifier(
    feature = 'PYGNormalizeFeatures',
    graph_models = (),
    hpo = 'tpe',
    ensemble = None,
    nas_algorithms='rl',
    nas_spaces='graphnasmacro',
    nas_estimators='scratch'
)

cora = build_dataset_from_name('cora')
solver.fit(cora)
```

The code above will first find the best architecture in space `graphnasmacro` using `rl` search algorithm. Then the searched architecture will be further optimized through hyperparameter-optimization `tpe`.

Note: The `graph_models` argument is not conflict with nas module. You can set `graph_models` to other hand-crafted models beside the ones found by nas. Once the architectures are derived from nas module, they act in the same way as hand-crafted models directly passed through graph_models.

3.7.2 Search Space

The space definition is base on mutable fashion used in NNI, which is defined as a model inheriting BaseSpace There are mainly two ways to define your search space, one can be performed with one-shot fashion while the other cannot. Currently, we support following search space:

Space	Description
`singlepath`[4]	Architectures with several sequential layers with each layer choosing only one path
`graphnas`[1]	The graph nas micro search space designed for fully supervised node classification models
`graphnasmacro`[1]	The graph nas macro search space designed for semi-superwised node classification models

[4] Guo, Zichao, et al. “Single Path One-Shot Neural Architecture Search with Uniform Sampling.” European Conference on Computer Vision, 2019, pp. 544–560.

[1] Gao, Yang, et al. “Graph neural architecture search.” IJCAI. Vol. 20. 2020.

You can also define your own nas search space. If you need one-shot fashion, you should use the function `setLayerChoice` and `setInputChoice` to construct the super network. Here is an example.

```python
# For example, create an NAS search space by yourself
from autogl.module.nas.space.base import BaseSpace
from autogl.module.nas.space.operation import gnn_map
class YourOneShotSpace(BaseSpace):
    # Get essential parameters at initialization
    def __init__(self, input_dim = None, output_dim = None):
        super().__init__()
        # must contain input_dim and output_dim in space, or you can initialize these two parameters in function `instantiate`
        self.input_dim = input_dim
        self.output_dim = output_dim

    # Instantiate the super network
    def instantiate(self, input_dim, output_dim):
        # must call super in this function
        super().instantiate()
        self.input_dim = input_dim or self.input_dim
        self.output_dim = output_dim or self.output_dim
        # define two layers with order 0 and 1
        self.layer0 = self.setLayerChoice(0, [gnn_map(op,self.input_dim,self.output_dim)for op in ['gcn', 'gat']])
        self.layer1 = self.setLayerChoice(1, [gnn_map(op,self.input_dim,self.output_dim)for op in ['gcn', 'gat']])
        # define an input choice two choose from the result of the two layer
        self.input_layer = self.setInputChoice(2, n_candidates = 2)

    # Define the forward process
    def forward(self, data):
        x, edges = data.x, data.edge_index
        x_0 = self.layer0(x, edges)
        x_1 = self.layer1(x, edges)
        y = self.input_layer([x_0, x_1])
        return y

    # For one-shot fashion, you can directly use following scheme in ``parse_model``
    def parse_model(self, selection, device) -> BaseModel:
        return self.wrap(device).fix(selection)
```

Also, you can use the way which does not support one shot fashion. In this way, you can directly copy you model with few changes. But you can only use sample-based search strategy.

```python
# For example, create an NAS search space by yourself
from autogl.module.nas.space.base import BaseSpace, map_nn
from autogl.module.nas.space.operation import gnn_map
# here we search from three types of graph convolution with `head` as a parameter
# we should search `heads` at the same time with the convolution
from torch_geometric.nn import GATConv, FeaStConv, TransformerConv
class YourNonOneShotSpace(BaseSpace):
    # Get essential parameters at initialization
    def __init__(self, input_dim = None, output_dim = None):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        # must contain input_dim and output_dim in space, or you can initialize these␣
↪two parameters in function `instantiate`
        self.input_dim = input_dim
        self.output_dim = output_dim

    # Instantiate the super network
    def instantiate(self, input_dim, output_dim):
        # must call super in this function
        super().instantiate()
        self.input_dim = input_dim or self.input_dim
        self.output_dim = output_dim or self.output_dim
        # set your choices as LayerChoices
        self.choice0 = self.setLayerChoice(0, map_nn(["gat", "feast", "transformer"]),␣
↪key="conv")
        self.choice1 = self.setLayerChoice(1, map_nn([1, 2, 4, 8]), key="head")

    # You do not need to define forward process here
    # For non-one-shot fashion, you can directly return your model based on the choices
    # ``YourModel`` must inherit BaseSpace.
    def parse_model(self, selection, device) -> BaseModel:
        model = YourModel(selection, self.input_dim, self.output_dim).wrap(device)
        return model

# YourModel can be defined as follows
class YourModel(BaseSpace):
    def __init__(self, selection, input_dim, output_dim):
        self.input_dim = input_dim
        self.output_dim = output_dim
        if selection["conv"] == "gat":
            conv = GATConv
        elif selection["conv"] == "feast":
            conv = FeaStConv
        elif selection["conv"] == "transformer":
            conv = TransformerConv
        self.layer = conv(input_dim, output_dim, selection["head"])

    def forward(self, data):
        x, edges = data.x, data.edge_index
        y = self.layer(x, edges)
        return y
```

3.7.3 Performance Estimator

The performance estimator estimates the performance of an architecture. Currently we support following estimators:

Estimator	Description
oneshot	Directly evaluating the given models without training
scratch	Train the models from scratch and then evaluate them

You can also write your own estimator. Here is an example of estimating an architecture without training (used in one-shot space).

```
# For example, create an NAS estimator by yourself
from autogl.module.nas.estimator.base import BaseEstimator
class YourOneShotEstimator(BaseEstimator):
    # The only thing you should do is defining ``infer`` function
    def infer(self, model: BaseSpace, dataset, mask="train"):
        device = next(model.parameters()).device
        dset = dataset[0].to(device)
        # Forward the architecture
        pred = model(dset)[getattr(dset, f"{mask}_mask")]
        y = dset.y[getattr(dset, f'{mask}_mask')]
        # Use default loss function and metrics to evaluate the architecture
        loss = getattr(F, self.loss_f)(pred, y)
        probs = F.softmax(pred, dim = 1)
        metrics = [eva.evaluate(probs, y) for eva in self.evaluation]
        return metrics, loss
```

3.7.4 Search Strategy

The space strategy defines how to find an architecture. We currently support following search strategies:

Strategy	Description
random	Random search by uniform sampling
rl[?]	Use rl as architecture generator agent
enas[2]	efficient neural architecture search
darts[3]	differentiable neural architecture search

Sample-based strategy without weight sharing is simpler than strategies with weight sharing. We show how to define your strategy here with DFS as an example. If you want to define more complex strategy, you can refer to Darts, Enas or other strategies in NNI.

```
from autogl.module.nas.algorithm.base import BaseNAS
class RandomSearch(BaseNAS):
    # Get the number of samples at initialization
    def __init__(self, n_sample):
        super().__init__()
        self.n_sample = n_sample

    # The key process in NAS algorithm, search for an architecture given space, dataset␣
↪and estimator
    def search(self, space: BaseSpace, dset, estimator):
        self.estimator=estimator
        self.dataset=dset
        self.space=space

        self.nas_modules = []
        k2o = get_module_order(self.space)
        # collect all mutables in the space
        replace_layer_choice(self.space, PathSamplingLayerChoice, self.nas_modules)
```

(continues on next page)

[2] Pham, Hieu, et al. "Efficient neural architecture search via parameters sharing." International Conference on Machine Learning. PMLR, 2018.

[3] Liu, Hanxiao, Karen Simonyan, and Yiming Yang. "DARTS: Differentiable Architecture Search." International Conference on Learning Representations. 2018.

(continued from previous page)

```
        replace_input_choice(self.space, PathSamplingInputChoice, self.nas_modules)
        # sort all mutables with given orders
        self.nas_modules = sort_replaced_module(k2o, self.nas_modules)
        # get a dict cantaining all chioces
        selection_range={}
        for k,v in self.nas_modules:
            selection_range[k]=len(v)
        self.selection_dict=selection_range

        arch_perfs=[]
        # define DFS process
        self.selection = {}
        last_k = list(self.selection_dict.keys())[-1]
        def dfs():
            for k,v in self.selection_dict.items():
                if not k in self.selection:
                    for i in range(v):
                        self.selection[k] = i
                        if k == last_k:
                            # evaluate an architecture
                            self.arch=space.parse_model(self.selection,self.device)
                            metric,loss=self._infer(mask='val')
                            arch_perfs.append([metric, self.selection.copy()])
                        else:
                            dfs()
                    del self.selection[k]
                    break
        dfs()

        # get the architecture with the best performance
        selection=arch_perfs[np.argmax([x[0] for x in arch_perfs])][1]
        arch=space.parse_model(selection,self.device)
        return arch
```

Different search strategies should be combined with different search spaces and estimators in usage.

Sapce	single path	GraphNAS[1]	GraphNAS-macro[1]
Random	✓	✓	✓
RL	✓	✓	✓
GraphNAS[?]	✓	✓	✓
ENAS[?]	✓		
DARTS[Page 26, 3]	✓		

Estimator	one-shot	Train
Random		✓
RL		✓
GraphNAS[?]		✓
ENAS[?]	✓	
DARTS[Page 26, 3]	✓	

3.8 Ensemble

We currently support voting and stacking methods.

3.8.1 Voting

A voter essentially constructs a weighted sum of the predictions of base learners. Given an evaluation metric, the weights of base learners are specified in some way to maximize the validation score.

We adopt Rich Caruana's method for weight specification. This method first finds a collection of (possibly redundant) base learners with equal weights via a greedy search, then specifies the weights in the voter by the number of occurrence in the collection.

You can customize your own weight specification method by overwriting the `_specify_weights` method.

```
# An example : use equal weights for all base learners.
class EqualWeightVoting(Voting):
    def _specify_weights(self, predictions, label, feval):
        return np.ones(self.n_models)/self.n_models
        # just allocate the same weight for each base learner
```

3.8.2 Stacking

A stacker trains a meta-model with the predictions of base learners as input to find an optimal combination of these base learners.

Currently we support generalized linear model (GLM) and gradient boosting model (GBM) as the meta-model.

3.8.3 Create a New Ensembler

You can create your own ensembler by inheriting the base ensembler, and overloading methods `fit` and `ensemble`.

```
# An example : use the currently available best model.
from autogl.module.ensemble.base import BaseEnsembler
import numpy as np
class BestModel(BaseEnsembler):
    def fit(self, predictions, label, identifiers, feval):
        if not isinstance(feval, list):
            feval = [feval]
        scores = np.array([feval[0].evaluate(pred, label) for pred in predictions]) * (1␣
↪if feval[0].is_higher_better else -1)
        self.scores = dict(zip(identifiers, scores)) # record validation score of base␣
↪learners
        ensemble_pred = predictions[np.argmax(scores)]
        return [fx.evaluate(ensemble_pred, label) for fx in feval]

    def ensemble(self, predictions, identifiers):
        best_idx = np.argmax([self.scores[model_name] for model_name in identifiers]) #␣
↪choose the currently best model in the identifiers
        return predictions[best_idx]
```

3.9 AutoGL Solver

AutoGL project use `solver` to handle the auto-solvation of tasks. Currently, we support the following tasks:

- `AutoNodeClassifier` for semi-supervised node classification
- `AutoGraphClassifier` for supervised graph classification
- `AutoLinkPredictor` for link prediction

3.9.1 Initialization

A solver can either be initialized from its `__init__()` or from a config dictionary or file.

Initialize from `__init__()`

If you want to build a solver by `__init__()`, you need to pass the key modules to it. You can either pass the keywords of corresponding modules or the initialized instances:

```
from autogl.solver import AutoNodeClassifier

# 1. initialize from keywords
solver = AutoNodeClassifier(
    feature_module='deepgl',
    graph_models=['gat','gcn'],
    hpo_module='anneal',
    ensemble_module='voting',
    device='auto'
)

# 2. initialize using instances
from autogl.module import AutoFeatureEngineer, AutoGCN, AutoGAT, AnnealAdvisorHPO, Voting
solver = AutoNodeClassifier(
    feature_module=AutoFeatureEngineer(),
    graph_models=[AutoGCN(device='cuda'), AutoGAT(device='cuda')],
    hpo_module=AnnealAdvisorHPO(max_evals=10),
    ensemble_module=Voting(size=2),
    device='cuda'
)
```

Where, the argument `device` means where to perform the training and searching, by setting to `auto`, the `cuda` is used when it is available.

If you want to disable one module, you can set it to `None`:

```
solver = AutoNodeClassifier(feature_module=None, hpo_module=None, ensemble_module=None)
```

You can also pass some important arguments of modules directly to solver, which will automatically be set for you:

```
solver = AutoNodeClassifier(hpo_module='anneal', max_evals=10)
```

Refer to *autogl.solver* for more details of argument default value or important argument lists.

Initialize from config dictionary or file

You can also initialize a solver directly from a config dictionary or file. Currently, the AutoGL solver supports config file type of `yaml` or `json`. You need to use `from_config()` when you want to initialize in this way:

```
# initialize from config file
path_to_config = 'your/path/to/config'
solver = AutoNodeClassifier.from_config(path_to_config)

# initialize from a dictionary
config = {
    'models':{'gcn': None, 'gat': None},
    'hpo': {'name': 'tpe', 'max_evals': 10},
    'ensemble': {'name': 'voting', 'size': 2}
}
solver = AutoNodeClassifier.from_config(config)
```

Refer to the config dictionary description *Config structure* for more details.

3.9.2 Optimization

After initializing a solver, you can optimize it on the given datasets (please refer to *AutoGL Dataset* and *autogl.datasets* for creating datasets).

You can use `fit()` or `fit_predict()` to perform optimization, which shares similar argument lists:

```
# load your dataset here
dataset = some_dataset()
solver.fit(dataset, inplace=True)
```

The inplace argument is used for saving memory if set to `True`. It will modify your dataset in an inplace manner during feature engineering.

You can also specify the `train_split` and `val_split` arguments to let solver auto-split the given dataset. If these arguments are given, the split dataset will be used instead of the default split specified by the dataset provided. All the models will be trained on `train dataset`. Their hyperparameters will be optimized based on the performance of `valid dataset`, as well as the final ensemble method. For example:

```
# split 0.2 of total nodes/graphs for train and 0.4 of nodes/graphs for validation,
# the rest 0.4 is left for test.
solver.fit(dataset, train_split=0.2, val_split=0.4)

# split 600 nodes/graphs for train and 400 nodes/graphs for validation,
# the rest nodes are left for test.
solver.fit(dataset, train_split=600, val_split=400)
```

For the node classification problem, we also support balanced sampling of train and valid: force the number of sampled nodes in different classes to be the same. The balanced mode can be turned on by setting `balanced=True` in `fit()`, which is by default set to `True`.

Note: Solver will maintain the models with the best hyper-parameter of each model architecture you pass to solver (the `graph_models` argument when initialized). The maintained models will then be ensembled by ensemble module.

After `fit()`, solver maintains the performances of every single model and the ensemble model in one leaderboard instance. You can output the performances on valid dataset by:

```
# get current leaderboard of the solver
lb = solver.get_leaderboard()
# show the leaderboard info
lb.show()
```

You can refer to the leaderboard documentation in *autogl.solver* for more usage.

3.9.3 Prediction

After optimized on the given dataset, you can make predictions using the fitted `solver`.

Prediction using ensemble

You can use the ensemble model constructed by solver to make the prediction, which is recommended and is the default choice:

```
solver.predict()
```

If you do not pass any dataset, the dataset during fitting will be used to give the prediction.

You can also pass the dataset when predicting, please make sure the `inplaced` argument is properly set.

```
solver.predict(dataset, inplaced=True, inplace=True)
```

The `predict()` function also has `inplace` argument, which is the same as the one in `fit()`. As for the `inplaced`, it means whether the passed dataset is already modified inplace or not (probably by `fit()` function). If you use `fit()` before, please make the `inplaced` of `predict()` stay the same with `inplace` in `fit()`.

Prediction using one single model

You can also make the prediction using the best single model the solver maintains by:

```
solver.predict(use_ensemble=False, use_best=True)
```

Also, you can name the single model maintained by solver to make predictions.

```
solver.predict(use_ensemble=False, use_best=False, name=the_name_of_model)
```

The names of models can be derived by calling `solver.trained_models.keys()`, which is the same as the names maintained by the leaderboard of solver.

Note: By default, solver will only make predictions on the `test` split of given datasets. Please make sure the passed dataset has the `test` split when making predictions. You can also change the default prediction split by setting argument `mask` to `train` or `valid`.

3.9.4 Appendix

Config structure

The structure of the config file or config is introduced here. The config should be a dict, with five optional keys, namely `feature`, `models`, `trainer`, `hpo` and `ensemble`. You can simply do not add one field if you want to use the default option. The default value of each module is the same as the one in `__init__()`.

For key `feature`, `hpo` and `ensemble`, their corresponding values are all dictionaries, which contains one must key `name` and other arguments when initializing the corresponding modules. The value of key `name` specifies which algorithm should be used, where `None` can be passed if you do not want to enable the module. Other arguments are used to initialize the specified algorithm.

For key `trainer`, you should specify the hyperparameter space of trainer. See *AutoGL Trainer* or *autogl.module.train* for the detailed hyperparameter space of different trainers.

For key `models`, the value is another dictionary with its keys being models that need optimized and the corresponding values being the hyperparameter space of that model. See *AutoGL Model* or *autogl.module.model* for the detailed hyperparameter space of different models.

Below shows some examples of the config dictionary.

```
config_for_node_classification = {
    'feature': {
        'name': 'deepgl',       # name of auto feature module
        # following are the deepgl specified auto feature engineer arguments
        'fixlen': 100,
        'max_epoch': 5
    },
    'models': {
        'gcn':
        # specify the hp space of gcn
        [
            {'parameterName': 'num_layers', 'type': 'DISCRETE', 'feasiblePoints': '2,3,4
↪'},
            {'parameterName': 'hidden', 'type': 'NUMERICAL_LIST', 'numericalType':
↪'INTEGER', 'length': 3,
                'minValue': [8, 8, 8], 'maxValue': [64, 64, 64], 'scalingType': 'LOG'},
            {'parameterName': 'dropout', 'type': 'DOUBLE', 'maxValue': 0.9, 'minValue':␣
↪0.1, 'scalingType': 'LINEAR'},
            {'parameterName': 'act', 'type': 'CATEGORICAL', 'feasiblePoints': ['leaky_
↪relu', 'relu', 'elu', 'tanh']}
        ],
        'gat': None,             # set to None to use default hp space
        'gin': None
    }
    'trainer': [
        # trainer hp space
        {'parameterName': 'max_epoch', 'type': 'INTEGER', 'maxValue': 300, 'minValue':␣
↪10, 'scalingType': 'LINEAR'},
        {'parameterName': 'early_stopping_round', 'type': 'INTEGER', 'maxValue': 30,
↪'minValue': 10, 'scalingType': 'LINEAR'},
        {'parameterName': 'lr', 'type': 'DOUBLE', 'maxValue': 0.001, 'minValue': 0.0001,
↪'scalingType': 'LOG'},
        {'parameterName': 'weight_decay', 'type': 'DOUBLE', 'maxValue': 0.005, 'minValue
↪': 0.0005, 'scalingType': 'LOG'}
```

(continues on next page)

(continued from previous page)

```
    ],
    'hpo': {
        'name': 'autone',       # name of hpo module
        # following are the autone specified auto hpo arguments
        'max_evals': 10,
        'subgraphs': 10,
        'sub_evals': 5
    },
    'ensemble': {
        'name': 'voting',       # name of ensemble module
        # following are the voting specified auto ensemble arguments
        'size': 2
    }
}

config_for_graph_classification = {
    'feature': None,            # set to None to disable this module
    # do not add field `model` to use default settings of solver
    'trainer': [
        # trainer hp space
        {'parameterName': 'max_epoch', 'type': 'INTEGER', 'maxValue': 300, 'minValue':␣
↪10, 'scalingType': 'LINEAR'},
        {'parameterName': 'batch_size', 'type': 'INTEGER', 'maxValue': 128, 'minValue':␣
↪32, 'scalingType': 'LOG'},
        {'parameterName': 'early_stopping_round', 'type': 'INTEGER', 'maxValue': 30,
↪'minValue': 10, 'scalingType': 'LINEAR'},
        {'parameterName': 'lr', 'type': 'DOUBLE', 'maxValue': 1e-3, 'minValue': 1e-4,
↪'scalingType': 'LOG'},
        {'parameterName': 'weight_decay', 'type': 'DOUBLE', 'maxValue': 5e-3, 'minValue
↪': 5e-4, 'scalingType': 'LOG'},
    ],
    'hpo': {
        'name': 'random',       # name of hpo module
        # following are the random specified auto hpo arguments
        'max_evals': 10
    },
    'ensemble': None            # set to None to disable this module
}
```

3.10 autogl.data

class `autogl.data.`**`Batch`**(*batch=None*, ***kwargs*)

A plain old python object modeling a batch of graphs as one big (dicconnected) graph. With `cogdl.data.Data` being the base class, all its methods can also be used here. In addition, single graphs can be reconstructed via the assignment vector `batch`, which maps each node to its respective graph identifier.

`cumsum`(*key*, *item*)

If `True`, the attribute `key` with content `item` should be added up cumulatively before concatenated together.

Note: This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

static from_data_list(*data_list*, *follow_batch=[]*)

Constructs a batch object from a python list holding `torch_geometric.data.Data` objects. The assignment vector `batch` is created on the fly. Additionally, creates assignment batch vectors for each key in `follow_batch`.

property num_graphs

Returns the number of graphs in the batch.

to_data_list()

Reconstructs the list of `torch_geometric.data.Data` objects from the batch object. The batch object must have been created via *`from_data_list()`* in order to be able reconstruct the initial objects.

class autogl.data.**Data**(*x=None*, *edge_index=None*, *edge_attr=None*, *y=None*, *pos=None*)

A plain old python object modeling a single graph with various (optional) attributes:

Parameters

- **x** (*Tensor, optional*) – Node feature matrix with shape `[num_nodes, num_node_features]`. (default: `None`)
- **edge_index** (*LongTensor, optional*) – Graph connectivity in COO format with shape `[2, num_edges]`. (default: `None`)
- **edge_attr** (*Tensor, optional*) – Edge feature matrix with shape `[num_edges, num_edge_features]`. (default: `None`)
- **y** (*Tensor, optional*) – Graph or node targets with arbitrary shape. (default: `None`)
- **pos** (*Tensor, optional*) – Node position matrix with shape `[num_nodes, num_dimensions]`. (default: `None`)

The data object is not restricted to these attributes and can be extented by any other additional data.

__call__(**keys*)

Iterates over all attributes `*keys` in the data, yielding their attribute names and content. If `*keys` is not given this method will iterative over all present attributes.

__contains__(*key*)

Returns `True`, if the attribute `key` is present in the data.

__getitem__(*key*)

Gets the data of the attribute `key`.

__inc__(*key*, *value*)

"Returns the incremental count to cumulatively increase the value of the next attribute of `key` when creating batches.

Note: This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

__iter__()

Iterates over all present attributes in the data, yielding their attribute names and content.

__len__()

Returns the number of all present attributes.

__setitem__(*key*, *value*)

Sets the attribute `key` to `value`.

`apply`(*func*, **keys*)

Applies the function `func` to all attributes `*keys`. If `*keys` is not given, `func` is applied to all present attributes.

`cat_dim`(*key*, *value*)

Returns the dimension in which the attribute `key` with content `value` gets concatenated when creating batches.

Note: This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

`contiguous`(**keys*)

Ensures a contiguous memory layout for all attributes `*keys`. If `*keys` is not given, all present attributes are ensured to have a contiguous memory layout.

`static from_dict`(*dictionary*)

Creates a data object from a python dictionary.

`get_label_number()`

Get the number of labels in this dataset as dict.

`is_coalesced()`

Returns `True`, if edge indices are ordered and do not contain duplicate entries.

`property keys`

Returns all names of graph attributes.

`property num_edges`

Returns the number of edges in the graph.

`property num_features`

Returns the number of features per node in the graph.

`random_splits_mask`(*train_ratio*, *val_ratio*, *seed=None*)

If the data has masks for train/val/test, return the splits with specific ratio.

Parameters

- **train_ratio** (*float*) – the portion of data that used for training.
- **val_ratio** (*float*) – the portion of data that used for validation.
- **seed** (*int*) – random seed for splitting dataset.

`random_splits_mask_class`(*num_train_per_class*, *num_val*, *num_test*, *seed=None*)

If the data has masks for train/val/test, return the splits with specific number of samples from every class for training.

Parameters

- **num_train_per_class** (*int*) – the number of samples from every class used for training.
- **num_val** (*int*) – the total number of nodes that used for validation.
- **num_test** (*int*) – the total number of nodes that used for testing.
- **seed** (*int*) – random seed for splitting dataset.

`random_splits_nodes`(*train_ratio*, *val_ratio*, *seed=None*)

If the data uses id of nodes for train/val/test, return the splits with specific ratio.

Parameters

- **train_ratio** (*float*) – the portion of data that used for training.
- **val_ratio** (*float*) – the portion of data that used for validation.
- **seed** (*int*) – random seed for splitting dataset.

`random_splits_nodes_class`(*num_train_per_class*, *num_val*, *num_test*, *seed=None*)

If the data uses id of nodes for train/val/test, return the splits with specific number of samples from every class for training.

Parameters

- **num_train_per_class** (*int*) – the number of samples from every class used for training.
- **num_val** (*int*) – the total number of nodes that used for validation.
- **num_test** (*int*) – the total number of nodes that used for testing.
- **seed** (*int*) – random seed for splitting dataset.

`to`(*device*, **keys*)

Performs tensor dtype and/or device conversion to all attributes `*keys`. If `*keys` is not given, the conversion is applied to all present attributes.

class autogl.data.**DataListLoader**(*dataset*, *batch_size=1*, *shuffle=True*, ***kwargs*)

Data loader which merges data objects from a `cogdl.data.dataset` to a python list.

Note: This data loader should be used for multi-gpu support via `cogdl.nn.DataParallel`.

Parameters

- **dataset** (Dataset) – The dataset from which to load the data.
- **batch_size** (*int, optional*) – How may samples per batch to load. (default: `1`)
- **shuffle** (*bool, optional*) – If set to `True`, the data will be reshuffled at every epoch (default: `True`)

class autogl.data.**DataLoader**(*dataset*, *batch_size=1*, *shuffle=True*, ***kwargs*)

Data loader which merges data objects from a `cogdl.data.dataset` to a mini-batch.

Parameters

- **dataset** (Dataset) – The dataset from which to load the data.
- **batch_size** (*int, optional*) – How may samples per batch to load. (default: `1`)
- **shuffle** (*bool, optional*) – If set to `True`, the data will be reshuffled at every epoch (default: `True`)

class autogl.data.**Dataset**(*root*, *transform=None*, *pre_transform=None*, *pre_filter=None*)

Dataset base class for creating graph datasets. See here for the accompanying tutorial.

Parameters

- **root** (*string*) – Root directory where the dataset should be saved.
- **transform** (*callable, optional*) – A function/transform that takes in an `cogdl.data.Data` object and returns a transformed version. The data object will be transformed before every access. (default: `None`)

- **pre_transform** (*callable, optional*) – A function/transform that takes in an `cogdl.data.Data` object and returns a transformed version. The data object will be transformed before being saved to disk. (default: `None`)
- **pre_filter** (*callable, optional*) – A function that takes in an `cogdl.data.Data` object and returns a boolean value, indicating whether the data object should be included in the final dataset. (default: `None`)

`__getitem__`(*idx*)
Gets the data object at index `idx` and transforms it (in case a `self.transform` is given).

`__len__`()
The number of examples in the dataset.

`download`()
Downloads the dataset to the `self.raw_dir` folder.

`get`(*idx*)
Gets the data object at index `idx`.

property **`get_label_number`**
Get the number of labels in this dataset as dict.

property **`num_features`**
Returns the number of features per node in the graph.

`process`()
Processes the dataset to the `self.processed_dir` folder.

property **`processed_file_names`**
The name of the files to find in the `self.processed_dir` folder in order to skip the processing.

property **`processed_paths`**
The filepaths to find in the `self.processed_dir` folder in order to skip the processing.

property **`raw_file_names`**
The name of the files to find in the `self.raw_dir` folder in order to skip the download.

property **`raw_paths`**
The filepaths to find in order to skip the download.

class `autogl.data.`**`DenseDataLoader`**(*dataset*, *batch_size=1*, *shuffle=True*, ***kwargs*)
Data loader which merges data objects from a `cogdl.data.dataset` to a mini-batch.

> **Note:** To make use of this data loader, all graphs in the dataset needs to have the same shape for each its attributes. Therefore, this data loader should only be used when working with *dense* adjacency matrices.

Parameters

- **dataset** (`Dataset`) – The dataset from which to load the data.
- **batch_size** (*int, optional*) – How may samples per batch to load. (default: `1`)
- **shuffle** (*bool, optional*) – If set to `True`, the data will be reshuffled at every epoch (default: `True`)

`autogl.data.`**`download_url`**(*url*, *folder*, *name=None*, *log=True*)
Downloads the content of an URL to a specific folder.

Parameters

- **url** (*string*) – The url.
- **folder** (*string*) – The folder.
- **log** (*bool, optional*) – If False, will not print anything to the console. (default: True)

autogl.data.**extract_tar**(*path*, *folder*, *mode='r:gz'*, *log=True*)

Extracts a tar archive to a specific folder.

Parameters

- **path** (*string*) – The path to the tar archive.
- **folder** (*string*) – The folder.
- **mode** (*string, optional*) – The compression mode. (default: "r:gz")
- **log** (*bool, optional*) – If False, will not print anything to the console. (default: True)

autogl.data.**extract_zip**(*path*, *folder*, *log=True*)

Extracts a zip archive to a specific folder.

Parameters

- **path** (*string*) – The path to the tar archive.
- **folder** (*string*) – The folder.
- **log** (*bool, optional*) – If False, will not print anything to the console. (default: True)

3.11 autogl.datasets

We integrate the datasets from PyTorch Geometric, CogDL and OGB. We also list some datasets from *CogDL* for simplicity.

3.12 autogl.module.feature

Several feature engineering operations are collected manually, or from PyTorch Geometric, NetworkX, etc.

3.13 autogl.module.model

3.14 autogl.module.train

3.15 autogl.module.hpo

3.16 autogl.module.nas

3.17 autogl.module.ensemble

3.18 autogl.solver

CHAPTER FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX