Graphormer

Release 1.0

MSRA Graphormer Team

Aug 12, 2022

GETTING STARTED

Graphormer is a deep learning package extended from fairseq that allows researchers and developers to train custom models for molecule modeling tasks. It aims to accelerate the research and application in AI for molecule science, such as material discovery, drug discovery, etc.

CHAPTER

ONE

INSTALLATION GUIDE

This is a guide to install Graphormer. Currently Graphormer supports intallation on Linux only.

1.1 Linux

On Linux, Graphormer can be easily installed with the install.sh script with prepared python environments.

1. Please use Python3.9 for Graphormer. It is recommended to create a virtual environment with conda or virtualenv. For example, to create and activate a conda environment with Python3.9

```
conda create -n graphormer python=3.9
conda activate graphormer
```

2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/Graphormer.git
cd Graphormer
bash install.sh
```

CHAPTER

TWO

START WITH EXAMPLE

Graphormer provides example scripts to train your own models on several datasets. For example, to train a Graphormer-slim on ZINC-500K on a single GPU card:

```
> cd examples/property_prediction/
> bash zinc.sh
```

The content of `zinc.sh` is simply a `fairseq-train` command:

```
CUDA_VISIBLE_DEVICES=0 fairseq-train \
    --user-dir ../../graphormer \
    --num-workers 16 \
    --ddp-backend=legacy_ddp \
    --dataset-name zinc \
    --dataset-source pyg \
    --task graph_prediction \
    --criterion l1_loss \
    --arch graphormer_slim \
    --num-classes 1 \
    --attention-dropout 0.1 --act-dropout 0.1 --dropout 0.0 \
    --optimizer adam --adam-betas '(0.9, 0.999)' --adam-eps 1e-8 --clip-norm 5.0 --
↪weight-decay 0.01 \
    --lr-scheduler polynomial_decay --power 1 --warmup-updates 60000 --total-num-update␣
↪400000 \
    --lr 2e-4 --end-learning-rate 1e-9 \
    --batch-size 64 \
    --fp16 \
    --data-buffer-size 20 \
    --encoder-layers 12 \
    --encoder-embed-dim 80 \
    --encoder-ffn-embed-dim 80 \
    --encoder-attention-heads 8 \
    --max-epoch 10000 \
    --save-dir ./ckpts
```

`CUDA_VISIBLE_DEVICES` specifies the GPUs to use. With multiple GPUs, the GPU IDs should be separated by commas. A `fairseq-train` with Graphormer model is used to launch training. *Command-line Tools* gives detailed explanations to the parameters.

Similarily, to train a Graphormer-base on PCQM4M dataset on multiple GPU cards:

```
> cd examples/property_prediction/
> bash pcqv1.sh
```

By runing the instructions in the scripts, Graphormer will automatically download the needed datasets and pre-process them.

CHAPTER

THREE

EVALUATE PRE-TRAINED MODELS

Graphormer provides pretrained models so that users can easily evaluate, and finetune. To evaluate a pre-trained model, use the script `graphormer/evaluate/evaluate.py`.

```
python evaluate.py \
    --user-dir ../../graphormer \
    --num-workers 16 \
    --ddp-backend=legacy_ddp \
    --dataset-name pcqm4m \
    --dataset-source ogb \
    --task graph_prediction \
    --criterion l1_loss \
    --arch graphormer_base \
    --num-classes 1 \
    --batch-size 64 \
    --pretrained-model-name pcqm4mv1_graphormer_base \
    --load-pretrained-model-output-layer \
    --split valid \
    --seed 1
```

`--pretrained-model-name` specifies the pre-trained model to be valuated. The pre-trained model will be automatically downloaded. And `--load-pretrained-model-output-layer` is set so that weights of the final fully connected layer in the pre-trained model is loaded. And `--split` specifies the split of the dataset to be evaluated, can be `train` or `valid`.

CHAPTER

FOUR

FINE-TUNING PRE-TRAINED MODELS

To fine-tune pre-trained models, use `--pretrained-model-name` to set the model name. For example, the script `examples/property_prediction/hiv_pre.sh` fine-tunes our model `pcqm4mv1_graphormer_base` on the `ogbg-molhiv` dataset. The command for fine-tune is

```
fairseq-train \
    --user-dir ../../graphormer \
    --num-workers 16 \
    --ddp-backend=legacy_ddp \
    --dataset-name ogbg-molhiv \
    --dataset-source ogb \
    --task graph_prediction_with_flag \
    --criterion binary_logloss_with_flag \
    --arch graphormer_base \
    --num-classes 1 \
    --attention-dropout 0.1 --act-dropout 0.1 --dropout 0.0 \
    --optimizer adam --adam-betas '(0.9, 0.999)' --adam-eps 1e-8 --weight-decay 0.0 \
    --lr-scheduler polynomial_decay --power 1 --warmup-updates $warmup_updates --total-
↪num-update $tot_updates \
    --lr 2e-4 --end-learning-rate 1e-9 \
    --batch-size $batch_size \
    --fp16 \
    --data-buffer-size 20 \
    --encoder-layers 12 \
    --encoder-embed-dim 768 \
    --encoder-ffn-embed-dim 768 \
    --encoder-attention-heads 32 \
    --max-epoch $max_epoch \
    --save-dir ./ckpts \
    --pretrained-model-name pcqm4mv1_graphormer_base \
    --flag-m 3 \
    --flag-step-size 0.01 \
    --flag-mag 0 \
    --seed 1 \
    --pre-layernorm
```

After fine-tuning, use `graphormer/evaluate/evaluate.py` to evaluate the performance of all checkpoints:

```
python evaluate.py \
    --user-dir ../../graphormer \
    --num-workers 16 \
    --ddp-backend=legacy_ddp \
```

(continues on next page)

(continued from previous page)

```
    --dataset-name ogbg-molhiv \
    --dataset-source ogb \
    --task graph_prediction \
    --arch graphormer_base \
    --num-classes 1 \
    --batch-size 64 \
    --save-dir ../../examples/property_prediction/ckpts/ \
    --split test \
    --metric auc \
    --seed 1 \
    --pre-layernorm
```

CHAPTER

FIVE

TRAINING A NEW MODEL

We take OC20 as an example to show how to train a new model on your own datasets.

First, download IS2RE train, validation, and test data in LMDB format by:

```
> cd examples/oc20/ && mkdir data && cd data/
> wget -c https://dl.fbaipublicfiles.com/opencatalystproject/data/is2res_train_val_test_
↪lmdbs.tar.gz && tar -xzvf is2res_train_val_test_lmdbs.tar.gz
```

Create `ckpt` folder to save checkpoints during the training:

```
> cd ../ && mkdir ckpt/
```

Now we train a 48-layer `graphormer-3D` architecture, which has 4 blocks and each block contains 12 Graphormer layers. The parameters are sharing across blocks. The total training steps are 1 million, and we warmup the learning rate by 10 thousand steps.

```
> fairseq-train --user-dir ../../graphormer  \
   ./data/is2res_train_val_test_lmdbs/data/is2re/all --valid-subset val_id,val_ood_ads,
↪val_ood_cat,val_ood_both --best-checkpoint-metric loss \
   --num-workers 0 --ddp-backend=c10d \
   --task is2re --criterion mae_deltapos --arch graphormer3d_base  \
   --optimizer adam --adam-betas '(0.9, 0.98)' --adam-eps 1e-6 --clip-norm $clip_norm \
   --lr-scheduler polynomial_decay --lr 3e-4 --warmup-updates  --total-num-update␣
↪1000000 --batch-size 4 \
   --dropout 0.0 --attention-dropout 0.1 --weight-decay 0.001 --update-freq 1 --seed 1 \
   --fp16 --fp16-init-scale 4 --fp16-scale-window 256 --tensorboard-logdir ./tsbs \
   --embed-dim 768 --ffn-embed-dim 768 --attention-heads 48 \
   --max-update 1000000 --log-interval 100 --log-format simple \
   --save-interval-updates 5000 --validate-interval-updates 2500 --keep-interval-updates␣
↪30 --no-epoch-checkpoints  \
   --save-dir ./ckpt --layers 12 --blocks 4 --required-batch-size-multiple 1  --node-
↪loss-weight 15
```

Please note that `--batch-size 4` requires at least 32GB of GPU memory. If out of GPU momery occuars, one may try to reduce the batchsize then train with more GPU cards, or increase the `--update-freq` to accumulate the gradients.

CHAPTER

SIX

COMMAND-LINE TOOLS

Graphormer reuses the `fairseq-train` command-line tools of fairseq for training, and here we mainly document the additional parameters in Graphormer and parameters of `fairseq-train` used by Graphormer.

6.1 Model

- `--arch`, type=enum, options: `graphormer_base`, `graphormer_slim`, `graphormer_large`
  - Predefined graphormer architectures
- `--encoder-ffn-embed-dim`, type = float
  - encoder embedding dimension for FFN
- `--encoder-layers`, type = int
  - number of graphormer encoder layers
- `--encoder-embed-dim`, type = int
  - encoder embedding dimension
- `--share-encoder-input-output-embed`, type = bool
  - if set, share encoder input and output embeddings
- `--share-encoder-input-output-embed`, type = bool
  - if set, share encoder input and output embeddings
- `--encoder-learned-pos`, type = bool
  - if set, use learned positional embeddings in the encoder
- `--no-token-positional-embeddings`, type = bool
  - if set, disables positional embeddings” ” (outside self attention)
- `--max-positions`, type = int
  - number of positional embeddings to learn
- `--activation-fn`, type = enum, options: `gelu`, `relu`
  - activation function to use
- `--encoder-normalize-before`
  - if set, apply layernorm before each encoder block

6.2 Training

- `--task`, type = enum, options: `graph_prediction`, `is2re`
  - the task for training
    - `graph_prediction`: ordinary graph-level prediction task, predict a single target for a graph
    - `is2re`: for IS2RE task of Open Catalyst Challenge
- `--criterion`, type = enum, options: `l1_loss`, `binary_logloss`, `multiclass_cross_entropy`, `mae_deltapos`, `l1_loss_with_flag`, `binary_logloss_with_flag`, `multiclass_cross_entropy_with_flag`
  - the criterion, or objective function for training.
    - `l1_loss`: mean absolute error (MAE) for regression tasks
    - `binary_logloss`: binary cross entropy for binary classification
    - `multiclass_cross_entropy`: multi-class cross entropy for multi-class classification
    - `mae_deltapos`: criterion for IS2RE task of Open Catalyst Challenge
    - `l1_loss_with_flag`: `l1_loss` with FLAG
    - `binary_logloss_with_flag`: `binary_logloss` with FLAG
    - `multiclass_cross_entropy_with_flag`: `multiclass_cross_entropy` with FLAG
- `--apply-graphormer-init`, type = bool
  - if set, use custom param initialization for Graphormer
- `--dropout`, type = float
  - dropout probability
- `--attention-dropout`, type = float
  - dropout probability for attention weights
- `--act-dropout`, type = float
  - dropout probability after activation in FFN
- `--seed`, type = int
  - random seed
- `--pretrained-model-name`, type = enum, default= `none`, options: `pcqm4mv1_graphormer_base`, `pcqm4mv2_graphormer_base`
  - name of used pretrained model
    - `pcqm4mv1_graphormer_base`: Pretrained Graphormer base model with PCQM4M v1 dataset.
    - `pcqm4mv2_graphormer_base`: Pretrained Graphormer base model with PCQM4M v2 dataset.
- `--load-pretrained-model-output-layer`, type = bool
  - if set, the weights of the final fully connected layer in the pre-trained model is loaded
- `--optimizer`, type = enum
  - optimizers from fairseq
- `--lr`, type = float
  - learning rate

- `--lr-scheduler`, type=enum
  - learning rate scheduler from fairseq
- `--fp16`, type=bool
  - if set, use mixed precision training
- `--data-buffer-size`, type=int, default=10
  - number of batches to preload
- `--batch-size`, type=int
  - number of examples in a batch
- `--max-epoch`, type=int, default=0
  - force stop training at specified epoch
- `--save-dir`, type=str, default=``checkpoints``
  - path to save checkpoints

6.3 Dataset

- `--dataset-name`, type = str, default= `pcqm4m`
  - name of the dataset
- `--dataset-source`, type = str, default= `ogb`
  - source of graph dataset, can be: `pyg`, `dgl`, `ogb`
- `--num-classes`, type = int, default=-1
  - number of classes or regression targets
- `--num-atoms`, type = int, default=512 * 9
  - number of atom types in the graph
- `--num-edges`, type = int, default=512 * 3
  - number of edge types in the graph
- `--num-in-degree`, type = int, default=512
  - number of in degree types in the graph
- `--num-out-degree`, type = int, default=512
  - number of out degree types in the graph
- `--num-spatial`, type = int, default=512
  - number of spatial types in the graph
- `--num-edge-dis`, type = int, default=128
  - number of edge dis types in the graph
- `--multi-hop-max-dist`, type = int, default=5
  - max number of edges considered in the edge encoding
- `--spatial-pos-max`, type = int, default=1024

  - max distance of attention in graph
- `--edge-type`, type = str, default="multi_hop"
  - edge type in the graph
- `--edge-type`, type = str, default="multi_hop"
  - edge type in the graph
- `--user-data-dir`, type = str, default=""
  - path to the module of user-defined dataset

CHAPTER

SEVEN

DATASETS

Graphormer supports training with both existing datasets in graph libraries and customized datasets.

7.1 Existing Datasets

Graphormer supports training with datasets in existing libraries. Users can easily exploit datasets in these libraries by specifying the `--dataset-source` and `--dataset-name` parameters.

`--dataset-source` specifies the source for the dataset, can be:

1. `dgl` for DGL
2. `pyg` for Pytorch Geometric
3. `ogb` for OGB

`--dataset-name` specifies the dataset in the source. For example, by specifying `--dataset-source pyg` and `--dataset-name zinc`, Graphormer will load the ZINC dataset from Pytorch Geometric. When a dataset requires additional parameters to construct, the parameters are specified as `<dataset_name>:<param_1>=<value_1>,<param_2>=<value_2>,...,<param_n>=<value_n>`. When the type of a parameter value is a list, the value is represented as a string with the list elements concatenated by `+`. For example, if we want to specify multiple `label_keys` with `mu`, `alpha`, and `homo` for QM9 dataset, `--dataset-name` should be `qm9:label_keys=mu+alpha+homo`.

When dataset split (`train`, `valid` and `test` subsets) is not configured in the original dataset source, we randomly partition the full set into `train`, `valid` and `test` with ratios `0.7`, `0.2` and `0.1`, respectively. If you want customized split of a dataset, you may implement a `customized dataset `. Currently, only integer features of nodes and edges in the datasets are used.

A full list of supported datasets of each data source:

Dataset Source	Dataset Name	Link	#Label/#Class
`dgl`	`qm7b`	QM7B dataset	14
	`qm9`	QM9 dataset	Depending on `label_keys`
	`qm9edge`	QM9Edge dataset	Depending on `label_keys`
	`minigc`	MiniGC dataset	8
	`gin`	Graph Isomorphism Network dataset	1
	`fakenews`	FakeNewsDataset dataset	1
`pgy`	`moleculenet`	MoleculeNet dataset	1
	`zinc`	ZINC dataset	1
`ogb`	`ogbg-molhiv`	ogbg-molhiv dataset	1
	`ogbg-molpcba`	ogbg-molpcba dataset	128
	`pcqm4m`	PCQM4M dataset	1
	`pcqm4mv2`	PCQM4Mv2 dataset	1

7.2 Customized Datasets

Users may create their own datasets. To use customized dataset:

1. Create a folder (for example, with name *customized_dataset*), and a python script with arbitrary name in the folder.

2. In the created python script, define a function which returns the created dataset. And register the function with `register_dataset`. Here is a sample python script. We define a QM9 dataset from `dgl` with customized split.

```
from graphormer.data import register_dataset
from dgl.data import QM9
import numpy as np
from sklearn.model_selection import train_test_split

@register_dataset("customized_qm9_dataset")
def create_customized_dataset():
    dataset = QM9(label_keys=["mu"])
    num_graphs = len(dataset)

    # customized dataset split
    train_valid_idx, test_idx = train_test_split(
        np.arange(num_graphs), test_size=num_graphs // 10, random_state=0
    )
    train_idx, valid_idx = train_test_split(
        train_valid_idx, test_size=num_graphs // 5, random_state=0
    )
    return {
        "dataset": dataset,
        "train_idx": train_idx,
        "valid_idx": valid_idx,
        "test_idx": test_idx,
        "source": "dgl"
    }
```

The function returns a dictionary. In the dictionary, `dataset` is the dataset object. `train_idx` is the graph indices used for training. Similarly we have `valid_idx` and `test_idx`. Finally `source` records the underlying graph library used by the dataset.

3. Specify the `--user-data-dir` as `customized_dataset` when training. And set `--dataset-name` as `customized_qm9_dataset`. Note that `--user-data-dir` should not be used together with `--dataset-source`. All datasets defined in all python scripts under the `customized_dataset` will be registered automatically.

CHAPTER

EIGHT

PRETRAINED MODELS

Graphormer provides a series of pre-trained model to help users leverage the power of the model quickly and smoothly. Contributing your pre-trained model by creating a pull request.

- Quick-Start: as a exmple using pre-trained models.

Pre-trained models from specific papers:

- Do Transformers Really Perform Badly for Graph Representation?

CHAPTER

NINE

OVERVIEW

Basically, Graphormer inherits the extending usage of fairseq, which means it could easily support user-defined plug-ins.

For example, the Graphormer-base model could be defined through `GraphormerModel`, which inherits the `FairseqModel` class.

It's also easy to extend the Graphormer-base model, which means you could define your own model and criterion, and then use them in Graphormer.

Also, development of new model is easy. We provide a tutorial of how to implement a simple MLP model on graph in Tutorials.

CHAPTER

TEN

SIMPLE MLP TUTORIAL

In this tutorial, we will extend Graphormer by adding a new `GraphMLP` that transforms the node features, and uses a sum pooling layer to combine the output of the MLP as graph representation.

This tutorial covers:

1. **Writing a new Model** so that the node token embeddings can be transformed by the MLP.
2. **Training the Model** using the existing command-line tools.

10.1 1. Writing a new GraphMLP Model

First, we create a new file with filename `graphormer/models/graphmlp.py`:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from fairseq.models import FairseqEncoderModel, register_model

@register_model("graphmlp")
class GraphMLP(FairseqEncoderModel):
    def __init__(self, args, encoder):
        super().__init__(encoder)
        self.args = args

    @staticmethod
    def add_args(parser):
        """Add model-specific arguments to the parser."""
        parser.add_argument(
            "--encoder-layers", type=int, metavar="N", help="num encoder layers"
        )
        parser.add_argument(
            "--max-nodes", type=int, metavar="N", help="num max nodes"
        )
        parser.add_argument(
            "--encoder-embed-dim",type=int, metavar="N", help="encoder embedding␣
↪dimension",
        )

    def max_nodes(self):
        return self.encoder.max_nodes
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def build_model(cls, args, task):
        """Build a new model instance."""
        # make sure all arguments are present in older models
        graphmlp_architecture(args)
        encoder = GraphMLPEncoder(args)
        return cls(args, encoder)

    def forward(self, batched_data, **kwargs):
       return self.encoder(batched_data, **kwargs)
```

The main component in GraphMLP is the GraphMLPEncoder. Here we implement it by adding following codes in graphormer/models/graphmlp.py:

```
from fairseq.models import FairseqEncoder
from ..modules import GraphNodeFeature

class GraphMLPEncoder(FairseqEncoder):
    def __init__(self, args):
        super().__init__(dictionary=None)
        self.max_nodes = args.max_nodes
        self.emb_dim = args.encoder_embed_dim
        self.num_layer = args.encoder_layers
        self.num_classes = args.num_classes

        self.atom_encoder = GraphNodeFeature(
            num_heads=1,
            num_atoms=512*9,
            num_in_degree=512,
            num_out_degree=512,
            hidden_dim=self.emb_dim,
            n_layers=self.num_layer,
        )

        self.linear = torch.nn.ModuleList()
        self.batch_norms = torch.nn.ModuleList()

        for layer in range(self.num_layer):
            self.linear.append(torch.nn.Linear(self.emb_dim, self.emb_dim))
            self.batch_norms.append(torch.nn.BatchNorm1d(self.emb_dim))


        self.graph_pred_linear = torch.nn.Linear(self.emb_dim, self.num_classes)


    def forward(self, batched_data, **unused):
        h=self.atom_encoder(batched_data)
        for layer in range(self.num_layer):
            h = self.linear[layer](h)
            h = h.transpose(1,2)
            h = self.batch_norms[layer](h)
```

(continues on next page)

(continued from previous page)

```
            h = h.transpose(1,2)

            if layer != self.num_layer - 1:
                h = F.relu(h)

        h = h.sum(dim=1)
        out = self.graph_pred_linear(h)

        return out.unsqueeze(1)


    def max_nodes(self):
        return self.max_nodes
```

Since we will validate our GraphMLP model on a graph representation task, we choose dataset in MoleculeNet. Therefore, we employ the `GraphNodeFeature` to encode the node features.

And finally, we register the model architecture by adding following codes in `graphormer/models/graphmlp.py`:

```
from fairseq.models import register_model_architecture
@register_model_architecture("graphmlp", "graphmlp")
def graphmlp_architecture(args):
    args.encoder_embed_dim = getattr(args, "encoder_embed_dim", 768)
    args.encoder_layers = getattr(args, "encoder_layers", 12)

    args.max_nodes = getattr(args, "max_nodes", 512)
```

10.2 2. Training the Model

Next, we prepare the training script for the model. We create a bash file `examples/property_prediction/graphmlp.sh`:

```
#!/bin/bash
CUDA_VISIBLE_DEVICES=0 fairseq-train \
--user-dir ../../graphormer \
--num-workers 16 \
--ddp-backend=legacy_ddp \
--dataset-name moleculenet:name=bbbp   \
--dataset-source pyg \
--task graph_prediction \
--criterion binary_logloss \
--arch graphmlp \
--num-classes 1 \
--optimizer adam --adam-betas '(0.9, 0.999)' --adam-eps 1e-8 --clip-norm 5.0 --weight-
↪decay 0.0 \
--lr-scheduler polynomial_decay --power 1 --total-num-update 1000000 \
--lr 0.001 --end-learning-rate 1e-9 \
--batch-size 32 \
--fp16 \
--data-buffer-size 20 \
--encoder-layers 5 \
```

(continues on next page)

(continued from previous page)

```
--encoder-embed-dim 256 \
--max-epoch 100 \
--save-dir ./ckpts \
--save-interval-updates 50000 \
--no-epoch-checkpoints
```

By executing the script, after the dataset is downloaded and processed, the training of the GraphMLP model starts.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search