# pygmtools Documentation

**Runzhong Wang, Ziao Guo**

**Oct 08, 2022**

# CONTENTS

`pygmtools` provides graph matching solvers in Python and is easily accessible via:

```
$ pip install pygmtools
```

Official documentation: https://pygmtools.readthedocs.io

Source code: https://github.com/Thinklab-SJTU/pygmtools

Graph matching is a fundamental yet challenging problem in pattern recognition, data mining, and others. Graph matching aims to find node-to-node correspondence among multiple graphs, by solving an NP-hard combinatorial optimization problem.

Doing graph matching in Python used to be non-trivial, and this library wants to make researchers' lives easier. To highlight, `pygmtools` has the following features:

- *Support various solvers*, including traditional combinatorial solvers (including linear, quadratic, and multi-graph) and novel deep learning-based solvers;
- *Support various backends*, including `numpy` which is universally accessible, and some state-of-the-art deep learning architectures with GPU support: `pytorch`, `paddle`, `jittor`.
- *Deep learning friendly*, the operations are designed to best preserve the gradient during computation and batched operations support for the best performance.

CHAPTER

ONE

# INSTALLATION

You can install the stable release on PyPI:

```
$ pip install pygmtools
```

or get the latest version by running:

```
$ pip install -U https://github.com/Thinklab-SJTU/pygmtools/archive/master.zip # with --
↪user for user install (no root)
```

Now the pygmtools is available with the `numpy` backend.

The following packages are required, and shall be automatically installed by `pip`:

```
Python >= 3.5
requests >= 2.25.1
scipy >= 1.4.1
Pillow >= 7.2.0
numpy >= 1.18.5
easydict >= 1.7
appdirs >= 1.4.4
tqdm >= 4.64.1
```

CHAPTER

TWO

# AVAILABLE GRAPH MATCHING SOLVERS

This library offers user-friendly API for the following solvers:

- Two-Graph Matching Solvers
  - Linear assignment solvers including the differentiable soft Sinkhorn algorithm [1], and the exact solver Hungarian [2].
  - Soft and differentiable quadratic assignment solvers, including spectral graph matching [3] and random-walk-based graph matching [4].
  - Discrete (non-differentiable) quadratic assignment solver integer projected fixed point method [5].
- Multi-Graph Matching Solvers
  - Composition based Affinity Optimization (CAO) solver [6] by optimizing the affinity score, meanwhile gradually infusing the consistency.
  - Multi-Graph Matching based on Floyd shortest path algorithm [7].
  - Graduated-assignment based multi-graph matching solver [8][9] by graduated annealing of Sinkhorn's temperature.
- Neural Graph Matching Solvers
  - Intra-graph and cross-graph embedding based neural graph matching solvers PCA-GM and IPCA-GM [10] for matching individual graphs.
  - Channel independent embedding (CIE) [11] based neural graph matching solver for matching individual graphs.
  - Neural graph matching solver (NGM) [12] for the general quadratic assignment formulation.

CHAPTER

THREE

# AVAILABLE BACKENDS

This library is designed to support multiple backends with the same set of API. Please follow the official instructions to install your backend.

The following backends are available:

- Numpy (**default** backend, CPU only)
- PyTorch (**recommended** backend, GPU friendly, deep learning friendly)
- PaddlePaddle (GPU friendly, deep learning friendly)
- Jittor (GPU friendly, deep learning friendly)

For more details, please read the documentation.

CHAPTER

FOUR

# THE DEEP GRAPH MATCHING BENCHMARK

`pygmtools` is also featured with a standard data interface of several graph matching benchmarks. We also maintain a repository containing non-trivial implementation of deep graph matching models, please check out ThinkMatch if you are interested!

CHAPTER

FIVE

# CONTRIBUTING

Any contributions/ideas/suggestions from the community is welcomed! Before starting your contribution, please read the Contributing Guide.

CHAPTER

SIX

# DEVELOPERS AND MAINTAINERS

`pygmtools` is currently developed and maintained by members from ThinkLab at Shanghai Jiao Tong University.

CHAPTER

SEVEN

# REFERENCES


[1] Sinkhorn, Richard, and Paul Knopp. "Concerning nonnegative matrices and doubly stochastic matrices." Pacific Journal of Mathematics 21.2 (1967): 343-348.

[2] Munkres, James. "Algorithms for the assignment and transportation problems." Journal of the society for industrial and applied mathematics 5.1 (1957): 32-38.

[3] Leordeanu, Marius, and Martial Hebert. "A spectral technique for correspondence problems using pairwise constraints." International Conference on Computer Vision (2005).

[4] Cho, Minsu, Jungmin Lee, and Kyoung Mu Lee. "Reweighted random walks for graph matching." European conference on Computer vision. Springer, Berlin, Heidelberg, 2010.

[5] Leordeanu, Marius, Martial Hebert, and Rahul Sukthankar. "An integer projected fixed point method for graph matching and map inference." Advances in neural information processing systems 22 (2009).

[6] Yan, Junchi, et al. "Multi-graph matching via affinity optimization with graduated consistency regularization." IEEE transactions on pattern analysis and machine intelligence 38.6 (2015): 1228-1242.

[7] Jiang, Zetian, Tianzhe Wang, and Junchi Yan. "Unifying offline and online multi-graph matching via finding shortest paths on supergraph." IEEE transactions on pattern analysis and machine intelligence 43.10 (2020): 3648-3663.

[8] Solé-Ribalta, Albert, and Francesc Serratosa. "Graduated assignment algorithm for multiple graph matching based on a common labeling." International Journal of Pattern Recognition and Artificial Intelligence 27.01 (2013): 1350001.

[9] Wang, Runzhong, Junchi Yan, and Xiaokang Yang. "Graduated assignment for joint multi-graph matching and clustering with application to unsupervised graph matching network learning." Advances in Neural Information Processing Systems 33 (2020): 19908-19919.

[10] Wang, Runzhong, Junchi Yan, and Xiaokang Yang. "Combinatorial learning of robust deep graph matching: an embedding based approach." IEEE Transactions on Pattern Analysis and Machine Intelligence (2020).

[11] Yu, Tianshu, et al. "Learning deep graph matching with channel-independent embedding and hungarian attention." International conference on learning representations. 2019.

[12] Wang, Runzhong, Junchi Yan, and Xiaokang Yang. "Neural graph matching network: Learning lawler's quadratic assignment problem with extension to hypergraph and multiple-graph matching." IEEE Transactions on Pattern Analysis and Machine Intelligence (2021).

CHAPTER

EIGHT

# CONTENTS OF OFFICIAL DOCUMENTATION

## 8.1 Introduction and Guidelines

This page provides a brief introduction to graph matching and some guidelines for using `pygmtools`. If you are seeking some background information, this is the right place!

---

**Note:** For more technical details, we recommend the following two surveys.

About **learning-based** deep graph matching: Junchi Yan, Shuang Yang, Edwin Hancock. "Learning Graph Matching and Related Combinatorial Optimization Problems." *IJCAI 2020*.

About **non-learning** two-graph matching and multi-graph matching: Junchi Yan, Xu-Cheng Yin, Weiyao Lin, Cheng Deng, Hongyuan Zha, Xiaokang Yang. "A Short Survey of Recent Advances in Graph Matching." *ICMR 2016*.

---

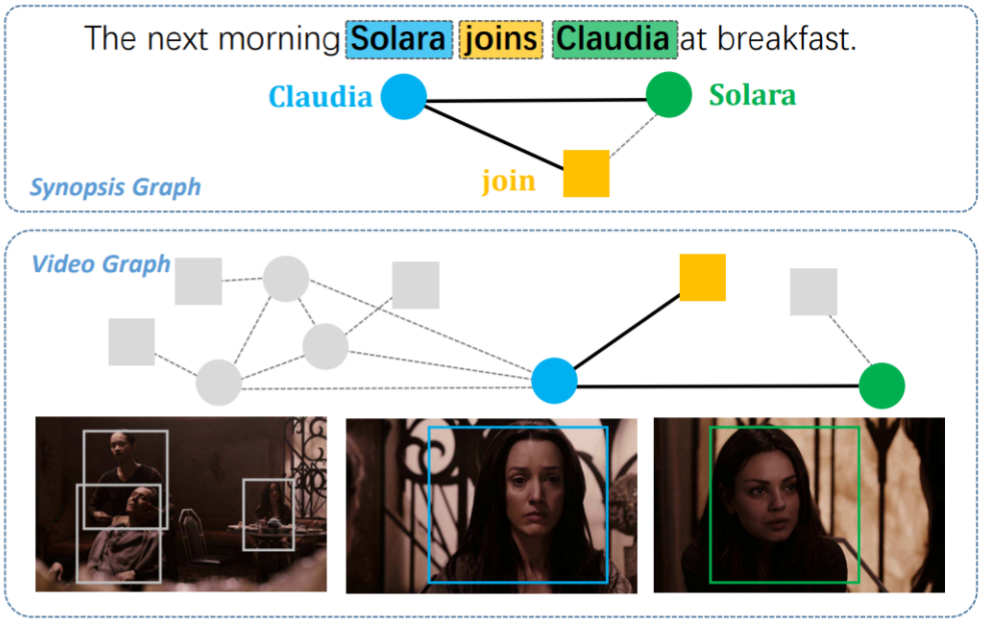
### 8.1.1 Why Graph Matching?

Graph Matching (GM) is a fundamental yet challenging problem in pattern recognition, data mining, and others. GM aims to find node-to-node correspondence among multiple graphs, by solving an NP-hard combinatorial problem. Recently, there is growing interest in developing deep learning-based graph matching methods.

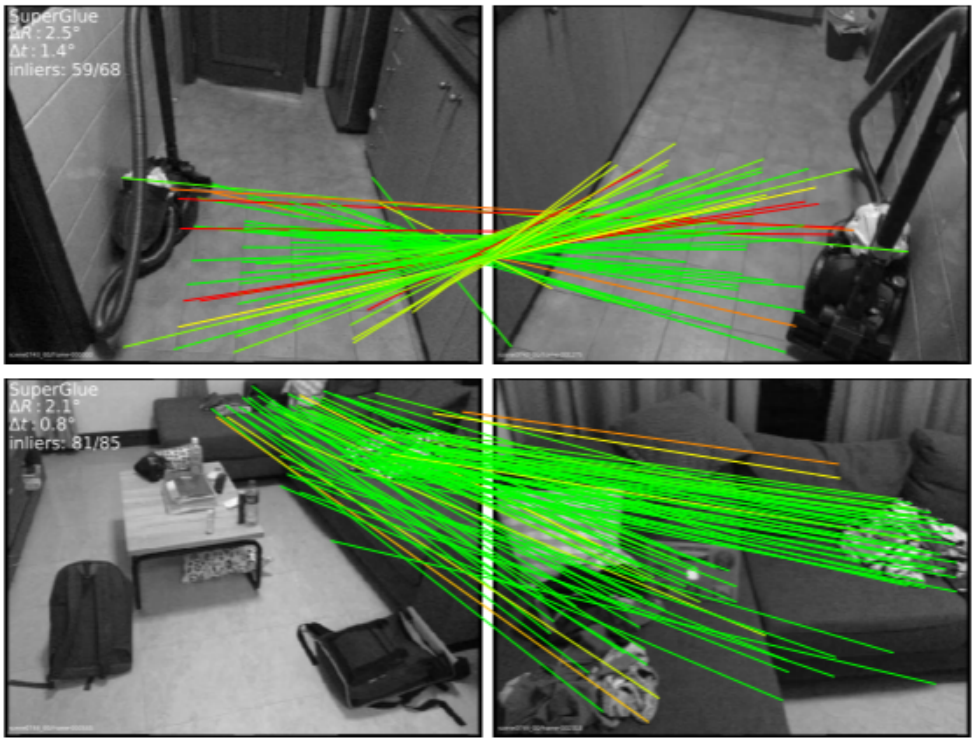
Compared to other straight-forward matching methods e.g. greedy matching, graph matching methods are more reliable because it is based on an optimization form. Besides, graph matching methods exploit both node affinity and edge affinity, thus graph matching methods are usually more robust to noises and outliers. The recent line of deep graph matching methods also enables many graph matching solvers to be integrated into a deep learning pipeline.

Graph matching techniques have been applied to the following applications:

- Bridging movie and synopses



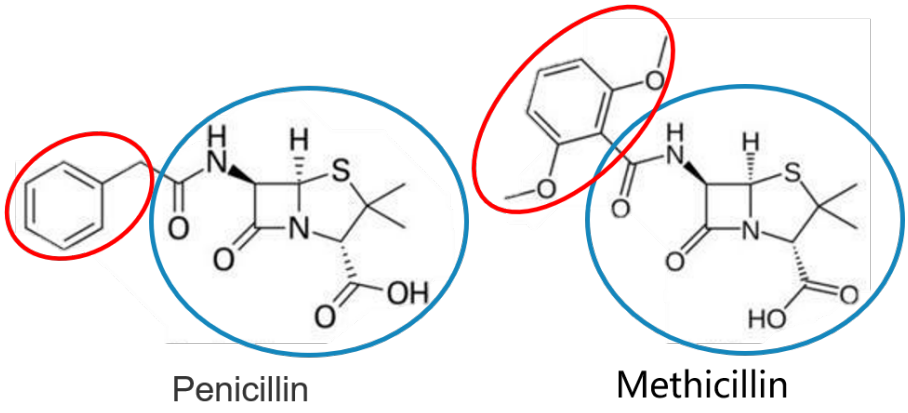

- Image correspondence




- Model ensemble and federated learning

images/federated_learning.png

- Molecules matching




- and more…

If your task involves matching two or more graphs, you should try the solvers in `pygmtools`!

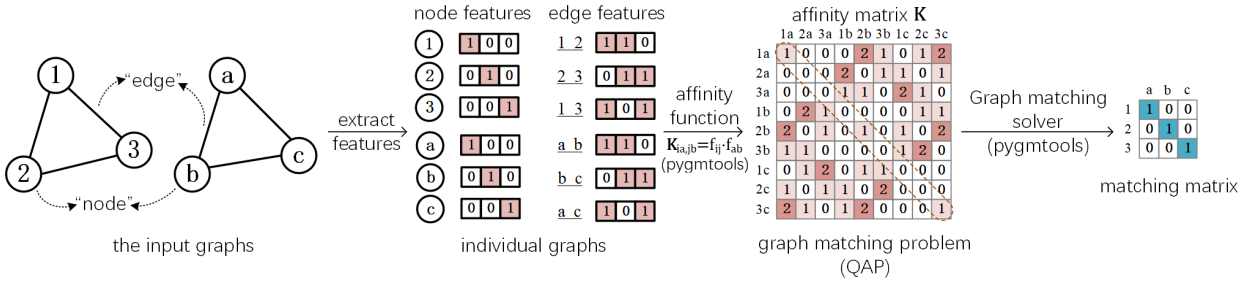
### 8.1.2 What is Graph Matching?

#### The Graph Matching Pipeline

Solving a real-world graph-matching problem may involve the following steps:

1. Extract node/edge features from the graphs you want to match.
2. Build an affinity matrix from node/edge features.
3. Solve the graph matching problem with GM solvers.

And Step 1 may be done by methods depending on your application, Step 2&3 can be handled by `pygmtools`. The following plot illustrates a standard deep graph matching pipeline.




#### The Math Form

Let's involve a little bit of math to better understand the graph matching pipeline. In general, graph matching is of the following form, known as **Quadratic Assignment Problem (QAP)**:

$$\max_{\mathbf{X}} \ \texttt{vec}(\mathbf{X})^\top \mathbf{K} \texttt{vec}(\mathbf{X})$$
$$s.t. \quad \mathbf{X} \in \{0,1\}^{n_1 \times n_2},\ \mathbf{X}\mathbf{1} = \mathbf{1},\ \mathbf{X}^\top \mathbf{1} \leq \mathbf{1}$$

The notations are explained as follows:

- $\mathbf{X}$ is known as the **permutation matrix** which encodes the matching result. It is also the decision variable in graph matching problem. $\mathbf{X}_{i,a} = 1$ means node $i$ in graph 1 is matched to node $a$ in graph 2, and $\mathbf{X}_{i,a} = 0$ means non-matched. Without loss of generality, it is assumed that $n_1 \leq n_2$. $\mathbf{X}$ has the following constraints:
  - The sum of each row must be equal to 1: $\mathbf{X}\mathbf{1} = \mathbf{1}$;
  - The sum of each column must be equal to, or smaller than 1: $\mathbf{X}\mathbf{1} \leq \mathbf{1}$.
- $\texttt{vec}(\mathbf{X})$ means the column-wise vectorization form of $\mathbf{X}$.
- $\mathbf{1}$ means a column vector whose elements are all 1s.
- $\mathbf{K}$ is known as the **affinity matrix** which encodes the information of the input graphs. Both node-wise and edge-wise affinities are encoded in $\mathbf{K}$:
  - The diagonal element $\mathbf{K}_{i+a\times n_1, i+a\times n_1}$ means the node-wise affinity of node $i$ in graph 1 and node $a$ in graph 2;
  - The off-diagonal element $\mathbf{K}_{i+a\times n_1, j+b\times n_1}$ means the edge-wise affinity of edge $ij$ in graph 1 and edge $ab$ in graph 2.

### 8.1.3 Graph Matching Best Practice

We need to understand the advantages and limitations of graph matching solvers. As discussed above, the major advantage of graph matching solvers is that they are more robust to noises and outliers. Graph matching also utilizes edge information, which is usually ignored in linear matching methods. The major drawback of graph matching solvers is their efficiency and scalability since the optimization problem is NP-hard. Therefore, to decide which matching method is most suitable, one needs to balance between the required matching accuracy and the affordable time and memory cost according to his/her application.

**Note:** Anyway, it does no harm to try graph matching first!

#### When to use pygmtools

`pygmtools` is recommended for the following cases, and you could benefit from the friendly API:

- If you want to integrate graph matching as a step of your pipeline (either learning or non-learning).
- If you want a quick benchmarking and profiling of the graph matching solvers available in `pygmtools`.
- If you do not want to dive too deep into the algorithm details and do not need to modify the algorithm.

We offer the following guidelines for your reference:

- If you want to integrate graph matching solvers into your end-to-end supervised deep learning pipeline, try *neural_solvers*.
- If no ground truth label is available for the matching step, try *classic_solvers*.
- If there are multiple graphs to be jointly matched, try *multi_graph_solvers*.
- If time and memory cost of the above methods are unacceptable for your task, try *linear_solvers*.

#### When not to use pygmtools

As a highly packed toolkit, `pygmtools` lacks some flexibilities in the implementation details, especially for experts in graph matching. If you are researching new graph matching algorithms or developing next-generation deep graph matching neural networks, `pygmtools` may not be suitable. We recommend ThinkMatch as the protocol for academic research.

## 8.2 Get Started

### 8.2.1 Basic Install by pip

You can install the stable release on PyPI:

```
$ pip install pygmtools
```

or get the latest version by running:

```
$ pip install -U https://github.com/Thinklab-SJTU/pygmtools/archive/master.zip # with --
↪user for user install (no root)
```

Now the pygmtools is available with the `numpy` backend:


You may jump to *Example: Matching Isomorphic Graphs* if you do not need other backends.

The following packages are required, and shall be automatically installed by `pip`:

```
Python >= 3.5
requests >= 2.25.1
scipy >= 1.4.1
Pillow >= 7.2.0
numpy >= 1.18.5
easydict >= 1.7
appdirs >= 1.4.4
tqdm >= 4.64.1
```

### 8.2.2 Install Other Backends

Currently, we also support deep learning architectures `pytorch`, `paddle`, `jittor` which are GPU-friendly and deep learning-friendly.

Once the backend is ready, you may switch to the backend globally by the following command:

```
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'  # replace 'pytorch' by other backend names
```

#### PyTorch Backend


PyTorch is an open-source machine learning framework developed and maintained by Meta Inc./Linux Foundation. PyTorch is popular, especially among the deep learning research community. The PyTorch backend of `pygmtools` is designed to support GPU devices and facilitate deep learning research.

Please follow the official PyTorch installation guide.

This package is developed with `torch==1.6.0` and shall work with any PyTorch versions `>=1.6.0`.

How to enable PyTorch backend:

```
>>> import pygmtools as pygm
>>> import torch
>>> pygm.BACKEND = 'pytorch'
```

#### Paddle Backend


PaddlePaddle is an open-source deep learning platform originated from industrial practice, which is developed and maintained by Baidu Inc. The Paddle backend of `pygmtools` is designed to support GPU devices and deep learning applications.

Please follow the official PaddlePaddle installation guide.

This package is developed with `paddlepaddle==2.3.1` and shall work with any PaddlePaddle versions `>=2.3.1`.

How to enable Paddle backend:

```
>>> import pygmtools as pygm
>>> import paddle
>>> pygm.BACKEND = 'paddle'
```

#### Jittor Backend


Jittor is an open-source deep learning platform based on just-in-time (JIT) for high performance, which is developed and maintained by the CSCG group from Tsinghua University. The Jittor backend of `pygmtools` is designed to support GPU devices and deep learning applications.

Please follow the official Jittor installation guide.

This package is developed with `jittor==1.3.4.16` and shall work with any Jittor versions `>=1.3.4.16`.

How to enable Jittor backend:

```
>>> import pygmtools as pygm
>>> import jittor
>>> pygm.BACKEND = 'jittor'
```

### 8.2.3 Example: Matching Isomorphic Graphs

Here we provide a basic example of matching two isomorphic graphs (i.e. two graphs have the same nodes and edges, but the node permutations are unknown).

Step 0: Import packages and set backend

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'
>>> np.random.seed(1)
```

Step 1: Generate a batch of isomorphic graphs

```
>>> batch_size = 3
>>> X_gt = np.zeros((batch_size, 4, 4))
>>> X_gt[:, np.arange(0, 4, dtype=np.int64), np.random.permutation(4)] = 1
>>> A1 = np.random.rand(batch_size, 4, 4)
>>> A2 = np.matmul(np.matmul(X_gt.transpose((0, 2, 1)), A1), X_gt)
>>> n1 = n2 = np.repeat([4], batch_size)
```

Step 2: Build an affinity matrix and select an affinity function

```
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1, n2,␣
↪ne2, edge_aff_fn=gaussian_aff)
```

Step 3: Solve graph matching by RRWM

```
>>> X = pygm.rrwm(K, n1, n2, beta=100)
>>> X = pygm.hungarian(X)
>>> X # X is the permutation matrix
[[[0. 0. 0. 1.]
  [0. 0. 1. 0.]
  [1. 0. 0. 0.]
  [0. 1. 0. 0.]]

 [[0. 0. 0. 1.]
  [0. 0. 1. 0.]
  [1. 0. 0. 0.]
  [0. 1. 0. 0.]]

 [[0. 0. 0. 1.]
  [0. 0. 1. 0.]
  [1. 0. 0. 0.]
  [0. 1. 0. 0.]]]
```

Final Step: Evaluate the accuracy

```
>>> (X * X_gt).sum() / X_gt.sum()
1.0
```

## 8.3 Graph Matching Benchmark

**pygmtools** also provides a protocol to fairly compare existing deep graph matching algorithms under different datasets & experiment settings. The `Benchmark` module provides a unified data interface and an evaluating platform for different datasets.

If you are interested in the performance and the full deep learning pipeline, please refer to our ThinkMatch project.

### 8.3.1 Evaluation Metrics and Results

Our evaluation metrics include **matching_precision (p)**, **matching_recall (r)** and **f1_score (f1)**. Also, to measure the reliability of the evaluation result, we define **coverage (cvg)** for each class in the dataset as *the number of evaluated pairs in the class/number of all possible pairs* in the class. Therefore, larger coverage refers to higher reliability.

An example of evaluation result (`p==r==f1` because this evaluation does not involve partial matching/outliers):

```
Matching accuracy
Car: p = 0.8395±0.2280, r = 0.8395±0.2280, f1 = 0.8395±0.2280, cvg = 1.0000
Duck: p = 0.7713±0.2255, r = 0.7713±0.2255, f1 = 0.7713±0.2255, cvg = 1.0000
Face: p = 0.9656±0.0913, r = 0.9656±0.0913, f1 = 0.9656±0.0913, cvg = 0.2612
Motorbike: p = 0.8821±0.1821, r = 0.8821±0.1821, f1 = 0.8821±0.1821, cvg = 1.0000
Winebottle: p = 0.8929±0.1569, r = 0.8929±0.1569, f1 = 0.8929±0.1569, cvg = 0.9662
average accuracy: p = 0.8703±0.1767, r = 0.8703±0.1767, f1 = 0.8703±0.1767
Evaluation complete in 1m 55s
```

### 8.3.2 Available Datasets

Dataset can be automatically downloaded and unzipped, but you can also download the dataset yourself, and make sure it in the right path.

#### PascalVOC-Keypoint Dataset

1. Download VOC2011 dataset and make sure it looks like `data/PascalVOC/TrainVal/VOCdevkit/VOC2011`
1. Download keypoint annotation for VOC2011 from Berkeley server or google drive and make sure it looks like `data/PascalVOC/annotations`
1. Download the train/test split file and make sure it looks like `data/PascalVOC/voc2011_pairs.npz`

Please cite the following papers if you use PascalVOC-Keypoint dataset:

```
@article{EveringhamIJCV10,
  title={The pascal visual object classes (voc) challenge},
  author={Everingham, Mark and Van Gool, Luc and Williams, Christopher KI and Winn, John␣
↪and Zisserman, Andrew},
  journal={International Journal of Computer Vision},
  volume={88},
  pages={303-338},
  year={2010}
}

@inproceedings{BourdevICCV09,
```

(continues on next page)

(continued from previous page)

```
  title={Poselets: Body part detectors trained using 3d human pose annotations},
  author={Bourdev, L. and Malik, J.},
  booktitle={International Conference on Computer Vision},
  pages={1365--1372},
  year={2009},
  organization={IEEE}
}
```

### Willow-Object-Class Dataset

1. Download Willow-ObjectClass dataset

1. Unzip the dataset and make sure it looks like `data/WillowObject/WILLOW-ObjectClass`

Please cite the following paper if you use Willow-Object-Class dataset:

```
@inproceedings{ChoICCV13,
  author={Cho, Minsu and Alahari, Karteek and Ponce, Jean},
  title = {Learning Graphs to Match},
  booktitle = {International Conference on Computer Vision},
  pages={25--32},
  year={2013}
}
```

### CUB2011 Dataset

1. Download CUB-200-2011 dataset.

1. Unzip the dataset and make sure it looks like `data/CUB_200_2011/CUB_200_2011`

Please cite the following report if you use CUB2011 dataset:

```
@techreport{CUB2011,
  Title = {{The Caltech-UCSD Birds-200-2011 Dataset}},
  Author = {Wah, C. and Branson, S. and Welinder, P. and Perona, P. and Belongie, S.},
  Year = {2011},
  Institution = {California Institute of Technology},
  Number = {CNS-TR-2011-001}
}
```

### IMC-PT-SparseGM Dataset

1. Download the IMC-PT-SparseGM dataset from google drive or baidu drive (code: 0576)

1. Unzip the dataset and make sure it looks like `data/IMC_PT_SparseGM/annotations`

Please cite the following papers if you use IMC-PT-SparseGM dataset:

```
@article{JinIJCV21,
  title={Image Matching across Wide Baselines: From Paper to Practice},
  author={Jin, Yuhe and Mishkin, Dmytro and Mishchuk, Anastasiia and Matas, Jiri and Fua,
↪ Pascal and Yi, Kwang Moo and Trulls, Eduard},
```

(continues on next page)

(continued from previous page)

```
  journal={International Journal of Computer Vision},
  pages={517--547},
  year={2021}
}
```

### 8.3.3 API Reference

See *the API doc of Benchmark module* and *the API doc of datasets* for details.

### 8.3.4 File Organization

- `dataset.py`: The file includes 5 dataset classes, used to automatically download the dataset and process the dataset into a json file, and also save the training set and the testing set.
- `benchmark.py`: The file includes Benchmark class that can be used to fetch data from the json file and evaluate prediction results.
- `dataset_config.py`: The default dataset settings, mostly dataset path and classes.

### 8.3.5 Example

```
import pygmtools as pygm
from pygm.benchmark import Benchmark

# Define Benchmark on PascalVOC.
bm = Benchmark(name='PascalVOC', sets='train',
               obj_resize=(256, 256), problem='2GM',
               filter='intersection')

# Random fetch data and ground truth.
data_list, gt_dict, _ = bm.rand_get_data(cls=None, num=2)
```

## 8.4 API and Modules

| | |
|---|---|
| *linear_solvers* | Classic (learning-free) **linear assignment problem** solvers. |
| *classic_solvers* | Classic (learning-free) **two-graph matching** solvers. |
| *multi_graph_solvers* | Classic (learning-free) **multi-graph matching** solvers. |
| *neural_solvers* | **Neural network-based** graph matching solvers. |
| *utils* | Utility functions: problem formulating, data processing, and beyond. |
| *benchmark* | The Benchmark module with a unified data interface to evaluate graph matching methods. |
| *dataset* | The implementations of data loading and data processing. |

### 8.4.1 pygmtools.linear_solvers

Classic (learning-free) **linear assignment problem** solvers. These linear assignment solvers are recommended to solve matching problems with only nodes (i.e. linear matching problems), or large-scale graph matching problems where the cost of QAP formulation is too high.

The linear assignment problem only considers nodes, and is also known as bipartite graph matching and linear matching:

$$\begin{aligned} &\max_{\mathbf{X}} \ \mathtt{tr}(\mathbf{X}^\top \mathbf{M}) \\ s.t. \quad &\mathbf{X} \in \{0,1\}^{n_1 \times n_2},\ \mathbf{X}\mathbf{1} = \mathbf{1},\ \mathbf{X}^\top \mathbf{1} \leq \mathbf{1} \end{aligned}$$

#### Functions

| | |
|---|---|
| *hungarian* | Solve optimal LAP permutation by hungarian algorithm. |
| *sinkhorn* | Sinkhorn algorithm turns the input matrix into a doubly-stochastic matrix. |

#### pygmtools.linear_solvers.hungarian

pygmtools.linear_solvers.**hungarian**(*s*, *n1=None*, *n2=None*, *nproc: int = 1*, *backend=None*)

Solve optimal LAP permutation by hungarian algorithm. The time cost is $O(n^3)$.

**Parameters**

- **s** – $(b \times n_1 \times n_2)$ input 3d tensor. $b$: batch size. Non-batched input is also supported if `s` is of size $(n_1 \times n_2)$
- **n1** – $(b)$ (optional) number of objects in dim1
- **n2** – $(b)$ (optional) number of objects in dim2
- **nproc** – (default: 1, i.e. no parallel) number of parallel processes
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** $(b \times n_1 \times n_2)$ optimal permutation matrix

---

**Note:** The parallelization is based on multi-processing workers that run on multiple CPU cores.

---

**Note:** For all backends, `scipy.optimize.linear_sum_assignment` is called to solve the LAP, therefore the computation is based on `numpy` and `scipy`. The `backend` argument of this function only affects the input-output data type.

---

**Note:** We support batched instances with different number of nodes, therefore `n1` and `n2` are required to specify the exact number of objects of each dimension in the batch. If not specified, we assume the batched matrices are not padded and all elements in `n1` are equal, all in `n2` are equal.

---

### Numpy Example

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'
>>> np.random.seed(0)

# 2-dimensional (non-batched) input
>>> s_2d = np.random.rand(5, 5)
>>> s_2d
array([[0.5488135 , 0.71518937, 0.60276338, 0.54488318, 0.4236548 ],
       [0.64589411, 0.43758721, 0.891773  , 0.96366276, 0.38344152],
       [0.79172504, 0.52889492, 0.56804456, 0.92559664, 0.07103606],
       [0.0871293 , 0.0202184 , 0.83261985, 0.77815675, 0.87001215],
       [0.97861834, 0.79915856, 0.46147936, 0.78052918, 0.11827443]])

>>> x = pygm.hungarian(s_2d)
>>> x
array([[0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0.]])

# 3-dimensional (batched) input
>>> s_3d = np.random.rand(3, 5, 5)
>>> n1 = n2 = np.array([3, 4, 5])
>>> x = pygm.hungarian(s_3d, n1, n2)
>>> x
array([[[0., 0., 1., 0., 0.],
        [0., 1., 0., 0., 0.],
        [1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]],

       [[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0.]],

       [[0., 0., 1., 0., 0.],
        [1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1.],
        [0., 1., 0., 0., 0.],
        [0., 0., 0., 1., 0.]]])
```

### Pytorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> np.random.seed(0)

# 2-dimensional (non-batched) input
>>> s_2d = torch.from_numpy(np.random.rand(5, 5))
>>> s_2d
tensor([[0.5488, 0.7152, 0.6028, 0.5449, 0.4237],
        [0.6459, 0.4376, 0.8918, 0.9637, 0.3834],
        [0.7917, 0.5289, 0.5680, 0.9256, 0.0710],
        [0.0871, 0.0202, 0.8326, 0.7782, 0.8700],
        [0.9786, 0.7992, 0.4615, 0.7805, 0.1183]], dtype=torch.float64)
>>> x = pygm.hungarian(s_2d)
>>> x
tensor([[0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.],
        [1., 0., 0., 0., 0.]], dtype=torch.float64)

# 3-dimensional (batched) input
>>> s_3d = torch.from_numpy(np.random.rand(3, 5, 5))
>>> n1 = n2 = torch.tensor([3, 4, 5])
>>> x = pygm.hungarian(s_3d, n1, n2)
>>> x
tensor([[[0., 0., 1., 0., 0.],
         [0., 1., 0., 0., 0.],
         [1., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]],

        [[1., 0., 0., 0., 0.],
         [0., 1., 0., 0., 0.],
         [0., 0., 1., 0., 0.],
         [0., 0., 0., 1., 0.],
         [0., 0., 0., 0., 0.]],

        [[0., 0., 1., 0., 0.],
         [1., 0., 0., 0., 0.],
         [0., 0., 0., 0., 1.],
         [0., 1., 0., 0., 0.],
         [0., 0., 0., 1., 0.]]], dtype=torch.float64)
```

#### Paddle Example

```
>>> import paddle
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'paddle'
>>> np.random.seed(0)

# 2-dimensional (non-batched) input
>>> s_2d = paddle.to_tensor(np.random.rand(5, 5))
>>> s_2d
Tensor(shape=[5, 5], dtype=float64, place=Place(cpu), stop_gradient=True,
       [[0.54881350, 0.71518937, 0.60276338, 0.54488318, 0.42365480],
        [0.64589411, 0.43758721, 0.89177300, 0.96366276, 0.38344152],
        [0.79172504, 0.52889492, 0.56804456, 0.92559664, 0.07103606],
        [0.08712930, 0.02021840, 0.83261985, 0.77815675, 0.87001215],
        [0.97861834, 0.79915856, 0.46147936, 0.78052918, 0.11827443]])
>>> x = pygm.hungarian(s_2d)
>>> x
Tensor(shape=[5, 5], dtype=float64, place=Place(cpu), stop_gradient=True,
       [[0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.],
        [1., 0., 0., 0., 0.]])

# 3-dimensional (batched) input
>>> s_3d = paddle.to_tensor(np.random.rand(3, 5, 5))
>>> n1 = n2 = paddle.to_tensor([3, 4, 5])
>>> x = pygm.hungarian(s_3d, n1, n2)
>>> x
Tensor(shape=[3, 5, 5], dtype=float64, place=Place(cpu), stop_gradient=True,
       [[[0., 0., 1., 0., 0.],
         [0., 1., 0., 0., 0.],
         [1., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]],

        [[1., 0., 0., 0., 0.],
         [0., 1., 0., 0., 0.],
         [0., 0., 1., 0., 0.],
         [0., 0., 0., 1., 0.],
         [0., 0., 0., 0., 0.]],

        [[0., 0., 1., 0., 0.],
         [1., 0., 0., 0., 0.],
         [0., 0., 0., 0., 1.],
         [0., 1., 0., 0., 0.],
         [0., 0., 0., 1., 0.]]])
```

### Jittor Example

```
>>> import jittor as jt
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'jittor'
>>> np.random.seed(0)

# 2-dimensional (non-batched) input
>>> s_2d = pygm.utils.from_numpy(np.random.rand(5, 5))
>>> s_2d
jt.Var([[0.5488135  0.71518934 0.60276335 0.5448832  0.4236548 ]
        [0.6458941  0.4375872  0.891773   0.96366274 0.3834415 ]
        [0.79172504 0.5288949  0.56804454 0.92559665 0.07103606]
        [0.0871293  0.0202184  0.83261985 0.77815676 0.87001216]
        [0.9786183  0.7991586  0.46147937 0.7805292  0.11827443]], dtype=float32)
>>> x = pygm.hungarian(s_2d)
>>> x
jt.Var([[0. 1. 0. 0. 0.]
        [0. 0. 1. 0. 0.]
        [0. 0. 0. 1. 0.]
        [0. 0. 0. 0. 1.]
        [1. 0. 0. 0. 0.]], dtype=float32)

# 3-dimensional (batched) input
>>> s_3d = pygm.utils.from_numpy(np.random.rand(3, 5, 5))
>>> n1 = n2 = jt.Var([3, 4, 5])
>>> x = pygm.hungarian(s_3d, n1, n2)
>>> x
jt.Var([[[0. 0. 1. 0. 0.]
        [0. 1. 0. 0. 0.]
        [1. 0. 0. 0. 0.]
        [0. 0. 0. 0. 0.]
        [0. 0. 0. 0. 0.]]

        [[1. 0. 0. 0. 0.]
        [0. 1. 0. 0. 0.]
        [0. 0. 1. 0. 0.]
        [0. 0. 0. 1. 0.]
        [0. 0. 0. 0. 0.]]

        [[0. 0. 1. 0. 0.]
        [1. 0. 0. 0. 0.]
        [0. 0. 0. 0. 1.]
        [0. 1. 0. 0. 0.]
        [0. 0. 0. 1. 0.]]], dtype=float32)
```

**Note:** If you find this graph matching solver useful for your research, please cite:

```
@article{hungarian,
  title={Algorithms for the assignment and transportation problems},
  author={Munkres, James},
  journal={Journal of the society for industrial and applied mathematics},
```

(continues on next page)

(continued from previous page)

```
  volume={5},
  number={1},
  pages={32--38},
  year={1957},
  publisher={SIAM}
}
```

### pygmtools.linear_solvers.sinkhorn

pygmtools.linear_solvers.**sinkhorn**(*s*, *n1=None*, *n2=None*, *dummy_row: bool = False*, *max_iter: int = 10*, *tau: float = 1.0*, *batched_operation: bool = False*, *backend=None*)

Sinkhorn algorithm turns the input matrix into a doubly-stochastic matrix.

Sinkhorn algorithm firstly applies an `exp` function with temperature $\tau$:

$$\mathbf{S}_{i,j} = \exp\left(\frac{\mathbf{s}_{i,j}}{\tau}\right)$$

And then turns the matrix into doubly-stochastic matrix by iterative row- and column-wise normalization:

$$\mathbf{S} = \mathbf{S} \oslash (\mathbf{1}_{n_2}\mathbf{1}_{n_2}^\top \cdot \mathbf{S})$$
$$\mathbf{S} = \mathbf{S} \oslash (\mathbf{S} \cdot \mathbf{1}_{n_2}\mathbf{1}_{n_2}^\top)$$

where $\oslash$ means element-wise division, $\mathbf{1}_n$ means a column-vector with length $n$ whose elements are all 1s.

**Parameters**

- **s** – $(b \times n_1 \times n_2)$ input 3d tensor. $b$: batch size. Non-batched input is also supported if `s` is of size $(n_1 \times n_2)$
- **n1** – (optional) $(b)$ number of objects in dim1
- **n2** – (optional) $(b)$ number of objects in dim2
- **dummy_row** – (default: False) whether to add dummy rows (rows whose elements are all 0) to pad the matrix to square matrix.
- **max_iter** – (default: 10) maximum iterations
- **tau** – (default: 1) the hyper parameter $\tau$ controlling the temperature
- **batched_operation** – (default: False) apply batched_operation for better efficiency (but may cause issues for back-propagation)
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** $(b \times n_1 \times n_2)$ the computed doubly-stochastic matrix

You need not dive too deep into the math details if you are simply using Sinkhorn. However, you should be aware of one important hyper parameter. `tau` controls the distance between the predicted doubly- stochastic matrix, and the discrete permutation matrix computed by Hungarian algorithm (see *hungarian()*). Given a small `tau`, Sinkhorn performs more closely to Hungarian, at the cost of slower convergence speed and reduced numerical stability.

**Note:** We support batched instances with different number of nodes, therefore `n1` and `n2` are required to specify the exact number of objects of each dimension in the batch. If not specified, we assume the batched matrices are not padded and all elements in `n1` are equal, all in `n2` are equal.

**Note:** The original Sinkhorn algorithm only works for square matrices. To handle cases where the graphs to be matched have different number of nodes, it is a common practice to add dummy rows to construct a square matrix. After the row and column normalizations, the padded rows are discarded.

**Note:** Setting `batched_operation=True` may be preferred when you are doing inference with this module and do not need the gradient. It is assumed that `row number <= column number`. If not, the input matrix will be transposed.

### Numpy Example

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'
>>> np.random.seed(0)

# 2-dimensional (non-batched) input
>>> s_2d = np.random.rand(5, 5)
>>> s_2d
array([[0.5488135 , 0.71518937, 0.60276338, 0.54488318, 0.4236548 ],
       [0.64589411, 0.43758721, 0.891773  , 0.96366276, 0.38344152],
       [0.79172504, 0.52889492, 0.56804456, 0.92559664, 0.07103606],
       [0.0871293 , 0.0202184 , 0.83261985, 0.77815675, 0.87001215],
       [0.97861834, 0.79915856, 0.46147936, 0.78052918, 0.11827443]])
>>> x = pygm.sinkhorn(s_2d)
>>> x
array([[0.18880224, 0.24990915, 0.19202217, 0.16034278, 0.20892366],
       [0.18945066, 0.17240445, 0.23345011, 0.22194762, 0.18274716],
       [0.23713583, 0.204348  , 0.18271243, 0.23114583, 0.1446579 ],
       [0.11731039, 0.1229692 , 0.23823909, 0.19961588, 0.32186549],
       [0.26730088, 0.2503692 , 0.15357619, 0.18694789, 0.1418058 ]])

# 3-dimensional (batched) input
>>> s_3d = np.random.rand(3, 5, 5)
>>> x = pygm.sinkhorn(s_3d)
>>> print('row_sum:', x.sum(2))
row_sum: [[1.         1.         1.         1.         1.        ]
 [0.99999998 1.00000002 0.99999999 1.00000003 0.99999999]
 [1.         1.         1.         1.         1.        ]]
>>> print('col_sum:', x.sum(1))
col_sum: [[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]

 # If the 3-d tensor are with different number of nodes
>>> n1 = np.array([3, 4, 5])
>>> n2 = np.array([3, 4, 5])
>>> x = pygm.sinkhorn(s_3d, n1, n2)
>>> x[0] # non-zero size: 3x3
array([[0.36665934, 0.21498158, 0.41835906, 0.        , 0.        ],
```

(continues on next page)

(continued from previous page)

```
       [0.27603621, 0.44270207, 0.28126175, 0.        , 0.        ],
       [0.35730445, 0.34231636, 0.3003792 , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ]])
>>> x[1] # non-zero size: 4x4
array([[0.28847831, 0.20583051, 0.34242091, 0.16327021, 0.        ],
       [0.22656752, 0.30153021, 0.19407969, 0.27782262, 0.        ],
       [0.25346378, 0.19649853, 0.32565049, 0.22438715, 0.        ],
       [0.23149039, 0.29614075, 0.13784891, 0.33452002, 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ]])
>>> x[2] # non-zero size: 5x5
array([[0.20147352, 0.19541986, 0.24942798, 0.17346397, 0.18021467],
       [0.21050732, 0.17620948, 0.18645469, 0.20384684, 0.22298167],
       [0.18319623, 0.18024007, 0.17619871, 0.1664133 , 0.29395169],
       [0.20754376, 0.2236443 , 0.19658101, 0.20570847, 0.16652246],
       [0.19727917, 0.22448629, 0.19133762, 0.25056742, 0.13632951]])

# non-squared input
>>> s_non_square = np.random.rand(4, 5)
>>> x = pygm.sinkhorn(s_non_square, dummy_row=True) # set dummy_row=True for non-
↪squared cases
>>> print('row_sum:', x.sum(1), 'col_sum:', x.sum(0))
row_sum: [1. 1. 1. 1.] col_sum: [0.78239609 0.80485526 0.80165627 0.80004254 0.
↪81104984]
```

### Pytorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> np.random.seed(0)

# 2-dimensional (non-batched) input
>>> s_2d = torch.from_numpy(np.random.rand(5, 5))
>>> s_2d
tensor([[0.5488, 0.7152, 0.6028, 0.5449, 0.4237],
        [0.6459, 0.4376, 0.8918, 0.9637, 0.3834],
        [0.7917, 0.5289, 0.5680, 0.9256, 0.0710],
        [0.0871, 0.0202, 0.8326, 0.7782, 0.8700],
        [0.9786, 0.7992, 0.4615, 0.7805, 0.1183]], dtype=torch.float64)
>>> x = pygm.sinkhorn(s_2d)
>>> x
tensor([[0.1888, 0.2499, 0.1920, 0.1603, 0.2089],
        [0.1895, 0.1724, 0.2335, 0.2219, 0.1827],
        [0.2371, 0.2043, 0.1827, 0.2311, 0.1447],
        [0.1173, 0.1230, 0.2382, 0.1996, 0.3219],
        [0.2673, 0.2504, 0.1536, 0.1869, 0.1418]], dtype=torch.float64)
>>> print('row_sum:', x.sum(1), 'col_sum:', x.sum(0))
row_sum: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000], dtype=torch.float64) col_
↪sum: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000], dtype=torch.float64)
```

(continues on next page)

(continued from previous page)

```
# 3-dimensional (batched) input
>>> s_3d = torch.from_numpy(np.random.rand(3, 5, 5))
>>> x = pygm.sinkhorn(s_3d)
>>> print('row_sum:', x.sum(2))
row_sum: tensor([[1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000, 1.0000, 1.0000]], dtype=torch.float64)
>>> print('col_sum:', x.sum(1))
col_sum: tensor([[1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000, 1.0000, 1.0000]], dtype=torch.float64)

# If the 3-d tensor are with different number of nodes
>>> n1 = torch.tensor([3, 4, 5])
>>> n2 = torch.tensor([3, 4, 5])
>>> x = pygm.sinkhorn(s_3d, n1, n2)
>>> x[0] # non-zero size: 3x3
tensor([[0.3667, 0.2150, 0.4184, 0.0000, 0.0000],
        [0.2760, 0.4427, 0.2813, 0.0000, 0.0000],
        [0.3573, 0.3423, 0.3004, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]], dtype=torch.float64)
>>> x[1] # non-zero size: 4x4
tensor([[0.2885, 0.2058, 0.3424, 0.1633, 0.0000],
        [0.2266, 0.3015, 0.1941, 0.2778, 0.0000],
        [0.2535, 0.1965, 0.3257, 0.2244, 0.0000],
        [0.2315, 0.2961, 0.1378, 0.3345, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]], dtype=torch.float64)
>>> x[2] # non-zero size: 5x5
tensor([[0.2015, 0.1954, 0.2494, 0.1735, 0.1802],
        [0.2105, 0.1762, 0.1865, 0.2038, 0.2230],
        [0.1832, 0.1802, 0.1762, 0.1664, 0.2940],
        [0.2075, 0.2236, 0.1966, 0.2057, 0.1665],
        [0.1973, 0.2245, 0.1913, 0.2506, 0.1363]], dtype=torch.float64)

# non-squared input
>>> s_non_square = torch.from_numpy(np.random.rand(4, 5))
>>> x = pygm.sinkhorn(s_non_square, dummy_row=True) # set dummy_row=True for non-
↪squared cases
>>> print('row_sum:', x.sum(1), 'col_sum:', x.sum(0))
row_sum: tensor([1.0000, 1.0000, 1.0000, 1.0000], dtype=torch.float64) col_sum:␣
↪tensor([0.7824, 0.8049, 0.8017, 0.8000, 0.8110], dtype=torch.float64)
```

### Paddle Example

```
>>> import paddle
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'paddle'
>>> np.random.seed(0)

# 2-dimensional (non-batched) input
>>> s_2d = paddle.to_tensor(np.random.rand(5, 5))
>>> s_2d
Tensor(shape=[5, 5], dtype=float64, place=Place(cpu), stop_gradient=True,
       [[0.54881350, 0.71518937, 0.60276338, 0.54488318, 0.42365480],
        [0.64589411, 0.43758721, 0.89177300, 0.96366276, 0.38344152],
        [0.79172504, 0.52889492, 0.56804456, 0.92559664, 0.07103606],
        [0.08712930, 0.02021840, 0.83261985, 0.77815675, 0.87001215],
        [0.97861834, 0.79915856, 0.46147936, 0.78052918, 0.11827443]])
>>> x = pygm.sinkhorn(s_2d)
>>> x
Tensor(shape=[5, 5], dtype=float64, place=Place(cpu), stop_gradient=True,
       [[0.18880224, 0.24990915, 0.19202217, 0.16034278, 0.20892366],
        [0.18945066, 0.17240445, 0.23345011, 0.22194762, 0.18274716],
        [0.23713583, 0.20434800, 0.18271243, 0.23114583, 0.14465790],
        [0.11731039, 0.12296920, 0.23823909, 0.19961588, 0.32186549],
        [0.26730088, 0.25036920, 0.15357619, 0.18694789, 0.14180580]])
>>> print('row_sum:', x.sum(1), 'col_sum:', x.sum(0))
row_sum: Tensor(shape=[5], dtype=float64, place=Place(cpu), stop_gradient=True,
                [1.00000000, 1.00000001, 0.99999998, 1.00000005, 0.99999997])
col_sum: Tensor(shape=[5], dtype=float64, place=Place(cpu), stop_gradient=True,
                [1.00000000, 1.00000000, 1.00000000, 1.        , 1.00000000])

# 3-dimensional (batched) input
>>> s_3d = paddle.to_tensor(np.random.rand(3, 5, 5))
>>> x = pygm.sinkhorn(s_3d)
>>> print('row_sum:', x.sum(2))
row_sum: Tensor(shape=[3, 5], dtype=float64, place=Place(cpu), stop_gradient=True,
                [[1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000],
                 [0.99999998, 1.00000002, 0.99999999, 1.00000003, 0.99999999],
                 [1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000]])
>>> print('col_sum:', x.sum(1))
col_sum: Tensor(shape=[3, 5], dtype=float64, place=Place(cpu), stop_gradient=True,
                [[1.00000000, 1.        , 1.        , 1.00000000, 1.00000000],
                 [1.        , 1.        , 1.        , 1.        , 1.        ],
                 [1.        , 1.        , 1.        , 1.        , 1.00000000]])

# If the 3-d tensor are with different number of nodes
>>> n1 = paddle.to_tensor([3, 4, 5])
>>> n2 = paddle.to_tensor([3, 4, 5])
>>> x = pygm.sinkhorn(s_3d, n1, n2)
>>> x[0] # non-zero size: 3x3
Tensor(shape=[5, 5], dtype=float64, place=Place(cpu), stop_gradient=True,
       [[0.36665934, 0.21498158, 0.41835906, 0.00000000, 0.00000000],
        [0.27603621, 0.44270207, 0.28126175, 0.00000000, 0.00000000],
        [0.35730445, 0.34231636, 0.30037920, 0.00000000, 0.00000000],
```

(continues on next page)

(continued from previous page)

```
        [0.00000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000],
        [0.00000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000]])
>>> x[1] # non-zero size: 4x4
Tensor(shape=[5, 5], dtype=float64, place=Place(cpu), stop_gradient=True,
       [[0.28847831, 0.20583051, 0.34242091, 0.16327021, 0.00000000],
        [0.22656752, 0.30153021, 0.19407969, 0.27782262, 0.00000000],
        [0.25346378, 0.19649853, 0.32565049, 0.22438715, 0.00000000],
        [0.23149039, 0.29614075, 0.13784891, 0.33452002, 0.00000000],
        [0.00000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000]])
>>> x[2] # non-zero size: 5x5
Tensor(shape=[5, 5], dtype=float64, place=Place(cpu), stop_gradient=True,
       [[0.20147352, 0.19541986, 0.24942798, 0.17346397, 0.18021467],
        [0.21050732, 0.17620948, 0.18645469, 0.20384684, 0.22298167],
        [0.18319623, 0.18024007, 0.17619871, 0.16641330, 0.29395169],
        [0.20754376, 0.22364430, 0.19658101, 0.20570847, 0.16652246],
        [0.19727917, 0.22448629, 0.19133762, 0.25056742, 0.13632951]])

# non-squared input
>>> s_non_square = paddle.to_tensor(np.random.rand(4, 5))
>>> x = pygm.sinkhorn(s_non_square, dummy_row=True) # set dummy_row=True for non-
↪squared cases
>>> print('row_sum:', x.sum(1), 'col_sum:', x.sum(0))
row_sum: Tensor(shape=[4], dtype=float64, place=Place(cpu), stop_gradient=True,
[1.00000000, 1.00000000, 1.00000000, 1.00000000])
col_sum: Tensor(shape=[5], dtype=float64, place=Place(cpu), stop_gradient=True,
[0.78239609, 0.80485526, 0.80165627, 0.80004254, 0.81104984])
```

### Jittor Example

```
>>> import jittor as jt
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'jittor'
>>> np.random.seed(0)

# 2-dimensional (non-batched) input
>>> s_2d = pygm.utils.from_numpy(np.random.rand(5, 5))
>>> s_2d
jt.Var([[0.5488135  0.71518934 0.60276335 0.5448832  0.4236548 ]
        [0.6458941  0.4375872  0.891773   0.96366274 0.3834415 ]
        [0.79172504 0.5288949  0.56804454 0.92559665 0.07103606]
        [0.0871293  0.0202184  0.83261985 0.77815676 0.87001216]
        [0.9786183  0.7991586  0.46147937 0.7805292  0.11827443]], dtype=float32)
>>> x = pygm.sinkhorn(s_2d)
>>> x
jt.Var([[0.18880227 0.24990915 0.19202219 0.1603428  0.20892365]
        [0.18945065 0.17240447 0.23345011 0.22194763 0.18274714]
        [0.23713583 0.20434798 0.18271242 0.23114584 0.1446579 ]
        [0.11731039 0.1229692  0.23823905 0.19961584 0.3218654 ]
        [0.2673009  0.2503692  0.1535762  0.1869479  0.1418058 ]], dtype=float32)
>>> print('row_sum:', x.sum(1), 'col_sum:', x.sum(0))
```

(continues on next page)

(continued from previous page)

```
row_sum: jt.Var([1.0000001  0.99999994 1.         0.9999999  1.        ],␣
↪dtype=float32)
col_sum: jt.Var([1.         1.         1.         1.         0.9999999],␣
↪dtype=float32)

# 3-dimensional (batched) input
>>> s_3d = pygm.utils.from_numpy(np.random.rand(3, 5, 5))
>>> x = pygm.sinkhorn(s_3d)
>>> print('row_sum:', x.sum(2))
row_sum: jt.Var([[1.0000001  0.9999999  0.99999994 1.         0.99999994]
                 [1.         1.0000001  1.         0.99999994 1.        ]
                 [1.         1.         0.99999994 0.99999994 1.        ]],␣
↪dtype=float32)
>>> print('col_sum:', x.sum(1))
col_sum: jt.Var([[1.         0.99999994 1.         0.99999994 1.        ]
                 [1.         1.         1.0000001  1.         0.9999999 ]
                 [0.99999994 1.0000001  0.9999999  1.         1.        ]],␣
↪dtype=float32)

# If the 3-d tensor are with different number of nodes
>>> n1 = jt.Var([3, 4, 5])
>>> n2 = jt.Var([3, 4, 5])
>>> x = pygm.sinkhorn(s_3d, n1, n2)
>>> x[0] # non-zero size: 3x3
jt.Var([[0.3666593  0.21498157 0.41835907 0.         0.        ]
        [0.2760362  0.44270205 0.28126174 0.         0.        ]
        [0.35730445 0.34231633 0.30037922 0.         0.        ]
        [0.         0.         0.         0.         0.        ]
        [0.         0.         0.         0.         0.        ]], dtype=float32)
>>> x[1] # non-zero size: 4x4
jt.Var([[0.28847834 0.20583051 0.34242094 0.16327024 0.        ]
        [0.22656752 0.3015302  0.1940797  0.2778226  0.        ]
        [0.2534638  0.1964985  0.32565048 0.22438715 0.        ]
        [0.23149039 0.2961407  0.13784888 0.33452    0.        ]
        [0.         0.         0.         0.         0.        ]], dtype=float32)
>>> x[2] # non-zero size: 5x5
jt.Var([[0.20147353 0.19541988 0.24942797 0.17346397 0.18021466]
        [0.21050733 0.1762095  0.18645467 0.20384683 0.22298168]
        [0.18319622 0.18024008 0.17619869 0.16641329 0.2939517 ]
        [0.20754376 0.2236443  0.19658099 0.20570846 0.16652244]
        [0.19727917 0.2244863  0.1913376  0.25056744 0.13632952]], dtype=float32)

# non-squared input
>>> s_non_square = pygm.utils.from_numpy(np.random.rand(4, 5))
>>> x = pygm.sinkhorn(s_non_square, dummy_row=True) # set dummy_row=True for non-
↪squared cases
>>> print('row_sum:', x.sum(1), 'col_sum:', x.sum(0))
row_sum: jt.Var([1.         1.         1.         0.99999994], dtype=float32)
col_sum: jt.Var([0.78239614 0.8048552  0.80165625 0.8000425  0.8110498],␣
↪dtype=float32)
```

**Note:** If you find this graph matching solver useful for your research, please cite:

```
@article{sinkhorn,
  title={Concerning nonnegative matrices and doubly stochastic matrices},
  author={Sinkhorn, Richard and Knopp, Paul},
  journal={Pacific Journal of Mathematics},
  volume={21},
  number={2},
  pages={343--348},
  year={1967},
  publisher={Mathematical Sciences Publishers}
}
```

### 8.4.2 pygmtools.classic_solvers

Classic (learning-free) **two-graph matching** solvers. These two-graph matching solvers are recommended to solve matching problems with two explicit graphs, or problems formulated as Quadratic Assignment Problem (QAP).

The two-graph matching problem considers both nodes and edges, formulated as a QAP:

$$\max_{\mathbf{X}} \ \mathtt{vec}(\mathbf{X})^\top \mathbf{K} \mathtt{vec}(\mathbf{X})$$
$$s.t. \quad \mathbf{X} \in \{0,1\}^{n_1 \times n_2},\ \mathbf{X}\mathbf{1} = \mathbf{1},\ \mathbf{X}^\top \mathbf{1} \leq \mathbf{1}$$

#### Functions

| | |
|---|---|
| *ipfp* | Integer Projected Fixed Point (IPFP) method for graph matching (QAP). |
| *rrwm* | Reweighted Random Walk Matching (RRWM) solver for graph matching (QAP). |
| *sm* | Spectral Graph Matching solver for graph matching (QAP). |

#### pygmtools.classic_solvers.ipfp

pygmtools.classic_solvers.**ipfp**(*K*, *n1=None*, *n2=None*, *n1max=None*, *n2max=None*, *x0=None*, *max_iter: int = 50*, *backend=None*)

Integer Projected Fixed Point (IPFP) method for graph matching (QAP).

**Parameters**

- **K** – $(b \times n_1 n_2 \times n_1 n_2)$ the input affinity matrix, $b$: batch size.
- **n1** – $(b)$ number of nodes in graph1 (optional if n1max is given, and all n1=n1max).
- **n2** – $(b)$ number of nodes in graph2 (optional if n2max is given, and all n2=n2max).
- **n1max** – $(b)$ max number of nodes in graph1 (optional if n1 is given, and n1max=max(n1)).
- **n2max** – $(b)$ max number of nodes in graph2 (optional if n2 is given, and n2max=max(n2)).
- **x0** – $(b \times n_1 \times n_2)$ an initial matching solution for warm-start. If not given, x0 will filled with $\frac{1}{n_1 n_2}$).

- **max_iter** – (default: 50) max number of iterations in IPFP. More iterations will be lead to more accurate result, at the cost of increased inference time.
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** $(b \times n_1 \times n_2)$ the solved matching matrix

---

**Note:** Either `n1` or `n1max` should be specified because it cannot be inferred from the input tensor size. Same for `n2` or `n2max`.

---

**Note:** We support batched instances with different number of nodes, therefore `n1` and `n2` are required to specify the exact number of objects of each dimension in the batch. If not specified, we assume the batched matrices are not padded and all elements in `n1` are equal, all in `n2` are equal.

---

**Note:** This function also supports non-batched input, by ignoring all batch dimensions in the input tensors.

---

**Note:** This solver is non-differentiable. The output is a discrete matching matrix (i.e. permutation matrix).

---

### Numpy Example

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'
>>> np.random.seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = np.zeros((batch_size, 4, 4))
>>> X_gt[:, np.arange(0, 4, dtype=np.int64), np.random.permutation(4)] = 1
>>> A1 = np.random.rand(batch_size, 4, 4)
>>> A2 = np.matmul(np.matmul(X_gt.transpose((0, 2, 1)), A1), X_gt)
>>> n1 = n2 = np.repeat([4], batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by IPFP
>>> X = pygm.ipfp(K, n1, n2)
>>> X[0]
array([[0., 0., 0., 1.],
       [0., 0., 1., 0.],
```

(continues on next page)

(continued from previous page)

```
       [1., 0., 0., 0.],
       [0., 1., 0., 0.]])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
1.0
```

### Pytorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = torch.zeros(batch_size, 4, 4)
>>> X_gt[:, torch.arange(0, 4, dtype=torch.int64), torch.randperm(4)] = 1
>>> A1 = torch.rand(batch_size, 4, 4)
>>> A2 = torch.bmm(torch.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> n1 = torch.tensor([4] * batch_size)
>>> n2 = torch.tensor([4] * batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by IPFP
>>> X = pygm.ipfp(K, n1, n2)
>>> X[0]
tensor([[0., 1., 0., 0.],
        [0., 0., 0., 1.],
        [0., 0., 1., 0.],
        [1., 0., 0., 0.]])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
tensor(1.)
```

### Paddle Example

```
>>> import paddle
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'paddle'
>>> _ = paddle.seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = paddle.zeros((batch_size, 4, 4))
>>> X_gt[:, paddle.arange(0, 4, dtype=paddle.int64), paddle.randperm(4)] = 1
>>> A1 = paddle.rand((batch_size, 4, 4))
>>> A2 = paddle.bmm(paddle.bmm(X_gt.transpose((0, 2, 1)), A1), X_gt)
>>> n1 = paddle.to_tensor([4] * batch_size)
>>> n2 = paddle.to_tensor([4] * batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by IPFP
>>> X = pygm.ipfp(K, n1, n2)
>>> X[0]
Tensor(shape=[4, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
       [[0., 1., 0., 0.],
        [0., 0., 0., 1.],
        [0., 0., 1., 0.],
        [1., 0., 0., 0.]])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
Tensor(shape=[1], dtype=float32, place=Place(cpu), stop_gradient=True, [1.])
```

### Jittor Example

```
>>> import jittor as jt
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'jittor'
>>> _ = jt.seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = jt.zeros((batch_size, 4, 4))
>>> X_gt[:, jt.arange(0, 4, dtype=jt.int64), jt.randperm(4)] = 1
>>> A1 = jt.rand(batch_size, 4, 4)
>>> A2 = jt.bmm(jt.bmm(X_gt.transpose(1, 2), A1), X_gt)
```

(continues on next page)

(continued from previous page)

```
>>> n1 = jt.Var([4] * batch_size)
>>> n2 = jt.Var([4] * batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by IPFP
>>> X = pygm.ipfp(K, n1, n2)
>>> X[0]
jt.Var([[1. 0. 0. 0.]
        [0. 0. 1. 0.]
        [0. 0. 0. 1.]
        [0. 1. 0. 0.]], dtype=float32)

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
jt.Var([1.], dtype=float32)
```

**Note:** If you find this graph matching solver useful in your research, please cite:

```
@article{ipfp,
  title={An integer projected fixed point method for graph matching and map␣
↪inference},
  author={Leordeanu, Marius and Hebert, Martial and Sukthankar, Rahul},
  journal={Advances in neural information processing systems},
  volume={22},
  year={2009}
}
```

### pygmtools.classic_solvers.rrwm

pygmtools.classic_solvers.**rrwm**(*K*, *n1=None*, *n2=None*, *n1max=None*, *n2max=None*, *x0=None*, *max_iter: int = 50*, *sk_iter: int = 20*, *alpha: float = 0.2*, *beta: float = 30*, *backend=None*)

Reweighted Random Walk Matching (RRWM) solver for graph matching (QAP). This algorithm is implemented by power iteration with Sinkhorn reweighted jumps.

The official matlab implementation is available at https://cv.snu.ac.kr/research/~RRWM/

**Parameters**

- **K** – $(b \times n_1 n_2 \times n_1 n_2)$ the input affinity matrix, $b$: batch size.
- **n1** – $(b)$ number of nodes in graph1 (optional if n1max is given, and all n1=n1max).
- **n2** – $(b)$ number of nodes in graph2 (optional if n2max is given, and all n2=n2max).

- **n1max** – $(b)$ max number of nodes in graph1 (optional if n1 is given, and n1max=max(n1)).
- **n2max** – $(b)$ max number of nodes in graph2 (optional if n2 is given, and n2max=max(n2)).
- **x0** – $(b \times n_1 \times n_2)$ an initial matching solution for warm-start. If not given, x0 will filled with $\frac{1}{n_1 n_2}$).
- **max_iter** – (default: 50) max number of iterations (i.e. number of random walk steps) in RRWM. More iterations will be lead to more accurate result, at the cost of increased inference time.
- **sk_iter** – (default: 20) max number of Sinkhorn iterations. More iterations will be lead to more accurate result, at the cost of increased inference time.
- **alpha** – (default: 0.2) the parameter controlling the importance of the reweighted jump. alpha should lie between 0 and 1. If `alpha=0`, it means no reweighted jump; if alpha=1, the reweighted jump provides all information.
- **beta** – (default: 30) the temperature parameter of exponential function before the Sinkhorn operator. `beta` should be larger than 0. A larger `beta` means more confidence in the jump. A larger `beta` will usually require a larger `sk_iter`.
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** $(b \times n_1 \times n_2)$ the solved matching matrix

---

**Note:** Either `n1` or `n1max` should be specified because it cannot be inferred from the input tensor size. Same for `n2` or `n2max`.

---

**Note:** We support batched instances with different number of nodes, therefore `n1` and `n2` are required to specify the exact number of objects of each dimension in the batch. If not specified, we assume the batched matrices are not padded and all elements in `n1` are equal, all in `n2` are equal.

---

**Note:** This function also supports non-batched input, by ignoring all batch dimensions in the input tensors.

---

**Note:** This solver is differentiable and supports gradient back-propagation.

---

**Warning:** The solver's output is normalized with a sum of 1, which is in line with the original implementation. If a doubly- stochastic matrix is required, please call `sinkhorn()` after this. If a discrete permutation matrix is required, please call `hungarian()`. Note that the Hungarian algorithm will truncate the gradient and the Sinkhorn algorithm will not.

### Numpy Example

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'
>>> np.random.seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = np.zeros((batch_size, 4, 4))
>>> X_gt[:, np.arange(0, 4, dtype=np.int64), np.random.permutation(4)] = 1
>>> A1 = np.random.rand(batch_size, 4, 4)
>>> A2 = np.matmul(np.matmul(X_gt.transpose((0, 2, 1)), A1), X_gt)
>>> n1 = n2 = np.repeat([4], batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by RRWM. Note that X is normalized with a sum of 1
>>> X = pygm.rrwm(K, n1, n2, beta=100)
>>> X.sum(axis=(1, 2))
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
1.0
```

### Pytorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = torch.zeros(batch_size, 4, 4)
>>> X_gt[:, torch.arange(0, 4, dtype=torch.int64), torch.randperm(4)] = 1
>>> A1 = torch.rand(batch_size, 4, 4)
>>> A2 = torch.bmm(torch.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> n1 = n2 = torch.tensor([4] * batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
```

(continues on next page)

(continued from previous page)

```
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by RRWM. Note that X is normalized with a sum of 1
>>> X = pygm.rrwm(K, n1, n2, beta=100)
>>> X.sum(dim=(1, 2))
tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
        1.0000])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
tensor(1.)

# This solver supports gradient back-propogation
>>> K = K.requires_grad_(True)
>>> pygm.rrwm(K, n1, n2, beta=100).sum().backward()
>>> len(torch.nonzero(K.grad))
272
```

### Paddle Example

```
>>> import paddle
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'paddle'
>>> _ = paddle.seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = paddle.zeros((batch_size, 4, 4))
>>> X_gt[:, paddle.arange(0, 4, dtype=paddle.int64), paddle.randperm(4)] = 1
>>> A1 = paddle.rand((batch_size, 4, 4))
>>> A2 = paddle.bmm(paddle.bmm(X_gt.transpose((0, 2, 1)), A1), X_gt)
>>> n1 = n2 = paddle.to_tensor([4] * batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by RRWM. Note that X is normalized with a sum of 1
>>> X = pygm.rrwm(K, n1, n2, beta=100)
>>> X.sum(axis=(1, 2))
Tensor(shape=[10], dtype=float32, place=Place(cpu), stop_gradient=True,
```

(continues on next page)

(continued from previous page)

```
        [0.99999988, 0.99999988, 0.99999994, 0.99999994, 1.        ,
         1.        , 1.        , 1.00000012, 1.00000012, 1.        ])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
Tensor(shape=[1], dtype=float32, place=Place(cpu), stop_gradient=True, [1.])

# This solver supports gradient back-propogation
>>> K.stop_gradient = False
>>> pygm.rrwm(K, n1, n2, beta=100).sum().backward()
>>> len(paddle.nonzero(K.grad))
544
```

### Jittor Example

```
>>> import jittor as jt
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'jittor'
>>> _ = jt.seed(1)


# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = jt.zeros((batch_size, 4, 4))
>>> X_gt[:, jt.arange(0, 4, dtype=jt.int64), jt.randperm(4)] = 1
>>> A1 = jt.rand(batch_size, 4, 4)
>>> A2 = jt.bmm(jt.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> n1 = n2 = jt.Var([4] * batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by RRWM. Note that X is normalized with a sum of 1
>>> X = pygm.rrwm(K, n1, n2, beta=100)
>>> X.sum(dims=(1, 2))
jt.Var([1.         1.0000001  1.         0.99999976 1.
        1.         1.         1.0000001  0.99999994 1.        ], dtype=float32)

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
jt.Var([1.], dtype=float32)
```

**Note:** If you find this graph matching solver useful in your research, please cite:

```
@inproceedings{rrwm,
  title={Reweighted random walks for graph matching},
  author={Cho, Minsu and Lee, Jungmin and Lee, Kyoung Mu},
  booktitle={European conference on Computer vision},
  pages={492--505},
  year={2010},
  organization={Springer}
}
```

### pygmtools.classic_solvers.sm

pygmtools.classic_solvers.**sm**(*K*, *n1=None*, *n2=None*, *n1max=None*, *n2max=None*, *x0=None*, *max_iter: int = 50*, *backend=None*)

Spectral Graph Matching solver for graph matching (QAP). This algorithm is also known as Power Iteration method, because it works by computing the leading eigenvector of the input affinity matrix by power iteration.

For each iteration,

$$\mathbf{v}_{k+1} = \mathbf{K}\mathbf{v}_k / ||\mathbf{K}\mathbf{v}_k||_2$$

**Parameters**

- **K** – $(b \times n_1 n_2 \times n_1 n_2)$ the input affinity matrix, $b$: batch size.
- **n1** – $(b)$ number of nodes in graph1 (optional if n1max is given, and all n1=n1max).
- **n2** – $(b)$ number of nodes in graph2 (optional if n2max is given, and all n2=n2max).
- **n1max** – $(b)$ max number of nodes in graph1 (optional if n1 is given, and n1max=max(n1)).
- **n2max** – $(b)$ max number of nodes in graph2 (optional if n2 is given, and n2max=max(n2)).
- **x0** – $(b \times n_1 \times n_2)$ an initial matching solution for warm-start. If not given, x0 will be randomly generated.
- **max_iter** – (default: 50) max number of iterations. More iterations will help the solver to converge better, at the cost of increased inference time.
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** $(b \times n_1 \times n_2)$ the solved doubly-stochastic matrix

**Note:** Either `n1` or `n1max` should be specified because it cannot be inferred from the input tensor size. Same for `n2` or `n2max`.

**Note:** We support batched instances with different number of nodes, therefore `n1` and `n2` are required to specify the exact number of objects of each dimension in the batch. If not specified, we assume the batched matrices are not padded and all elements in `n1` are equal, all in `n2` are equal.

**Note:** This function also supports non-batched input, by ignoring all batch dimensions in the input tensors.

**Note:** This solver is differentiable and supports gradient back-propagation.

**Warning:** The solver's output is normalized with a squared sum of 1, which is in line with the original implementation. If a doubly-stochastic matrix is required, please call `sinkhorn()` after this. If a discrete permutation matrix is required, please call `hungarian()`. Note that the Hungarian algorithm will truncate the gradient and the Sinkhorn algorithm will not.

### Numpy Example

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'
>>> np.random.seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = np.zeros((batch_size, 4, 4))
>>> X_gt[:, np.arange(0, 4, dtype=np.int64), np.random.permutation(4)] = 1
>>> A1 = np.random.rand(batch_size, 4, 4)
>>> A2 = np.matmul(np.matmul(X_gt.transpose((0, 2, 1)), A1), X_gt)
>>> n1 = n2 = np.repeat([4], batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by SM. Note that X is normalized with a squared sum of 1
>>> X = pygm.sm(K, n1, n2)
>>> (X ** 2).sum(axis=(1, 2))
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
1.0
```

### Pytorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = torch.zeros(batch_size, 4, 4)
>>> X_gt[:, torch.arange(0, 4, dtype=torch.int64), torch.randperm(4)] = 1
>>> A1 = torch.rand(batch_size, 4, 4)
>>> A2 = torch.bmm(torch.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> n1 = n2 = torch.tensor([4] * batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by SM. Note that X is normalized with a squared sum of 1
>>> X = pygm.sm(K, n1, n2)
>>> (X ** 2).sum(dim=(1, 2))
tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
        1.0000])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
tensor(1.)

# This solver supports gradient back-propogation
>>> K = K.requires_grad_(True)
>>> pygm.sm(K, n1, n2).sum().backward()
>>> len(torch.nonzero(K.grad))
2560
```

### Paddle Example

```
>>> import paddle
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'paddle'
>>> _ = paddle.seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = paddle.zeros((batch_size, 4, 4))
>>> X_gt[:, paddle.arange(0, 4, dtype=paddle.int64), paddle.randperm(4)] = 1
```

(continues on next page)

(continued from previous page)

```
>>> A1 = paddle.rand((batch_size, 4, 4))
>>> A2 = paddle.bmm(paddle.bmm(X_gt.transpose((0, 2, 1)), A1), X_gt)
>>> n1 = n2 = paddle.to_tensor([4] * batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by SM. Note that X is normalized with a squared sum of 1
>>> X = pygm.sm(K, n1, n2)
>>> (X ** 2).sum(axis=(1, 2))
Tensor(shape=[10], dtype=float32, place=Place(cpu), stop_gradient=True,
       [1.        , 1.        , 0.99999994, 0.99999994, 1.00000012,
        1.        , 1.00000012, 1.        , 1.        , 0.99999994])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
Tensor(shape=[1], dtype=float32, place=Place(cpu), stop_gradient=True, [1.])

# This solver supports gradient back-propogation
>>> K.stop_gradient = False
>>> pygm.sm(K, n1, n2).sum().backward()
>>> len(paddle.nonzero(K.grad))
2560
```

### Jittor Example

```
>>> import jittor as jt
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'jittor'
>>> _ = jt.seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = jt.zeros((batch_size, 4, 4))
>>> X_gt[:, jt.arange(0, 4, dtype=jt.int64), jt.randperm(4)] = 1
>>> A1 = jt.rand(batch_size, 4, 4)
>>> A2 = jt.bmm(jt.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> n1 = n2 = jt.Var([4] * batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
```

(continues on next page)

(continued from previous page)

```
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by SM. Note that X is normalized with a squared sum of 1
>>> X = pygm.sm(K, n1, n2)
>>> (X ** 2).sum(dim=1).sum(dim=1)
jt.Var([0.9999998  1.         0.9999999  1.0000001  1.         1.
        0.9999999  0.99999994 1.0000001  1.        ], dtype=float32)

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
jt.Var([1.], dtype=float32)
```

**Note:** If you find this graph matching solver useful for your research, please cite:

```
@inproceedings{sm,
  title={A spectral technique for correspondence problems using pairwise␣
↪constraints},
  author={Leordeanu, Marius and Hebert, Martial},
  year={2005},
  pages={1482-1489},
  booktitle={International Conference on Computer Vision},
  publisher={IEEE}
}
```

### 8.4.3 pygmtools.multi_graph_solvers

Classic (learning-free) **multi-graph matching** solvers. These multi-graph matching solvers are recommended to solve the joint matching problem of multiple graphs.

#### Functions

| | |
|---|---|
| *cao* | Composition based Affinity Optimization (CAO) solver for multi-graph matching. |
| *gamgm* | Graduated Assignment-based multi-graph matching solver. |
| *mgm_floyd* | Multi-Graph Matching based on Floyd shortest path algorithm. |

### pygmtools.multi_graph_solvers.cao

pygmtools.multi_graph_solvers.**cao**(*K*, *x0=None*, *qap_solver=None*, *mode='accu'*, *max_iter=6*, *lambda_init=0.3*, *lambda_step=1.1*, *lambda_max=1.0*, *iter_boost=2*, *backend=None*)

Composition based Affinity Optimization (CAO) solver for multi-graph matching. This solver builds a supergraph for matching update to incorporate the two aspects by optimizing the affinity score, meanwhile gradually infusing the consistency.

Each update step is described as follows:

$$\arg\max_k (1-\lambda) J(\mathbf{X}_{ik}\mathbf{X}_{kj}) + \lambda C_p(\mathbf{X}_{ik}\mathbf{X}_{kj})$$

where $J(\mathbf{X}_{ik}\mathbf{X}_{kj})$ is the objective score, and $C_p(\mathbf{X}_{ik}\mathbf{X}_{kj})$ measures a consistency score compared to other matchings. These two terms are balanced by $\lambda$, and $\lambda$ starts from a smaller number and gradually grows.

**Parameters**

- **K** – $(m \times m \times n^2 \times n^2)$ the input affinity matrix, where `K[i,j]` is the affinity matrix of graph `i` and graph `j` ($m$: number of nodes)
- **x0** – (optional) $(m \times m \times n \times n)$ the initial two-graph matching result, where `X[i,j]` is the matching matrix result of graph `i` and graph `j`. If this argument is not given, `qap_solver` will be used to compute the two-graph matching result.
- **qap_solver** – (default: pygm.rrwm) a function object that accepts a batched affinity matrix and returns the matching matrices. It is suggested to use `functools.partial` and the QAP solvers provided in the *classic_solvers* module (see examples below).
- **mode** – (default: `'accu'`) the operation mode of this algorithm. Options: `'accu'`, `'c'`, `'fast'`, `'pc'`, where `'accu'` is equivalent to `'c'` (accurate version) and `'fast'` is equivalent to `'pc'` (fast version).
- **max_iter** – (default: 6) max number of iterations
- **lambda_init** – (default: 0.3) initial value of $\lambda$, with $\lambda \in [0, 1]$
- **lambda_step** – (default: 1.1) the increase step size of $\lambda$, updated by `lambda = step * lambda`
- **lambda_max** – (default: 1.0) the max value of lambda
- **iter_boost** – (default: 2) to boost the convergence of the CAO algorithm, $\lambda$ will be forced to update every `iter_boost` iterations.
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** $(m \times m \times n \times n)$ the multi-graph matching result

---

**Note:** The input graphs must have the same number of nodes for this algorithm to work correctly.

---

**Note:** Multi-graph matching methods process all graphs at once and do not support the additional batch dimension. Please note that this behavior is different from two-graph matching solvers in *classic_solvers*.

---

### Pytorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate 10 isomorphic graphs
>>> graph_num = 10
>>> As, X_gt = pygm.utils.generate_isomorphic_graphs(node_num=4, graph_num=10)
>>> As_1, As_2 = [], []
>>> for i in range(graph_num):
...     for j in range(graph_num):
...         As_1.append(As[i])
...         As_2.append(As[j])
>>> As_1 = torch.stack(As_1, dim=0)
>>> As_2 = torch.stack(As_2, dim=0)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(As_1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(As_2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, None, None,
↪ None, None, edge_aff_fn=gaussian_aff)
>>> K = K.reshape(graph_num, graph_num, 4*4, 4*4)
>>> K.shape
torch.Size([10, 10, 16, 16])

# Solve the multi-matching problem
>>> X = pygm.cao(K)
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)

# Use the IPFP solver for two-graph matching
>>> ipfp_func = functools.partial(pygmtools.ipfp, n1max=4, n2max=4)
>>> X = pygm.cao(K, qap_solver=ipfp_func)
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)

# Run the faster version of CAO algorithm
>>> X = pygm.cao(K, mode='fast')
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)
```

**Note:** If you find this graph matching solver useful in your research, please cite:

```
@article{cao,
  title={Multi-graph matching via affinity optimization with graduated consistency␣
↪regularization},
  author={Yan, Junchi and Cho, Minsu and Zha, Hongyuan and Yang, Xiaokang and Chu,␣
↪Stephen M},
```

(continues on next page)

(continued from previous page)

```
  journal={IEEE transactions on pattern analysis and machine intelligence},
  volume={38},
  number={6},
  pages={1228--1242},
  year={2015},
  publisher={IEEE}
}
```

### pygmtools.multi_graph_solvers.gamgm

pygmtools.multi_graph_solvers.**gamgm**(*A*, *W*, *ns=None*, *n_univ=None*, *U0=None*, *sk_init_tau=0.5*, *sk_min_tau=0.1*, *sk_gamma=0.8*, *sk_iter=20*, *max_iter=100*, *param_lambda=1.0*, *converge_thresh=1e-05*, *outlier_thresh=- 1*, *bb_smooth=0.1*, *verbose=False*, *backend=None*)

Graduated Assignment-based multi-graph matching solver. Graduated assignment is a classic approach for hard assignment problems like graph matching, based on graduated annealing of Sinkhorn's temperature $\tau$ to enforce the matching constraint.

The objective score is described as

$$\max_{\mathbf{X}_{i,j},i,j\in[m]} \sum_{i,j\in[m]} \left(\lambda \operatorname{tr}(\mathbf{X}_{ij}^\top \mathbf{A}_i \mathbf{X}_{ij} \mathbf{A}_j) + \operatorname{tr}(\mathbf{X}_{ij}^\top \mathbf{W}_{ij})\right)$$

Once the algorithm converges at a fixed $\tau$ value, $\tau$ shrinks as:

$$\tau = \tau \times \gamma$$

and the iteration continues. At last, Hungarian algorithm is applied to ensure the result is a permutation matrix.

**Note:** This algorithm is based on the Koopmans-Beckmann's QAP formulation and you should input the adjacency matrices `A` and node-wise similarity matrices `W` instead of the affinity matrices.

**Parameters**

- **A** – $(m \times n \times n)$ the adjacency matrix ($m$: number of nodes). The graphs may have different number of nodes (specified by the `ns` argument).
- **W** – $(m \times m \times n \times n)$ the node-wise similarity matrix, where `W[i,j]` is the similarity matrix
- **ns** – (optional) $(m)$ the number of nodes. If not given, it will be inferred based on the size of `A`.
- **n_univ** – (optional) the size of the universe node set. If not given, it will be the largest number of nodes.
- **U0** – (optional) the initial multi-graph matching result. If not given, it will be randomly initialized.
- **sk_init_tau** – (default: 0.05) initial value of $\tau$ for Sinkhorn algorithm
- **sk_min_tau** – (default: 1.0e-3) minimal value of $\tau$ for Sinkhorn algorithm
- **sk_gamma** – (default: 0.8) the shrinking parameter of $\tau$: $\tau = \tau \times \gamma$

- **sk_iter** – (default: 200) max number of iterations for Sinkhorn algorithm
- **max_iter** – (default: 1000) max number of iterations for graduated assignment
- **param_lambda** – (default: 1) the weight $\lambda$ of the quadratic term
- **converge_thresh** – (default: 1e-5) if the Frobenius norm of the change of U is smaller than this, the iteration is stopped.
- **outlier_thresh** – (default: -1) if > 0, pairs with node+edge similarity score smaller than this threshold will be discarded. This threshold is designed to handle outliers.
- **bb_smooth** – (default: 0.1) the black-box differentiation smoothing parameter.
- **verbose** – (default: False) print verbose information for parameter tuning
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** the multi-graph matching result (a *MultiMatchingResult* object)

---

**Note:** In PyTorch backend, this function is differentiable through the black-box trick. See the following paper for details:

```
Vlastelica M, Paulus A., Differentiation of Blackbox Combinatorial Solvers, ICLR␣
↪2020
```

If you want to disable this differentiable feature, please detach the input tensors from the computational graph.

---

**Note:** Setting `verbose=True` may help you tune the parameters.

---

### Pytorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> import itertools
>>> import time
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate 10 isomorphic graphs
>>> graph_num = 10
>>> As, X_gt, Fs = pygm.utils.generate_isomorphic_graphs(node_num=4, graph_num=10,␣
↪node_feat_dim=20)

# Compute node-wise similarity by inner-product and Sinkhorn
>>> W = torch.matmul(Fs.unsqueeze(1), Fs.transpose(1, 2).unsqueeze(0))
>>> W = pygm.sinkhorn(W.reshape(graph_num ** 2, 4, 4)).reshape(graph_num, graph_num,
↪ 4, 4)

# Solve the multi-matching problem
>>> X = pygm.gamgm(As, W)
>>> matched = 0
>>> for i, j in itertools.product(range(graph_num), repeat=2):
```

(continues on next page)

(continued from previous page)

```
...     matched += (X[i,j] * X_gt[i,j]).sum()
>>> acc = matched / X_gt.sum()
>>> acc
tensor(1.)

# This function is differentiable by the black-box trick
>>> W.requires_grad_(True)  # tell PyTorch to track the gradients
>>> X = pygm.gamgm(As, W)
>>> matched = 0
>>> for i, j in itertools.product(range(graph_num), repeat=2):
...     matched += (X[i,j] * X_gt[i,j]).sum()
>>> acc = matched / X_gt.sum()

# Backward pass via black-box trick
>>> acc.backward()
>>> torch.sum(W.grad != 0)
tensor(128)

# This function supports graphs with different nodes (also known as partial␣
↪matching)
# In the following we ignore the last node from the last 5 graphs
>>> ns = torch.tensor([4, 4, 4, 4, 4, 3, 3, 3, 3, 3], dtype=torch.int)
>>> for i in range(graph_num):
...     As[i, ns[i]:, :] = 0
...     As[i, :, ns[i]:] = 0
>>> for i, j in itertools.product(range(graph_num), repeat=2):
...     X_gt[i, j, ns[i]:, :] = 0
...     X_gt[i, j, :, ns[j]:] = 0
...     W[i, j, ns[i]:, :] = 0
...     W[i, j, :, ns[j]:] = 0
>>> W = W.detach() # detach tensor if gradient is not needed

# Partial matching is challenging and the following parameters are carefully tuned
>>> X = pygm.gamgm(As, W, ns, n_univ=4, sk_init_tau=.1, sk_min_tau=0.01, param_
↪lambda=0.3)

# Check the partial matching result
>>> matched = 0
>>> for i, j in itertools.product(range(graph_num), repeat=2):
...     matched += (X[i,j] * X_gt[i, j, :ns[i], :ns[j]]).sum()
>>> matched / X_gt.sum()
tensor(1.)
```

**Note:** If you find this graph matching solver useful in your research, please cite:

```
@article{gamgm1,
  title={Graduated assignment algorithm for multiple graph matching based on a␣
↪common labeling},
  author={Sol{\'e}-Ribalta, Albert and Serratosa, Francesc},
  journal={International Journal of Pattern Recognition and Artificial Intelligence}
↪,
```

(continues on next page)

(continued from previous page)

```
  volume={27},
  number={01},
  pages={1350001},
  year={2013},
  publisher={World Scientific}
}

@article{gamgm2,
  title={Graduated assignment for joint multi-graph matching and clustering with␣
↪application to unsupervised graph matching network learning},
  author={Wang, Runzhong and Yan, Junchi and Yang, Xiaokang},
  journal={Advances in Neural Information Processing Systems},
  volume={33},
  pages={19908--19919},
  year={2020}
}
```

This algorithm is originally proposed by paper `gamgm1`, and further improved by paper `gamgm2` to fit modern computing architectures like GPU.

---

### pygmtools.multi_graph_solvers.mgm_floyd

pygmtools.multi_graph_solvers.**mgm_floyd**(*K*, *x0=None*, *qap_solver=None*, *mode='accu'*, *param_lambda=0.2*, *backend=None*)

Multi-Graph Matching based on Floyd shortest path algorithm. A supergraph is considered by regarding each input graph as a node, and the matching between graphs are regraded as edges in the supergraph. Floyd algorithm is used to discover a shortest path on this supergraph for matching update.

The length of edges on the supergraph is described as follows:

$$\arg\max_k (1-\lambda) J(\mathbf{X}_{ik}\mathbf{X}_{kj}) + \lambda C_p(\mathbf{X}_{ik}\mathbf{X}_{kj})$$

where $J(\mathbf{X}_{ik}\mathbf{X}_{kj})$ is the objective score, and $C_p(\mathbf{X}_{ik}\mathbf{X}_{kj})$ measures a consistency score compared to other matchings. These two terms are balanced by $\lambda$.

**Parameters**

- **K** – $(m \times m \times n^2 \times n^2)$ the input affinity matrix, where `K[i,j]` is the affinity matrix of graph `i` and graph `j` ($m$: number of nodes)
- **x0** – (optional) $(m \times m \times n \times n)$ the initial two-graph matching result, where `X[i,j]` is the matching matrix result of graph `i` and graph `j`. If this argument is not given, `qap_solver` will be used to compute the two-graph matching result.
- **qap_solver** – (default: pygm.rrwm) a function object that accepts a batched affinity matrix and returns the matching matrices. It is suggested to use `functools.partial` and the QAP solvers provided in the *classic_solvers* module (see examples below).
- **mode** – (default: `'accu'`) the operation mode of this algorithm. Options: `'accu'`, `'c'`, `'fast'`, `'pc'`, where `'accu'` is equivalent to `'c'` (accurate version) and `'fast'` is equivalent to `'pc'` (fast version).
- **param_lambda** – (default: 0.3) value of $\lambda$, with $\lambda \in [0, 1]$
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** $(m \times m \times n \times n)$ the multi-graph matching result

### Pytorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate 10 isomorphic graphs
>>> graph_num = 10
>>> As, X_gt = pygm.utils.generate_isomorphic_graphs(node_num=4, graph_num=10)
>>> As_1, As_2 = [], []
>>> for i in range(graph_num):
...     for j in range(graph_num):
...         As_1.append(As[i])
...         As_2.append(As[j])
>>> As_1 = torch.stack(As_1, dim=0)
>>> As_2 = torch.stack(As_2, dim=0)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(As_1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(As_2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, None, None,
↪ None, None, edge_aff_fn=gaussian_aff)
>>> K = K.reshape(graph_num, graph_num, 4*4, 4*4)
>>> K.shape
torch.Size([10, 10, 16, 16])

# Solve the multi-matching problem
>>> X = pygm.mgm_floyd(K)
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)

# Use the IPFP solver for two-graph matching
>>> ipfp_func = functools.partial(pygmtools.ipfp, n1max=4, n2max=4)
>>> X = pygm.mgm_floyd(K, qap_solver=ipfp_func)
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)

# Run the faster version of CAO algorithm
>>> X = pygm.mgm_floyd(K, mode='fast')
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)
```

**Note:** If you find this graph matching solver useful in your research, please cite:

```
@article{mgm_floyd,
```

(continues on next page)

(continued from previous page)

```
  title={Unifying offline and online multi-graph matching via finding shortest␣
↪paths on supergraph},
  author={Jiang, Zetian and Wang, Tianzhe and Yan, Junchi},
  journal={IEEE transactions on pattern analysis and machine intelligence},
  volume={43},
  number={10},
  pages={3648--3663},
  year={2020},
  publisher={IEEE}
}
```

### 8.4.4 pygmtools.neural_solvers

**Neural network-based** graph matching solvers. It is recommended to integrate these networks as modules into your existing deep learning pipeline (either supervised, unsupervised or reinforcement learning).

#### Functions

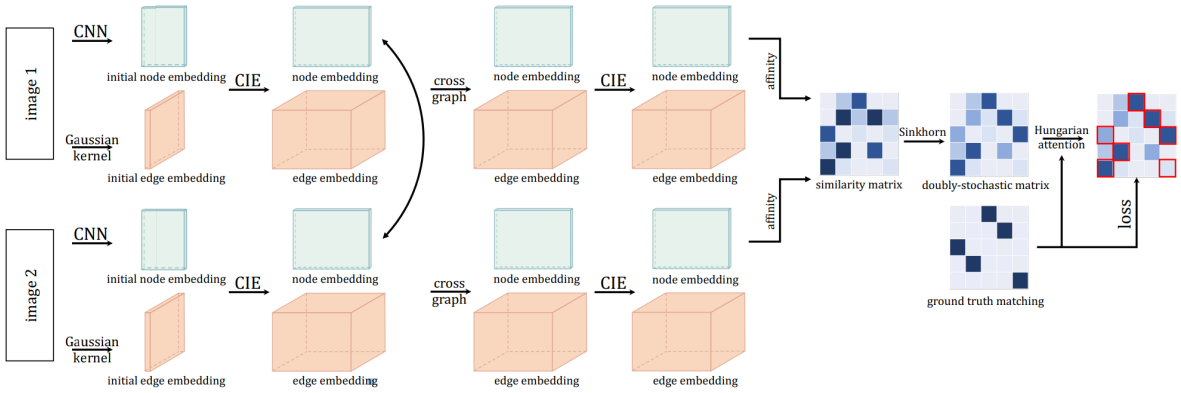
| | |
|---|---|
| *cie* | The **CIE** (Channel Independent Embedding) graph matching neural network model for processing two individual graphs (KB-QAP). |
| *ipca_gm* | The **IPCA-GM** (Iterative Permutation loss and Cross-graph Affinity Graph Matching) neural network model for processing two individual graphs (KB-QAP). |
| *ngm* | The **NGM** (Neural Graph Matching) model for processing the affinity matrix (the most general form of Lawler's QAP). |
| *pca_gm* | The **PCA-GM** (Permutation loss and Cross-graph Affinity Graph Matching) neural network model for processing two individual graphs (KB-QAP). |

#### pygmtools.neural_solvers.cie

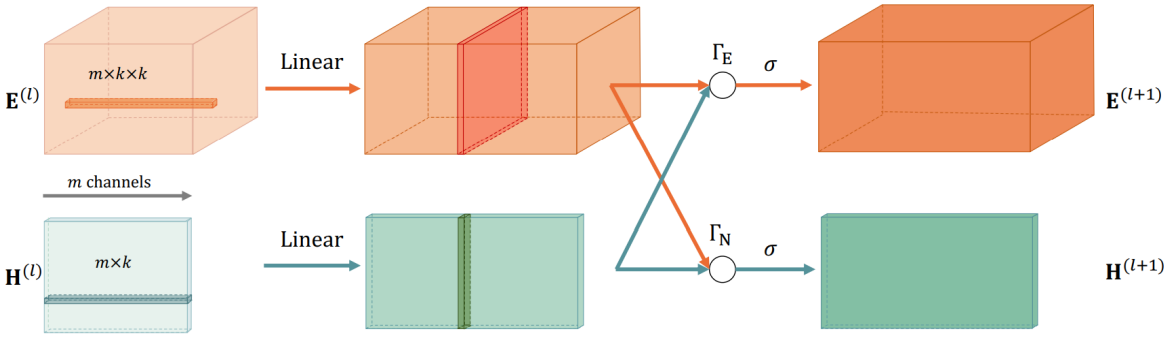
pygmtools.neural_solvers.**cie**(*feat_node1*, *feat_node2*, *A1*, *A2*, *feat_edge1*, *feat_edge2*, *n1=None*, *n2=None*, *in_node_channel=1024*, *in_edge_channel=1*, *hidden_channel=2048*, *out_channel=2048*, *num_layers=2*, *sk_max_iter=20*, *sk_tau=0.05*, *network=None*, *return_network=False*, *pretrain='voc'*, *backend=None*)

The **CIE** (Channel Independent Embedding) graph matching neural network model for processing two individual graphs (KB-QAP). The graph matching module is composed of several intra-graph embedding layers, a cross-graph embedding layer, and a Sinkhorn matching layer. Only the second last layer has a cross-graph update layer. The graph embedding layers are based on channel-independent embedding, under the assumption that such a message passing scheme may offer higher model capacity especially with high-dimensional edge features.

See the following pipeline for an example, with application to visual graph matching:




The graph embedding layer (CIE layer) involves both node embeding and edge embedding:




See the following paper for more technical details: "Yu et al. Learning Deep Graph Matching with Channel-Independent Embedding and Hungarian Attention. ICLR 2020."

**Parameters**

- **feat_node1** – $(b \times n_1 \times d_n)$ input node feature of graph1
- **feat_node2** – $(b \times n_2 \times d_n)$ input node feature of graph2
- **A1** – $(b \times n_1 \times n_1)$ input adjacency matrix of graph1
- **A2** – $(b \times n_2 \times n_2)$ input adjacency matrix of graph2
- **feat_edge1** – $(b \times n_1 \times n_1 \times d_e)$ input edge feature of graph1
- **feat_edge2** – $(b \times n_2 \times n_2 \times d_e)$ input edge feature of graph2
- **n1** – $(b)$ number of nodes in graph1. Optional if all equal to :math:`n_1`
- **n2** – $(b)$ number of nodes in graph2. Optional if all equal to :math:`n_2`
- **in_node_channel** – (default: 1024) Node channel size of the input layer. It must match the feature dimension $(d_n)$ of `feat_node1, feat_node2`. Ignored if the network object is given (ignored if `network!=None`)
- **in_edge_channel** – (default: 1) Edge channel size of the input layer. It must match the feature dimension $(d_e)$ of `feat_edge1, feat_edge2`. Ignored if the network object is given (ignored if `network!=None`)
- **hidden_channel** – (default: 2048) Channel size of hidden layers (node channel == edge channel). Ignored if the network object is given (ignored if `network!=None`)
- **out_channel** – (default: 2048) Channel size of the output layer (node channel == edge channel). Ignored if the network object is given (ignored if `network!=None`)

- **num_layers** – (default: 2) Number of graph embedding layers. Must be >=2. Ignored if the network object is given (ignored if `network!=None`)
- **sk_max_iter** – (default: 20) Max number of iterations of Sinkhorn. See `sinkhorn()` for more details about this argument.
- **sk_tau** – (default: 0.05) The temperature parameter of Sinkhorn. See `sinkhorn()` for more details about this argument.
- **network** – (default: None) The network object. If None, a new network object will be created, and load the model weights specified in `pretrain` argument.
- **return_network** – (default: False) Return the network object (saving model construction time if calling the model multiple times).
- **pretrain** – (default: 'voc') If `network==None`, the pretrained model weights to be loaded. Available pretrained weights: `voc` (on Pascal VOC Keypoint dataset), `willow` (on Willow Object Class dataset), or `False` (no pretraining).
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns**

if `return_network==False`, $(b \times n_1 \times n_2)$ the doubly-stochastic matching matrix

if `return_network==True`, $(b \times n_1 \times n_2)$ the doubly-stochastic matching matrix, the network object

---

**Note:** You may need a proxy to load the pretrained weights if Google drive is not accessible in your contry/region.

---

### PyTorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = torch.zeros(batch_size, 4, 4)
>>> X_gt[:, torch.arange(0, 4, dtype=torch.int64), torch.randperm(4)] = 1
>>> A1 = 1. * (torch.rand(batch_size, 4, 4) > 0.5)
>>> torch.diagonal(A1, dim1=1, dim2=2)[:] = 0 # discard self-loop edges
>>> e_feat1 = (torch.rand(batch_size, 4, 4) * A1).unsqueeze(-1) # shape: (10, 4, 4,␣
↪1)
>>> A2 = torch.bmm(torch.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> e_feat2 = torch.bmm(torch.bmm(X_gt.transpose(1, 2), e_feat1.squeeze(-1)), X_gt).
↪unsqueeze(-1)
>>> feat1 = torch.rand(batch_size, 4, 1024) - 0.5
>>> feat2 = torch.bmm(X_gt.transpose(1, 2), feat1)
>>> n1 = n2 = torch.tensor([4] * batch_size)

# Match by CIE (load pretrained model)
>>> X, net = pygm.cie(feat1, feat2, A1, A2, e_feat1, e_feat2, n1, n2, return_
↪network=True)
```

(continues on next page)

(continued from previous page)

```
Downloading to ~/.cache/pygmtools/cie_voc_pytorch.pt...
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum() # accuracy
tensor(1.)

# Pass the net object to avoid rebuilding the model agian
>>> X = pygm.cie(feat1, feat2, A1, A2, e_feat1, e_feat2, n1, n2, network=net)

# You may also load other pretrained weights
>>> X, net = pygm.cie(feat1, feat2, A1, A2, e_feat1, e_feat2, n1, n2, return_
↪network=True, pretrain='willow')
Downloading to ~/.cache/pygmtools/cie_willow_pytorch.pt...

# You may configure your own model and integrate the model into a deep learning␣
↪pipeline. For example:
>>> net = pygm.utils.get_network(pygm.cie, in_node_channel=1024, in_edge_channel=1,␣
↪hidden_channel=2048, out_channel=512, num_layers=3, pretrain=False)
>>> optimizer = torch.optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
# feat1/feat2/e_feat1/e_feat2 may be outputs by other neural networks
>>> X = pygm.cie(feat1, feat2, A1, A2, e_feat1, e_feat2, n1, n2, network=net)
>>> loss = pygm.utils.permutation_loss(X, X_gt)
>>> loss.backward()
>>> optimizer.step()
```

**Note:** If you find this model useful in your research, please cite:

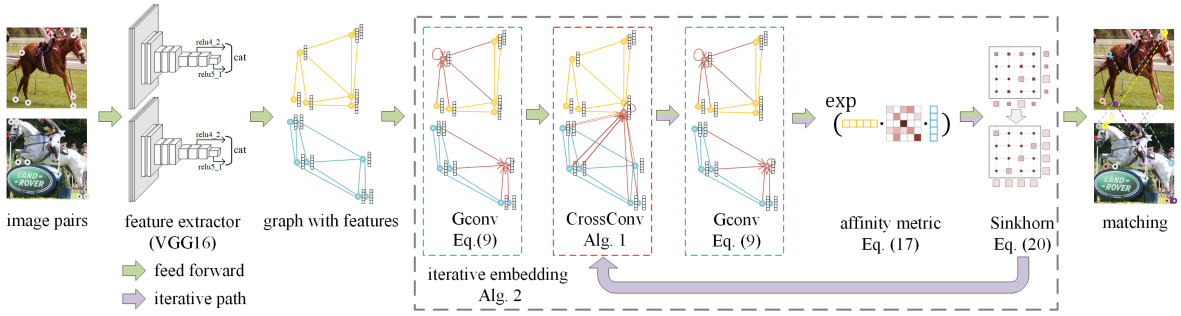
```
@inproceedings{YuICLR20,
  title={Learning deep graph matching with channel-independent embedding and␣
↪Hungarian attention},
  author={Yu, Tianshu and Wang, Runzhong and Yan, Junchi and Li, Baoxin},
  booktitle={International Conference on Learning Representations},
  year={2020}
}
```

### pygmtools.neural_solvers.ipca_gm

pygmtools.neural_solvers.**ipca_gm**(*feat1*, *feat2*, *A1*, *A2*, *n1=None*, *n2=None*, *in_channel=1024*, *hidden_channel=2048*, *out_channel=2048*, *num_layers=2*, *cross_iter=3*, *sk_max_iter=20*, *sk_tau=0.05*, *network=None*, *return_network=False*, *pretrain='voc'*, *backend=None*)

The **IPCA-GM** (Iterative Permutation loss and Cross-graph Affinity Graph Matching) neural network model for processing two individual graphs (KB-QAP). The graph matching module is composed of several intra-graph embedding layers, a cross-graph embedding layer, and a Sinkhorn matching layer. The weight matrix of the cross-graph embedding layer is updated iteratively. Only the second last layer has a cross-graph update layer. IPCA-GM is the extended version of PCA-GM (see *pca_gm()*). The dfference is that the cross-graph weight in PCA-GM is computed in one shot, and in IPCA-GM it is updated iteratively.

See the following pipeline for an example, with application to visual graph matching (layers in gray box are implemented by pygmtools):




See the following paper for more technical details: "Wang et al. Combinatorial Learning of Robust Deep Graph Matching: an Embedding based Approach. TPAMI 2020."

**Parameters**

- **feat1** – $(b \times n_1 \times d)$ input feature of graph1
- **feat2** – $(b \times n_2 \times d)$ input feature of graph2
- **A1** – $(b \times n_1 \times n_1)$ input adjacency matrix of graph1
- **A2** – $(b \times n_2 \times n_2)$ input adjacency matrix of graph2
- **n1** – $(b)$ number of nodes in graph1. Optional if all equal to :math:`n_1`
- **n2** – $(b)$ number of nodes in graph2. Optional if all equal to :math:`n_2`
- **in_channel** – (default: 1024) Channel size of the input layer. It must match the feature dimension $(d)$ of `feat1, feat2`. Ignored if the network object is given (ignored if `network!=None`)
- **hidden_channel** – (default: 2048) Channel size of hidden layers. Ignored if the network object is given (ignored if `network!=None`)
- **out_channel** – (default: 2048) Channel size of the output layer. Ignored if the network object is given (ignored if `network!=None`)
- **num_layers** – (default: 2) Number of graph embedding layers. Must be >=2. Ignored if the network object is given (ignored if `network!=None`)
- **cross_iter** – (default: 3) Number of iterations for the cross-graph embedding layer.
- **sk_max_iter** – (default: 20) Max number of iterations of Sinkhorn. See `sinkhorn()` for more details about this argument.
- **sk_tau** – (default: 0.05) The temperature parameter of Sinkhorn. See `sinkhorn()` for more details about this argument.
- **network** – (default: None) The network object. If None, a new network object will be created, and load the model weights specified in `pretrain` argument.
- **return_network** – (default: False) Return the network object (saving model construction time if calling the model multiple times).
- **pretrain** – (default: 'voc') If `network==None`, the pretrained model weights to be loaded. Available pretrained weights: `voc` (on Pascal VOC Keypoint dataset), `willow` (on Willow Object Class dataset), or `False` (no pretraining).
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns**

if `return_network==False`, $(b \times n_1 \times n_2)$ the doubly-stochastic matching matrix

if `return_network==True`, $(b \times n_1 \times n_2)$ the doubly-stochastic matching matrix, the network object

---

**Note:** You may need a proxy to load the pretrained weights if Google drive is not accessible in your country/region.

---

### PyTorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = torch.zeros(batch_size, 4, 4)
>>> X_gt[:, torch.arange(0, 4, dtype=torch.int64), torch.randperm(4)] = 1
>>> A1 = 1. * (torch.rand(batch_size, 4, 4) > 0.5)
>>> torch.diagonal(A1, dim1=1, dim2=2)[:] = 0 # discard self-loop edges
>>> A2 = torch.bmm(torch.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> feat1 = torch.rand(batch_size, 4, 1024) - 0.5
>>> feat2 = torch.bmm(X_gt.transpose(1, 2), feat1)
>>> n1 = n2 = torch.tensor([4] * batch_size)

# Match by IPCA-GM (load pretrained model)
>>> X, net = pygm.ipca_gm(feat1, feat2, A1, A2, n1, n2, return_network=True)
Downloading to ~/.cache/pygmtools/ipca_gm_voc_pytorch.pt...
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum() # accuracy
tensor(1.)

# Pass the net object to avoid rebuilding the model agian
>>> X = pygm.ipca_gm(feat1, feat2, A1, A2, n1, n2, network=net)

# You may also load other pretrained weights
>>> X, net = pygm.ipca_gm(feat1, feat2, A1, A2, n1, n2, return_network=True,␣
↪pretrain='willow')
Downloading to ~/.cache/pygmtools/ipca_gm_willow_pytorch.pt...

# You may configure your own model and integrate the model into a deep learning␣
↪pipeline. For example:
>>> net = pygm.utils.get_network(in_channel=1024, hidden_channel=2048, out_
↪channel=512, num_layers=3, cross_iter=10, pretrain=False)
>>> optimizer = torch.optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
# feat1/feat2 may be outputs by other neural networks
>>> X = pygm.ipca_gm(feat1, feat2, A1, A2, n1, n2, network=net)
>>> loss = pygm.utils.permutation_loss(X, X_gt)
>>> loss.backward()
>>> optimizer.step()
```

---

**Note:** If you find this model useful in your research, please cite:

```
@article{WangPAMI20,
  author = {Wang, Runzhong and Yan, Junchi and Yang, Xiaokang},
  title = {Combinatorial Learning of Robust Deep Graph Matching: an Embedding based␣
↪Approach},
  journal = {IEEE Transactions on Pattern Analysis and Machine Intelligence},
  year = {2020}
}
```

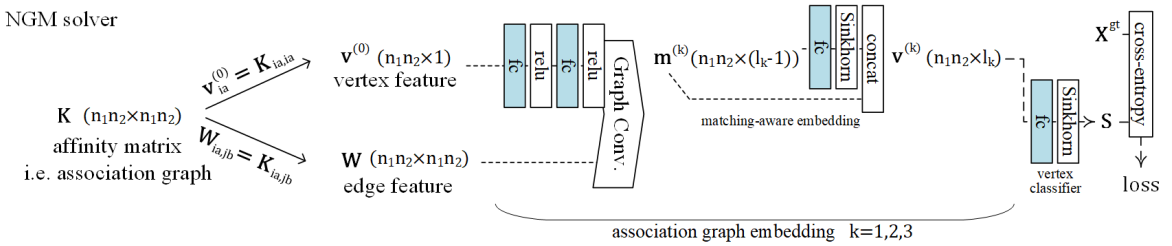
---

### pygmtools.neural_solvers.ngm

pygmtools.neural_solvers.**ngm**(*K*, *n1=None*, *n2=None*, *n1max=None*, *n2max=None*, *x0=None*, *gnn_channels=(16, 16, 16)*, *sk_emb=1*, *sk_max_iter=20*, *sk_tau=0.05*, *network=None*, *return_network=False*, *pretrain='voc'*, *backend=None*)

The **NGM** (Neural Graph Matching) model for processing the affinity matrix (the most general form of Lawler's QAP). The math form of graph matching (Lawler's QAP) is equivalent to a vertex classification problem on the **association graph**, which is an equivalent formulation based on the affinity matrix $\mathbf{K}$. The graph matching module is composed of several graph convolution layers, Sinkhorn embedding layers and finally a Sinkhorn layer to output a doubly-stochastic matrix.

See the following pipeline for an example:




See the following paper for more technical details: "Wang et al. Neural Graph Matching Network: Learning Lawler's Quadratic Assignment Problem With Extension to Hypergraph and Multiple-Graph Matching. TPAMI 2022."

**Parameters**

- **K** – $(b \times n_1 n_2 \times n_1 n_2)$ the input affinity matrix, $b$: batch size.
- **n1** – $(b)$ number of nodes in graph1 (optional if n1max is given, and all n1=n1max).
- **n2** – $(b)$ number of nodes in graph2 (optional if n2max is given, and all n2=n2max).
- **n1max** – $(b)$ max number of nodes in graph1 (optional if n1 is given, and n1max=max(n1)).
- **n2max** – $(b)$ max number of nodes in graph2 (optional if n2 is given, and n2max=max(n2)).
- **x0** – $(b \times n_1 \times n_2)$ an initial matching solution to warm-start the vertex embedding. If not given, the vertex embedding is initialized as a vector of all 1s.
- **gnn_channels** – (default: `(16, 16, 16)`) A list/tuple of channel sizes of the GNN. Ignored if the network object is given (ignored if `network!=None`)
- **sk_emb** – (default: 1) Number of Sinkhorn embedding channels. Sinkhorn embedding is designed to encode the matching constraints inside GNN layers. How it works: a Sinkhorn embedding channel accepts the vertex feature from the current layer and computes a doubly-stochastic matrix, which is then concatenated to the vertex feature. Ignored if the network object is given (ignored if `network!=None`)

- **sk_max_iter** – (default: 20) Max number of iterations of Sinkhorn. See `sinkhorn()` for more details about this argument.
- **sk_tau** – (default: 0.05) The temperature parameter of Sinkhorn. See `sinkhorn()` for more details about this argument.
- **network** – (default: None) The network object. If None, a new network object will be created, and load the model weights specified in `pretrain` argument.
- **return_network** – (default: False) Return the network object (saving model construction time if calling the model multiple times).
- **pretrain** – (default: 'voc') If `network==None`, the pretrained model weights to be loaded. Available pretrained weights: `voc` (on Pascal VOC Keypoint dataset), `willow` (on Willow Object Class dataset), or `False` (no pretraining).
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns**

if `return_network==False`, $(b \times n_1 \times n_2)$ the doubly-stochastic matching matrix

if `return_network==True`, $(b \times n_1 \times n_2)$ the doubly-stochastic matching matrix, the network object

---

**Note:** You may need a proxy to load the pretrained weights if Google drive is not accessible in your contry/region.

---

### PyTorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = torch.zeros(batch_size, 4, 4)
>>> X_gt[:, torch.arange(0, 4, dtype=torch.int64), torch.randperm(4)] = 1
>>> A1 = torch.rand(batch_size, 4, 4)
>>> A2 = torch.bmm(torch.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> n1 = n2 = torch.tensor([4] * batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by NGM
>>> X, net = pygm.ngm(K, n1, n2, return_network=True)
Downloading to ~/.cache/pygmtools/ngm_voc_pytorch.pt...
```

(continues on next page)

(continued from previous page)

```
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum() # accuracy
tensor(1.)

# Pass the net object to avoid rebuilding the model agian
>>> X = pygm.ngm(K, n1, n2, network=net)

# You may also load other pretrained weights
>>> X, net = pygm.ngm(K, n1, n2, return_network=True, pretrain='willow')
Downloading to ~/.cache/pygmtools/ngm_willow_pytorch.pt...

# You may configure your own model and integrate the model into a deep learning␣
↪pipeline. For example:
>>> net = pygm.utils.get_network(pygm.ngm, gnn_channels=(32, 64, 128, 64, 32), sk_
↪emb=8, pretrain=False)
>>> optimizer = torch.optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
# K may be outputs by other neural networks (constructed K from node/edge features␣
↪by pygm.utils.build_aff_mat)
>>> X, net = pygm.ngm(K, n1, n2, network=net)
>>> loss = pygm.utils.permutation_loss(X, X_gt)
>>> loss.backward()
>>> optimizer.step()
```

**Note:** If you find this model useful in your research, please cite:

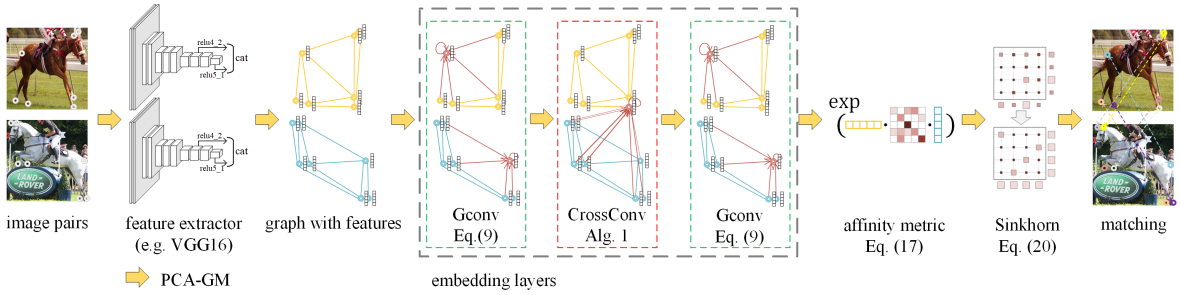
```
@ARTICLE{WangPAMI22,
  author={Wang, Runzhong and Yan, Junchi and Yang, Xiaokang},
  journal={IEEE Transactions on Pattern Analysis and Machine Intelligence},
  title={Neural Graph Matching Network: Learning Lawler's Quadratic Assignment␣
↪Problem With Extension to Hypergraph and Multiple-Graph Matching},
  year={2022},
  volume={44},
  number={9},
  pages={5261-5279},
  doi={10.1109/TPAMI.2021.3078053}
}
```

### pygmtools.neural_solvers.pca_gm

pygmtools.neural_solvers.**pca_gm**(*feat1*, *feat2*, *A1*, *A2*, *n1=None*, *n2=None*, *in_channel=1024*, *hidden_channel=2048*, *out_channel=2048*, *num_layers=2*, *sk_max_iter=20*, *sk_tau=0.05*, *network=None*, *return_network=False*, *pretrain='voc'*, *backend=None*)

The **PCA-GM** (Permutation loss and Cross-graph Affinity Graph Matching) neural network model for processing two individual graphs (KB-QAP). The graph matching module is composed of several intra-graph embedding layers, a cross-graph embedding layer, and a Sinkhorn matching layer. Only the second last layer has a cross-graph update layer.

See the following pipeline for an example, with application to visual graph matching (layers in the gray box are implemented by pygmtools):




See the following paper for more technical details: "Wang et al. Combinatorial Learning of Robust Deep Graph Matching: an Embedding based Approach. TPAMI 2020."

You may be also interested in the extended version IPCA-GM (see *ipca_gm()*).

**Parameters**

- **feat1** – $(b \times n_1 \times d)$ input feature of graph1
- **feat2** – $(b \times n_2 \times d)$ input feature of graph2
- **A1** – $(b \times n_1 \times n_1)$ input adjacency matrix of graph1
- **A2** – $(b \times n_2 \times n_2)$ input adjacency matrix of graph2
- **n1** – $(b)$ number of nodes in graph1. Optional if all equal to :math:`n_1`
- **n2** – $(b)$ number of nodes in graph2. Optional if all equal to :math:`n_2`
- **in_channel** – (default: 1024) Channel size of the input layer. It must match the feature dimension $(d)$ of `feat1, feat2`. Ignored if the network object is given (ignored if `network!=None`)
- **hidden_channel** – (default: 2048) Channel size of hidden layers. Ignored if the network object is given (ignored if `network!=None`)
- **out_channel** – (default: 2048) Channel size of the output layer. Ignored if the network object is given (ignored if `network!=None`)
- **num_layers** – (default: 2) Number of graph embedding layers. Must be >=2. Ignored if the network object is given (ignored if `network!=None`)
- **sk_max_iter** – (default: 20) Max number of iterations of Sinkhorn. See `sinkhorn()` for more details about this argument.
- **sk_tau** – (default: 0.05) The temperature parameter of Sinkhorn. See `sinkhorn()` for more details about this argument.
- **network** – (default: None) The network object. If None, a new network object will be created, and load the model weights specified in `pretrain` argument.
- **return_network** – (default: False) Return the network object (saving model construction time if calling the model multiple times).
- **pretrain** – (default: 'voc') If `network==None`, the pretrained model weights to be loaded. Available pretrained weights: `voc` (on Pascal VOC Keypoint dataset), `willow` (on Willow Object Class dataset), `voc-all` (on Pascal VOC Keypoint dataset, without filtering), or `False` (no pretraining).
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns**

if `return_network==False`, $(b \times n_1 \times n_2)$ the doubly-stochastic matching matrix

if `return_network==True`, $(b \times n_1 \times n_2)$ the doubly-stochastic matching matrix, the network object

---

**Note:** You may need a proxy to load the pretrained weights if Google drive is not accessible in your country/region.

---

### PyTorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = torch.zeros(batch_size, 4, 4)
>>> X_gt[:, torch.arange(0, 4, dtype=torch.int64), torch.randperm(4)] = 1
>>> A1 = 1. * (torch.rand(batch_size, 4, 4) > 0.5)
>>> torch.diagonal(A1, dim1=1, dim2=2)[:] = 0 # discard self-loop edges
>>> A2 = torch.bmm(torch.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> feat1 = torch.rand(batch_size, 4, 1024) - 0.5
>>> feat2 = torch.bmm(X_gt.transpose(1, 2), feat1)
>>> n1 = n2 = torch.tensor([4] * batch_size)

# Match by PCA-GM (load pretrained model)
>>> X, net = pygm.pca_gm(feat1, feat2, A1, A2, n1, n2, return_network=True)
Downloading to ~/.cache/pygmtools/pca_gm_voc_pytorch.pt...
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum() # accuracy
tensor(1.)

# Pass the net object to avoid rebuilding the model agian
>>> X = pygm.pca_gm(feat1, feat2, A1, A2, n1, n2, network=net)

# You may also load other pretrained weights
>>> X, net = pygm.pca_gm(feat1, feat2, A1, A2, n1, n2, return_network=True,␣
↪pretrain='willow')
Downloading to ~/.cache/pygmtools/pca_gm_willow_pytorch.pt...

# You may configure your own model and integrate the model into a deep learning␣
↪pipeline. For example:
>>> net = pygm.utils.get_network(pygm.pca_gm, in_channel=1024, hidden_channel=2048,␣
↪out_channel=512, num_layers=3, pretrain=False)
>>> optimizer = torch.optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
# feat1/feat2 may be outputs by other neural networks
>>> X = pygm.pca_gm(feat1, feat2, A1, A2, n1, n2, network=net)
>>> loss = pygm.utils.permutation_loss(X, X_gt)
>>> loss.backward()
>>> optimizer.step()
```

---

**Note:** If you find this model useful in your research, please cite:

```
@article{WangPAMI20,
  author = {Wang, Runzhong and Yan, Junchi and Yang, Xiaokang},
  title = {Combinatorial Learning of Robust Deep Graph Matching: an Embedding based Approach},
  journal = {IEEE Transactions on Pattern Analysis and Machine Intelligence},
  year = {2020}
}
```

### 8.4.5 pygmtools.utils

Utility functions: problem formulating, data processing, and beyond.

#### Functions

| | |
|---|---|
| *build_aff_mat* | Build affinity matrix for graph matching from input node/edge features. |
| *build_batch* | Build a batched tensor from a list of tensors. |
| *compute_affinity_score* | Compute the affinity score of graph matching. |
| *dense_to_sparse* | Convert a dense connectivity/adjacency matrix to a sparse connectivity/adjacency matrix and an edge weight tensor. |
| *download* | Check if content exits. |
| *from_numpy* | Convert a numpy ndarray to a tensor. |
| *gaussian_aff_fn* | Gaussian kernel affinity function. |
| *generate_isomorphic_graphs* | Generate a set of isomorphic graphs, for testing purposes and examples. |
| *get_network* | Get the network object of a neural network solver. |
| *inner_prod_aff_fn* | Inner product affinity function. |
| *permutation_loss* | Binary cross entropy loss between two permutations, also known as "permutation loss". |
| *to_numpy* | Convert a tensor to a numpy ndarray. |

#### pygmtools.utils.build_aff_mat

pygmtools.utils.**build_aff_mat**(*node_feat1*, *edge_feat1*, *connectivity1*, *node_feat2*, *edge_feat2*, *connectivity2*, *n1=None*, *ne1=None*, *n2=None*, *ne2=None*, *node_aff_fn=None*, *edge_aff_fn=None*, *backend=None*)

Build affinity matrix for graph matching from input node/edge features. The affinity matrix encodes both node-wise and edge-wise affinities and formulates the Quadratic Assignment Problem (QAP), which is the mathematical form of graph matching.

**Parameters**

- **node_feat1** – $(b \times n_1 \times f_{node})$ the node feature of graph1
- **edge_feat1** – $(b \times ne_1 \times f_{edge})$ the edge feature of graph1
- **connectivity1** – $(b \times ne_1 \times 2)$ sparse connectivity information of graph 1. `connectivity1[i, j, 0]` is the starting node index of edge `j` at batch `i`, and `connectivity1[i, j, 1]` is the ending node index of edge `j` at batch `i`

- **node_feat2** – $(b \times n_2 \times f_{node})$ the node feature of graph2
- **edge_feat2** – $(b \times ne_2 \times f_{edge})$ the edge feature of graph2
- **connectivity2** – $(b \times ne_2 \times 2)$ sparse connectivity information of graph 2. `connectivity2[i, j, 0]` is the starting node index of edge `j` at batch `i`, and `connectivity2[i, j, 1]` is the ending node index of edge `j` at batch `i`
- **n1** – $(b)$ number of nodes in graph1. If not given, it will be inferred based on the shape of `node_feat1` or the values in `connectivity1`
- **ne1** – $(b)$ number of edges in graph1. If not given, it will be inferred based on the shape of `edge_feat1`
- **n2** – $(b)$ number of nodes in graph2. If not given, it will be inferred based on the shape of `node_feat2` or the values in `connectivity2`
- **ne2** – $(b)$ number of edges in graph2. If not given, it will be inferred based on the shape of `edge_feat2`
- **node_aff_fn** – (default: inner_prod_aff_fn) the node affinity function with the characteristic `node_aff_fn(2D Tensor, 2D Tensor) -> 2D Tensor`, which accepts two node feature tensors and outputs the node-wise affinity tensor. See *inner_prod_aff_fn()* as an example.
- **edge_aff_fn** – (default: inner_prod_aff_fn) the edge affinity function with the characteristic `edge_aff_fn(2D Tensor, 2D Tensor) -> 2D Tensor`, which accepts two edge feature tensors and outputs the edge-wise affinity tensor. See *inner_prod_aff_fn()* as an example.
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** $(b \times n_1 n_2 \times n_1 n_2)$ the affinity matrix

---

**Note:** This function also supports non-batched input, by ignoring all batch dimensions in the input tensors.

---

### Numpy Example

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'

# Generate a batch of graphs
>>> batch_size = 10
>>> A1 = np.random.rand(batch_size, 4, 4)
>>> A2 = np.random.rand(batch_size, 4, 4)
>>> n1 = n2 = np.repeat([4], batch_size)

# Build affinity matrix by the default inner-product function
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1,␣
↪n2, ne2)

# Build affinity matrix by gaussian kernel
>>> import functools
```

(continues on next page)

(continued from previous page)

```
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.)
>>> K2 = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1,␣
↪n2, ne2, edge_aff_fn=gaussian_aff)

# Build affinity matrix based on node features
>>> F1 = np.random.rand(batch_size, 4, 10)
>>> F2 = np.random.rand(batch_size, 4, 10)
>>> K3 = pygm.utils.build_aff_mat(F1, edge1, conn1, F2, edge2, conn2, n1, ne1, n2,␣
↪ne2, edge_aff_fn=gaussian_aff)

# The affinity matrices K, K2, K3 can be further processed by GM solvers
```

### Pytorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'

# Generate a batch of graphs
>>> batch_size = 10
>>> A1 = torch.rand(batch_size, 4, 4)
>>> A2 = torch.rand(batch_size, 4, 4)
>>> n1 = n2 = torch.tensor([4] * batch_size)

# Build affinity matrix by the default inner-product function
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1,␣
↪n2, ne2)

# Build affinity matrix by gaussian kernel
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.)
>>> K2 = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1,␣
↪n2, ne2, edge_aff_fn=gaussian_aff)

# Build affinity matrix based on node features
>>> F1 = torch.rand(batch_size, 4, 10)
>>> F2 = torch.rand(batch_size, 4, 10)
>>> K3 = pygm.utils.build_aff_mat(F1, edge1, conn1, F2, edge2, conn2, n1, ne1, n2,␣
↪ne2, edge_aff_fn=gaussian_aff)

# The affinity matrices K, K2, K3 can be further processed by GM solvers
```

### Paddle Example

::

```
>>> import paddle
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'paddle'
```

# Generate a batch of graphs >>> batch_size = 10 >>> A1 = paddle.rand((batch_size, 4, 4)) >>> A2 = paddle.rand((batch_size, 4, 4)) >>> n1 = n2 = paddle.t0_tensor([4] * batch_size)

# Build affinity matrix by the default inner-product function >>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1) >>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2) >>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1, n2, ne2)

# Build affinity matrix by gaussian kernel >>> import functools >>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) >>> K2 = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1, n2, ne2, edge_aff_fn=gaussian_aff)

# Build affinity matrix based on node features >>> F1 = paddle.rand((batch_size, 4, 10)) >>> F2 = paddle.rand((batch_size, 4, 10)) >>> K3 = pygm.utils.build_aff_mat(F1, edge1, conn1, F2, edge2, conn2, n1, ne1, n2, ne2, edge_aff_fn=gaussian_aff)

# The affinity matrices K, K2, K3 can be further processed by GM solvers

## pygmtools.utils.build_batch

pygmtools.utils.**build_batch**(*input*, *return_ori_dim=False*, *backend=None*)

Build a batched tensor from a list of tensors. If the list of tensors are with different sizes of dimensions, it will be padded to the largest dimension.

The batched tensor and the number of original dimensions will be returned.

**Parameters**

- **input** – list of input tensors
- **return_ori_dim** – (default: False) return the original dimension
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** batched tensor, (if `return_ori_dim=True`) a list of the original dimensions

### Numpy Example

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'

# batched adjacency matrices
>>> A1 = np.random.rand(4, 4)
>>> A2 = np.random.rand(5, 5)
>>> A3 = np.random.rand(3, 3)
>>> batched_A, ori_shape = pygm.utils.build_batch([A1, A2, A3], return_ori_dim=True)
>>> batched_A.shape
(3, 5, 5)
```

(continues on next page)

(continued from previous page)

```
>>> ori_shape
([4, 5, 3], [4, 5, 3])

# batched node features (feature dimension=10)
>>> F1 = np.random.rand(4, 10)
>>> F2 = np.random.rand(5, 10)
>>> F3 = np.random.rand(3, 10)
>>> batched_F = pygm.utils.build_batch([F1, F2, F3])
>>> batched_F.shape
(3, 5, 10)
```

### Pytorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'

# batched adjacency matrices
>>> A1 = torch.rand(4, 4)
>>> A2 = torch.rand(5, 5)
>>> A3 = torch.rand(3, 3)
>>> batched_A, ori_shape = pygm.utils.build_batch([A1, A2, A3], return_ori_dim=True)
>>> batched_A.shape
torch.Size([3, 5, 5])
>>> ori_shape
(tensor([4, 5, 3]), tensor([4, 5, 3]))

# batched node features (feature dimension=10)
>>> F1 = torch.rand(4, 10)
>>> F2 = torch.rand(5, 10)
>>> F3 = torch.rand(3, 10)
>>> batched_F = pygm.utils.build_batch([F1, F2, F3])
>>> batched_F.shape
torch.Size([3, 5, 10])
```

### Paddle Example

```
>>> import paddle
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'paddle'

# batched adjacency matrices
>>> A1 = paddle.rand((4, 4))
>>> A2 = paddle.rand((5, 5))
>>> A3 = paddle.rand((3, 3))
>>> batched_A, ori_shape = pygm.utils.build_batch([A1, A2, A3], return_ori_dim=True)
>>> batched_A.shape
[3, 5, 5]
>>> ori_shape
```

(continues on next page)

(continued from previous page)

```
(Tensor(shape=[3], dtype=int64, place=Place(cpu), stop_gradient=True, [4, 5, 3]),
 Tensor(shape=[3], dtype=int64, place=Place(cpu), stop_gradient=True, [4, 5, 3]))

# batched node features (feature dimension=10)
>>> F1 = paddle.rand((4, 10))
>>> F2 = paddle.rand((5, 10))
>>> F3 = paddle.rand((3, 10))
>>> batched_F = pygm.utils.build_batch([F1, F2, F3])
>>> batched_F.shape
[3, 5, 10]
```

### pygmtools.utils.compute_affinity_score

pygmtools.utils.**compute_affinity_score**(*X*, *K*, *backend=None*)

Compute the affinity score of graph matching. It is the objective score of the corresponding Quadratic Assignment Problem.

$$\texttt{vec}(\mathbf{X})^\top \mathbf{K} \texttt{vec}(\mathbf{X})$$

here vec means column-wise vectorization.

**Parameters**

- **X** – $(b \times n_1 \times n_2)$ the permutation matrix that represents the matching result
- **K** – $(b \times n_1 n_2 \times n_1 n_2)$ the affinity matrix
- **backend** – (default: pygmtools.BACKEND variable) the backend for computation.

**Returns** $(b)$ the objective score

---

**Note:** This function also supports non-batched input if the batch dimension of X, K is ignored.

---

#### Pytorch Example

```
>>> import pygmtools as pygm
>>> import torch
>>> pygm.BACKEND = 'pytorch'

# Generate a graph matching problem
>>> X_gt = torch.zeros(4, 4)
>>> X_gt[torch.arange(0, 4, dtype=torch.int64), torch.randperm(4)] =1
>>> A1 = torch.rand(4, 4)
>>> A2 = torch.mm(torch.mm(X_gt.transpose(0,1), A1), X_gt)
>>> conn1, edge1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.)
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, None, None,
↪ None, None, edge_aff_fn=gaussian_aff)
```

(continues on next page)

(continued from previous page)

```
# Compute the objective score of ground truth matching
>>> pygm.utils.compute_affinity_score(X_gt, K)
tensor(16.)
```

### pygmtools.utils.dense_to_sparse

pygmtools.utils.**dense_to_sparse**(*dense_adj*, *backend=None*)

Convert a dense connectivity/adjacency matrix to a sparse connectivity/adjacency matrix and an edge weight tensor.

**Parameters**

- **dense_adj** – $(b \times n \times n)$ the dense adjacency matrix. This function also supports non-batched input where the batch dimension b is ignored
- **backend** – (default: pygmtools.BACKEND variable) the backend for computation.

**Returns**

if batched input: $(b \times ne \times 2)$ sparse connectivity matrix, $(b \times ne \times 1)$ edge weight tensor, $(b)$ number of edges

if non-batched input: $(ne \times 2)$ sparse connectivity matrix, $(ne \times 1)$ edge weight tensor,

#### Numpy Example

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'
>>> np.random.seed(0)

>>> batch_size = 10
>>> A = np.random.rand(batch_size, 4, 4)
>>> A[:, np.arange(4), np.arange(4)] = 0 # remove the diagonal elements
>>> A.shape
(10, 4, 4)

>>> conn, edge, ne = pygm.utils.dense_to_sparse(A)
>>> conn.shape # connectivity: (batch x num_edge x 2)
(10, 12, 2)

>>> edge.shape # edge feature (batch x num_edge x feature_dim)
(10, 12, 1)

>>> ne
[12, 12, 12, 12, 12, 12, 12, 12, 12, 12]
```

### Pytorch Example

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(0)

>>> batch_size = 10
>>> A = torch.rand(batch_size, 4, 4)
>>> torch.diagonal(A, dim1=1, dim2=2)[:] = 0 # remove the diagonal elements
>>> A.shape
torch.Size([10, 4, 4])

>>> conn, edge, ne = pygm.utils.dense_to_sparse(A)
>>> conn.shape # connectivity: (batch x num_edge x 2)
torch.Size([10, 12, 2])

>>> edge.shape # edge feature (batch x num_edge x feature_dim)
torch.Size([10, 12, 1])

>>> ne
tensor([12, 12, 12, 12, 12, 12, 12, 12, 12, 12])
```

### Paddle Example

```
>>> import paddle
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'paddle'
>>> paddle.seed(0)

>>> batch_size = 10
>>> A = paddle.rand((batch_size, 4, 4))
>>> paddle.diagonal(A, axis1=1, axis2=2)[:] = 0 # remove the diagonal elements
>>> A.shape
[10, 4, 4]

>>> conn, edge, ne = pygm.utils.dense_to_sparse(A)
>>> conn.shape # connectivity: (batch x num_edge x 2)
torch.Size([10, 16, 2])

>>> edge.shape # edge feature (batch x num_edge x feature_dim)
torch.Size([10, 16, 1])

>>> ne
Tensor(shape=[10], dtype=int64, place=Place(cpu), stop_gradient=True,
       [16, 16, 16, 16, 16, 16, 16, 16, 16, 16])
```

### pygmtools.utils.download

pygmtools.utils.**download**(*filename*, *url*, *md5=None*, *retries=5*)

Check if content exits. If not, download the content to <user cache path>/pygmtools/<filename>. <user cache path> depends on your system. For example, on Debian, it should be $HOME/.cache.

**Parameters**

- **filename** – the destination file name
- **url** – the url
- **md5** – (optional) the md5sum to verify the content. It should match the result of `md5sum file` on Linux.
- **retries** – (default: 5) max number of retries

**Returns** the full path to the file: <user cache path>/pygmtools/<filename>

### pygmtools.utils.from_numpy

pygmtools.utils.**from_numpy**(*input*, *device=None*, *backend=None*)

Convert a numpy ndarray to a tensor. This is the helper function to convert tensors across different backends via numpy.

**Parameters**

- **input** – input ndarray/*MultiMatchingResult*
- **device** – (default: None) the target device
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** tensor for the backend

### pygmtools.utils.gaussian_aff_fn

pygmtools.utils.**gaussian_aff_fn**(*feat1*, *feat2*, *sigma=1.0*, *backend=None*)

Gaussian kernel affinity function. The affinity is defined as

$$\exp(-\frac{(\mathbf{f}_1 - \mathbf{f}_2)^2}{\sigma})$$

**Parameters**

- **feat1** – $(b \times n_1 \times f)$ the feature vectors $\mathbf{f}_1$
- **feat2** – $(b \times n_2 \times f)$ the feature vectors $\mathbf{f}_2$
- **sigma** – (default: 1) the parameter $\sigma$ in Gaussian kernel
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns** $(b \times n_1 \times n_2)$ element-wise Gaussian affinity matrix

### pygmtools.utils.generate_isomorphic_graphs

pygmtools.utils.**generate_isomorphic_graphs**(*node_num*, *graph_num=2*, *node_feat_dim=0*, *backend=None*)

Generate a set of isomorphic graphs, for testing purposes and examples.

**Parameters**

- **node_num** – number of nodes in each graph
- **graph_num** – (default: 2) number of graphs
- **node_feat_dim** – (default: 0) number of node feature dimensions
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

**Returns**

if `graph_num==2`, this function returns $(m \times n \times n)$ the adjacency matrix, and $(n \times n)$ the permutation matrix;

else, this function returns $(m \times n \times n)$ the adjacency matrix, and $(m \times m \times n \times n)$ the multi-matching permutation matrix

### pygmtools.utils.get_network

pygmtools.utils.**get_network**(*nn_solver_func*, ***params*)

Get the network object of a neural network solver.

**Parameters**

- **nn_solver_func** – the neural network solver function, for example `pygm.pca_gm`
- **params** – keyword parameters to define the neural network

**Returns** the network object

#### Pytorch Example

```
>>> import pygmtools as pygm
>>> import torch
>>> pygm.BACKEND = 'pytorch'
>>> pygm.utils.get_network(pygm.pca_gm, pretrain='willow')
PCA_GM_Net(
  (gnn_layer_0): Siamese_Gconv(
    (gconv): Gconv(
      (a_fc): Linear(in_features=1024, out_features=2048, bias=True)
      (u_fc): Linear(in_features=1024, out_features=2048, bias=True)
    )
  )
  (cross_graph_0): Linear(in_features=4096, out_features=2048, bias=True)
  (affinity_0): WeightedInnerProdAffinity()
  (affinity_1): WeightedInnerProdAffinity()
  (gnn_layer_1): Siamese_Gconv(
    (gconv): Gconv(
      (a_fc): Linear(in_features=2048, out_features=2048, bias=True)
      (u_fc): Linear(in_features=2048, out_features=2048, bias=True)
```

(continues on next page)

(continued from previous page)

```
    )
  )
)

# the neural network can be integrated into a deep learning pipeline
>>> net = pygm.utils.get_network(pygm.pca_gm, in_channel=1024, hidden_channel=2048,␣
↪out_channel=512, num_layers=3, pretrain=False)
>>> optimizer = torch.optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
```

### pygmtools.utils.inner_prod_aff_fn

pygmtools.utils.**inner_prod_aff_fn**(*feat1*, *feat2*, *backend=None*)

Inner product affinity function. The affinity is defined as

$$\mathbf{f}_1^\top \cdot \mathbf{f}_2$$

**Parameters**

- **feat1** – $(b \times n_1 \times f)$ the feature vectors $\mathbf{f}_1$
- **feat2** – $(b \times n_2 \times f)$ the feature vectors $\mathbf{f}_2$
- **backend** – (default: pygmtools.BACKEND variable) the backend for computation.

**Returns** $(b \times n_1 \times n_2)$ element-wise inner product affinity matrix

### pygmtools.utils.permutation_loss

pygmtools.utils.**permutation_loss**(*pred_dsmat*, *gt_perm*, *n1=None*, *n2=None*, *backend=None*)

Binary cross entropy loss between two permutations, also known as "permutation loss". Proposed by "Wang et al. Learning Combinatorial Embedding Networks for Deep Graph Matching. ICCV 2019."

$$L_{perm} = -\sum_{i\in\mathcal{V}_1, j\in\mathcal{V}_2} \left(\mathbf{X}_{i,j}^{gt}\log\mathbf{S}_{i,j} + (1-\mathbf{X}_{i,j}^{gt})\log(1-\mathbf{S}_{i,j})\right)$$

where $\mathcal{V}_1, \mathcal{V}_2$ are vertex sets for two graphs.

**Parameters**

- **pred_dsmat** – $(b \times n_1 \times n_2)$ predicted doubly-stochastic matrix $(\mathbf{S})$
- **gt_perm** – $(b \times n_1 \times n_2)$ ground truth permutation matrix $(\mathbf{X}^{gt})$
- **n1** – (optional) $(b)$ number of exact pairs in the first graph.
- **n2** – (optional) $(b)$ number of exact pairs in the second graph.
- **backend** – (default: pygmtools.BACKEND variable) the backend for computation.

**Returns** $(1)$ averaged permutation loss

---

**Note:** We support batched instances with different number of nodes, therefore n1 and n2 are required if you want to specify the exact number of nodes of each instance in the batch.

---

**Note:** For batched input, this loss function computes the averaged loss among all instances in the batch. This function also supports non-batched input if the batch dimension ($b$) is ignored.

### pygmtools.utils.to_numpy

pygmtools.utils.**to_numpy**(*input*, *backend=None*)

Convert a tensor to a numpy ndarray. This is the helper function to convert tensors across different backends via numpy.

**Parameters**

- **input** – input tensor/*MultiMatchingResult*
- **backend** – (default: pygmtools.BACKEND variable) the backend for computation.

**Returns** numpy ndarray

### Classes

| | |
|---|---|
| *MultiMatchingResult* | A memory-efficient class for multi-graph matching results. |

### MultiMatchingResult

**class** pygmtools.utils.**MultiMatchingResult**(*cycle_consistent=False*, *backend=None*)

A memory-efficient class for multi-graph matching results. For non-cycle consistent results, the dense storage for $m$ graphs with $n$ nodes requires a size of $(m \times m \times n \times n)$, and this implementation requires $((m-1) \times m \times n \times n/2)$. For cycle consistent result, this implementation requires only $(m \times n \times n)$.

#### Numpy Example

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> np.random.seed(0)

>>> X = pygm.utils.MultiMatchingResult(backend='numpy')
>>> X[0, 1] = np.zeros((4, 4))
>>> X[0, 1][np.arange(0, 4, dtype=np.int64), np.random.permutation(4)] = 1
>>> X
MultiMatchingResult:
{'0,1': array([[0., 0., 1., 0.],
    [0., 0., 0., 1.],
    [0., 1., 0., 0.],
    [1., 0., 0., 0.]])}
>>> X[1, 0]
array([[0., 0., 0., 1.],
    [0., 0., 1., 0.],
    [1., 0., 0., 0.],
    [0., 1., 0., 0.]])
```

**static from_numpy**(*data*, *device=None*, *new_backend=None*)

Convert a numpy-backend MultiMatchingResult data to another backend.

**Parameters**

- **data** – the numpy-backend data
- **device** – (default: None) the target device
- **new_backend** – (default: `pygmtools.BACKEND` variable) the target backend

**Returns** a new MultiMatchingResult instance for `new_backend` on `device`

**from_numpy_**(*device=None*, *new_backend=None*)

In-place operation for *from_numpy()*.

**static to_numpy**(*data*)

Convert an any-type MultiMatchingResult to numpy backend.

**Parameters** **data** – the any-type data

**Returns** a new MultiMatchingResult instance for numpy

**to_numpy_**()

In-place operation for *to_numpy()*.

### 8.4.6 pygmtools.benchmark

The Benchmark module with a unified data interface to evaluate graph matching methods.

If you are interested in the performance and the deep learning framework, please refer to our ThinkMatch project.

#### Classes

| | |
|---|---|
| *Benchmark* | The *Benchmark* module provides a unified data interface and an evaluating platform for different datasets. |

#### Benchmark

**class** pygmtools.benchmark.**Benchmark**(*name*, *sets*, *obj_resize=(256, 256)*, *problem='2GM'*, *filter='intersection'*, ***args*)

The *Benchmark* module provides a unified data interface and an evaluating platform for different datasets.

**Parameters**

- **name** – str, dataset name, currently support `'PascalVOC'`, `'WillowObject'`, `'IMC_PT_SparseGM'`, `'CUB2011'`, `'SPair71k'`
- **sets** – str, problem set, `'train'` for training set and `'test'` for test set
- **obj_resize** – tuple, (default: `(256, 256)`) resized object size
- **problem** – str, (default: `'2GM'`) problem type, `'2GM'` for 2-graph matching and `'MGM'` for multi-graph matching

- **filter** – str, (default: `'intersection'`) filter of nodes, `'intersection'` refers to retaining only common nodes; `'inclusion'` is only for 2GM and refers to filtering only one graph to make its nodes a subset of the other graph, and `'unfiltered'` refers to retaining all nodes in all graphs
- **args** – keyword settings for specific dataset

**compute_img_num**(*classes*)

Compute number of images in specified classes.

**Parameters** **classes** – list of dataset classes

**Returns** list of numbers of images in each class

**compute_length**(*cls=None*, *num=2*)

Compute the length of image combinations in specified class.

**Parameters**

- **cls** – int or str, class of expected data. None for all classes
- **num** – int, number of images in each image ID list; for example, 2 for two-graph matching problem

**Returns** length of combinations

**eval**(*prediction*, *classes*, *verbose=False*)

Evaluate test results and compute matching accuracy and coverage.

**Parameters**

- **prediction** – list, prediction result, like `[{'ids': (id1, id2), 'cls': cls, 'permmat': np.array or scipy.sparse}, ...]`
- **classes** – list of evaluated classes
- **verbose** – bool, whether to print the result

**Returns** evaluation result in each class and their averages, including p, r, f1 and their standard deviation and coverage

**eval_cls**(*prediction*, *cls*, *verbose=False*)

Evaluate test results and compute matching accuracy and coverage on one specified class.

**Parameters**

- **prediction** – list, prediction result on one class, like `[{'ids': (id1, id2), 'cls': cls, 'permmat': np.array or scipy.sparse}, ...]`
- **cls** – str, evaluated class
- **verbose** – bool, whether to print the result

**Returns** evaluation result on the specified class, including p, r, f1 and their standard deviation and coverage

**get_data**(*ids*, *test=False*, *shuffle=True*)

Fetch a data pair or pairs of data by image ID for training or test.

**Parameters**

- **ids** – list of image ID, usually in `train.json` or `test.json`
- **test** – bool, whether the fetched data is used for test; if true, this function will not return ground truth

- **shuffle** – bool, whether to shuffle the order of keypoints

**Returns**

**data_list**: list of data, like `[{'img': np.array, 'kpts': coordinates of kpts}, ...]`

**perm_mat_dict**: ground truth, like `{(0,1):scipy.sparse, (0,2):scipy.sparse, ...}`, `(0,1)` refers to data pair `(ids[0],ids[1])`

**ids**: list of image ID

`get_id_combination`(*cls=None*, *num=2*)

Get the combination of images and length of combinations in specified class.

**Parameters**

- **cls** – int or str, class of expected data. None for all classes
- **num** – int, number of images in each image ID list; for example, 2 for 2GM

**Returns**

**id_combination_list**: list of combinations of image ids

**length**: length of combinations

`rand_get_data`(*cls=None*, *num=2*, *test=False*, *shuffle=True*)

Randomly fetch data for training or test. Implemented by calling `get_data` function.

**Parameters**

- **cls** – int or str, class of expected data. None for random class
- **num** – int, number of images; for example, 2 for 2GM
- **test** – bool, whether the fetched data is used for test; if true, this function will not return ground truth
- **shuffle** – bool, whether to shuffle the order of keypoints

**Returns**

**data_list**: list of data, like `[{'img': np.array, 'kpts': coordinates of kpts}, ...]`

**perm_mat_dict**: ground truth, like `{(0,1):scipy.sparse, (0,2):scipy.sparse, ...}`, `(0,1)` refers to data pair `(ids[0],ids[1])`

**ids**: list of image ID

`rm_gt_cache`(*last_epoch=False*)

Remove ground truth cache.

**Parameters** **last_epoch** – Boolean variable, whether this epoch is last epoch; if true, the directory of cache will also be removed.

### 8.4.7 pygmtools.dataset

The implementations of data loading and data processing.

#### Classes

| | |
|---|---|
| *CUB2011* | Download and preprocess **CUB2011** dataset. |
| *IMC_PT_SparseGM* | Download and preprocess **IMC_PT_SparseGM** dataset. |
| *PascalVOC* | Download and preprocess **PascalVOC Keypoint** dataset. |
| *SPair71k* | Download and preprocess **SPair71k** dataset. |
| *WillowObject* | Download and preprocess **Willow Object Class** dataset. |

#### CUB2011

**class** pygmtools.dataset.**CUB2011**(*sets*, *obj_resize*, *ds_dict=None*, ***args*)

Download and preprocess **CUB2011** dataset.

**Parameters**

- **sets** – str, problem set, `'train'` for training set and `'test'` for testing set
- **obj_resize** – tuple, resized image size
- **ds_dict** – settings of dataset, containing at most 1 param(key) for CUB2011:
  - **CLS_SPLIT**: str, `'ori'` (original split), `'sup'` (super class) or `'all'` (all birds as one class)

**download**(*url=None*)

Automatically download CUB2011 dataset.

**Parameters** **url** – str, web url of CUB2011

**process**()

Process the dataset and generate `data-(size, size).json` for preprocessed dataset, `train.json` for training set, and `test.json` for testing set.

#### IMC_PT_SparseGM

**class** pygmtools.dataset.**IMC_PT_SparseGM**(*sets*, *obj_resize*, *ds_dict=None*, ***args*)

Download and preprocess **IMC_PT_SparseGM** dataset.

**Parameters**

- **sets** – str, problem set, `'train'` for training set and `'test'` for testing set
- **obj_resize** – tuple, resized image size
- **ds_dict** – settings of dataset, containing at most 1 param(key) for IMC_PT_SparseGM:
  - **TOTAL_KPT_NUM**: int, maximum kpt_num in an image

**download**(*url=None*)

Automatically download IMC_PT_SparseGM dataset.

**Parameters** **url** – str, web url of IMC_PT_SparseGM

**process**()

Process the dataset and generate `data-(size, size).json` for preprocessed dataset, `train.json` for training set, and `test.json` for testing set.

### PascalVOC

**class** pygmtools.dataset.**PascalVOC**(*sets*, *obj_resize*, ***args*)

Download and preprocess **PascalVOC Keypoint** dataset.

**Parameters**

- **sets** – str, problem set, `'train'` for training set and `'test'` for testing set
- **obj_resize** – tuple, resized image size

**download**(*url=None*, *name=None*)

Automatically download PascalVOC dataset.

**Parameters**

- **url** – str, web url of PascalVOC and PascalVOC annotation
- **name** – str, `"PascalVOC"` to download PascalVOC and `"PascalVOC_anno"` to download PascalVOC annotation

**process**()

Process the dataset and generate `data-(size, size).json` for preprocessed dataset, `train.json` for training set, and `test.json` for testing set.

### SPair71k

**class** pygmtools.dataset.**SPair71k**(*sets*, *obj_resize*, *problem='2GM'*, *ds_dict=None*, ***args*)

Download and preprocess **SPair71k** dataset.

**Parameters**

- **sets** – str, problem set, `'train'` for training set and `'test'` for testing set
- **obj_resize** – tuple, resized image size
- **problem** – str, problem type, only `'2GM'` is supported in SPair71k
- **ds_dict** – settings of dataset, containing at most 4 params(keys) for SPair71k:
  - **TRAIN_DIFF_PARAMS**: list of images that should be dumped in train set
  - **EVAL_DIFF_PARAMS**: list of images that should be dumped in testing set
  - **COMB_CLS**: bool, whether to combine images in different classes
  - **SIZE**: str, `'large'` for SPair71k-large and `'small'` for SPair71k-small

**download**(*url=None*)

Automatically download SPair71k dataset.

**Parameters** **url** – str, web url of SPair71k

**process**()

Process the dataset and generate `data-(size, size).json` for preprocessed dataset, `train.json` for training set, and `test.json` for testing set.

### WillowObject

**class** pygmtools.dataset.**WillowObject**(*sets*, *obj_resize*, *ds_dict=None*, ***args*)

Download and preprocess **Willow Object Class** dataset.

**Parameters**

- **sets** – str, problem set, `'train'` for training set and `'test'` for testing set
- **obj_resize** – tuple, resized image size
- **ds_dict** – settings of dataset, containing at most 4 params(keys) for WillowObject:
  - **TRAIN_NUM**: int, number of images for train in each class
  - **SPLIT_OFFSET**: int, offset when split train and testing set
  - **TRAIN_SAME_AS_TEST**: bool, whether to use same images for training and test
  - **RAND_OUTLIER**: int, number of added outliers in one image

**download**(*url=None*)

Automatically download WillowObject dataset.

**Parameters** **url** – str, web url of WillowObject

**process**()

Process the dataset and generate `data-(size, size).json` for preprocessed dataset, `train.json` for training set, and `test.json` for testing set.

**Warning:** By default the API functions and modules run on `numpy` backend. You could set the default backend by setting `pygm.BACKEND`. If you enable other backends than `numpy`, the corresponding package should be installed. See *the installation guide* for details.

## 8.5 Contributing to pygmtools

First, thank you for contributing to `pygmtools`!

### 8.5.1 How to contribute

The preferred workflow for contributing to `pygmtools` is to fork the main repository on GitHub, clone, and develop on a branch. Steps:

1. Fork the project repository by clicking on the 'Fork' button near the top right of the page. This creates a copy of the code under your GitHub user account. For more details on how to fork a repository see this guide.
2. Clone your fork of the repo from your GitHub account to your local disk:

```
$ git clone git@github.com:YourUserName/pygmtools.git
$ cd pygmtools
```

3. Create a `feature` branch to hold your development changes:

```
$ git checkout -b my-feature
```

   Always use a `feature` branch. It is good practice to never work on the `master` branch!

4. Develop the feature on your feature branch. Add changed files using `git add` and then `git commit` files:

```
$ git add modified_files
$ git commit
```

   to record your changes in Git, then push the changes to your GitHub account with:

```
$ git push -u origin my-feature
```

5. Follow these instructions to create a pull request from your fork. This will email the committers and an automatic check will run.

(If any of the above seems like magic to you, please look up the Git documentation on the web, or ask a friend or another contributor for help.)

### 8.5.2 Pull Request Checklist

We recommended that your contribution complies with the following rules before you submit a pull request:

- Follow the PEP8 Guidelines.
- If your pull request addresses an issue, please use the pull request title to describe the issue and mention the issue number in the pull request description. This will make sure a link back to the original issue is created.
- All public methods should have informative docstrings with sample usage presented as doctests when appropriate.
- When adding additional functionality, provide at least one example script in the `examples/` folder. Have a look at other examples for reference. Examples should demonstrate why the new functionality is useful in practice and, if possible, compare it to other methods available in `pygmtools`.
- Documentation and high-coverage tests are necessary for enhancements to be accepted. Bug-fixes or new features should be provided with non-regression tests. These tests verify the correct behavior of the fix or feature. In this manner, further modifications on the code base are granted to be consistent with the desired behavior. For the Bug-fixes case, at the time of the PR, these tests should fail for the code base in master and pass for the PR code.
- At least one paragraph of narrative documentation with links to references in the literature and the example.

You can also check for common programming errors with the following tools:

- No pyflakes warnings, check with:

```
$ pip install pyflakes
$ pyflakes path/to/module.py
```

- No PEP8 warnings, check with:

```
$ pip install pep8
$ pep8 path/to/module.py
```

- AutoPEP8 can help you fix some of the easy redundant errors:

```
$ pip install autopep8
$ autopep8 path/to/pep8.py
```

### 8.5.3 Filing bugs

We use Github issues to track all bugs and feature requests; feel free to open an issue if you have found a bug or wish to see a feature implemented.

It is recommended to check that your issue complies with the following rules before submitting:

- Verify that your issue is not being currently addressed by other issues or pull requests.
- Please ensure all code snippets and error messages are formatted in appropriate code blocks. See Creating and highlighting code blocks.
- Please include your operating system type and version number, as well as your Python, pygmtools, numpy, and scipy versions. Please also provide the name of your running backend, and the GPU/CUDA versions if you are using GPU. This information can be found by running the following environment report (`pygmtools>=0.2.9`):

  ```
  $ python3 -c 'import pygmtools; pygmtools.env_report()'
  ```

  If you are using GPU, make sure to install `pynvml` before running the above script: `pip install pynvml`.
- Please be specific about what estimators and/or functions are involved and the shape of the data, as appropriate; please include a reproducible code snippet or link to a gist. If an exception is raised, please provide the traceback.

### 8.5.4 Documentation

We are glad to accept any sort of documentation: function docstrings, reStructuredText documents, tutorials, etc. reStructuredText documents live in the source code repository under the doc/ directory.

You can edit the documentation using any text editor and then generate the HTML output by typing `make html` from the `docs/` directory. The resulting HTML index is `docs/_build/index.html` and is viewable in a web browser.

For building the documentation, you will need sphinx, matplotlib, and pillow.

When you are writing documentation, it is important to keep a good compromise between mathematical and algorithmic details, and give intuition to the reader on what the algorithm does. It is best to always start with a small paragraph with a hand-waving explanation of what the method does to the data and a figure (coming from an example) illustrating it.

This Contribution guide is strongly inpired by the one of the scikit-learn team.

# PYTHON MODULE INDEX

## p

# INDEX

## R



## S



## T



## W