Learning Multiresolution Matrix Factorization and Wavelet Networks

Software release

Truong Son Hy

October 2021

1 Mathematical introduction

1.1 Multiresolution matrix factorization

The *Multiresolution Matrix Factorization* (MMF) [Kondor et al., 2014] of a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is a factorization of the form

$$\boldsymbol{A} = \boldsymbol{U}_1^T \boldsymbol{U}_2^T \dots \boldsymbol{U}_L^T \boldsymbol{H} \boldsymbol{U}_L \dots \boldsymbol{U}_2 \boldsymbol{U}_1,$$

where the $\boldsymbol{H}$ and $\boldsymbol{U}_1, \dots, \boldsymbol{U}_L$ matrices conform to the following constraints: (i) Each $\boldsymbol{U}_\ell$ is an orthogonal matrix that is a k-point rotation for some small k, meaning that it only rotates k coordinates at a time; (ii) There is a nested sequence of sets $\mathbb{S}_L \subseteq \dots \subseteq \mathbb{S}_1 \subseteq \mathbb{S}_0 = [n]$ such that the coordinates rotated by $\boldsymbol{U}_\ell$ are a subset of $\mathbb{S}_\ell$; and (iii) $\boldsymbol{H}$ is an $\mathbb{S}_L$-core-diagonal matrix meaning that is diagonal with a an additional small $\mathbb{S}_L \times \mathbb{S}_L$ dimensional "core". Finding the best MMF factorization to a symmetric matrix $\boldsymbol{A}$ involves solving

$$\min_{\substack{\mathbb{S}_L \subseteq \dots \subseteq \mathbb{S}_1 \subseteq \mathbb{S}_0 = [n] \\ \boldsymbol{H} \in \mathbb{H}_n^{\mathbb{S}_L}; \boldsymbol{U}_1, \dots, \boldsymbol{U}_L \in \mathbb{O}}} ||\boldsymbol{A} - \boldsymbol{U}_1^T \dots \boldsymbol{U}_L^T \boldsymbol{H} \boldsymbol{U}_L \dots \boldsymbol{U}_1||. \quad (1)$$

Assuming that we measure error in the Frobenius norm, (1) is equivalent to

$$\min_{\substack{\mathbb{S}_L \subseteq \dots \subseteq \mathbb{S}_1 \subseteq \mathbb{S}_0 = [n] \\ \boldsymbol{U}_1, \dots, \boldsymbol{U}_L \in \mathbb{O}}} ||\boldsymbol{U}_L \dots \boldsymbol{U}_1 \boldsymbol{A} \boldsymbol{U}_1^T \dots \boldsymbol{U}_L^T||^2_{\text{resi}}, \quad (2)$$

where $||\cdot||^2_{\text{resi}}$ is the squared residual norm $||\boldsymbol{H}||^2_{\text{resi}} = \sum_{i \neq j; (i,j) \notin \mathbb{S}_L \times \mathbb{S}_L} |\boldsymbol{H}_{i,j}|^2$. There are two fundamental difficulties in MMF optimization: finding the optimal nested sequence of $\mathbb{S}_\ell$ is a combinatorially hard (e.g., there are $\binom{d_\ell}{k}$ ways to choose k indices out of $\mathbb{S}_\ell$); and the solution for $\boldsymbol{U}_\ell$ must satisfy the orthogonality constraint such that $\boldsymbol{U}_\ell^T \boldsymbol{U}_\ell = \boldsymbol{I}$. The existing literature on solving this optimization problem [Kondor et al., 2014] [Teneva et al., 2016] [Ithapu et al., 2017] [Ding et al., 2017] has various heuristic elements and has a number of limitations:

- There is no guarantee that the greedy heuristics (e.g., clustering) used in selecting k rows/columns $\mathbb{I}_\ell = \{i_1, .., i_k\} \subset \mathbb{S}_\ell$ for each rotation return a globally optimal factorization.
- Instead of direct optimization for each rotation $\boldsymbol{U}_\ell \triangleq \boldsymbol{I}_{n-k} \oplus_{\mathbb{I}_\ell} \boldsymbol{O}_\ell$ where $\boldsymbol{O}_\ell \in \mathbb{SO}(k)$ globally and simultaneously with the objective (1), Jacobi MMFs (see Proposition 2 of [Kondor et al., 2014]) apply the greedy strategy of optimizing them locally and sequentially. Again, this does not necessarily lead to a *globally* optimal combination of rotations.
- Most MMF algorithms are limited to the simplest case of $k = 2$ where $\boldsymbol{U}_\ell$ is just a Given rotation, which can be parameterized by a single variable, the rotation angle θ_ℓ. This makes it possible to optimize the greed objective by simple gradient descent, but larger rotations would yield more expressive factorizations and better approximations.

We show that the resulting learning-based MMF algorithm outperforms existing greedy MMFs and other traditional baselines for matrix approximation in various scenarios.

1.2 Stiefel manifold optimization

The MMF optimization problem in (1) and (2) is equivalent to

$$\min_{\mathbb{S}_L \subseteq \cdots \subseteq \mathbb{S}_1 \subseteq \mathbb{S}_0 = [n]} \min_{\boldsymbol{U}_1, \ldots, \boldsymbol{U}_L \in \mathbb{O}} ||\boldsymbol{U}_L \ldots \boldsymbol{U}_1 \boldsymbol{A} \boldsymbol{U}_1^T \ldots \boldsymbol{U}_L^T||^2_{\text{resi}}, \tag{3}$$

In order to solve the inner optimization problem of (3), we consider the following generic optimization with orthogonality constraints:

$$\min_{\boldsymbol{X} \in \mathbb{R}^{n \times p}} \mathcal{F}(\boldsymbol{X}), \quad \text{s.t.} \quad \boldsymbol{X}^T \boldsymbol{X} = \boldsymbol{I}_p, \tag{4}$$

where $\boldsymbol{I}_p$ is the identity matrix and $\mathcal{F}(\boldsymbol{X}) : \mathbb{R}^{n \times p} \rightarrow \mathbb{R}$ is a differentiable function. The feasible set $\mathcal{V}_p(\mathbb{R}^n) = \{\boldsymbol{X} \in \mathbb{R}^{n \times p} : \boldsymbol{X}^T \boldsymbol{X} = \boldsymbol{I}_p\}$ is referred to as the Stiefel manifold of p orthonormal vectors in $\mathbb{R}^n$ that has dimension equal to $np - \frac{1}{2}p(p+1)$. We will view $\mathcal{V}_p(\mathbb{R}^n)$ as an embedded submanifold of $\mathbb{R}^{n \times p}$.

When there is more than one orthogonal constraint, (4) is written as

$$\min_{\boldsymbol{X}_1 \in \mathcal{V}_{p_1}(\mathbb{R}^{n_1}), \ldots, \boldsymbol{X}_q \in \mathcal{V}_{p_q}(\mathbb{R}^{n_q})} \mathcal{F}(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_q) \tag{5}$$

where there are q variables with corresponding q orthogonal constraints. For example, in the MMF optimization problem (1), suppose we are already given $\mathbb{S}_L \subseteq \cdots \subseteq \mathbb{S}_1 \subseteq \mathbb{S}_0 = [n]$ meaning that the indices of active rows/columns at each resolution were already determined, for simplicity. In this case, we have $q = L$ number of variables such that each variable $\boldsymbol{X}_\ell = \boldsymbol{O}_\ell \in \mathbb{R}^{k \times k}$, where $\boldsymbol{U}_\ell = \boldsymbol{I}_{n-k} \oplus_{\mathbb{I}_\ell} \boldsymbol{O}_\ell \in \mathbb{R}^{n \times n}$ in which $\mathbb{I}_\ell$ is a subset of k indices from $\mathbb{S}_\ell$, must satisfy the orthogonality constraint. The corresponding objective function is

$$\mathcal{F}(\boldsymbol{O}_1, \ldots, \boldsymbol{O}_L) = ||\boldsymbol{U}_L \ldots \boldsymbol{U}_1 \boldsymbol{A} \boldsymbol{U}_1^T \ldots \boldsymbol{U}_L^T||^2_{\text{resi}}. \tag{6}$$

1.3 Wavelet networks

In the case $\boldsymbol{A}$ is the normalized graph Laplacian of a graph $\mathcal{G} = (V, E)$, the wavelet transform (up to level L) expresses a graph signal (function over the vertex domain) $f : V \rightarrow \mathbb{R}$, without loss of generality $f \in \mathbb{V}_0$, as:

$$f(v) = \sum_{\ell=1}^{L} \sum_m \alpha_m^\ell \psi_m^\ell(v) + \sum_m \beta_m \phi_m^L(v), \qquad \text{for each} \quad v \in V,$$

where $\alpha_m^\ell = \langle f, \psi_m^\ell \rangle$ and $\beta_m = \langle f, \phi_m^L \rangle$ are the wavelet coefficients. At each level, a set of coordinates $\mathbb{T}_\ell \subset \mathbb{S}_{\ell-1}$ are selected to be the wavelet indices, and then to be eliminated from the active set by setting $\mathbb{S}_\ell = \mathbb{S}_{\ell-1} \setminus \mathbb{T}_\ell$. Practically, we make the assumption that we only select 1 wavelet index for each level that results in a single mother wavelet $\psi^\ell = [\boldsymbol{A}_\ell]_{i^*,:}$ where i^* is the selected index. We get exactly L mother wavelets $\overline{\psi} = \{\psi^1, \psi^2, \ldots, \psi^L\}$. On the another hand, the active rows of $\boldsymbol{H} = \boldsymbol{A}_L$ make exactly $N - L$ father wavelets $\overline{\phi} = \{\phi_m^L = \boldsymbol{H}_{m,:}\}_{m \in \mathbb{S}_L}$. In total, a graph of N vertices has exactly N wavelets (both mothers and fathers). Analogous to the convolution based on graph Fourier transform (GFT) [Bruna et al., 2014], each convolution layer $k = 1, .., K$ of our wavelet network transforms an input vector $\boldsymbol{f}^{(k-1)}$ of size $|V| \times F_{k-1}$ into an output $\boldsymbol{f}^{(k)}$ of size $|V| \times F_k$ as

$$\boldsymbol{f}_{:,j}^{(k)} = \sigma\left(\boldsymbol{W} \sum_{i=1}^{F_{k-1}} \boldsymbol{g}_{i,j}^{(k)} \boldsymbol{W}^T \boldsymbol{f}_{:,i}^{(k-1)}\right) \quad \text{for} \quad j = 1, \ldots, F_k, \tag{7}$$

where $\boldsymbol{W}$ is our wavelet basis matrix as we concatenate $\overline{\phi}$ and $\overline{\psi}$ column-by-column, $\boldsymbol{g}_{i,j}^{(k)}$ is a parameter/filter in the form of a diagonal matrix learned in spectral domain, and σ is an element-wise linearity (e.g., ReLU, sigmoid, etc.).

2 Implementation and usage

2.1 Requirement

- Python 3.7.10
- PyTorch 1.8.0

Recommend using Conda environment for easy installation.

2.2 Organization

- `data/`: Datasets.
- `doc/`: Documentation in LaTeX.
- `experiments/`: Experiments of wavelet networks learning graphs (e.g., graph classification).
- `source/`: Implementation of Multiresolution Matrix Factorization (MMF) including the original (baseline), learnable and sparse; and several examples.

2.3 Datasets

Located in `data/` directory:

- `citeseer`: Citeseer citation graph of $N = 3,312$ nodes and $E = 4,723$ edges in which each node represents a paper and each directed edge represents a citation [Sen et al., 2008].
- `cora`: Cora citation graph of $N = 2,708$ nodes and $E = 5,429$ edges [Sen et al., 2008].
- `DD`: A protein dataset of 1,113 molecular graphs with binary labels, where nodes are secondary structure elements (SSEs) and there is an edge between two nodes if they are neighbors in the amino-acid sequence or in 3D space [Borgwardt et al., 2005].
- `ENZYMES`: A balanced dataset of 600 protein tertiary structures obtained from [Borgwardt et al., 2005].
- `karate`: The Karate club network ($N = 34$, $E = 78$) [Zachary, 1976].
- `minnesota`: The road network of Minnesota state ($N = 2,642$, $E = 3,303$) [Rossi and Ahmed, 2015].
- `mnist`: MNIST dataset `http://yann.lecun.com/exdb/mnist/` [Lecun et al., 1998]. We randomly select 1,000 (100 samples per one digit) samples and compute its RBF kernel gram matrix.
- `MUTAG`: 188 mutagenic aromatic and heteroaromatic nitro compounds with 7 discrete labels [Debnath et al., 1991].
- `NCI1`: 4,110 compounds with binary labels, each screened for activity against small cell lung cancer and ovarian cancer lines [Wale et al., 2008].
- `NCI109`: Similar to `NCI1` with 4,127 compounds [Wale et al., 2008].
- `PTC`: 344 chemical compounds with 19 discrete labels that have been tested for positive or negative toxicity in lab rats [Toivonen et al., 2003].
- `WebKB`: 877 scientific publications classified into one of five classes and 1,608 citation links [Lu and Getoor, 2003].

2.4 Dense implementation

This section details the implementation of MMFs using dense matrices as the storage of sparse rotation matrices. The codes are located in `source/`.

2.4.1 Built-in data loader

Built-in data loaders implemented in `data_loader.py` for:

- Synthetic datasets such as Kronecker product matrix, cycle graph and Cayley tree;
- Real datasets including citation graphs such as Cora, Citeseer and WebKB; RBF kernel gram matrix on MNIST digit images; Minnesota road network; and Karate club network.

Examples of data loading are in `test_data_loader.py`.

```
>>> import torch
>>> from data_loader import *
>>> karate_laplacian = karate_def('../data/')
Number of vertices: 34
Number of edges: 78
Done reading the Karate club network
Done computing the graph Laplacian
>>> karate_laplacian
tensor([[ 1.0000, -0.0833, -0.0791,  ..., -0.1021,  0.0000,  0.0000],
        [-0.0833,  1.0000, -0.1054,  ...,  0.0000,  0.0000,  0.0000],
        [-0.0791, -0.1054,  1.0000,  ...,  0.0000, -0.0913,  0.0000],
        ...,
        [-0.1021,  0.0000,  0.0000,  ...,  1.0000, -0.1179, -0.0990],
        [ 0.0000,  0.0000, -0.0913,  ..., -0.1179,  1.0000, -0.0700],
        [ 0.0000,  0.0000,  0.0000,  ..., -0.0990, -0.0700,  1.0000]])
>>> karate_laplacian.size()
torch.Size([34, 34])
```

Listing 1: Load the Karate club network data.

2.4.2 Baseline (original) MMF

The original (baseline) MMF [Kondor et al., 2014] is implemented in `baseline_mmf_model.py`. First, we initialize a PyTorch module `Baseline_MMF` given 3 parameters `N`, `L` and `dim` that are the input matrix size, number of rotations and the number of columns/rows active at the end. For each rotation, the original MMF drops exactly one column, so we must have `N = L + dim`. The original MMF limits to the case of `K = 2`. Remark that the original MMF does not have any learnable parameters, thus there is no need for training. Furthermore, the algorithm exhaustively search for an optimal pair of indices to rotate each rotation. The forward pass of this module returns the following outputs:

1. `A_rec`: The approximate (reconstructed) matrix of the input matrix A defined as
$$A \approx U_1^T U_2^T \dots U_L^T D U_L \dots U_2 U_1,$$
where $\{U_\ell\}_{\ell=1}^L$ are sparse orthogonal matrices, and D is close to diagonal.
2. `U`: The product of rotation matrices defined as $U_L \dots U_2 U_1$, so we have $A \approx U^T DU$.
3. `D`: The block diagonal matrix (close to diagonal). The block size has exactly size `dim` × `dim`.
4. `mother_coefficients`: There are exactly `L` = `N` − `dim` mother wavelets in this case. This output is a matrix containing the mother wavelet coefficients stored in its diagonal.
5. `father_coefficients`: There are exactly `dim` father wavelets. This output is the father wavelet coefficients.
6. `mother_wavelets`: This is mother wavelet bases, a matrix of size (`N` − `dim`) × `N`.
7. `father_wavelets`: This is father wavelet bases, a matrix of size `dim` × `N`. Both mother and father wavelet bases construct a wavelet basis, a matrix of size `N` × `N`.

The following are examples of using the original MMF (directly from Python interactive command line) and Nyström method (implemented in nystrom_model.py). The original MMF outperforms the Nyström method with a big margin. Script and training program (for large cases and other datasets) can be found at baseline_mmf_run.sh and baseline_mmf_run.py.

```
>>> from baseline_mmf_model import Baseline_MMF
>>> N = karate_laplacian.size(0)
>>> original_mmf = Baseline_MMF(N = N, L = N - 8, dim = 8)
>>> A_rec, U, D, mother_coefficients, father_coefficients, mother_wavelets, father_wavelets
    = original_mmf(karate_laplacian)
>>> torch.norm(karate_laplacian - A_rec, p = 'fro')
tensor(1.9621)
>>> torch.norm(karate_laplacian - torch.matmul(torch.matmul(U.transpose(0, 1), D), U), p = '
    fro')
tensor(1.9621)
>>> mother_coefficients.size()
torch.Size([26, 26])
>>> mother_wavelets.size()
torch.Size([26, 34])
>>> father_coefficients.size()
torch.Size([8, 8])
>>> father_wavelets.size()
torch.Size([8, 34])
```

Listing 2: Original MMF on the Karate club network with 26 rotations and 8 columns remained at the end. The Frobenius norm error is 1.9621.

```
>>> from nystrom_model import nystrom_model
>>> A_rec, C, W_inverse = nystrom_model(karate_laplacian, dim = 8)
Error =  tensor(5.2881)
```

Listing 3: Nyström method with 8 columns selected randomly. The Frobenius norm error is 5.2881 far worse comparing to the baseline MMF.

2.4.3 Learnable MMF

The learnable MMF applies Stiefel manifold optimization to find the optimal rotation matrices with arbitrary `K`. There are 3 algorithms (heuristics) implemented in heuristics.py to find `K` indices for each rotation:

1. `heuristics_random`: The input arguments are the input matrix in sparse format, number of rotations `L`, the size of rotation matrices `K`, number of columns dropped for each rotation `drop`, and the number of columns/rows active at the end `dim`. We need to make sure that `N` $=$ `L` $\times$ `drop` $+$ `dim`. All the indices are selected by uniformly random. The function returns 2 outputs `wavelet_indices` and `rest_indices` that are 2 lists of `L` elements. For each of `L` rotations, we select `drop` number of indices for mother wavelets (stored in `wavelet_indices`) and `K` $-$ `drop` number of indices (stored in `rest_indices`) to have exactly `K` indices to rotate. We drop the indices corresponding to mother wavelets after each rotation.

2. `heuristics_k_neighbors_single_wavelet`: The API is exactly the same. This is the heuristics of selecting indices in a neighborhood of the mother wavelet index (restricting to the case of `drop` $=$ `1`, only a single wavelet for each rotation).

3. `heuristics_k_neighbors_multiple_wavelets`: Heuristics for multiple wavelet indices for each rotation (`drop` $>$ `1`).

The learnable MMF model is implemented in learnable_mmf_model.py:

- `Learnable_MMF`: The PyTorch module (network).

- `learnable_mmf_train`: The training procedure to train the learnable MMF network, given the input hyperparamters including the indices selected by heuristics, number of epochs, and learning rate.

The following are 2 interactive examples of training the learnable MMF with random indices and indices selected by neighborhood heuristics, respectively. **Indeed, users can input customized indices instead of the built-in ones**. The Frobenius norm errors are better comparing to the original (baseline) MMF and the Nyström method. Short example code for usage is included in `example_1.py`. Script and training program (for large cases and other datasets) can be found at `learnable_mmf_run.sh` and `learnable_mmf_run.py`.

```
>>> from heuristics import *
>>> wavelet_indices, rest_indices = heuristics_random(karate_laplacian.to_sparse(), L = 26,
    K = 8, drop = 1, dim = 8)
>>> wavelet_indices
[[17], [8], [2], [11], [28], [19], [32], [15], [26], [9], [33], [4], [22], [14], [13], [31],
     [16], [10], [6], [29], [18], [5], [21], [30], [7], [0]]
>>> rest_indices
[[19, 2, 13, 5, 27, 20, 4], [6, 24, 4, 20, 27, 30, 19], [30, 7, 9, 3, 12, 23, 1], [17, 28,
    33, 25, 9, 19, 20], [7, 4, 23, 0, 19, 6, 24], [4, 30, 13, 2, 27, 25, 21], [20, 11, 22,
    8, 7, 27, 12], [9, 4, 6, 13, 23, 12, 29], [17, 31, 23, 32, 21, 20, 2], [3, 28, 18, 11,
    13, 20, 14], [10, 23, 14, 26, 6, 12, 30], [22, 8, 24, 21, 33, 16, 2], [13, 15, 30, 14,
    7, 4, 32], [26, 21, 8, 1, 18, 12, 33], [21, 2, 20, 19, 32, 25, 4], [10, 17, 7, 18, 33,
    26, 20], [2, 32, 29, 6, 5, 20, 22], [16, 2, 18, 11, 32, 30, 20], [28, 12, 17, 32, 3, 26,
     15], [5, 30, 2, 8, 25, 31, 7], [23, 25, 1, 4, 31, 3, 29], [18, 6, 25, 27, 32, 21, 10],
    [22, 1, 31, 25, 14, 9, 26], [25, 21, 10, 31, 6, 8, 24], [25, 10, 24, 20, 15, 4, 28],
    [28, 8, 7, 29, 27, 14, 16]]
>>> from learnable_mmf_model import *
>>> A_rec, U, D, mother_coefficients, father_coefficients, mother_wavelets, father_wavelets
    = learnable_mmf_train(karate_laplacian, L = 26, K = 8, drop = 1, dim = 8,
    wavelet_indices = wavelet_indices, rest_indices = rest_indices, epochs = 10000,
    learning_rate = 1e-4, early_stop = True)
Initialization loss: tensor(1.2220, grad_fn=<CopyBackwards>)
---- Epoch 0 ----
Loss = 1.2220197916030884
Time = 0.09203886985778809
---- Epoch 1000 ----
Loss = 1.200061321258545
Time = 0.0033674240112304688
---- Epoch 2000 ----
Loss = 1.1807942390441895
Time = 0.0033948421478271484
---- Epoch 3000 ----
Loss = 1.1633180379867554
Time = 0.003404378890991211
---- Epoch 4000 ----
Loss = 1.1471084356307983
Time = 0.003468036651611328
---- Epoch 5000 ----
Loss = 1.1318330764770508
Time = 0.0033025741577148438
---- Epoch 6000 ----
Loss = 1.11728835105896
Time = 0.003514528274536133
---- Epoch 7000 ----
Loss = 1.1032865047454834
Time = 0.0033206939697265625
---- Epoch 8000 ----
Loss = 1.089708685874939
Time = 0.0033156871795654297
---- Epoch 9000 ----
Loss = 1.0764575004577637
Time = 0.003555774688720703
---- Final loss ----
Loss = 1.0635088682174683
```

Listing 4: Learnable MMF on the Karate club network with 26 rotations, 8 columns remained at the end,

and drop 1 column for every rotation. The indices are randomly selected. The final Frobenius norm error is 1.0635 that is better than both the original MMF and Nyström method.

```
>>> wavelet_indices, rest_indices = heuristics_k_neighbors_single_wavelet(karate_laplacian.
    to_sparse(), L = 26, K = 8, drop = 1, dim = 8)
>>> A_rec, U, D, mother_coefficients, father_coefficients, mother_wavelets, father_wavelets
    = learnable_mmf_train(karate_laplacian, L = 26, K = 8, drop = 1, dim = 8,
    wavelet_indices = wavelet_indices, rest_indices = rest_indices, epochs = 10000,
    learning_rate = 1e-3, early_stop = True)
Initialization loss: tensor(1.1630, grad_fn=<CopyBackwards>)
---- Epoch 0 ----
Loss = 1.1629701852798462
Time = 0.10514163970947266
---- Epoch 1000 ----
Loss = 1.037307620048523
Time = 0.003313779830932617
---- Epoch 2000 ----
Loss = 0.9769641757011414
Time = 0.003489971160888672
---- Epoch 3000 ----
Loss = 0.9305171370506287
Time = 0.003305196762084961
---- Epoch 4000 ----
Loss = 0.8888177275657654
Time = 0.0034813880920410156
---- Epoch 5000 ----
Loss = 0.8487063646316528
Time = 0.003463268280029297
---- Epoch 6000 ----
Loss = 0.8089708685874939
Time = 0.0033228397369384766
---- Epoch 7000 ----
Loss = 0.769409716129303
Time = 0.003598928451538086
---- Epoch 8000 ----
Loss = 0.7304494380950928
Time = 0.003465414047241211
---- Epoch 9000 ----
Loss = 0.6928210258483887
Time = 0.003907918930053711
---- Final loss ----
Loss = 0.65743088722229
```

Listing 5: Learnable MMF on the Karate club network with 26 rotations, 8 columns remained at the end, and drop 1 column for every rotation. The indices are selected by heuristics. The final Frobenius norm error is 0.6574 that is better than randomly selecting indices.

2.4.4 Learnable MMF with smooth wavelets

Here is an interactive example of learnable MMF with smoothing loss function (regularization) $\lambda||f^T L f||^\alpha$ where f is each individual wavelet, L is the normalized graph Laplacian, λ and α are hyperparameters. The PyTorch network module and its training procedure are in `learnable_mmf_smooth_wavelets_model.py`. The API is exactly the same with other versions of MMFs. Script and training program (for large cases and other datasets) can be found at `learnable_mmf_smooth_wavelets_run.*` where `* = sh, py`.

Figure 1 and figure 2 show the visualization of some *smoothed* wavelets on Cayley tree (check `example_2.py`) and cycle graph (check `example_3.py`), respectively. Drawing functions (outputing to `.pdf` files) are implemented in `drawing_utils.py`.

```
>>> import torch
>>> from data_loader import *
```

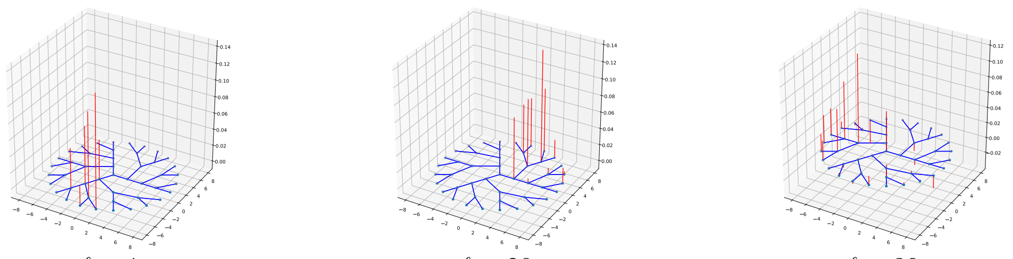

Figure 1: Visualization of some of the wavelets on the Cayley tree of 46 vertices. The low index wavelets (low ℓ) are highly localized, whereas the high index ones are smoother and spread out over large parts of the graph.

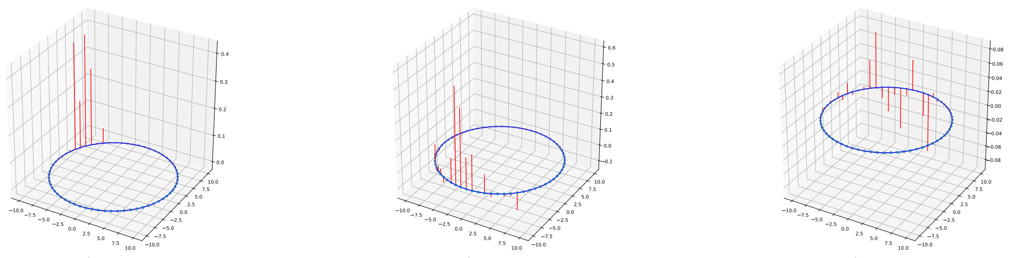

Figure 2: Visualization of some of the wavelets on the cycle graph of 64 vertices.

```
>>> from learnable_mmf_smooth_wavelets_model import *
>>> from drawing_utils import *
>>> A, cayley_node_x, cayley_node_y, edges = cayley_def(cayley_order = 2, cayley_depth = 4)
Done creating the Cayley graph
Cayley order = 2
Cayley depth = 4
Number of vertices = 46
>>> A_rec, U, D, mother_coefficients, father_coefficients, mother_wavelets, father_wavelets
    = learnable_mmf_smooth_wavelets_train(A, L = 42, K = 4, drop = 1, dim = 4, epochs =
    20000, learning_rate = 1e-3, Lambda = 1, alpha = 0.5, early_stop = True)
---- Epoch 0 ----
Reconstruction loss = 1.948068618774414
Smoothing loss = 9.003425598144531
Loss = 10.951494216918945
Time = 0.12453103065490723
... <Many lines>
---- Epoch 19000 ----
Reconstruction loss = 1.3875497579574585
Smoothing loss = 6.445069313049316
Loss = 7.8326191902160645
Time = 0.007997274398803711
---- Final loss ----
Reconstruction loss = 1.3746501207351685
Smooth loss = 6.391794204711914
Loss = 7.766444206237793
>>> for l in range(42):
...     wavelet = mother_wavelets[l].unsqueeze(dim = 0).detach().cpu().numpy()
...     draw_cayley(A.detach().cpu().numpy(), wavelet, cayley_node_x, cayley_node_y, edges,
    'wavelet_' + str(l))
```

Listing 6: Learning MMF with smoothing loss on Cayley tree and plot mother wavelets.

2.5 Sparse implementation

One problem with the dense implementation of MMFs is: storing sparse rotation matrices $\{U_\ell\}_{\ell=1}^L$ in dense format (i.e. $N \times N$) is wasteful and computationally expensive (e.g., for matrix multiplication). Indeed, each rotation matrix has exactly a block of size $K \times K$ and the diagonal that are non-zero. We implement the sparse version by storing rotation matrices in PyTorch's COO sparse format and replace dense matrix multiplication operations by the sparse ones. The model (`Sparse_MMF`) and training procedure (`sparse_mmf_train`) are implemented in `sparse_mmf_model.py`. Here is an example of usage for the Karate club network in Python interactive command line (check `example_4.py` also). Script and training program (for large cases and other datasets) can be found at `sparse_mmf_run.sh` and `sparse_mmf_run.py`.

```
>>> import torch
>>> from data_loader import *
>>> from heuristics import *
>>> from sparse_mmf_model import *
>>> karate_laplacian = karate_def('../data/')
Number of vertices: 34
Number of edges: 78
Done reading the Karate club network
Done computing the graph Laplacian
>>> wavelet_indices, rest_indices = heuristics_k_neighbors_single_wavelet(karate_laplacian.
    to_sparse(), L = 26, K = 8, drop = 1, dim = 8)
>>> A_rec, U, D, mother_coefficients, father_coefficients, mother_wavelets, father_wavelets
    = sparse_mmf_train(karate_laplacian, L = 26, K = 8, drop = 1, dim = 8, wavelet_indices =
     wavelet_indices, rest_indices = rest_indices, epochs = 10000, learning_rate = 1e-3,
    early_stop = True)
Initialized rotation 0
Initialization loss: tensor(1.1506, grad_fn=<CopyBackwards>)
---- Epoch 0 ----
Loss = 1.1506097316741943
Time = 0.09698176383972168
---- Epoch 1000 ----
Loss = 0.9675350189208984
Time = 0.008771419525146484
---- Epoch 2000 ----
Loss = 0.8891613483428955
Time = 0.008894681930541992
---- Epoch 3000 ----
Loss = 0.8363781571388245
Time = 0.008858680725097656
---- Epoch 4000 ----
Loss = 0.7941299080848694
Time = 0.008779764175415039
---- Epoch 5000 ----
Loss = 0.7579137086868286
Time = 0.008807659149169922
---- Epoch 6000 ----
Loss = 0.7262439131736755
Time = 0.008814096450805664
---- Epoch 7000 ----
Loss = 0.6985877156257629
Time = 0.008789777755737305
---- Epoch 8000 ----
Loss = 0.6744832396507263
Time = 0.008812665939331055
---- Epoch 9000 ----
Loss = 0.6534818410873413
Time = 0.008777379989624023
---- Final loss ----
Loss = 0.6350235342979431
```

Listing 7: Sparse MMF on Karate club network (N = 34). The API is exactly the same as in the dense implementation.

2.6 Training the wavelet networks

We also tested our wavelet networks on standard graph classification benchmarks including four bioinformatics datasets: (1) MUTAG, which is a dataset of 188 mutagenic aromatic and heteroaromatic nitro compounds with 7 discrete labels [Debnath et al., 1991]; (2) PTC, which consists of 344 chemical compounds with 19 discrete labels that have been tested for positive or negative toxicity in lab rats [Toivonen et al., 2003]; (3) PROTEINS, which contains 1,113 molecular graphs with binary labels, where nodes are secondary structure elements (SSEs) and there is an edge between two nodes if they are neighbors in the amino-acid sequence or in 3D space [Borgwardt et al., 2005]; (4) NCI1, which has 4,110 compounds with binary labels, each screened for activity against small cell lung cancer and ovarian cancer lines [Wale et al., 2008]. Each molecule is represented by an adjacency matrix, and we represent each atomic type as a one-hot vector and use them as the node features.

We factorize all normalized graph Laplacian matrices in these datasets by MMF to obtain the wavelet bases. Again, MMF wavelets are **sparse** and suitable for fast transform via sparse matrix multiplication, with the following average percentages of non-zero elements for each dataset: 19.23% (MUTAG), 18.18% (PTC), 2.26% (PROTEINS) and 11.43% (NCI1).

Our WNNs contain 6 layers of spectral convolution, 32 hidden units for each node, and are trained with 256 epochs by Adam optimization with an initial learning rate of 10^{-3}. We follow the evaluation protocol of 10-fold cross-validation from [Zhang et al., 2018]. We compare our results to several deep learning methods and popular graph kernel methods. Baseline results are taken from [Maron et al., 2019]. Our WNNs outperform 7/8, 7/8, 8/8, and 2/8 baseline methods on MUTAG, PTC, PROTEINS, and NCI1, respectively (see Table 1).

Located in `experiments/graph_classification/`. Each dataset (implemented in `Dataset.py`) is a list of molecules (implemented in `Molecule.py`) which each is a list of atoms (implemented in `Atom.py`). First, we need to run MMF to factorize all normalized graph Laplacians in the dataset to get the wavelet basis (for each individual molecule):

- **Baseline MMF**: Script to run and program are `baseline_mmf_basis.sh` and `baseline_mmf_basis.py`.
- **Learnable MMF**: Script to run and program are `learnable_mmf_basis.sh` and `learnable_mmf_basis.py`.

Second, based on wavelet basis that we found, we train our wavelet networks (with spectral convolution): script and program are `train_wavelet_network.sh` and `train_wavelet_network.py`, respectively. The PyTorch network module is `Wavelet_Network` in the Python program.

Table 1: Graph classification. Baseline results are taken from [Maron et al., 2019].

Method	**MUTAG**	**PTC**	**PROTEINS**	**NCI1**
DGCNN [Zhang et al., 2018]	85.83 ± 1.7	58.59 ± 2.5	75.54 ± 0.9	74.44 ± 0.5
PSCN [Niepert et al., 2016]	88.95 ± 4.4	62.29 ± 5.7	75 ± 2.5	76.34 ± 1.7
DCNN [Atwood and Towsley, 2016]	N/A	N/A	61.29 ± 1.6	56.61 ± 1.0
CCN [Kondor et al., 2018]	$\mathbf{91.64 \pm 7.2}$	$\mathbf{70.62 \pm 7.0}$	N/A	76.27 ± 4.1
GK [Shervashidze et al., 2009]	81.39 ± 1.7	55.65 ± 0.5	71.39 ± 0.3	62.49 ± 0.3
RW [Vishwanathan et al., 2010]	79.17 ± 2.1	55.91 ± 0.3	59.57 ± 0.1	N/A
PK [Neumann et al., 2015]	76 ± 2.7	59.5 ± 2.4	73.68 ± 0.7	82.54 ± 0.5
WL [Shervashidze et al., 2011]	84.11 ± 1.9	57.97 ± 2.5	74.68 ± 0.5	$\mathbf{84.46 \pm 0.5}$
IEGN [Maron et al., 2019]	84.61 ± 10	59.47 ± 7.3	75.19 ± 4.3	73.71 ± 2.6
MMF	86.31 ± 9.47	67.99 ± 8.55	$\mathbf{78.72 \pm 2.53}$	71.04 ± 1.53

References

[Atwood and Towsley, 2016] Atwood, J. and Towsley, D. (2016). Diffusion-convolutional neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 2001–2009, Red Hook, NY, USA. Curran Associates Inc.

[Borgwardt et al., 2005] Borgwardt, K., underlineCS, Schönauer, S., Vishwanathan, S., Smola, A., and Kriegel, H. (2005). Protein function prediction via graph kernels. *Bioinformatics*, 21 Suppl 1:i47–56.

[Bruna et al., 2014] Bruna, J., Zaremba, W., Szlam, A., and Lecun, Y. (2014). Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations (ICLR2014), CBLS, April 2014*.

[Debnath et al., 1991] Debnath, A. K., Lopez de Compadre, R. L., Debnath, G., Shusterman, A. J., and Hansch, C. (1991). Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786–797.

[Ding et al., 2017] Ding, Y., Kondor, R., and Eskreis-Winkler, J. (2017). Multiresolution kernel approximation for gaussian process regression. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[Ithapu et al., 2017] Ithapu, V. K., Kondor, R., Johnson, S. C., and Singh, V. (2017). The incremental multiresolution matrix factorization algorithm. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 692–701.

[Kondor et al., 2018] Kondor, R., Hy, T. S., Pan, H., Trivedi, S., and Anderson, B. M. (2018). Covariant compositional networks for learning graphs.

[Kondor et al., 2014] Kondor, R., Teneva, N., and Garg, V. (2014). Multiresolution matrix factorization. In Xing, E. P. and Jebara, T., editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1620–1628, Bejing, China. PMLR.

[Lecun et al., 1998] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[Lu and Getoor, 2003] Lu, Q. and Getoor, L. (2003). Link-based classification. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, page 496–503. AAAI Press.

[Maron et al., 2019] Maron, H., Ben-Hamu, H., Shamir, N., and Lipman, Y. (2019). Invariant and equivariant graph networks. In *International Conference on Learning Representations*.

[Neumann et al., 2015] Neumann, M., Garnett, R., Bauckhage, C., and Kersting, K. (2015). Propagation kernels: efficient graph kernels from propagated information. *Machine Learning*, 102.

[Niepert et al., 2016] Niepert, M., Ahmed, M., and Kutzkov, K. (2016). Learning convolutional neural networks for graphs. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2014–2023, New York, New York, USA. PMLR.

[Rossi and Ahmed, 2015] Rossi, R. A. and Ahmed, N. K. (2015). The network data repository with interactive graph analytics and visualization. In *AAAI*.

[Sen et al., 2008] Sen, P., Namata, G. M., Bilgic, M., Getoor, L., Gallagher, B., , and Eliassi-Rad, T. (2008). Collective classification in network data. *AI Magazine*, 29(3):93–106.

[Shervashidze et al., 2011] Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Mehlhorn, K., and Borgwardt, K. M. (2011). Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(77):2539–2561.

[Shervashidze et al., 2009] Shervashidze, N., Vishwanathan, S., Petri, T., Mehlhorn, K., and Borgwardt, K. (2009). Efficient graphlet kernels for large graph comparison. In van Dyk, D. and Welling, M., editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 488–495, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA. PMLR.

[Teneva et al., 2016] Teneva, N., Mudrakarta, P. K., and Kondor, R. (2016). Multiresolution matrix compression. In Gretton, A. and Robert, C. C., editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 1441–1449, Cadiz, Spain. PMLR.

[Toivonen et al., 2003] Toivonen, H., Srinivasan, A., King, R. D., Kramer, S., and Helma, C. (2003). Statistical evaluation of the Predictive Toxicology Challenge 2000–2001. *Bioinformatics*, 19(10):1183–1193.

[Vishwanathan et al., 2010] Vishwanathan, S. V. N., Schraudolph, N. N., Kondor, R., and Borgwardt, K. M. (2010). Graph kernels. *J. Mach. Learn. Res.*, 11:1201–1242.

[Wale et al., 2008] Wale, N., Watson, I., and Karypis, G. (2008). Comparison of descriptor spaces for chemical compound retrieval and classification. *Knowl. Inf. Syst.*, 14:347–375.

[Zachary, 1976] Zachary, W. (1976). An information flow model for conflict and fission in small groups1. *Journal of anthropological research*, 33.

[Zhang et al., 2018] Zhang, M., Cui, Z., Neumann, M., and Chen, Y. (2018). An end-to-end deep learning architecture for graph classification. In *AAAI*.