Graph Filters & Graph Neural Networks

Luana Ruiz, Damian Owerko and Alejandro Ribeiro

September 15, 2020

This lab is our first approximation at learning with graph filters and graph neural networks (GNNs). You will learn how to train a graph filter and a GNN. You will also see evidence that the following three facts holds:

(F1) Graph filters produce better learning results than arbitrary linear parametrizations and GNNs produce better results than arbitrary (fully connected) neural networks.

(F2) GNNs work better than graph filters.

(F3) A GNN that is trained on a graph with a certain number of nodes can be executed in a graph with a larger number of nodes and still produce good rating estimates.

Facts (F1)-(F3) support advocacy for the use of GNNs. They also spark three interesting questions:

(Q1) Why do graph filters and GNNs outperform linear transformations and fully connected neural networks?

(Q2) Why do GNNs outperform graph filters?

(Q3) Why do GNNs transfer to networks with different number of nodes?

We will spend a sizable chunk of this course endeavoring to respond Questions (Q1)-(Q3).

Throughout the lab we use source localization as an example problem. This problems uses fake data that we generate so as to work in a controlled environment. We will soon be repeating this lab in a recommendation system using real data and will rediscover Facts (F1)-(F3) and reintroduce Questions (Q1)-(Q3). Working with real data is messier and better relegated to a second experience.

1 Source Localization

The source localization problem consists of identifying the sources of a graph diffusion process from an observation of the process at a given time $t = T$. A practical example is trying to identify a set of malicious agents responsible for the spread of fake news on a social network.

Consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{S})$ with node set $\mathcal{V}$, $|\mathcal{V}| = N$, edge set $\mathcal{E} \in \mathcal{V} \times \mathcal{V}$ and graph shift operator (GSO) $\mathbf{S} \in \mathbb{R}^{N \times N}$. Let $\mathcal{S} = \{s_1, \ldots, s_M\}$ denote a set of M sources $s_i \in \mathcal{V}$. At time $t = 0$, the graph signal $\mathbf{z}_0 \in \mathbb{R}^N$ is given by

$$[\mathbf{z}_0]_i = \begin{cases} z \sim \mathcal{U}(a, b) & \text{if } i \in \mathcal{S} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where $\mathcal{U}(a, b)$ stands for the uniform distribution on the $[a, b]$ interval. For $t > 0$, $\mathbf{z}_t$ is the output of the diffusion of $\mathbf{z}_{t-1}$ on the graph, i.e.,

$$\mathbf{z}_t = \mathbf{S}\mathbf{z}_{t-1} + \mathbf{w}_t \tag{2}$$

where $\mathbf{w}_t \in \mathbb{R}^N$ is a Gaussian noise. Given an observation $\mathbf{z}_T$ of this process at time $t = T$, our goal is to identify the sources $s_i \in \mathcal{S}$.

We will study the source localization problem on stochastic block model (SBM) graphs. SBM graphs are unweighted and undirected graphs made up of C communities. Nodes in the same community c_i are connected with probability $p_{c_i c_i}$ (called intra-community probability) and nodes in different communities c_i, c_j are connected with probability $p_{c_i c_j}$ (called inter-community probability).

1.1 Graph Generation. Generate the adjacency matrix of a SBM graph with $N = 50$ nodes, $C = 5$ communities of size N/C, intra-community

probability $p_{c_i c_i} = 0.6$ and inter-community probability $p_{c_i c_j} = 0.2$. Normalize it by the eigenvalue with largest absolute value.

1.2 Data Generation. Generate 2100 samples of $\mathbf{z}_0$ [cf. (1)] with $a = 0$, $b = 10$ and $|\mathcal{S}| = M = 10$. For each sample, the M sources $s_i \in \mathcal{S}$ should be drawn at random from $\mathcal{V}$. Use these samples to generate $\mathbf{z}_t$ for $t = 1, \ldots, 4$ according to (2). The Gaussian noise $\mathbf{w}_t$ has mean $\boldsymbol{\mu} = \mathbf{0}$ and covariance $\boldsymbol{\Sigma} = 10^{-3}\mathbf{I}$.

1.3 Training and Test Set. In the source localization problem, the output data $\mathbf{y}$ is given by $\mathbf{z}_0$ and the input data $\mathbf{x}$ is given by $\mathbf{z}_t$ for $t = T$. Fix $T = 4$ and store the input samples $\mathbf{x} = \mathbf{z}_T$ in a $2100 \times N$ matrix $\mathbf{X}$. Store the output samples $\mathbf{y} = \mathbf{x}_0$ in a $2100 \times N$ matrix $\mathbf{Y}$. Permute the rows of $\mathbf{X}$ and $\mathbf{Y}$ at random and split the samples between 2000 for training and 100 for testing.

2 Learning with Graph Filters

Our goal is to learn a map that will produce outputs $\mathbf{y}$ when presented with inputs $\mathbf{x}$. To do that we define the loss function

$$\ell\big(\boldsymbol{\Phi}(\mathbf{x}; \mathcal{H}), \mathbf{y}\big) \;=\; \sum_{i=1}^{N} \frac{([\boldsymbol{\Phi}(\mathbf{x}; \mathcal{H})]_i - [\mathbf{y}]_i)^2}{N}, \tag{3}$$

called mean squared error (MSE) or quadratic loss. This loss is minimized over the training samples to obtain the map $\boldsymbol{\Phi}(\mathbf{x}; \mathcal{H})$, where the set $\mathcal{H}$ groups the learnable parameters. At execution time, this map can be used to predict the sources of graph diffusion processes $\mathbf{x}_t$ where $\mathbf{x}_0$ is unseen. As we will see in this lab, successful predictions depend on the choice of parametrization. In particular, we are interested in parametrizations that depend on the graph through the GSO, i.e., $\boldsymbol{\Phi}(\mathbf{x}; \mathcal{H}) = \boldsymbol{\Phi}(\mathbf{x}; \mathcal{H}, \mathbf{S})$.

A parametrization that is convenient for processing graph signals is a graph convolutional filter. To define this operation introduce a filter order K along with filter coefficients h_k that we group in the vector $\mathbf{h} = [h_0, \ldots, h_{K-1}]$. A graph convolutional filter applied to the graph signal $\mathbf{x}$

is a polynomial in the GSO $\mathbf{S}$. In this case, $\mathbf{\Phi}(\mathbf{x}; \mathcal{H}, \mathbf{S})$ is given by

$$\mathbf{\Phi}(\mathbf{x}; \mathbf{h}, \mathbf{S}) = \mathbf{H}(\mathbf{S})\mathbf{x} = \sum_{k=0}^{K-1} \mathbf{S}^k \mathbf{x}\, h_k \tag{4}$$

where the output $\mathbf{\Phi}(\mathbf{x}; \mathbf{h}, \mathbf{S})$ is also a graph signal and the learnable parameters $\mathcal{H}$ are the filter coefficients h_k.

One advantage of graph filters is their locality. Indeed, we can define the diffusion sequence as the collection of graph signals $\mathbf{u}_k = \mathbf{S}^k\mathbf{x}$ to rewrite the filter in (4) as $\mathbf{u} = \sum_{k=0}^{K-1} \mathbf{u}_k h_k$. It is ready to see that the diffusion sequence is given by the recursion $\mathbf{u}_k = \mathbf{S}\mathbf{u}_{k-1}$ with $\mathbf{u}_0 = \mathbf{x}$. Further observing that $S_{ij} \neq 0$ only when the pair (i, j) is an edge of the graph, we see that the entries of the diffusion sequence satisfy

$$u_{k,i} \;=\; \sum_{j:(i,j)\in\mathcal{E}} S_{ij} u_{k-1,j}. \tag{5}$$

We can therefore interpret the graph filter in (4) as an operation that propagates information through adjacent nodes. This is a property that graph convolutional filters share with regular convolutional filters in time and offers motivation for their use in the processing of graph signals.

In the context of machine learning on graphs, a more important property of graph filters is their *equivariance to permutation*. Use $\mathbf{P}$ to denote a permutation matrix – entries P_{ij} are binary with exactly one nonzero entry in each row and column. The vector $\hat{\mathbf{x}} = \mathbf{P}\mathbf{x}$ is just a reordering of the entries of $\mathbf{x}$ which we can interpret as a graph signal supported on the graph $\hat{\mathbf{S}} = \mathbf{P}\mathbf{S}\mathbf{P}^T$ which is just a reordering of the graph $\mathbf{S}$. If we now consider the processing of $\hat{\mathbf{x}}$ on the graph $\hat{\mathbf{S}}$ with the graph filter $\mathbf{h}$ the following holds.

Proposition 1 *Graph filters are permutation equivariant,*

$$\mathbf{\Phi}(\hat{\mathbf{x}}; \mathbf{h}, \hat{\mathbf{S}}) = \mathbf{\Phi}(\mathbf{P}\mathbf{x}; \mathbf{h}, \mathbf{P}\mathbf{S}\mathbf{P}^T) = \mathbf{P}\mathbf{\Phi}(\mathbf{x}; \mathbf{h}, \mathbf{S}). \tag{6}$$

The immediate relevance of permutation equivariance is that it shows that processing a graph signal with a graph filter is independent of node labeling. This is something we know must hold in several applications but that is not true of, say, a generic linear parametrization of the signal components at each node. There is, however, further value in permutation

equivariance. To explain this, return to the ERM problem formulation and utilize the graph filter in (4) as a learning parametrization. This yields the learning problem

$$\mathbf{h}^* = \underset{\mathbf{h}}{\operatorname{argmin}} \frac{1}{Q} \sum_{q=1}^{Q} \ell\Big(\mathbf{\Phi}(\mathbf{x}_q; \mathbf{h}, \mathbf{S}),\, \mathbf{y}_q\Big). \tag{7}$$

An important observation is that we know that $\mathbf{\Phi}(\mathbf{x}; \mathbf{H}) = \mathbf{H}\mathbf{x}$ must yield a function $\mathbf{\Phi}(\mathbf{x}; \mathbf{H}^*)$ whose average loss is smaller than the average loss attained by the function $\mathbf{\Phi}(\mathbf{x}; \mathbf{h}^*, \mathbf{S})$ obtained from solving (7). This is because both are linear transformations and while $\mathbf{\Phi}(\mathbf{x}; \mathbf{H}) = \mathbf{H}\mathbf{x}$ is generic, the graph filter $\mathbf{\Phi}(\mathbf{x}; \mathbf{h}, \mathbf{S}) = \sum_{k=0}^{K-1} \mathbf{S}^k \mathbf{x} h_k$ belongs to a particular linear class. This is certainly true on the training set $\mathcal{T}$, but when operating on unobserved samples $\mathbf{x}$ the graph filter can and will do better because its permutation equivariance induces better generalization.

2.1 Graph filter function. Write a function that takes as inputs a graph shift operator $\mathbf{S}$, a graph signal $\mathbf{x}$ and a vector of coefficients $\mathbf{h}$ and returns the output of (4).

2.2 Graph filter module. Using the function from the previous item, write a `torch.nn.Module` class to implement the graph filter in (4) as a learning architecture in PyTorch.

Obs.: Every `torch.nn.Module` *has a constructor method called* `init` *and a method called* `forward` *which specifies the operations to be executed. The* `init` *method takes the architecture's hyperparameters as inputs and saves its parameters as attributes. In the case of the graph filter, your* `init` *function should take the hyperparameter K and the GSO $\mathbf{S}$ as inputs and declare the weights $\mathbf{h}$ as learnable parameters (*`nn.parameter.Parameter`*). The* `forward` *method should take $\mathbf{x}$ as an input and implement equation* (4) *using the learnable parameters, the GSO and the function you wrote in item 2.1. Check the* `torch.nn.Module` *example of a linear parametrization provided in the class website for more details.*

Like in Lab 1, we will consider batches of samples to reduce the computational complexity associated with training our models. We will also

consider multiple epochs, which are full passes over the training set. Multiple epochs are necessary because, typically, neural networks have a large number of parameters, which requires the optimization algorithm (e.g., SGD) to be run over a sufficient number of steps in order to converge to a local minimum. For a training set of size Q, the number of training steps n is determined by the batch size Q_t and the number of epochs E. Explicitly,

$$n = E \times \text{ceil}\left(\frac{Q}{Q_t}\right) \tag{8}$$

where ceil stands for the ceiling function. In this lab, all learning architectures will be trained over $E = 30$ epochs with batch size $Q_t = 200$. Instead of SGD, we will use another stochastic optimization algorithm called Adam[1] with step size $\epsilon = 0.05$. These hyperparameters will be the same for all architectures you are asked to train.

2.3 Training your first model. Instantiate a graph filter with $K = 8$ using the module you implemented in 2.2. Use the data you generated in Section 1 to train and test your model. Report the number of training steps n and plot the training loss vs. the number of training steps.

Obs.: You can set aside a portion of your training set (e.g., 10% of the training samples) for validation. Validation samples are not used for training but they are used to test the model at regular intervals during training. Validation is a good practice because it allows saving the model with best performance on the validation set at any point of the training process, which helps to avoid overfitting. For more details, check the sample code provided in the class website, where an arbitrary linear parametrization undergoes training and validation.

3 Learning with Graph Neural Networks

Graph neural networks (GNNs) extend graph filters by using pointwise nonlinearities which are nonlinear functions that are applied independently to each component of a vector. For a formal definition, begin by

[1]Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. arXiv:1412.6980.

introducing a single variable function $\sigma : \mathbb{R} \to \mathbb{R}$ which we extend to the vector function $\sigma : \mathbb{R}^n \to \mathbb{R}^n$ by independent application to each component. Thus, if we have $\mathbf{u} = [u_1, \ldots, u_n] \in \mathbb{R}^n$ the output vector $\sigma(\mathbf{u})$ is such that

$$\sigma\big(\, \mathbf{u} \,\big) \;:\; \big[\, \sigma\big(\, \mathbf{u} \,\big) \,\big]_i = \sigma\big(\, u_i \,\big). \tag{9}$$

I.e., the output vector is of the form $\sigma(\mathbf{u}) = [\sigma(u_1), \ldots, \sigma(u_n)]$. Observe that we are abusing notation and using σ to denote both the scalar function and the pointwise vector function.

In a single layer GNN, the graph signal $\mathbf{u} = \mathbf{H}(\mathbf{S})\mathbf{x}$ is passed through a pointwise nonlinear function satisfying (9) to produce the output

$$\mathbf{\Phi}(\mathbf{x}; \mathbf{h}, \mathbf{S}) = \sigma(\mathbf{u}) = \sigma(\mathbf{H}(\mathbf{S})\mathbf{x}) = \sigma\left(\sum_{k=0}^{K-1} \mathbf{S}^k \mathbf{x} h_k\right) . \tag{10}$$

We say that the transform in (10) is a graph perceptron. Different from the graph filter in (4), the graph perceptron is a nonlinear function of the input. It is, however, a very simple form of nonlinear processing because the nonlinearity does not mix signal components. Signal components are mixed by the graph filter but are then processed element-wise through σ. In particular, (10) retains the locality properties of graph convolutional filters as well as their permutation equivariance (cf. Proposition 1).

Note that the nonlinearity does not add any learnable parameters and, like in the graph filter, the learnable parameters $\mathcal{H}$ are the coefficients $\mathbf{h}$.

3.1 Graph perceptron. Using the graph filter module from 2.2 and the ReLU for σ, implement the graph perceptron in (10) and instantiate it with $K = 8$. Use the data you generated in Section 1 to train and test your model. Plot the training loss vs. the number of training steps.

Obs.: You can implement the graph perceptron either as a `torch.nn.Module` *class or by using* `torch.nn.Sequential`*, which allows stacking modules in a sequential way.*

Graph perceptrons can be stacked in layers to create multi-layer GNNs. This stacking is mathematically written as a function composition where the outputs of a layer become inputs to the next layer. For a formal definition let $\ell = 1, \ldots, L$ be a layer index and $\mathbf{h}_\ell = \{h_{\ell k}\}_{k=0}^{K-1}$ be collections of

K graph filter coefficients associated with each layer. Each of these sets of coefficients define a respective graph filter $\mathbf{\Phi}(\mathbf{x}; \mathbf{h}_\ell, \mathbf{S}) = \sum_{k=0}^{K-1} \mathbf{S}^k \mathbf{x} h_{\ell k}$. At layer ℓ we take as input the output $\mathbf{x}_{\ell-1}$ of layer $\ell - 1$ which we process with the filter $\mathbf{\Phi}(\mathbf{x}; \mathbf{h}_\ell, \mathbf{S})$ to produce the intermediate feature

$$\mathbf{u}_\ell \;=\; \mathbf{H}_\ell(\mathbf{S})\, \mathbf{x}_{\ell-1} \;=\; \sum_{k=0}^{K-1} \mathbf{S}^k\, \mathbf{x}_{\ell-1} h_{\ell k}, \tag{11}$$

where, by convention, we say that $\mathbf{x}_0 = \mathbf{x}$ so that the given graph signal $\mathbf{x}$ is the GNN input. As in the case of the graph perceptron, this feature is passed through a pointwise nonlinear function to produce the ℓth layer output

$$\mathbf{x}_\ell \;=\; \sigma(\mathbf{u}_\ell) \;=\; \sigma\left(\sum_{k=0}^{K-1} \mathbf{S}^k\, \mathbf{x}_{l-1} h_{\ell k} \right). \tag{12}$$

After recursive repetition of (11)-(12) for $\ell = 1, \ldots, L$ we reach the Lth layer whose output $\mathbf{x}_L$ is not further processed and is declared the GNN output $\mathbf{y} = \mathbf{x}_L$. To represent the output of the GNN we define the filter tensor $\mathbf{H} := \{\mathbf{h}_\ell\}_{\ell=1}^{L}$ grouping the L sets of filter coefficients at each layer, and define the operator $\mathbf{\Phi}(\,\cdot\,; \mathbf{H}, \mathbf{S})$ as the map

$$\mathbf{\Phi}(\mathbf{x}; \mathbf{H}, \mathbf{S}) = \mathbf{x}_L. \tag{13}$$

We repeat that in (13) the GNN output $\mathbf{\Phi}(\mathbf{x}; \mathbf{H}, \mathbf{S}) = \mathbf{x}_L$ follows from recursive application of (11)-(12) for $\ell = 1, \ldots, L$ with $\mathbf{x}_0 = \mathbf{x}$. Observe that this operator notation emphasizes that the output of a GNN depends on the filter tensor $\mathbf{H}$ and the graph shift operator $\mathbf{S}$. We also emphasize that, similar to the case of the graph filters in (4), the optimization is over the filter tensor $\mathbf{H}$ with the shift operator $\mathbf{S}$ given. Finally, note that since each perceptron is permutation equivariant, the whole GNN also inherits the permutation equivariance of graph filters.

3.2 Multi-layer GNN. Implement the multi-layer graph perceptron as defined in (12) and instantiate it with $L = 2$ layers, $K_1 = 8$ and $K_2 = 1$ (i.e., 8 and 1 filter taps in the first and second layers respectively). Use the data you generated in Section 1 to train and test your model. Plot the training loss vs. the number of training steps.

4 Multiple Feature Filters and GNNs

To further increase the representation power of multi-layer GNNs, we incorporate multiple features per layer that are the result of processing multiple input features with a bank of graph filters. Let F_{in} be the number of input features and F_{out} be the number of output features. Define the feature matrix $\mathbf{X} \in \mathbb{R}^{N \times F_{\text{in}}}$ as

$$\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^{F_{\text{in}}}]. \tag{14}$$

We have that $\mathbf{X} \in \mathbb{R}^{N \times F_{\text{in}}}$ and interpret each column of $\mathbf{X}$ as a graph signal. For a compact representation of a filterbank made up of $F_{\text{in}} \times F_{\text{out}}$ filters, consider coefficient matrices $\mathbf{H}_k \in \mathbb{R}^{F_{\text{in}} \times F_{\text{out}}}$. The filterbank is defined as

$$\mathbf{\Phi}(\mathbf{X}; \mathbf{H}, \mathbf{S}) \;=\; \sum_{k=0}^{K-1} \mathbf{S}^k \mathbf{X} \mathbf{H}_k \tag{15}$$

where the tensor $\mathbf{H}$ groups the filter coefficients $\mathbf{H}_k$, $\mathbf{H} = \{\mathbf{H}_k\}_k$. Each of the F_{out} columns of the matrix $\mathbf{\Phi}(\mathbf{x}; \mathbf{H}, \mathbf{S}) \in \mathbb{R}^{N \times F_{\text{out}}}$ is a separate graph signal. We say that (15) represents a multiple-input-multiple-output graph filter since it takes F_{in} graph signals as inputs and yields F_{out} graph signals at its output.

Note that filter banks can also be cascaded to obtain a layered linear architecture. Denoting the graph filter coefficients at layer ℓ by $\mathbf{H}_{\ell k} \in \mathbb{R}^{F_\ell \times F_{\ell-1}}$, the ℓth layer of a L-layer filterbank is given by

$$\mathbf{U}_\ell \;=\; \sum_{k=0}^{K-1} \mathbf{S}^k \mathbf{U}_{\ell-1} \mathbf{H}_{\ell k} \,. \tag{16}$$

The output of this filterbank is $\mathbf{\Phi}(\mathbf{X}; \mathbf{H}, \mathbf{S}) = \mathbf{U}_L$ with $\mathbf{U}_0 = \mathbf{X}$, and $\mathbf{H}$ is given by $\mathbf{H} = \{\mathbf{H}_{\ell k}\}_{\ell,k}$. Given that they are simply parallel and sequential compositions of graph filters, layered filter banks as the one in (16) inherit the locality and permutation equivariance properties from graph filters.

4.1 Multi-feature graph filter. Modify the graph filter function and the graph filter module you implemented in 2.1 and 2.2 to take the number of input features F_{in} and the number of output features F_{out} as hyperparameters and to process signals with multiple features. Then, use this

module to instantiate a graph filter with $L = 2$ layers, $K_1 = 8$, $K_2 = 1$ and $F_1 = 32$ features in the first layer. Use the data you generated in Section 1 to train and test your model. Plot the training loss vs. the number of training steps.

In their most general form, GNNs consist of compositions of filter banks as in (15) with pointwise nonlinearities σ. Explicitly, let F_ℓ be the number of features at layer ℓ and define the feature matrix $\mathbf{X}_\ell$ as

$$\mathbf{X}_\ell = [\mathbf{x}_\ell^1, \mathbf{x}_\ell^2, \ldots, \mathbf{x}_\ell^{F_\ell}]. \tag{17}$$

We have that $\mathbf{X}_\ell \in \mathbb{R}^{N \times F_\ell}$ where each column is a graph signal. The outputs of layer $\ell - 1$ are inputs to layer ℓ where the set of $F_{\ell-1}$ features in $\mathbf{X}_{\ell-1}$ are processed by a filterbank made up of $F_{\ell-1} \times F_\ell$ filters (cf. (15)) to obtain the intermediate feature matrix

$$\mathbf{U}_\ell \;=\; \sum_{k=0}^{K-1} \mathbf{S}^k \, \mathbf{X}_{\ell-1} \, \mathbf{H}_{\ell k} \; . \tag{18}$$

As in the case of the single feature GNN (cf. (12)) – and the graph perceptron in (10) – the intermediate feature $\mathbf{U}_\ell$ is passed through a pointwise nonlinearity to produce the ℓth layer output

$$\mathbf{X}_\ell \;=\; \sigma(\mathbf{U}_\ell) \;=\; \sigma\left(\sum_{k=0}^{K-1} \mathbf{S}^k \, \mathbf{X}_{\ell-1} \, \mathbf{H}_{\ell k} \right). \tag{19}$$

When $\ell = 0$ we convene that $\mathbf{X}_0 = \mathbf{X}$ is the input to the GNN which is made of F_0 graph signals. The output $\mathbf{X}_L$ of layer L is also the output of the GNN which is made up of F_L graph signals. To define a GNN operator we group filter coefficients $\mathbf{H}_{\ell k}$ in the tensor $\mathbf{H} = \{\mathbf{H}_{\ell k}\}_{\ell,k}$ and define the GNN operator

$$\mathbf{\Phi}(\mathbf{X}; \mathbf{H}, \mathbf{S}) = \mathbf{X}_L. \tag{20}$$

If the input is a single graph signal as in (10) and (13), we have $F_0 = 1$ and $\mathbf{X}_0 = \mathbf{x} \in \mathbb{R}^n$. If the output is also a single graph signal – as is also the case in (10) and (13) – we have $F_L = 1$ and $\mathbf{X}_L = \mathbf{x}_L \in \mathbb{R}^N$.

Each layer of the GNN is made up of filter banks which are permutation equivariant. Since pointwise nonlinearities do not mix signal components, each individual layer is permutation equivariant. It follows that the GNN, being a composition of permutation equivariant operators, is also permutation equivariant.

4.2 2-layer GNN. Use the multi-feature graph filter model from item 4.1 to implement the GNN as defined in (19). Then, instantiate a GNN with $L = 2$ layers, $K_1 = 8$, $K_2 = 1$ and $F_1 = 32$ features in the first layer. Use the data you generated in Section 1 to train and test your model. Plot the training loss vs. the number of training steps.

4.3 3-layer GNN. Instantiate a GNN with $L = 3$ layers, $K_1 = K_2 = 5$, $K_3 = 1$ and $F_1 = 16$, $F_2 = 4$ (i.e., 16 and 4 features in the first and second layers respectively). Use the data you generated in Section 1 to train and test your model. Plot the training loss vs. the number of training steps.

5 Generalization and Transferability

In the previous section, we went over linear and nonlinear graph parametrizations of the learning model $\mathbf{\Phi}(\mathbf{x}; \mathcal{H})$, but there are several other possible choices of parametrization. For instance, we could choose $\mathbf{\Phi}(\mathbf{x}; \mathcal{H})$ to be a simple linear transform (i.e., a $N \times N$ matrix) or a fully connected neural network (FCNN). This raises the question of why we favor graph filters, and even more so GNNs, over linear transforms and FCNNs. We start to answer this question by comparing generic linear transforms with graph filters and FCNNs with GNNs. Then, we will compare graph filters with GNNs.

5.1 Linear parametrization vs. graph filter. Implement a generic linear parametrization (i.e., a matrix mapping N nodes to N nodes) to solve the source localization problem. Train and test this parametrization and a graph filter with same hyperparameters as in 4.1 on 10 random realizations of the data and of the graph. Report the number of parameters of both models the mean error achieved by each of them. What do you observe?

Hint: Check `torch.nn.Linear`.

5.2 FCNN vs. GNN. Implement a FCNN with 2 layers and 25 hidden units in the first layer to solve the source localization problem. Train and test this parametrization and a GNN with same hyperparameters as in 4.2 on 10 random realizations of the data and of the graph. Report the number of parameters of both models the mean error achieved by each of them. What do you observe?

5.3 Graph filter vs. GNNs. Train and test a graph filter with same hyperparameters as in 4.1 and GNNs with same hyperparameters as in 4.2 and 4.3 on 10 random realizations of the data and of the graph. Report the number of parameters of both models and the mean error achieved by each of them. What do you observe?

In applications where graphs are large or dynamic, it is sometimes impractical to train the GNN on the full network or to re-train it every time the graph changes. This is the case, e.g., of source localization for fake news detection on social networks, which are large and dynamic graphs. In general, we want to be able to train GNNs on moderately sized graphs and apply them to similar graphs that are large and/or dynamic. Explicitly, given a GNN $\boldsymbol{\Phi}(\mathbf{x}_1; \mathbf{H}, \mathbf{S}_1)$ trained on the graph $\mathbf{S}_1 \in \mathbb{R}^{N_1 \times N_1}$, we want to make predictions on the graph $\mathbf{S}_2 \in \mathbb{R}^{N_2 \times N_2}$ using the GNN $\boldsymbol{\Phi}(\mathbf{x}_2; \mathbf{H}, \mathbf{S}_2)$, i.e., a GNN with same parameter tensor $\mathbf{H}$ but with different GSO. It is then important that to assess whether GNNs can retain good performance when they are transferred between graphs.

5.4 Adapting the GNN model. Adapt your GNN model by writing a method that takes a new GSO as input and replaces the old GSO with the new GSO.

5.5 Training the GNN on a small graph. Train and test GNNs with same hyperparameters as in 4.2 and 4.3 using the graph and the data you generated in Section 1. Report the test error achieved by each model and save both models.

5.6 Transferability to $N_2 = 500$. Repeat item 1.1 to build a SBM graph $\mathbf{S}_2$ with $N_2 = 500$ nodes, $C = 5$ communities of size N_2/C, $p_{c_i c_i} = 0.6$ and $p_{c_i c_j} = 0.2$. Use the method you implemented in 5.4 to change the GSO of the models you saved to $\mathbf{S}_2$. Repeat step 1.2 to generate 100 input-output graph diffusion samples with $|\mathcal{S}| = M = 100$ sources on the graph $\mathbf{S}_2$. Test both GNNs on this dataset and report the errors achieved by each of them. How do these errors compare with the errors you obtained in 5.5?

5.7 Transferability to $N_3 = 1000$. Repeat item 1.1 to build a SBM graph $\mathbf{S}_3$ with $N_3 = 1000$ nodes, $C = 5$ communities of size N_3/C, $p_{c_i c_i} = 0.6$ and $p_{c_i c_j} = 0.2$. Use the method you implemented in 5.4 to change the GSO of the models you saved to $\mathbf{S}_3$. Repeat step 1.2 to generate 100 input-output graph diffusion samples with $|\mathcal{S}| = M = 200$ sources on the graph $\mathbf{S}_3$. Test both GNNs on this dataset and report the errors achieved by each of them. How do these errors compare with the errors you obtained in 5.5 and 5.6? Which GNN transfers better?

6 Report

Do not take much time to prepare a lab report. We do not want you to report your code and we don't want you to report your work. Just give us answers to items we ask. Specifically, give us the following:

Item	Report deliverable
Item 1.1	Do not report.
Item 1.2	Do not report.
Item 1.3	Do not report.
Item 2.1	Do not report.
Item 2.2	Do not report.
Item 2.3	Number of training steps. Training loss plot.
Item 3.1	Training loss plot.
Item 3.2	Training loss plot.
Item 4.1	Training loss plot.
Item 4.2	Training loss plot.
Item 4.3	Training loss plot.
Item 5.1	Number of parameters. Loss over test set. Comment.
Item 5.2	Number of parameters. Loss over test set. Comment.
Item 5.3	Number of parameters. Loss over test set. Comment.
Item 5.4	Do not report.
Item 5.5	Loss over test set.
Item 5.6	Loss over test set. Comment.
Item 5.7	Loss over test set. Comment.

We will check that your answers are correct. If they are not, we will get back to you and ask you to correct them. As long as you submit responses, you get an A for the assignment. It counts for 16% of your final grade.